

Alba: The Dawn of Scalable Bridges for Blockchains

Giulia Scaffino*†§, Lukas Aumayr*§, Mahsa Bastankhah‡, Zeta Avarikioti*§ and Matteo Maffei*†
*TU Wien
†Christian Doppler Laboratory for Blockchain Technologies and the Internet of Things
‡Princeton University
§Common Prefix

***Abstract*—Over the past decade, cryptocurrencies have garnered attention from academia and industry alike, fostering a diverse blockchain ecosystem and novel applications. The inception of bridges improved interoperability, enabling asset transfers across different blockchains to capitalize on their unique features. Despite their surge in popularity and the emergence of Decentralized Finance (DeFi), trustless bridge protocols remain inefficient, either relaying too much information (e.g. light-client-based bridges) or demanding expensive computation (e.g. zk-based bridges). These inefficiencies arise because existing bridges securely prove a *transaction's on-chain inclusion* on another blockchain. Yet this is unnecessary as off-chain solutions, like payment and state channels, permit safe transactions without on-chain publication. However, existing bridges do not support the verification of off-chain payments.**

This paper fills this gap by introducing the concept of Pay2Chain bridges that leverage the advantages of off-chain solutions like payment channels to overcome current bridges' limitations. Our proposed Pay2Chain bridge, named Alba, facilitates the *efficient*, *secure*, and *trustless* execution of conditional payments or smart contracts on a target blockchain based on off-chain events. Alba, besides its technical advantages, enriches the source blockchain's ecosystem by facilitating DeFi applications, multi-asset payment channels, and optimistic stateful off-chain computation.

We formalize the security of Alba against Byzantine adversaries in the UC framework and complement it with a game theoretic analysis. We further introduce formal *scalability* metrics to demonstrate Alba's efficiency. Our empirical evaluation confirms Alba's efficiency in terms of communication complexity and on-chain costs, with its optimistic case incurring only twice the cost of a standard Ethereum transaction of token ownership transfer.

I. Introduction

Fourteen years after its genesis block was mined, Bitcoin [53] remains the leading blockchain in terms of market capitalization. However, a diverse range of other blockchains have emerged, each appealing to users due to their specific design goals such as enhanced privacy (e.g., Monero, ZCash), high transaction throughput (e.g., Algorand), DeFi support (e.g., Ethereum), and unique technical features such as their scripting language, consensus mechanism, and cryptographic primitives. In response to the diverse ecosystem, several solutions have been introduced to facilitate cross-chain functionalities – e.g., atomic swaps, cross-chain lending – thereby enabling users to exploit the unique benefits and features each blockchain offers. These solutions are called *bridges* and their fundamental goal is to condition the occurrence of an event (e.g., transaction execution) on a target blockchain $\mathcal{L}_D$ on the occurrence of a specific event on a source blockchain $\mathcal{L}_S$.

Bridges can generally be classified as centralized or decentralized. Centralized bridges based on trusted exchanges, are efficient but have often led to significant financial losses due to hacks [4], [34], [35], fraudulent activities, or bankruptcy [10]. On the other hand, decentralized bridges, based on cryptographic protocols [58], [20], [5], [46], [59] or smart contracts [43], [61], typically trade off between expressiveness and efficiency to maintain security. In particular, atomic swaps [5], [46], [59] relay and store minimal information but do not go beyond token swaps. The most popular light-client-based bridges are fully expressive but either incur a high computational overhead (e.g., zk-based bridges [61]) or relay and store information proportional to the length of $\mathcal{L}_S$ (e.g., Simplified Payment Verification (SPV)-based bridges [30], [64]). Recent improvements on light clients [48], [31] cannot yield practical bridges as they are not compatible with major existing blockchains, requiring instead a new consensus protocol. In addition, at least a logarithmic amount of information with respect to the length of $\mathcal{L}_S$ must be relayed to maintain security, even in the latest PoW light-client-based bridge, Glimpse [44], [56]. More importantly, all LC-based bridges, including [56], [48], [31], are consensus-specific, hence connecting two different blockchains depends on their unique technical features (e.g., consensus choice of $\mathcal{L}_S$, scripting language expressiveness, and cost of $\mathcal{L}_D$), which makes their design complex and time-consuming.

The inefficiencies of current bridging solutions stem from the fact that *the triggering event has to occur on-chain*, and then one has to reliably and securely relay its occurrence on another blockchain. The first operation exacerbates an existing problem of major blockchains: their limited transaction capacity. The second task imposes significant overhead on the bridge, that either relays a lot of information or wastes substantial computational resources to generate succinct proofs.

Network and Distributed System Security (NDSS) Symposium 2025
24 - 28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241286
www.ndss-symposium.org

Both these problems can potentially be mitigated by lifting $\mathcal{L}_S$'s event off-chain, leveraging Layer 2 (L2) solutions [45]. For instance, the Lightning Network [54], Bitcoin's premier scalability solution holding more than $200M, enables parties to transact with each other securely without publishing any information on the Bitcoin blockchain. Nevertheless, *existing bridges do not support off-chain protocols*: in other words, it is currently not possible to prove on a target *chain* $\mathcal{L}_D$ that an off-chain *payment* occurred on an L2 network.

It is an open question whether or not such bridge protocols, referred to as *Pay2Chain*, can be designed at all, since all existing solutions aim to prove that a transaction has been validated on-chain, i.e., by the consensus protocol itself. In contrast, off-chain transactions are ideally never posted on-chain (except in case of disputes); hence, proving their validity requires fundamentally new ideas. Moreover, the economic security of a Pay2Chain bridge should account for the game-theoretic arguments tailored to off-chain protocols. Finally, Pay2Chain bridges' foremost challenge is to improve the scalability and overall efficiency of bridge protocols, overcoming the limitations of current solutions.

In the pursuit of scalable Pay2Chain bridges,we explore two distinct design options based on leading L2 solutions: payment channel networks and rollups. Rollups, rapidly gaining momentum as an efficient off-chain solution, leverage the base blockchain (L1) for data availability, meaning transactions are sequenced on-chain but execution is off-chain [9], [17], [14]. This structure means verifying a rollup transaction is akin to verifying a transaction in a "dirty ledger", a blockchain that logs transactions including potentially invalid ones. Thus, rollups inherit its L1's bridging limitations. Given these constraints, our focus shifts to payment channels, that effectively lift the transaction workload and storage off-chain. This shift not only proposes a novel blueprint for bridge construction but also ensures compatibility with a wide spectrum of blockchains, as payment channels are virtually compatible with any chain that supports signature verification [60], [20].

Our Contributions. In this work, we present Alba, the first scalable Pay2Chain bridge connecting the Lightning Network to EVM-based chains. At its core, Alba encodes the logic to verify that a specific transaction occurred in a Lightning channel in a smart contract on a target chain $\mathcal{L}_D$, e.g., Ethereum. Unlike traditional bridges, Alba verifies off-chain transactions instead of on-chain ones, thus offering several key advantages.

Alba leverages the nature of payment channels that enable two parties to securely transact with each other by exchanging their signatures on a message. These signatures along with two transactions are the only data the Alba smart contract verifies on the target chain. As a result, the relayed information of the bridge is easily computable, constant in size, and independent of the source chain's length or its consensus mechanism. Thus, Alba offers the first lightweight bridge that scales in storage, computation, and communication.

Moreover, Alba's consensus-agnostic nature enables its seamless deployment on any chain with a payment channel network, regardless of its underlying consensus mechanism. Unlike on-chain bridges, Alba achieves instant finality of payments, solely bounded by network delay, rather than relying on the liveness guarantees of the $\mathcal{L}_S$'s consensus protocol.

Besides its improved technical features, Alba enhances the functionalities of payment channel networks, such as the Lighting Network. It serves as a trustless protocol for bringing backed assets from the target chain, e.g. Ethereum, into Lightning. As a result, users can transact off-chain virtual representations of their ($\mathcal{L}_D$) tokens, known as wrapped tokens, thereby reducing fees. Moreover, Alba can be used for collateralized, trustless, cross-chain lending, allowing users to hold coins on one chain but participate in applications of another chain. Another benefit of this functionality is bringing arbitrary, stateful computation to scriptless blockchains. Users can now perform arbitrary computations off-chain (e.g., play chess), store, and optimistically settle the outcome in their L2 channel. Security is ensured because the correct outcome can always be enforced on the target chain in case of disagreement.

Finally, by employing Alba, the source blockchain is relieved of on-chain transactions as all transactions remain off-chain within the Lighting Network. Additionally, lifting the Alba smart contract off-chain, as outlined in [56], can offer similar advantages to the destination chain, promoting the adoption of L2-L2 interoperable solutions.

We summarize the advantages of Alba over state-of-the-art trustless bridge solutions in Table I.

Our contributions are summarized as follows:

- We present the key building blocks of Alba, including formal definitions of bridge security and scalability, followed by an overview of the Alba protocol (Section II).
- We give an instantiation of Alba between the LN and Ethereum (Section III), and we describe the rationale behind its design choices.
- We identify three novel classes of applications which are enabled by Alba (Section IV) and we demonstrate how they improve the state-of-the-art of both the LN and the target chain. In particular, we describe how to use Alba for DeFi applications, trustless backed assets in payment channels, and to optimistically offload arbitrary, stateful computations to L2.
- We prove Alba is scalable (Section V-A), as well as secure under two different models. We first prove the security of Alba in the UC framework [32] (Section V-B) showing that honest users do not lose money even against byzantine adversaries (balance security property). We then prove via game-theoretic tools that rational parties follow correctly the protocol specification (Section V-C).
- We conduct a performance evaluation (Section VI), characterizing Alba's communication complexity, and its on-chain costs on EVM-based chains, demonstrating its practicality. In the optimistic scenario, Alba costs only twice as much as the simplest Ethereum transaction.
- We compare Alba with state-of-the-art bridges (Section VII), showing how Alba improves on prior constructions introduced in Table I.

	SPV-Based Bridges [64], [30], [43]	ZK-Based Bridges [61], [63]	Glimpse [56]	Multi-Hop Cross-Chain Atomic Swap [50]	**Alba**
Consensus Agnostic:	✗	✗	✗	✓	✓
Instant Finality:	✗	✗	✗	✓	✓
Transactions on $\mathcal{L}_S$:	1	1	1	$0^\dagger$	$0^\dagger$
Transaction Units on $\mathcal{L}_D$:	$O(\lvert\mathcal{L}_S\rvert)$	$O(\lvert\mathcal{L}_S\rvert)$	$O(\log\lvert\mathcal{L}_S\rvert)$	$O(1)$	$O(1)$
Expressiveness:	Arbitrary logic	Arbitrary logic	DNF formulas	Secret-based logic	Arbitrary logic

TABLE I: Comparison of Alba with other state-of-the-art trustless bridges regarding their consensus agnosticism, finality property, on chain transactions on $\mathcal{L}_S$ and $\mathcal{L}_D$, and expressiveness. We define the transactions on $\mathcal{L}_D$ in terms of transaction units, i.e., the size of a standard transaction of token ownership transfer on $\mathcal{L}_D$. We denote the length of a ledger $\mathcal{L}$ as $|\mathcal{L}|$. † For multi-hop cross-chain atomic swaps and Alba we omit considering the two transactions posted on-chain for opening and closing the channel, as they are posted only once for any arbitrary number of off-chain transactions.

II. Preliminaries & Overview

A. Preliminaries and Key Building Blocks

The UTXO Transaction Model. On a blockchain, each user U is identified by the public key of a digital signature key pair $(\mathsf{pk}_U, \mathsf{sk}_U)$, proving ownership over a coin. In the Unspent Transaction Output (UTXO) model, a UTXO object holds some units of currency, *coins*, and a set of instructions that specify the requirements in order to spend those coins, e.g., the signature corresponding to which public key can spend the coins. A transaction Tx is an atomic update of the state of the blockchain and maps a non-empty list of inputs, i.e., unspent outputs, to a non-empty list of newly created outputs. In other words, a transaction consumes some UTXOs and creates new UTXOs. In Bitcoin, a transaction includes (i) inputs, which uniquely identify unspent outputs, (ii) outputs, which specify the amount of currency held by the output and the conditions under which the coins can be spent, and (iii) witnesses, which store the data fulfilling the spending conditions of the inputs.

The Lightning Network. The core idea of payment channels is to let two users lock funds in a shared account – the payment channel – and thereafter execute transactions off-chain by adjusting the allocation of these funds between them, e.g. [54], [38], [26], [51], [19], [27], [40], [28]. Upon establishing the channel, the users can perform as many transactions as they want off-chain and only post one on-chain when closing the channel or when a disagreement arises. As a result, users forgo expensive transaction fees while the blockchain's limited capacity is relieved.

The Lightning Network (LN) [54] is the state-of-the-art payment channel solution operating on top of Bitcoin, moving funds worth more than 200M USD and counting more than 60k channels [3]. Concretely, the LN protocol consists of three phases: open, update, and close. These phases are shown in Figure 1. In the *opening* phase, two users, e.g., Alice and Bob, post the *funding transaction* $\mathtt{Tx_f}$ on the blockchain, where they jointly lock up some coins in a shared output. For example, in Figure 1 Alice locks x_1 coins, Bob locks y_1 coins, and they create an output θ holding $x_1 + y_1$ coins. This output can only be spent if both Alice and Bob provide their signatures $\sigma_{A,B} := \{\sigma_A, \sigma_B\}$ over the transaction that spends the shared output, thus expressing their agreement on how the coins should be spent. Before posting $\mathtt{Tx_f}$ on-chain, Alice creates a transaction that spends the shared output θ and returns to her and Bob their initial funds (of value x_1 and y_1 respectively, in our example). Alice signs this transaction – the first *commitment transaction* or *channel state* – and sends it along with her signature to Bob. Similarly, Bob creates, signs, and sends Alice an (asymmetric) commitment transaction, which spends the shared output and returns their initial funds. Upon exchanging the first commitment transactions, the users can post $\mathtt{Tx_f}$ on-chain, safely opening their channel, certain that their counterparty cannot hold their funds hostage.

After $\mathtt{Tx_f}$ is published on-chain, the *update* phase begins: Users can now perform arbitrarily many payments by simply exchanging commitment transactions spending the shared output of $\mathtt{Tx_f}$ and holding new balance distributions. Specifically, for each channel update, Alice and Bob sign and exchange an asymmetric version of the commitment transaction, denoted by $\mathtt{Tx^A}$ and $\mathtt{Tx^B}$ [54], [52]. Both transactions spend the funding output θ, and create two new outputs that allocate to each user their currently agreed share of the channel funds, albeit with different spending conditions. $\mathtt{Tx^A}$ allows Alice to redeem her coins only after a timeout T (timelocked output), while Bob can redeem his coins immediately. Additionally, Bob may spend Alice's coins if he provides the preimage of a specific hash (hashlocked output). $\mathtt{Tx^B}$ is asymmetric meaning that the timelocked and hashlocked outputs are the ones holding Bob's coins, while Alice can redeem her coins immediately providing only her signature.

Apart from the spending conditions, the safety of a channel also relies on the order of operations that take place during each update. In particular, a channel update s_i consists of three steps: (i) Alice chooses a secret $r_{A,i}$, Bob chooses a secret $r_{B,i}$ and they hash their respective (revocation) secrets obtaining $R_{A,i} := \mathcal{H}(r_{A,i})$ and $R_{B,i} := \mathcal{H}(r_{B,i})$, where $\mathcal{H}$ is a hash function modeled as a random oracle. Alice gives $R_{A,i}$ to Bob and Bob gives $R_{B,i}$ to Alice. (ii) Alice and Bob exchange *signed commitment transactions*, which means that they create new commitment transactions $\mathtt{Tx}_i^A$ and $\mathtt{Tx}_i^B$, and Alice gives to Bob $\mathtt{Tx}_i^B$ with her signature $\sigma_A(\mathtt{Tx}_i^B)$, while Bob gives Alice $\mathtt{Tx}_i^A$ with his signature $\sigma_B(\mathtt{Tx}_i^A)$. Finally, (iii) they exchange their *old revocation secrets*, i.e., Alice gives $\bar{r}_A := r_{A,i-1}$ to Bob and Bob gives $\bar{r}_B := r_{B,i-1}$ to Alice. This final step ensures that the previous state of the channel can be invalidated (aka revoked) if posted on-chain while the cheating party can be punished.

Finally, in the *closing* phase, users can close the channel by posting the last state of the channel on-chain, either in collaboration with the counterparty or unilaterally. In the former

case, users cooperate and sign a transaction $\mathtt{Tx_c}$ that allows them to claim their balances immediately. In the latter case, a user, say Alice, can close the channel single-handedly by publishing the latest signed commitment transaction, say $\mathtt{Tx^A}$. Now, Alice can only spend her output after the dispute period T elapses. This timelock acts as a protection mechanism for Bob: If Alice cheats and publishes on-chain an old state of the channel $\overline{\mathtt{Tx}}^{\mathtt{A}}$, Bob can steal Alice's coins by revealing the *old revocation secret* of Alice for that transaction $\bar{r}_A$ within the dispute period T. In this way, LN channels guarantee that honest users are always able to enforce on-chain either the latest agreed state of the channel or a state where they get all the funds of the channel.

Payment channel protocols offer several beneficial features. First, they guarantee *balance security*, meaning that an honest user will not lose money, even in the presence of arbitrarily many byzantine adversaries. Second, in a rational setting, *instant finality* is guaranteed: a channel update can be considered final as soon as users exchange their respective commitment transactions. Instant finality is solely constrained by the communication bandwidth.

Bridges. A blockchain is the output of a Byzantine fault-tolerant state machine replication (BFT-SMR) protocol. Breaking down this definition, a state machine stores, at any point in time, the state of the system. It receives a set of inputs, e.g., transactions, and it applies these inputs in a sequential order using a state transition function to generate an output and the newly updated state. BFT-SMR protocols ensure that a network of nodes, each running a replica of the same state machine, agree on the order and execution of state transitions to maintain a consistent state across all replicas, even if some nodes fail or behave maliciously. In other words, BFT-SMR protocols yield what we commonly refer to as *ledger*. We will use the terms ledger and blockchain interchangeably.

In the following, we refer to states and state transitions borrowing the terminology of distributing computing to provide formal definitions.

Definition 1 (Ledger)**.** A ledger is a finite ordered list of state transitions.

Starting from the definition of a ledger, we come to define what a bridge is, i.e., a protocol that ensures the atomic execution of events across two different ledgers.

Definition 2 (Bridge)**.** Let $\Gamma(\mathcal{L}_S, \Delta s_S)$ be the algorithm running on $\mathcal{L}_D$ that, on input $\mathcal{L}_S$ and a valid state transition Δs_S of $\mathcal{L}_S$, outputs a valid Δs_D to be applied to $\mathcal{L}_D$. A bridge is a protocol which runs $\Gamma(\mathcal{L}_S, \Delta s_S)$ as a subroutine and enforces Δs_D on $\mathcal{L}_D$ if and only if Δs_S was previously enforced on $\mathcal{L}_S$.

We now introduce, for the first time, *scalability metrics for bridges*, extending [25]. Our metrics capture the required resources in terms of storage, computational, and communication complexity per call to the bridge protocol. We denote as a *transaction unit* the size of an average state transition, e.g., a simple transaction that transfers ownership over a UTXO.

Definition 3 (Storage Complexity)**.** The storage complexity of one call to the bridge is the amount of data in transaction units that has to be stored in both $\mathcal{L}_S$ and $\mathcal{L}_D$ for that call.

Definition 4 (Computational Complexity)**.** The computational complexity of one call to the bridge is the runtime of $\Gamma(\mathcal{L}_S, \Delta s_S)$ with respect to $\mathcal{L}_S$.

We observe that Δs_S represents a single state transition (with polynomial complexity), whereas $\mathcal{L}_S$ encompasses the aggregate of all state transitions from the inception of the ledger. If the computational resources of a bridge depend on the ever-increasing $\mathcal{L}_S$, the necessary resources will continuously grow. Our goal is to design lightweight bridges that forgo this reliance on the source ledger $\mathcal{L}_S$.

Definition 5 (Communication Complexity)**.** The communication complexity of one call to the bridge is the size of the output of $\Gamma(\mathcal{L}_S, \Delta s_S)$.

We refer to a bridge as *scalable* when the storage, computational, and communication complexity is constant regardless of the size of the source chain $\mathcal{L}_S$.

Definition 6 (Scalable Bridge)**.** A scalable bridge is a bridge protocol with $O(1)$ storage, computational, and communication complexity.

We observe now that existing bridges each fail to satisfy at least one of the complexity measures of scalable bridges. For example, SPV-based designs relay the entire source chain and thus have linear storage and communication complexity. Zk-based bridges improve upon storage and communication complexity but exhibit excessively high computational complexity to produce succinct proofs.

In the context of L2 solutions (i.e., payment channels, state channels, and rollups), we highlight that *payment channels currently constitute the only possible choice for designing scalable bridges*. This stems from their distinctive attributes: state transitions within payment channels (i.e., commitment transactions) are independent of the consensus protocol of the underlying ledger. As a result, a bridge can prove to $\mathcal{L}_D$ that a state transition occurred by relaying and storing only a constant amount of data, irrespective of the state of their underlying ledger, and with no additional computation. Contrarily, rollups are intrinsically dependent on the underlying ledger for their operation, relying on it to ascertain the sequence of the rollup's state transitions, which in turn yield the rollup's state. Therefore, for rollups, a bridge must relay and store ledger data as evidence of the execution of a rollup's state transition. The storage and communication complexity of a bridge based on rollups mirrors that of on-chain bridges.

B. Alba Overview

We introduce Alba, a *Pay2Chain scalable bridge* which enables users to *efficiently*, *securely*, and *trustlessly* condition a state transition in an EVM-based destination chain $\mathcal{L}_D$ on a specific state transition occurring in a payment channel.

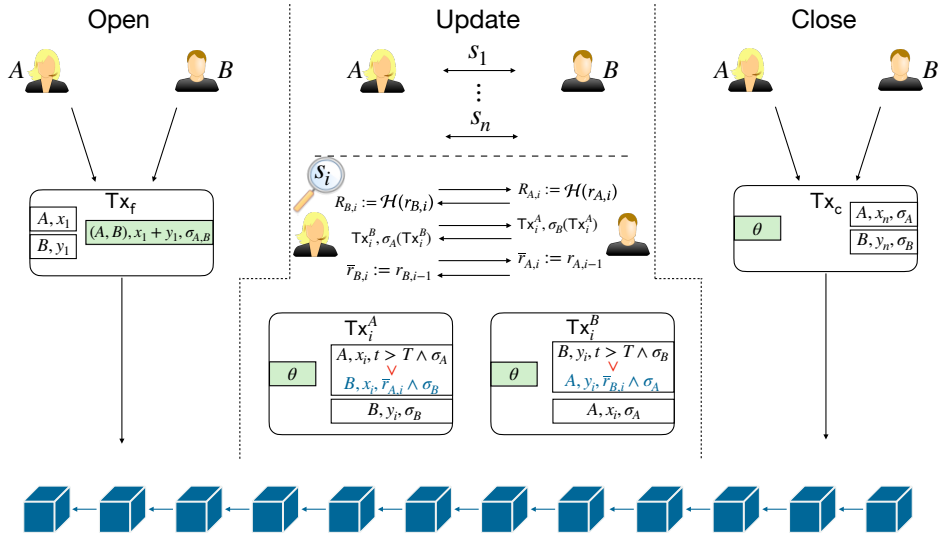

Fig. 1: Overview of the open, update, and close phase of a LN payment channel. Transactions have inputs on the left and outputs on the right. Each output is a tuple representing who owns the output, the coins it holds, and the data that unlocks it. We color in green the shared output of the funding transaction $\mathtt{Tx_f}$ and the inputs spending it. We show the communication steps of a channel update s_i and we illustrate the logic of the outputs of $\mathtt{Tx^A}$ and $\mathtt{Tx^B}$. In blue we outline the punishing mechanism logic.

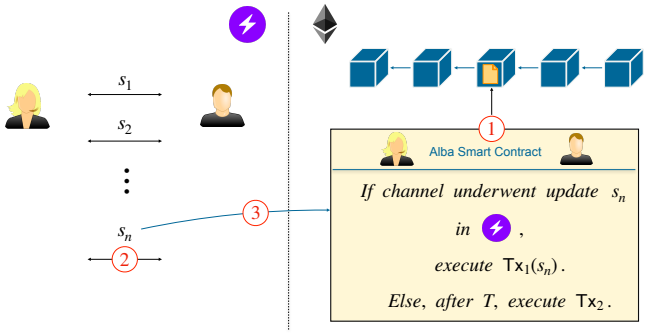

Fig. 2: Abstractly, the flow of our Alba protocol, with a LN payment channel on the left and Ethereum on the right and the three mains steps highlighted: (1) the two parties set up the contract on Ethereum with the Alba logic within, (2) they update the channel as many times as they want until they reach the update they want to verify on Ethereum, and (3) they relay a proof of such an update to the contract.

As depicted in Figure 2, Alba comprises three main phases: (1) users set up a contract on $\mathcal{L}_D$, they lock some coins in it, they specify to the contract the funding transaction of their channel as well as the channel update that conditions the execution of a transaction on $\mathcal{L}_D$. Then, (2) users perform arbitrarily many channel updates until they reach the update in question. Finally, (3) a user, acting as prover P, relays to the contract on $\mathcal{L}_D$ a proof π proving the particular update occurred. The contract checks the proof and, if it is valid, it enables the execution of the corresponding transaction.

We stress that P can convince the smart contract by exhibiting a proof π consisting only of the two signed commitment transactions representing the state of the LN channel after the update. This is possible because we enforce P's counterparty V, to embed some protocol-relevant information in the commitment transaction it gives to P, to prevent a malicious P from creating π out of two arbitrary, perhaps old, commitment transactions. Notably, Alba remains secure without verifying that the funding transaction of the channel has been actually included on $\mathcal{L}_S$. This is because in the setup phase of Alba parties agree on which is the funding transaction and acknowledge its existence in the contract: Rational parties will make sure the channel exists and never agree on an incorrect funding transaction as it conditions the distribution of funds on $\mathcal{L}_D$. For this process, parties do not need to read and use data from the source chain $\mathcal{L}_S$ to generate the proof. Finally, parties can only go on-chain with an old revoked state, thus exposing a malicious close to punishing. Hence, Alba's proof is *agnostic of the consensus of* $\mathcal{L}_S$.

To satisfy Definition 2, the Alba smart contract implements a punishing mechanism. We prove later that the Alba design is secure against a Byzantine counterparty, while rational parties are incentivized to adhere to the protocol specification (Section V). Furthermore, we demonstrate the superior performance of Alba, proving it scales in storage, computation, and communication (Section V-A), which we back with the findings of our evaluation (Section VI) and the comparison to existing bridges (Section VII). In the rational setting, Alba additionally achieves instant finality, inherited directly from the use of payment channels that ensure that any update is final

as soon as users complete the required communication steps. Note that instant finality is solely bounded by communication bandwidth, in contrast to, e.g., fast finality, which is bounded by the liveness parameter of the consensus mechnanism. Finally, Alba seamlessly integrates with existing payment channel solutions by design, offering an opt-in approach: it only requires one additional round of communication between parties, which can take place either on the payment channel system itself or it can occur concurrently.

III. Protocol Design

In this section, we present the construction of Alba and motivate its design choices by exemplifying Alba for a DeFi lending protocol, an application which we later describe in more detail in Section IV-A. Adopting a straw man approach, we start with a naive construction, delineate its vulnerabilities and challenges, and present solutions, systematically advancing towards the final, secure construction. We outline the protocol steps and the logic of the smart contract in Figure 3.

A. The Alba Protocol

We consider *mutually distrusted* parties having an *open channel* on the LN and an account, i.e., a key pair $(\mathsf{sk}, \mathsf{pk})$, on a destination chain $\mathcal{L}_D$, e.g., Ethereum. We assume parties to *continuously monitor $\mathcal{L}_D$ for the duration of the protocol* for certain transactions to appear: They can achieve this, e.g., by running a full node, a light client, or by querying a (trusted) Blockchain Explorer.

For simplicity, from now on, we assume that one coin in LN (Bitcoin) has the same value as one coin in Ethereum. In practice, an exchange rate can be fixed, or wrapped Bitcoin (WBTC) on Ethereum can be used.

A Naive Construction. Imagine that Bob wants to grant Alice a loan of 5 BTC on the LN, on the condition that Alice locks, e.g., 5 ETH in an Ethereum smart contract as collateral. Alice gets back her collateral only if she can prove, within a certain finite time T_0, to the smart contract that she has returned the loan to Bob by updating the channel accordingly, i.e., $A \xrightarrow{5} B$. If she defaults on the loan, Bob keeps the collateral. We expand on this application in Section IV-A, while for now we focus on the specific design choices of Alba.

In a preliminary setup phase, Alice and Bob inform the contract of their channel's funding transaction $\mathtt{Tx_f}$, of timelocks T_0 (timeout for submitting a proof), T_1 (timeout for resolving a dispute), T_2 (timeout for closing Alba in case no proof has been submitted nor dispute has been raised), and a function f that maps the distribution of coins in their channel to the balance distribution in the contract.

Let us demonstrate the potential vulnerabilities of a naive construction. Consider the case where Alice and Bob update their channel with $A \xrightarrow{5} B$, and then Alice naively proves to the smart contract that the update occurred by presenting $\mathtt{Tx^A}$ signed by her and by Bob. This exposes Alice and Bob to a significant risk: if the commitment transaction $\mathtt{Tx^A}$ and the two signatures $\sigma_A(\mathtt{Tx^A}), \sigma_B(\mathtt{Tx^A})$ are submitted to the contract, they are published on-chain and leaked to everyone, also to users external to the protocol. By having a commitment transaction and Alice's and Bob's signatures, anyone could close the channel between Alice and Bob. To prevent this, Alice could, instead, provide to the contract the two commitment transactions $\mathtt{Tx^A}$ and $\mathtt{Tx^B}$, along with $\sigma_B(\mathtt{Tx^A})$ and $\sigma_A(\mathtt{Tx^B})$.

This naive approach still presents some caveats: (i) the contract has no means to check whether Alice has indeed sent 5 coins to Bob within the update, as commitment transactions only contain information about the current channel's balance distribution, and not about the amount transferred from one party to the other. Then, a dishonest Alice could fool the contract in multiple ways: (ii) by submitting an old channel update, e.g., occurred prior to setting up the Alba contract, and (iii) by submitting commitment transactions corresponding to different channel updates, e.g., $\mathtt{Tx^A}$ and $\mathtt{Tx^B}$ corresponding to the i-th and $(i-4)$-th update of the channel, respectively.

Relay Channel Balance to $\mathcal{L}_D$. To verify that the channel update presented by Alice corresponds to $A \xrightarrow{5} B$, the contract needs to know the parties' balance distribution before the update. For instance, Alice and Bob could inform the contract about their channel balance distribution during the setup phase and then proceed with the update $A \xrightarrow{5} B$. This approach has, however, a major drawback: after setting up the contract, Alice and Bob have to update with $A \xrightarrow{5} B$ immediately after, without being able to perform any other intermediary update until Alba is resolved. To avoid this, one could give the contract access to the channel state prior to $A \xrightarrow{5} B$. For instance, if the contract had access to the commitment transactions of the latest update where, e.g., Alice holds 8 coins and Bob 2 coins, and then the contract is given the commitment transactions of the new update $A \xrightarrow{5} B$ where, e.g., Alice holds 3 coins and Bob holds 7 coins, the contract can infer that Alice has sent 5 coins to Bob. While solving problem (i), this approach requires a large proof (four commitment transactions) and still suffers from caveat (ii) where a dishonest Alice may submit old updates.

Embed Protocol-Relevant Information in $\mathtt{Tx^A}$. To prevent Alice from cheating and, at the same time, to enable the contract to verify the validity of the update $A \xrightarrow{5} B$, we propose a solution that allows for *arbitrarily many channel updates while the contract is active* and requires a *succinct proof π consisting of only two transactions*. We ask Bob to *embed protocol-relevant information in the commitment transaction* $\mathtt{Tx^A}$ of the update $A \xrightarrow{5} B$.

First, a unique identifier id for the update, which allows parties to perform as many updates as they want, and then mark the one they want the contract to verify, and also prevents Alice from submitting a fake proof consisting of an old update, addressing caveats (i) and (ii). Second, Bob embeds in $\mathtt{Tx^A}$ the hash $R_B := \mathcal{H}(r_B)$ of Bob's revocation secret r_B, the same one that in the LN, by default, is included in $\mathtt{Tx^B}$ for the punishment mechanism. This allows the contract to recognize $\mathtt{Tx^B}$ as the commitment transaction matching $\mathtt{Tx^A}$, addressing caveat (iii). In the UTXO transaction model, where transactions map a non-empty list of inputs

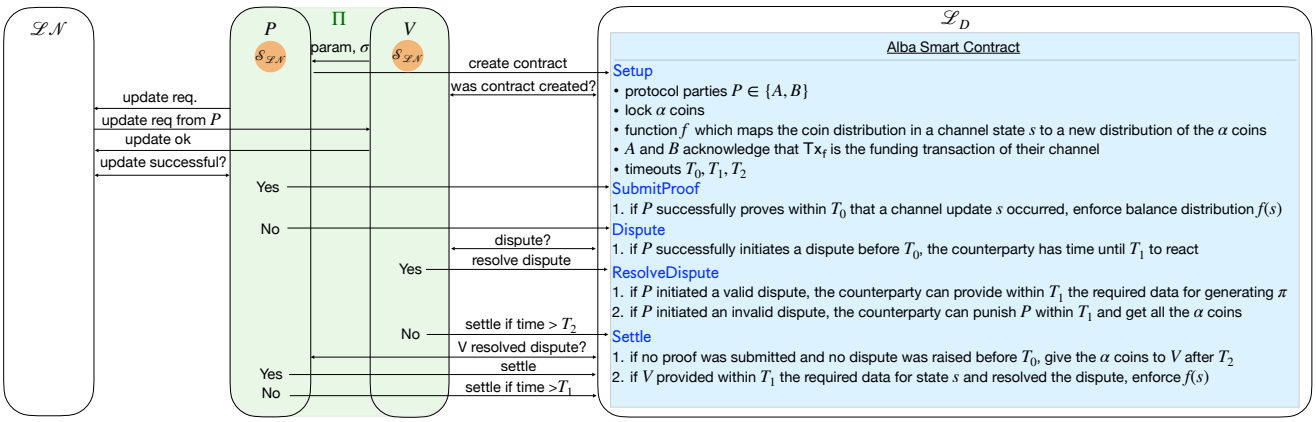

Fig. 3: Overview of the Alba protocol Π and of the smart contract on $\mathcal{L}_D$ (light blue). First, P and V agree on Alba's parameters and create a smart contract on $\mathcal{L}_D$ according to these (Setup). They perform an arbitrary number of LN channel updates until they hit the update conditioning the execution of a transaction on $\mathcal{L}_D$. If the update is successful within T_0, P relays π to the contract (SubmitProof), otherwise it opens a dispute (Dispute). Meanwhile, V monitors $\mathcal{L}_D$ looking for disputes: if P opened one, V can resolve it before T_1. If no π was submitted nor dispute raised before T_2, V get all the coins in the contract (Settle). In our protocol formalization, $\mathcal{L}_D$ and the LN are captured by generic ideal functionalities $\mathcal{G}_{Ledger}$ and $\mathcal{PC}$ respectively (see Section V). To extract LN-protocol-specific information for π (transactions and signatures), P and V internally run the simulator code $\mathcal{S}_{\mathcal{LN}}$.

(i.e., unspent outputs) to a non-empty list of newly created outputs, an easy way to store information in a transaction is to use OP_RETURN outputs, which can hold 0 coins and store arbitrary data of maximum 80 bytes. In Bitcoin, and therefore in the LN as well, such an output θ would look as follows: $\theta = (0, \texttt{OP_RETURN}(\mathsf{id}, R_B))$.

However, embedding information in $\mathtt{Tx}^{\mathtt{A}}$ does not lead us to a complete solution yet, as it does not prevent a malicious party from closing the channel with an old, un-revoked state before settling the contract, see below.

Impose Transaction-Level Timelocks on Updates. While the Alba contract is active, parties should not be able to go on-chain and close the channel with an *old, unrevoked* state. In normal channel operation, the latest state of the channel is an un-revoked state. However, during channel updates, one party will receive the counterparty's revocation secret before revealing its own (fair exchange problem). This results in one party having an advantage over the other, holding two un-revoked, valid states, and being able to close the channel with either of them without being punished. Note here that if a cheating party closes the channel with an old, revoked state, the security of Alba is not affected as the counterparty can punish it and claim its funds on the payment channel.

To avoid parties from closing the channel with an old, un-revoked state, we impose timelocks at the transaction level on all commitment transactions exchanged from the creation of Alba until its settlement. By definition, a transaction-level timelock invalidates a transaction until a certain time, even if its script and witness are correct.

During Alba, a channel update now consists of the following steps: first, parties exchange their new revocation keys; second, they exchange signed *locked* commitment transactions ($\mathtt{Tx}^{\mathtt{A}}_1$ and $\mathtt{Tx}^{\mathtt{B}}_1$); third, they exchange their old revocation secrets. Finally, when parties reach the update they want to verify via Alba, i.e., $A \xrightarrow{5} B$, an extra communication round occurs where parties exchange signed *unlocked* commitment transactions ($\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}}$ and $\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}$). The transaction-level timelock expires after the Alba contract has been settled (after T_2), and prevents malicious parties from going on-chain and closing the channel while Alba is still active. Since the Alba instance is closed upon revealing unlocked transactions, as soon as Alba is settled, standard LN updates and closure resume.

To summarize, the proof π given as input to the function of the Alba contract that verifies said proof, called SubmitProof, is

$$\pi := (\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}}, \sigma_B(\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}}), \mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}, \sigma_A(\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}})),$$

with $\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}}$ embedding id and R_B.[1]

Yet another issue remains unresolved: if one party does not cooperate in an update, e.g., by refusing to send the commitment transaction or the old revocation secret, the counterparty cannot immediately close the channel due to the transaction-level timelock we introduced, thus yielding a hostage situation. For example, Bob may not respond to Alice's request to update the channel, to keep the collateral on the Ethereum, and forfeit the money he lent on the LN. In this situation, Alice is at the mercy of Bob, as she cannot update or close the channel as well as redeem the coins in the contract.

Incentivize Cooperation in Channel Updates. To protect honest users from such hostage attacks, we empower Alice with some leverage to exert economic pressure on an uncooperative Bob, ensuring that updates can be carried out and

[1]When verifying the current latest state of the channel, all inputs refer to the current latest state. When verifying the occurrence of a new update, all inputs refer to the new update.

successfully completed. Specifically, we economically force (a rational) Bob to update the channel by introducing the following incentive mechanism. If Bob does not honestly update the channel within time T_0, Alice can *open a dispute* by calling the Dispute function of the contract. Alice inputs to this function the latest signed locked commitment transaction $\mathtt{Tx}_1^{\mathtt{A}}$ received from Bob, and the new signed unlocked commitment transaction $\mathtt{Tx}_{\mathtt{unl}}^{\mathtt{B}}$ she created for the update $A \xrightarrow{5} B$ she wants to enforce.[2] Bob has now time until T_1 to call the ResolveDispute function and reveal the signed unlocked commitment transaction $\mathtt{Tx}_{\mathtt{unl}}^{\mathtt{A}}$, thereby completing the proof π to the smart contract. If, however, Alice cheats and opens a dispute with an old state $\overline{\mathtt{Tx}}_1^{\mathtt{A}}$, Bob can punish Alice by calling the ResolveDispute function and revealing the revocation secret for $\overline{\mathtt{Tx}}_1^{\mathtt{A}}$.

Let us now examine a critical but subtle case where Alice halts during a channel update and subsequently claims on the Alba contract that Bob did so. In detail, suppose Alice does not share with Bob the revocation secret for the previous state during a channel update. Bob will then not share with Alice the unlocked transaction (i.e., timelock-free transaction) as for Bob this update is not complete. In turn, Alice opens a dispute on $\mathcal{L}_D$, reclaiming the unlocked transaction. Observe now that Alice cannot close the channel on-chain with the old state since the commitment transaction is locked until after T_2. Bob can now obtain the *unlocked* transaction $\mathtt{Tx}_{\mathtt{unl}}^{\mathtt{B}}$ from the contract, close the channel (as Alice proved to be malicious) and disclose $\mathtt{Tx}_{\mathtt{unl}}^{\mathtt{A}}$. Thus, even in this case, the closing state of the channel correctly reflects the outcome on $\mathcal{L}_D$.

B. Observations and Technical Features

The Alba protocol exposes two functionalities: (i) it enables parties to verify specific channel updates (as for the loan payback transaction); (ii) it enables parties to verify which is the current state of a channel. Note (ii) is a subcase of (i) where the channel is updated with the identity function which transforms a state into itself.

We observe that Alba, as described above, is a uni-directional bridge. It can be made bi-directional by simply instantiating it twice, symmetrically on L2-L1, where L2 is a payment channel and L1 is a chain supporting the scripting requirement described in detail in Section B. We also note that, generally, both parties can submit the proof to the smart contract: whoever submits the proof first or, alternatively, opens a dispute, acts as the *prover* P. The other party, instead, acts as the *verifier* V. Note that even though both parties can submit proofs and the other party reacts, there is still one party in this case, that gets all the funds after T_2, to incentivize the other party to close. We now look at how to remove this timelock.

Unlimited Lifetime of Alba. If Alice and Bob omit to specify T_0 and T_2 during setup, no time limit is imposed on parties for the submission of a proof, thereby deeming the lifetime of the Alba contract unlimited. In this case, should a dispute arise, a relative timelock T_1 starts counting from the moment the dispute is raised. Users simply need to ensure that do not exist un-revoked commitment transactions with an expired timelock (unless they wish to close Alba). This timelock acts as T_0 and can be repeatedly extended with state updates. With this, the contract is more flexible, while preventing hostage situations.

Compatibility of Alba. Many payment channel constructions beside the LN exist, and Alba can be run on top of all of them with only minor adjustments.

Furthermore, by construction, Alba is compatible with *virtual channel* constructions [23]. Virtual channels are protocols for payment channel networks that allow the exchange of off-chain transactions over multi-hop paths without involving intermediary users. One can think of a virtual channel as a payment channel within a payment channel, with the funding and closing transactions of the virtual channel remaining off-chain. Parties can use Alba for virtual channels exactly as they would use it for payment channels.

Alba for Stateful Applications. As we will see in the next section, our protocol can be used as a building block in different applications. In complex applications, commitment transactions store a succinct state of the application, and the smart contract needs to be able to verify that that state is a valid transition from the previous state. We stress that the Alba contract might need to encode the application logic in order to allow parties to reach the application's final state in case a dispute is raised before it is reached. For instance, consider the case where Alice and Bob are playing chess (Section IV-C) and Bob stops responding when he realizes that Alice will inevitably checkmate his king in a few moves. In this case, all the remaining moves must be enforced on-chain by the contract, until the game is over.

IV. Applications of Alba

Perhaps surprisingly, this relatively simple idea of bridging payment channel state updates enables a plethora of novel applications which can be divided into three main categories: *decentralized finance*, *multi-asset payment channels*, and *optimistic stateful computation* on scriptless blockchains.

A. Decentralized Finance

DeFi applications such as automated market makers (AMMs), liquidity pools, lending protocols, or flash loans, typically take place on-chain, mainly on Ethereum, resulting in a huge amount of transactions being broadcast to the network and competing for a limited block space [12]. Partially shifting the DeFi ecosystem off-chain would benefit users and the security of the chain itself, threatened by MEV-specific attacks [36], e.g., the undercutting attack.

In the following, we demonstrate how to enable collateralized lending protocols via Alba, by exposing a *functionality that allows users to prove to a smart contract on $\mathcal{L}_D$ that a specific channel update took place*.

Collateralized Lending with Alba. Assume Alice and Bob have an account on Ethereum and have a payment channel on

[2] For this, the smart contract needs to be able to check what constitutes a valid update, e.g., $A \xrightarrow{5} B$ in our lending example.

the LN. Alice wishes to borrow 5 BTC on the LN from Bob to participate, e.g., in online games [13]. Bob agrees to that on the condition that Alice locks, e.g., 5 ETH in an Ethereum smart contract as collateral. The smart contract's logic is as follows: (1) Alice locks the collateral while providing the contract with the signed commitment transaction $\texttt{Tx}^{\texttt{B}}$ where Bob grants her the loan; (2) Bob provides the contract with the signed commitment transaction $\texttt{Tx}^{\texttt{A}}$ where he grants Alice the loan, and the revocation secret $\bar{r}_B$ for the previous state of the channel; (3) If step (2) fails to take place within a timeout T_B, the contract gives back the collateral to Alice; (4) Alice provides the contract with her old revocation secret $\bar{r}_A$, thus revoking the previous state of the channel; (5) if step (4) fails to take place within a timeout T_A, the contract gives back the collateral to Bob. In other words, we have two actions which have to take place atomically: Alice locks collateral in the Ethereum contract, Bob grants the loan to Alice. To be sure that these two actions are taking place atomically, with the logic above, we require the users to grant the loan within a channel update whose communication steps take place on-chain. Alternatively, one could use an atomic swap.

Now, Alice is free to use the 5 BTC she borrowed from Bob in the way she desires. Finally, upon Alice paying back the loan, the contract needs to be able to give back to Alice her collateral. Therefore, the contract has two more steps in its logic: Alice can have back her collateral upon proving she paid back the loan. She can do this by providing the contract with a valid proof π as presented in Section III. After a timeout T_{loan}, if no valid proof has been submitted to the contract, Bob can redeem the collateral.

We note that the collateral locked in the contract could be used for *flash loans*, thus mitigating opportunity costs.

Lending to Participate in DeFi. This lending occurs between two chains, and indeed, the chain that holds the locked collateral in the ALBA contract requires Turing-complete scripting. Recent proposals, i.e., BitVM [49], conjecture that it is possible to have stateful and Turing-complete computation on Bitcoin. Should this hold, we can, in fact, use ALBA for a lending scenario that is even more interesting. Following the example above, Alice could give 5 BTC as collateral, and borrow 5 ETH on Ethereum from Bob. On Ethereum, there are many payment channel constructions, e.g., [8], [40], that facilitate this borrowing. This would effectively bring DeFi to Bitcoin, where people can hold Bitcoin, and then temporarily lend ETH to participate, e.g., in Ethereum-based DeFi applications.

B. Multi-Asset Payment Channels

By default, existing payment channels only allow transactions with the native currency of their underlying blockchain.

Recent developments such as Taproot Assets [15] and RGB [16] enable users to issue and transfer assets on Bitcoin and on the LN. However, assets issued with these two approaches are not provably backed by any fiat nor crypto asset: if an issuer claims that its Taproot Assets represent ETH, the issuer needs to be trusted to (i) hold an equal amount of ETH as collateral on Ethereum and (ii) allow users to change their Taproot Asset back to ETH on Ethereum.

Alba *trustlessy* enables *backed assets* into the LN, finally allowing users to (1) issue and transfer on the LN (and therefore, indirectly, on Bitcoin) *wrapped assets*, e.g., assets representation of non-native assets, (2) exchange them back and forth as many times as they want, and then (3) get back a corresponding amount of the original token in its native chain. Contrarily to the lending example, Alba achieves this by exposing a *functionality that allows users to prove to a smart contract on $\mathcal{L}_D$ which is the (balance distribution in their) latest state of their channel*. This application brings new life to the payment channel ecosystem by finally allowing users to transact in their own, desired tokens, while avoiding on-chain fees.

Issuing Non-Native Tokens on the LN. Consider two users who share an LN channel and wish to top it up with, e.g., some ETH. Since ETH is not a native currency in the LN, users create virtual representations of ETH, also known as wrapped ETH (WETH). They can do this by locking some ETH in an Alba contract on Ethereum (using it for flash loans to mitigate opportunity costs) and, for instance, naively representing WETH in their channel within a dedicated `OP_RETURN` output in their commitment transactions. Alternative solutions to the `OP_RETURN` can be found in [15], [16]. To create the lock on Ethereum and introduce WETH in the state of the channel atomically, similar solutions to the ones presented in Section IV-A can be used. Then, the users can transact WETH on the LN by exchanging new commitment transactions reflecting a new distribution thereof.

Back to the Token's Native Chain. To trustlessly transfer their WETH from their channel back to Ethereum, Alice or Bob can present to the smart contract a valid proof π for the latest state of their channel and the contract distributes the locked ETH according to their WETH.

Payment Channel Networks. We conjecture that in case WETH are used in several different channels of the LN, they can be sent across these channels via intermediaries, thus leveraging the payment channel network. Alternatively, if WETH exists in a payment A-B-C in channel A-B but not in B-C, A could send WETH to C via B only if B offers an ad-hoc exchange of WETH to BTC. We leave it to future work to explore this in more detail.

C. Stateful Computation on Scriptless Chains

Alba allows for the first time to build *state channels* on blockchains with limited-scripting capabilities, *without relying on additional trust assumptions external to the target chain $\mathcal{L}_D$*, contrarily to other existing solutions which use trusted execution environments [37], [42], trusted executioners, or honest majority of a quorum [62]. With Alba, one can have stateful and *quasi-Turing complete smart contracts optimistically executed fully off-chain*, with the outcome of the computation stored in an LN channel, while in case of misbehavior, the correct execution enforced by the smart contract on another chain.

Playing Chess on LN. A playful but nevertheless interesting example of stateful computation is the game of chess, where the state of the game (position of all non-captured pieces in the chessboard) has to be stored move after move, and the rules of the game have to be enforced by checking the validity of each move with respect to the current state of the game and the capabilities of the pieces. This is not possible with standard LN channels, as the underlying chain, i.e., Bitcoin, lacks statefulness and cannot enforce the rules of the game. We show how Alba enables users to play chess on the LN by exposing the *two functionalities* introduced for the two previous examples respectively: *(i) users need to be able to prove that a specific channel update occurred, and (ii) users need to be able to prove which is the current latest state of their channel*.

Our proposed construction involves two parties and an *untrusted* hub, which collateralizes the state channel on a Turing-complete blockchain. The hub is not strictly necessary, but we use it to remove the need for users to own and lock coins on a contract on $\mathcal{L}_D$. To understand how it works, we give the following example.

Setting Up the Game. Assume that two parties, Alice and Bob, wish to play a chess game in their LN channel on top of Bitcoin. We also assume that each party stakes 5 coins in the game, and the winner cashes in 10 coins.

A hub H has a LN channel with Alice and Bob (outbound capacity ≥ 10, inbound capacity ≥ 5) and some collateral ≥ 10 on Ethereum. All three *parties are mutually distrusted*. Alice, Bob, and the hub perform the following transaction atomically (e.g., with a secret-based atomic swap): (i) Alice pays 5 coins to the hub in the channel A-H, (ii) Bob pays 5 coins to the hub in the channel B-H, and (iii) the hub creates an Alba contract holding 10 coins on Ethereum.

Now, Alice and Bob start playing chess in their A-B channel, by storing in a dedicated output of their commitment transactions the current state of the game, i.e., an efficient representation of the chessboard [11]. In case both parties honestly cooperate, every time a player makes a move, parties exchange a new channel update, containing the new representation of the state of the game. Honest parties will only sign states corresponding to a valid chess move, i.e., representing a valid state transition.

Forcing Unresponsive Players to Move. The main challenge is if one player, say Bob, stops making moves, yielding to a stale situation where the game is not over, and Alice must wait for Bob's turn. This could happen, for instance, if Bob realizes he is going to lose. In this case, Alice can rationally motivate Bob to respond with a valid move by opening a dispute and posting the channel's current state (i.e., of the game) to the Alba contract on Ethereum. Bob, if he does not want to lose money, needs to make a move and post the new state of the game. Since the smart contract on Ethereum is Turing-complete, it can verify the validity of any chess move. Provided Bob's new commitment transaction, the game is not stale anymore, and parties can either continue playing off-chain, or, in the worst case, need to enforce every subsequent move on-chain (which essentially results in playing on Ethereum). If Alice opened the dispute by posting an old, revoked state of the game, Bob can prove that this is an old state using the properties of channel updates, and get away with all the money. We observe that *honest users need to monitor the Alba contract on Ethereum* to not lose money, as a malicious user can submit a claim even though the counterparty is responsive. At the end of the game, the smart contract can always acknowledge the winner and proceed with the payout.

Payout of Winnings in the LN. Since Alice and Bob started playing chess in the LN, they presumably want to cash in their winnings in LN as well. When the game is over, players provide the hub with the commitment transactions reporting the final state of the chessboard. If the hub is honest, it pays the winner 10 coins via a channel update and then discloses to the smart contract the commitment transactions of such an update. In this way, the hub gets back its collateral on Ethereum. If the hub is malicious and does not reward the correct amount of coins to the winner in LN, the winner can still cash in the ETH via the smart contract on Ethereum.

V. Analysis

In this section, we initially analyze Alba's efficiency before delving into its security aspects. We aim to demonstrate that Alba adheres to Definition 2, ensuring the atomicity of operations across participating ledgers despite potential malicious parties. Subsequently, we illustrate that, provided parties remain active (online and responsive) and behave rationally, the protocol's correct execution is assured.

To the best of our knowledge, no practical framework integrating Byzantine adversaries and rational agents yields meaningful results for complex protocols instantiated across multiple blockchains such as ours. Therefore, we split our proof technique into two parts: we first provide a standard UC proof to show Alba remains secure against Byzantine parties (Section V-B), and then perform a game theoretic analysis to demonstrate rational parties will adhere to the correct protocol execution (Section V-C).

A. Efficiency Analysis

Following the scalability metrics of Section II, we now prove that Alba has constant storage, computation, and communication complexity in the length of the source chain $\mathcal{L}_S$.

Theorem 1. *Alba has $O(1)$ storage, computation, and communication complexity with respect to $\mathcal{L}_S$, and it is therefore a scalable bridge according to Definition 6.*

Proof. The storage complexity of Alba is constant with respect to the length of $\mathcal{L}_S$, i.e., no data are stored on $\mathcal{L}_S$ and only two signed transactions are stored on $\mathcal{L}_D$ (see inputs to the SubmitProof function and *sp* variable in the Alba smart contract pseudocode in Section B). The computation complexity of Alba with respect to $\mathcal{L}_S$ is constant (zero), as Alba is independent on $\mathcal{L}_S$ and the proof is generated

out of the data of the payment channel. The communication complexity of Alba with respect to $\mathcal{L}_S$ is constant, as only two signed transactions are relayed to $\mathcal{L}_D$. For the computation and communication complexity of Alba see step 7 in the *Channel Update and Proof* phase of the protocol in Section C. □

B. Security in the UC Framework

To formally model our construction, we use the global universal composability (GUC) framework [33]. Our analysis follows [41], [39], [40], [20], [22]. We use the ideal functionality of generalized channels [20] to capture the functionality of payment channels. Exactly as in the model of generalized channels, we model synchrony with a global clock [47], authenticated communication with guaranteed delivery [39], and, finally, the ledger as an idealized append-only data structure keeping track of all published transactions [20]. We instantiate the ledger twice, once for the destination chain (which holds the Alba contract), and once indirectly for the source chain, via the channel functionality on top of the source chain. The ledger is parameterized by a delay Δ, which is an upper bound for the time it takes for a valid transaction to appear on the ledger. Due to space constraints, we defer the full security analysis in Section A-C and only give an overview.

We present the ideal functionality $\mathcal{F}_{\mathsf{Alba}}$, which formally defines the input/output behavior, any side-effects on the ledger, and the properties we want to capture. More specifically, we capture the *atomicity with punish* property. Informally, this property ensures consistency between the current state of the payment channel and the target ledger, or in case of a cheating party posting either an old state or refusing to perform a valid update, all the funds go to the non-cheating party (punish). Provided this property it is straightforward to show Alba satisfies Definition 2.

We assume static corruption, i.e., the adversary $\mathcal{A}$ chooses at the beginning of the protocol execution whom to corrupt. There is an environment $\mathcal{Z}$ that captures anything external to the protocol execution. The UC proof aims to show that the formalized Alba protocol Π is as secure as the ideal functionality $\mathcal{F}_{\mathsf{Alba}}$, or *GUC-realizes* $\mathcal{F}_{\mathsf{Alba}}$, thus having the same security properties. This is done by defining a simulator $\mathcal{S}$, that can convert any attack on the real-world protocol Π to an attack on the ideal-world functionality $\mathcal{F}_{\mathsf{Alba}}$. In other words, it should be computationally indistinguishable for any PPT environment $\mathcal{Z}$ whether it is interacting with Π or with $\mathcal{F}_{\mathsf{Alba}}$ and $\mathcal{S}$. We give formal definitions of $\mathcal{F}_{\mathsf{Alba}}$ (Section A-C), Π (Section B), GUC security, and the proof of the following main security theorem (Section D).

Theorem 2. *The Alba protocol* Π *GUC-realizes the ideal functionality* $\mathcal{F}_{\mathsf{Alba}}$.

C. Game Theoretic Analysis

While our UC-based analysis shows that our protocol is secure against Byzantine parties, we now explore Alba's robustness against rational players who may deviate from the correct protocol execution if they are to gain from it. This analysis assumes parties are rational utility-maximizing agents, capturing better the behavior of parties who generally do not engage in irrational decisions that will cost them money (Byzantine adversaries) nor blindly adhere to the protocol when a more profitable strategy exists (honest parties assumption). In the following, we present a summary of our analysis, while the complete analysis, including formal definitions and omitted proofs, can be found in Section E.

To model the decision-making process of Alba, our focus narrows to Alba's smart contract interactions. Since the channel cannot be closed with an un-revoked state while the Alba instance remains active on the contract, we can infer the decisions of players with respect to the channel from the smart contract execution. Deviations from the channel protocol are limited to refusal to update or halting cooperation during updates. Both scenarios allow for enforcing the correct outcome through disputes or proof submissions to the smart contract.

We adopt extensive form games to map out Alba 's decision-making process, identifying players, possible actions, and payoffs. This model, characterized by perfect information, means every player is fully informed of other players' past actions and outcomes, enabling strategy optimization at every decision point. The game's extensive form is visualized as a tree, with nodes representing players' decisions and edges their actions, leading to payoffs at terminal nodes. Figure 4 depicts the game tree of Alba's smart contract.

Using the game tree, we can now identify the SPNE, which is the set of players' strategies from which active, utility-maximizing parties do not deviate at any point in the game, regardless of what happened before. We do so by applying *backward induction* to the tree of Figure 4: Starting from terminal nodes going upwards toward the root of the tree, we select the action with the highest payoff for the decision-making player, based on the future decisions that are already determined (as they are lower in the depth of the tree).

In Theorem 6, we prove Alba has one SPNE, where P and V cooperate to submit a valid proof to the smart contract (action (i) in the tree). Thus, Alba adheres to Definition 2 even under rational participants.

Theorem 3. *Consider the tree in Figure 4 reflecting the game between two parties* (P, V). *For* at least one of two parties*, say* P*, the following always holds:* $c_P < (f_P + u_P)(s_2)$. *The following strategy profile* Σ *is the SPNE of the game:*

$$\Sigma(P, V) = \{[(i)]_P, [(v), (viii)]_V\}.$$

VI. Evaluation

We evaluate Alba' s performance regarding communication and on-chain costs. We further discuss possible optimizations.

LN Transaction Size. We recall from Section III that in Alba, the two protocol parties need to embed, for security reasons, some protocol-relevant information within their commitment transactions. In particular, compared to standard LN transactions, the commitment transaction of P has to include an additional `OP_RETURN` output which stores the hash of V's

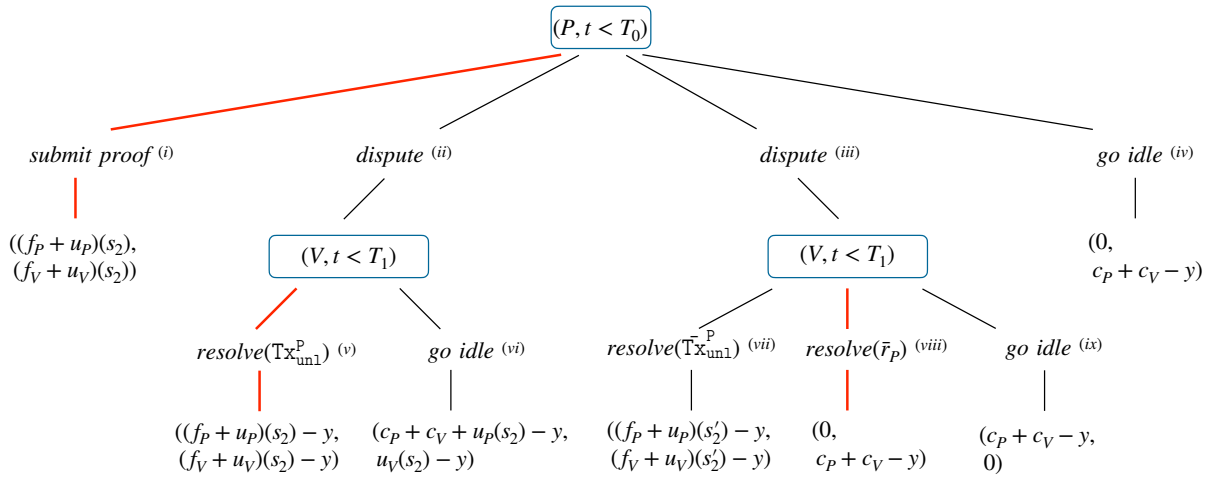

Fig. 4: P and V deposit c_P and $c_V \geq 0$ coins, updating channel from s_1 to $s_2 = s_1 + \Delta s$ before T_0. Without proof or dispute by T_0, $c_P + c_V$ go to V. Utility in state s_i and on the contract on $\mathcal{L}_D$ are denoted by $u(s_i) = (u_P(s_i), u_V(s_i))$ and $f(s_i) = (f_P(s_i), f_V(s_i))$, respectively. s_2' denotes state after applying Δs to revoked state s_r. Game tree: P actions comprise submitting a valid proof *(i)*, opening a valid dispute with the latest channel state *(ii)*, presenting an old, revoked state *(iii)*, or going idle *(iv)*. V responds to a valid dispute with an unlocked transaction *(v)* or goes idle *(vi)*, and to an invalid dispute with the old state *(vii)*, the revocation secret $\bar{r}_P$ *(viii)*, or by going idle *(ix)*. y denotes the opportunity and fee costs. The SPNE is illustrated in red.

revocation secret, i.e., $R_V := \mathcal{H}(r_V)$. Another `OP_RETURN` output storing a unique update identifier might be needed depending on the application. We have generated such an augmented commitment transaction using a Python library [7], broadcast it to the Bitcoin testnet [2], and compared its size to LN transactions. Adding the two `OP_RETURN` outputs results in a 406-byte transaction, i.e., 126 bytes larger than the 280-byte standard ones, whereas including the hash of the revocation secret alone results in a 395-byte transaction. We also recall that this larger transaction needs to be exchanged only once, as all other channel updates can be performed by simply exchanging locked versions of standard transactions.

The size of commitment transactions also depends on the application: for instance, when two parties are playing chess, they also need to store an efficient representation of the state of the game. If the application state exceeds the 80 bytes allowed by the `OP_RETURN`, additional outputs can be created.

Communication Overhead. When updating the LN channel, our protocol requires one more round of communication with respect to standard LN updates. This is because parties must exchange locked and unlocked versions of signed commitment transactions. This additional step takes place entirely off-chain, incurring no additional cost.

On-Chain Costs in Ethereum. To give insights on the overhead introduced by our smart contract, we spin up a local blockchain instance using Hardhat network 2.17.4, a development environment for Ethereum that allows to deploy, run, and test smart contracts and provides per-function detailed reports on gas costs. A proof-of-concept implementation and evaluation of the ALBA contract is publicly available at [1]. We have evaluated the gas consumption, which verifies that a simple coin transfer occurred in the channel as, for instance, in the case of a payback transaction within a lending protocol. For more sophisticated applications, the gas costs pertaining to the on-chain application logic have to be taken into account.

We specifically look at the gas consumption of the Setup, SubmitProof, Dispute, ResolveDispute, and Settle functions, presented in Table II.

We observe that the Setup function consumes 390k gas as a result of initializing the state variables and storing the protocol specifics, such as the hash of the funding transaction, index of the funding output, parties' public keys, and timelocks. The SubmitProof and Dispute functions consume 250k and 515k gas, respectively, as they need to parse both commitment transactions: this means that from raw transactions they (i) extract the timelock and check whether transactions are locked or unlocked, (ii) extract the input and verify that they spend from the funding transaction output, (iii) extract the outputs and check their well-formedness, their balances, and the data in the `OP_RETURN`, (iv) verify parties' signatures. The Dispute function incurs some additional storage cost, as it stores the commitment transaction state in global variables, which needs to be checked against the state of the transaction provided when resolving the dispute. The ResolveDispute functions require 168k gas in the optimistic case (the dispute was opened with the latest state), and 37k gas in the pessimistic case (the dispute was opened with an old state). Resolving a dispute is significantly cheaper compared to the gas consumption of verifying a proof because only a single commitment transaction needs to be checked (optimistic case), or a hash's

	Gas Cost
Setup	393401
SubmitProof	253566
OptimisticSubmitProof	48027
Dispute	515860
ResolveDispute($\mathsf{Tx}^{\mathsf{P}}_{\mathsf{unl}}, \sigma_V$)	168046
ResolveDispute($\bar{r}_P$)	37333
Settle	49814

TABLE II: On-chain gas costs evaluation for EVM-based blockchains. For more details, see our implementation and evaluation [1].

preimage must be verified (pessimistic case). Finally, funds in the contract can be unlocked by calling the Settle function, which has an overhead of 50k gas.

Optimizations. From Table II, on-chain costs incurred by Alba are on par with those of other cross-chain protocols: zkBridge uses 230k gas for zk-proof verification [63], Glimpse 290k gas [56], and SPV-based bridges 205k gas [30] or more [43], [61]. On top of these, SPV-based bridges, have high maintenance costs which greatly increase overall users' costs. Nonetheless, any optimizations to bring the costs down towards the costs of a standard transaction of token ownership transfer (a *transaction unit*), which is 21k gas in Ethereum, are desirable. A natural optimization involves utilizing the optimistic scenario where parties act honestly and collaboratively. In this case, parties can simply replace the proof π with their signatures over a message of the form, e.g., $(\mathsf{id}, \mathsf{ProofSubmitted}, \mathsf{true})$ or $(\mathsf{id}, \mathsf{GameOver}, (\mathsf{Winner}, P))$, where they both acknowledge that some event occurred within the channel. The smart contract only needs to verify two signatures (consuming 48k gas, see OptimisticSubmitProof in Table II), before unlocking the coins, significantly reducing by 5x the cost of Alba in the optimistic case. Additional cost optimizations may be feasible in a production-level implementation as our smart contract in [1] is a research prototype.

VII. Related Work

SPV-Based Bridges. *SPV light clients* have been presented in Nakamoto's original paper itself [53]. The idea is to store and verify block headers, as opposed to whole blocks, to identify which chain carries the most Proof-of-Work. SPV-based light clients are at the core of chain relays [6], [43], which run a light client protocol of a source chain within a smart contract on a target chain. However, the high maintenance costs of these constructions (100k gas per block header submitted + 62k gas per transaction inclusion verification [6]), associated with the lack of incentives for relaying blocks from one chain to the other, are arguably the reasons why relays are not widely used in practice. Some bridge constructions, such as [64], [30] used by Interlay, however, heavily subsidize relayers, and still use BTC-Relay.

Contrary to SPV-based bridges, Alba's smart contract does not need to verify any information related to the consensus mechanism of the source chain. Instead, it only has to parse two transactions and their signatures, with all necessary incentives for parties embedded directly within the smart contract.

ZK-Based Bridges. More recently, zero-knowledge (zk) interoperability solutions have emerged, leveraging the completeness and soundness properties of zk protocols to ensure security. While zk-based chain relays have been proposed [61] but not used in practice, zkBridge [63] is getting attention from the community. zkBridge is an EVM-compatible bridge with constant size storage, where a zk-SNARK proof guarantees that the blockchain has undergone a state update (either a single block or a batch of them). Each verified state update is stored within a stateful contract, and recent block headers of the source chain are relayed to and stored within the contract until a zero-knowledge proof is made available to finalize and verify the state update. Similarly to SPV-based bridges, also zkBridge incurs high maintenance costs (230k gas per zk proof verification, without considering the non-reported costs for storage of recent blocks) and suffers from lack of incentives (authors leave incentives for future work) for relayers. In addition, given the high computation complexity of zk proof generation, generating the zk proof is *very resource intensive* on the prover's side, and it can only be computed with very high-performance (and expensive) hardware machines.

Contrarily, Alba being agnostic to $\mathcal{L}_S$'s consensus eliminates the need to account for maintenance costs during incentive design and allows for the generation of an Alba proof that is not computationally expensive to the point that it can be done in resource-constrained environments. Moreover, the on-chain verification of zk proofs requires $\mathcal{L}_D$ to support complicated crypto operations, whereas Alba only requires $\mathcal{L}_D$ to allow for signature verification.

Cryptographic Constructions. A radically different approach is taken by interoperability solutions relying on cryptographic primitives such as Hashed TimeLock Contracts (HTLCs) and adaptor signatures. These solutions are solely used for simple applications like atomic swaps [5], [46], [20], [50], [58], as they favor a simple design over expressiveness. Surprisingly, despite their simplicity, their gas fees are still high (270k gas per HTLC [59]). Similarly to Alba, these solutions exhibit good properties: they are agnostic to the consensus of the blockchains they operate between, they have instant finality, and a lightweight on-chain footprint on both $\mathcal{L}_S$ and $\mathcal{L}_D$. Nonetheless, their application is confined to token swaps only, whereas Alba can be used for a wide spectrum of applications, from multi-asset swaps to DeFi protocols.

VIII. Conclusion

In this work, we introduced, for the first time, Pay2Chain bridges, i.e., bridges supporting the verification of off-chain transactions. We defined the advantages of scalable bridges over existing ones and instantiated them for payment channels, which are among the most well-established, decentralized, effective, and compatible off-chain solutions.

In particular, we presented Alba, a new, *Pay2Chain bridge* which enables users to *efficiently*, *securely*, and *trustlessly* condition the execution of a transaction in a target chain $\mathcal{L}_D$ on a specific transaction occurring in a payment channel. We illustrated how Alba serves as a building block for new, exciting applications in the realm of *DeFi* (e.g., trustless lending protocols for borrowing assets from another chain

in order to use its features, such as DeFi applications), but also towards *multi-asset payment channels*, and *optimistic off-chain computation* (e.g., bringing stateful, Turing-complete to limited chains like Bitcoin).

We formally analyzed Alba in the UC framework. Furthermore, we completed the analysis by incorporating a game-theoretic study, capturing the case where participants act rationally. Lastly, we assessed the performance of Alba in terms of communication complexity and on-chain costs. Notably, in optimistic scenarios, Alba only costs twice as much as the simplest Ethereum transactions.

Acknowledgments

The work was partially supported by CoBloX Labs, by the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement 771527-BROWSEC), by the Austrian Science Fund (FWF) through the SFB SpyCode project F8510-N and F8512-N, and the project CoRaF (grant agreement ESP 68-N), by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association through the Christian Doppler Laboratory Blockchain Technologies for the Internet of Things (CDL-BOT), and by the WWTF through the project 10.47379/ICT22045.

References

[1] "ALBA Proof of Concept and Evaluation," https://github.com/ALBA-blockchain/ALBA-Protocol.

[2] "Prover's commitment transaction," https://shorturl.at/abhlv.

[3] "Real-Time Lightning Network Statistics," https://1ml.com/statistics.

[4] "Ronin Attack Shows Cross-Chain Crypto Is a 'Bridge' Too Far." [Online]. Available: https://www.coindesk.com/layer2/2022/04/05/ronin-attack-shows-cross-chain-crypto-is-a-bridge-too-far/

[5] "What is atomic swap and how to implement it," https://www.axiomadev.com/blog/what-is-atomic-swap-and-how-to-implement-it/.

[6] "BTC Relay," https://github.com/ethereum/btcrelay,http://btcrelay.org/, 2016, Gas costs: https://github.com/interlay/btc-relay-solidity.

[7] "python-bitcoin-utils library," https://github.com/karask/python-bitcoin-utils, 2016.

[8] "Update from the raiden team on development progress, announcement of raidex," Feb. 2017, https://tinyurl.com/z2snp9e.

[9] "A "literature review" on Rollups and Validium," 2023, https://ethresear.ch/t/a-literature-review-on-rollups-and-validium/16370.

[10] "Bankruptcy of FTX," 2023, https://en.wikipedia.org/wiki/Bankruptcy_of_FTX.

[11] "Board representation (computer chess)," https://en.wikipedia.org/wiki/Board_representation_(computer_chess), 2023.

[12] "Flashbot Docs," 2023, https://shorturl.at/tuBSZ.

[13] "Online games for the Lightning Network," 2023, https://shorturl.at/hGHR0, https://shorturl.at/gwORZ, https://shorturl.at/qtzG4.

[14] "Optimistic Rollups," 2023, https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/.

[15] "Taproot assets," 2023, https://docs.lightning.engineering/the-lightning-network/taproot-assets/taproot-assets-protocol.

[16] "What is RGB?" https://www.rgbfaq.com/what-is-rgb, 2023.

[17] "Zero-Knowledge Rollups," 2023, https://ethereum.org/en/developers/docs/scaling/zk-rollups/.

[18] L. Aumayr, K. Abbaszadeh, and M. Maffei, "Thora: Atomic and privacy-preserving multi-channel updates," in *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[19] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostáková, M. Maffei, P. Moreno-Sanchez, and S. Riahi, "Bitcoin-compatible virtual channels," *IACR Cryptology ePrint Archive, Report 2020/554*, 2020.

[20] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostáková, M. Maffei, P. Moreno-Sanchez, and S. Riahi, "Generalized channels from limited blockchain scripts and adaptor signatures," in *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2021.

[21] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostáková, M. Maffei, P. Moreno-Sanchez, and S. Riahi, "Bitcoin-Compatible Virtual Channels," in *IEEE Symposium on Security and Privacy*, 2021.

[22] L. Aumayr, P. Moreno-Sanchez, A. Kate, and M. Maffei, "Blitz: Secure Multi-Hop Payments Without Two-Phase Commits," in *USENIX Security Symposium*, 2021.

[23] L. Aumayr, P. M. Sanchez, A. Kate, and M. Maffei, "Breaking and Fixing Virtual Channels: Domino Attack and Donner," in *NDSS*, 2023.

[24] L. Aumayr, S. A. Thyagarajan, G. Malavolta, P. Moreno-Sanchez, and M. Maffei, "Sleepy channels: Bitcoin-compatible bi-directional payment channels without watchtowers," Cryptology ePrint Archive, Report 2021/1445, 2021, https://ia.cr/2021/1445.

[25] Z. Avarikioti, A. Desjardins, L. Kokoris-Kogias, and R. Wattenhofer, "Divide & scale: Formalization and roadmap to robust sharding," in *Structural Information and Communication Complexity: 30th International Colloquium, SIROCCO 2023, Alcalá de Henares, Spain, June 6–9, 2023, Proceedings*. Springer-Verlag, 2023, p. 199–245. [Online]. Available: https://doi.org/10.1007/978-3-031-32733-9_10

[26] Z. Avarikioti, E. Kokoris-Kogias, R. Wattenhofer, and D. Zindros, "Brick: Asynchronous incentive-compatible payment channels," in *Financial Cryptography and Data Security FC*, 2021. [Online]. Available: https://doi.org/10.1007/978-3-662-64331-0_11

[27] Z. Avarikioti, O. S. T. Litos, and R. Wattenhofer, "Cerberus channels: Incentivizing watchtowers for bitcoin," in *International Conference on Financial Cryptography and Data Security*. Springer, 2020, pp. 346–366.

[28] Z. Avarikioti, L. Thyfronitis, and S. Orfeas, "Suborn channels: Incentives against timelock bribes," in *Financial Cryptography and Data Security*, I. Eyal and J. Garay, Eds. Springer International Publishing, 2022.

[29] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas, "Bitcoin as a transaction ledger: A composable treatment," in *Advances in Cryptology – CRYPTO 2017*, 2017.

[30] T. Bugnet and A. Zamyatin, "XCC: Theft-resilient and collateral-optimized cryptocurrency-backed assets," Cryptology ePrint Archive, Paper 2022/113, 2022.

[31] B. Bünz, L. Kiffer, L. Luu, and M. Zamani, "Flyclient: Super-light clients for cryptocurrencies," in *IEEE Symposium on Security and Privacy, SP*, 2020. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00049

[32] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Foundations of Computer Science FOCS*, 2001, pp. 136–145.

[33] R. Canetti, Y. Dodis, R. Pass, and S. Walfish, "Universally composable security with global setup," in *Theory of Cryptography*, 2007.

[34] CoinDesk, by Shaurya Malwa, "North korean hacking group tied to $100m harmony hack moves 41,000 ether over weekend," 2023, https://shorturl.at/hoAD5.

[35] Cointelegraph, by Brian Quarmby, "Wormhole hacker moves $155M in biggest shift of stolen funds in months," 2023, https://cointelegraph.com/news/wormhole-hacker-moves-155m-in-biggest-shift-of-stolen-funds-in-months.

[36] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges," *CoRR*, 2019. [Online]. Available: http://arxiv.org/abs/1904.05234

[37] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi, "Fastkitten: Practical smart contracts on bitcoin." USENIX Association, 2019.

[38] C. Decker and R. Wattenhofer, "A fast and scalable payment network with bitcoin duplex micropayment channels," in *Stabilization, Safety, and Security of Distributed Systems*. Springer, 2015, pp. 3–18.

[39] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková, "Multi-party Virtual State Channels," in *Advances in Cryptology - EUROCRYPT*, 2019, pp. 625–656.

[40] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment hubs over cryptocurrencies," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 106–123.

[41] S. Dziembowski, S. Faust, and K. Hostáková, "General State Channel Networks," in *Computer and Communications Security, CCS*, 2018.

[42] T. Frassetto, P. Jauernig, D. Koisser, D. Kretzler, B. Schlosser, S. Faust, and A.-R. Sadeghi, "POSE: Practical off-chain smart contract execution," in *Proceedings 2023 Network and Distributed System Security Symposium*, 2023. [Online]. Available: https://doi.org/10.14722%2Fndss.2023.23118
[43] P. Frauenthaler, M. Sigwart, C. Spanring, M. Sober, and S. Schulte, "ETH relay: A cost-efficient relay for ethereum-based blockchains," in *IEEE International Conference on Blockchain*. IEEE, 2020.
[44] J. A. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," in *Advances in Cryptology - EUROCRYPT*. Springer, 2015.
[45] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, "Sok: Layer-two blockchain protocols." Berlin, Heidelberg: Springer-Verlag, 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-51280-4_12
[46] M. Herlihy, "Atomic cross-chain swaps," *CoRR*, vol. abs/1801.09515, 2018. [Online]. Available: http://arxiv.org/abs/1801.09515
[47] J. Katz, U. Maurer, B. Tackmann, and V. Zikas, "Universally composable synchronous computation," in *Theory of Cryptography*, 2013.
[48] A. Kiayias, A. Miller, and D. Zindros, "Non-interactive Proofs of Proof-of-Work," in *Financial Cryptography and Data Security*. Springer International Publishing, 2020.
[49] R. Linus, "Bitvm: Compute anything on bitcoin," dec 2023.
[50] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei, "Anonymous multi-hop locks for blockchain scalability and interoperability," in *Network and Distributed System Security Symposium, NDSS*, 2019.
[51] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry, "Sprites and state channels: Payment networks that go faster than lightning," in *FC 2019: Financial Cryptography and Data Security*, 2019.
[52] E. Mouton, "LN Things Part 2: Updating State," https://ellemouton.com/posts/updating-state/, 2021.
[53] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009, http://bitcoin.org/bitcoin.pdf.
[54] J. Poon and T. Dryja, "The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments," Jan. 2016, draft version 0.5.9.2, available at https://lightning.network/lightning-network-paper.pdf.
[55] S. Reinhard, "Reexamination of the perfectness concept for equilibrium points in extensive games," *International Journal of Game Theory*, 1975. [Online]. Available: https://doi.org/10.1007/BF01766400
[56] G. Scaffino, L. Aumayr, Z. Avarikioti, and M. Maffei, "Glimpse: On-Demand PoW Light Client with Constant-Size Storage for DeFi," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/scaffino
[57] R. Selten, "Multistage game models and delay supergames," *Nobel lecture*, 1994. [Online]. Available: https://www.nobelprize.org/uploads/2018/06/selten-lecture.pdf
[58] E. Tairi, P. Moreno-Sanchez, and M. Maffei, "A2l: Anonymous atomic locks for scalability in payment channel hubs," in *2021 2021 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2021, pp. 1919–1936. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP40001.2021.00111
[59] S. Thyagarajan, G. Malavolta, and P. Moreno-Sanchez, "Universal atomic swaps: Secure exchange of coins across all blockchains," in *2022 IEEE Symposium on Security and Privacy (SP) (SP)*, 2022.
[60] S. A. K. Thyagarajan, G. Malavolta, F. Schmidt, and D. Schröder, "PayMo: Payment Channels For Monero," *IACR Cryptol. ePrint Arch.*, 2020. [Online]. Available: https://eprint.iacr.org/2020/1441
[61] M. Westerkamp and J. Eberhardt, "zkRelay: Facilitating Sidechains using zkSNARK-based Chain-Relays," in *IEEE European Symposium on Security and Privacy Workshops*, 2020.
[62] K. Wüst, L. Diana, K. Kostiainen, G. O. Karame, S. Matetic, and S. Capkun, "Bitcontracts: Supporting Smart Contracts in Legacy Blockchains," in *Network and Distributed System Security Symposium*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:231860152
[63] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. Song, "zkBridge: Trustless Cross-chain Bridges Made Practical," in *ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2022.
[64] A. Zamyatin, D. Harz, J. Lind, P. Panayiotou, A. Gervais, and W. Knottenbelt, "XCLAIM: Trustless, Interoperable, Cryptocurrency-Backed Assets," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

APPENDIX A
MODELING ALBA IN THE UC-FRAMEWORK

A. Extended Notation

For the following formal analysis, we introduce the following notation. A transaction $\mathsf{Tx} = (\mathsf{cntr}_{in}, \mathsf{inputs}, \mathsf{cntr}_{out}, \mathsf{outputs}, \mathsf{witnesses})$ is an atomic update of the blockchain state and is associated to a unique identifier $\mathsf{txid} \in \{0,1\}^{256}$ defined as the *hash* $\mathcal{H}([\mathsf{Tx}])$ *of the transaction*, where $[\mathsf{Tx}] := (\mathsf{cntr}_{in}, \mathsf{inputs}, \mathsf{cntr}_{out}, \mathsf{outputs})$ is the *body of the transaction*. Intuitively, a transaction maps a non-empty list of inputs to a non-empty list of newly created outputs, describing a redistribution of funds from the users identified in the inputs to those identified in the outputs.

$\mathsf{cntr}_{in}, \mathsf{cntr}_{out} \in \mathbb{N}_{>0}$ represent the number of elements in the inputs and $\mathsf{outputs}$ lists. Any input ζ in the list of inputs is an unspent output from an older transaction, defined by the tuple $\zeta := (\mathsf{txid}, \mathsf{outid})$, with $\mathsf{txid} \in \{0,1\}^{256}$ representing the hash of the old transaction containing the to-be-spent output, and $\mathsf{outid} \in \mathbb{R}_{\geq 0}$ the index of such an output within the output list of the old transaction. These two fields uniquely identify the to-be-spent output. For short, we use the notation $\mathsf{Tx.txid}\|1$, to refer to the first output of a transaction Tx. $\mathsf{witnesses} \in \{0,1\}^*$, also known as *scriptSig* or *unlocking script*, is a list of witnesses ω, i.e., the data that only the entity entitled to spend the output can provide, thereby authenticating and validating the transaction. Any output θ in the list of $\mathsf{outputs}$ is a pair $\theta := (\mathsf{coins}, \phi)$ and can be consumed by at most one transaction (i.e., no double-spend). The amount of coins in an output θ is denoted by $\mathsf{coins} \in \mathbb{R}_{\geq 0}$, whereas the spendability of θ is restricted by the conditions in ϕ, also known as the *scriptPubKey* or *locking script*. Such conditions are modeled in the native scripting language of the blockchain and can vary from single-user $\mathsf{OneSig}(\mathsf{pk}_U)$ and multi-user $\mathsf{MuSig}(\mathsf{pk}_{U1}, \mathsf{pk}_{U2})$ ownership, to time locks, hash locks, and more complex scripts.

B. Modeling in the UC-Framework

To analyze Alba's security, we make use of the global universal composability (GUC) framework [33], i.e., an extension to the original UC framework [32]. Our analysis closely follows [41], [39], [40], [20], [21], [22], [24], [23], [18].

1) Preliminaries: Our protocol Π is executed between a set of parties $\mathcal{P}$ (interactive Turing machines), who exchange messages in the presence of an adversary $\mathcal{A}$. We assume static corruption, which means that $\mathcal{A}$ announces at the beginning of the protocol execution which parties out of $\mathcal{P}$ she wants to corrupt. Corrupting a party P means taking control of P, its internal state and being able to send any message and execute code on P's behalf. There is a special entity called the environment $\mathcal{Z}$, which can send inputs to every party in $\mathcal{P}$ and the adversary and which observes every response output by those parties. The intuition behind $\mathcal{Z}$ is to capture anything external to the protocol execution. $\mathcal{Z}$ and in extension $\mathcal{A}$, are given as input a security parameter $\lambda \in \mathbb{N}$ and an auxiliary input $z \in \{0,1\}^*$.

Communication. To model synchronous communication, we assume there is a global clock, which divides the protocol execution into discrete rounds and allows for a more intuitive arguing about time. This is captured by the functionality $\mathcal{G}_{clock}$ (defined in [47]), which ticks off each round after every honest party reports they are completed with the current time. Note that every party is aware of the current round. Additionally, we make use of the functionality $\mathcal{F}_{GDC}$ (defined in [39]) to model communication channels that are authenticated and have guaranteed delivery. Any message m sent in round t from one party $A \in \mathcal{P}$ to another party $B \in \mathcal{P}$, is received by B exactly in round $t+1$. The adversary can see messages sent between parties and reorder messages sent in the same round, but cannot drop, delay, or modify them. Any other message that is not sent between two protocol parties of $\mathcal{P}$, but instead involves one other entity, for example, $\mathcal{Z}$ or $\mathcal{A}$, takes zero rounds to be delivered. We further assume that all computations can be executed in the same round.

To ease readability, we use $\mathcal{G}_{clock}$ and $\mathcal{F}_{GDC}$ implicitly in the following way. We denote $(msg) \overset{t}{\hookrightarrow} A$ as sending a message msg to party A in round t. Similarly, we denote $(msg) \overset{t+1}{\hookleftarrow} B$ as B receiving a message msg in round $t+1$.

Ledgers and Contracts. For the ledger, we take the functionality defined in [20]. This idealized ledger keeps an append-only list of transactions. The functionality allows the environment $\mathcal{Z}$ to generate and register public keys for users to the ledger. Further, users can post transactions, which if valid, are added to the ledger $\mathcal{L}$ after at most Δ rounds; the exact number is chosen by the adversary. The ledger is global, publicly accessible by parties and from it, the current state of the ledger can be derived. Aside from Δ, the ledger is parameterized by a digital signature scheme Σ (used to register parties). We note that there are models that more accurately capture ledgers, e.g., [29], [44]. These functionalities introduce a lot of additional complexity. To increase readability, we opt for this simplified ledger functionality.

Ledgers can support smart contracts. Following [39], [41], smart contracts (or simply contracts) are self-executing agreements specified in some programming language. A contract is deployed by one party on the ledger, which can receive, store, and send coins. It has a dynamic, internal storage which represents the contract's current state. A contract is idle by default, which means that a contract never acts on its own accord, but only when a party calls a function defined by its code. A function executes the code according to its current internal state. Finally, a contract lives on a unique address $\{0,1\}^*$ and can be called via this address. We capture these smart contract capabilities, as well as the rules by which a transaction is considered valid, as parameter $\mathcal{V}$ of the ledger.

In our case, we need two specific ledger functionalities. One of them we call $\mathcal{L}_D$, which represents the destination ledger. We consider this ledger to have Turing-complete smart contract capabilities (similar to Ethereum) and write its smart contracts in pseudocode. We write function calls to a smart contract as $(\text{CallFunction}, address = \text{address}, function = \text{functionName}, args = \text{arguments}, coins = \text{coins})$, where $address$ specifies the address of the contract, $function$ the name of the to-be-called function, $args$ the (optional) arguments passed to the function and $coins$ the optional amount of coins passed to the function. A function can return a value. There is a special function $(\text{InitiateContract}, code = \text{code}, function = \text{Constructor}, args = \text{arguments}, coins = \text{coins})$ which creates a new contract with the specified $code$, runs the $Constructor$ function with the specified $args$ and returns the address where the smart contract was created. A call to a contract function (including contract creation) can fail and return $\perp$. A contract can learn the value and sender of a message via $msg.value()$ and $msg.sender()$.

The other ledger instance we use is for the payment channels and is used by the payment channel functionalities we define in Section A-B3. It does not (necessarily) have the capability for Turing-complete smart contracts, but can be thought of as more similar to Bitcoin. Indeed, this instance is the same as the one used in [20]. We proceed now to give the API of the $\mathcal{G}_{Ledger}$ functionality.

Interface of $\mathcal{G}_{Ledger}(\Delta, \Sigma, \mathcal{V})$ [20]

This functionality keeps a record of the public keys of parties. Also, all transactions that are posted (and accpeted, see below) are stored in the publicly accessible set $\mathcal{L}$ containing tuples of all accepted transactions.

Parameters:

- Δ: upper bound on the number of rounds it takes a valid transaction to be published on $\mathcal{L}$
- Σ: a digital signature scheme
- $\mathcal{C}$: the smart contract capabilities and transaction validity rules of the ledger

API: Messages from $\mathcal{Z}$ via a dummy user $A \in \mathcal{P}$:

- $(\texttt{sid}, \texttt{REGISTER}, \mathsf{pk}_A) \overset{\delta}{\hookleftarrow} A$:
  This function adds an entry (pk_A, A) to PKI consisting of the public key pk_A and the user A, if it does not already exist.
- $(\texttt{sid}, \texttt{POST}, [\mathsf{Tx}]) \overset{\delta}{\hookleftarrow} A$:
  This function checks if $[\mathsf{Tx}]$ is a valid transaction and if yes, accepts it on $\mathcal{L}$ after at most Δ rounds.

2) UC-Security Definition: We proceed to present UC-security definition. Let Π denote some *hybrid* protocol which has access to a set of auxiliary ideal functionalities $\mathcal{F}_{aux}$. An environment interacting with $\mathcal{A}$ will on input λ and z sees the transcript or execution ensemble $\text{EXEC}^{\mathcal{F}_{aux}}_{\Pi,\mathcal{A},\mathcal{Z}}(\lambda, z)$ as the set of all outputs and side-effects produced by the interacting with Π observable by $\mathcal{Z}$. Further, let $\phi\mathcal{F}$ denote the idealized protocol of some ideal functionality $\mathcal{F}$, where the messages between $\mathcal{F}$ and $\mathcal{Z}$ are sent through dummy parties. Say $\phi\mathcal{F}$ also has access to some ideal functionalities $\mathcal{F}_{aux}$. We define the execution ensemble observed by $\mathcal{Z}$ when interacting with $\phi\mathcal{F}$, a simulator $\mathcal{S}$ and on input λ and z as $\text{EXEC}^{\mathcal{F}_{aux}}_{\phi\mathcal{F},\mathcal{A},\mathcal{Z}}(\lambda, z)$. If a protocol Π GUC-realizes a functionality $\mathcal{F}$, it means that any attack on the real world protocol Π can be carried out against the idealized protocol $\phi\mathcal{F}$, and vice versa. In other words, Π shares the security properties of $\phi\mathcal{F}$. The formal

security is as follows.

Definition 7. A protocol Π GUC-realizes an ideal functionality $\mathcal{F}$, w.r.t. $\mathcal{F}_{aux}$, if for every adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that

$$\left\{\text{EXEC}^{\mathcal{F}_{aux}}_{\Pi,\mathcal{A},\mathcal{Z}}(\lambda,z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}} \overset{c}{\approx} \left\{\text{EXEC}^{\mathcal{F}_{aux}}_{\phi\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda,z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}}$$

where $\approx^c$ denotes computational indistinguishability.

3) Payment Channels: We reuse the functionality introduced in [20]. This functionality defines the intended behavior of channels, including the phases for *create*, *update*, *close* and *punish*. One of the properties achieved by this functionality is *instant finality with punish*, which on a high level means that an honest party is guaranteed that she can either enforce the current state of the channel or else get all the money in the channel. The functionality is parameterized by the upper bound on the number of consecutive off-chain communication rounds between channel users T and the number of ways a channel can be published on-chain k. It is also defined over a ledger, which we already described earlier.

We define the following channel tuple $\gamma := (\mathsf{id}, \mathsf{users}, \mathsf{cash}, \mathsf{st}, \mathsf{Aid}, \mathsf{ADeadline}, \mathsf{id}_{\mathsf{ch}})$. $\gamma.\mathsf{id} \in \{0,1\}^*$ is the unique identifier of the channel, the two channel users are stored as $\gamma.\mathsf{users} \in \mathcal{P}^2$. The coins held in the channel are stored as $\gamma.\mathsf{cash} \in \mathbb{R}_{\geq 0}$, and the current state of the channel as $\gamma.\mathsf{st} := (\mathsf{out}_1, \ldots, \mathsf{out}_n)$, a list of outputs.

We note that the functionality we present below differs from the one in [20] in the following two ways: (i) In all update messages, we require a boolean flag to be sent which indicates whether or not this update is done while there is an active instance of Alba running between these users, and we note that we restrict the environment to send this flag correctly (this is not unrealistic, as users are generally aware of the applications they are running on top of the channel). For the normal channel execution (without any *Alba* application, this is set to $\perp$ and nothing changes. And (ii), we add additional fields $\gamma.\mathsf{Aid}$, $\gamma.\mathsf{ADeadline}$ and $\gamma.\mathsf{idle}$ to the channel tuple, along with a functionality wrapper. This functionality wrapper delays any force-closure of the channel to the point when a dispute in Alba is resolved. An active Alba instance is indicated by $\gamma.\mathsf{Aid}$, $\gamma.\mathsf{idle}$ indicates a dispute and $\gamma.\mathsf{ADeadline}$ the time when the dispute will be resolved. We note that this delaying changes the property *instant finality with punish*, such that now the finality is delayed until $\gamma.\mathsf{ADeadline}$, in the case of an active Alba application and dispute.

The ideal functionality $\mathcal{F}^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}(T,k)$

Upon $(\texttt{CREATE}, \gamma, tid_P) \xleftarrow{\tau_0} P$, distinguish:

Both agreed: If already received $(\texttt{CREATE}, \gamma, tid_Q) \xleftarrow{\tau} Q$, where $\tau_0 - \tau \leq T$: If Tx s.t. $\mathsf{Tx.In} = (tid_P, tid_Q)$ and $\mathsf{Tx.Out} = (\gamma.\mathsf{cash}, \varphi)$, for some φ, appears on $\mathcal{L}$ in round $\tau_1 \leq \tau + \Delta + T$, set $\Gamma(\gamma.\texttt{id}) := (\{\gamma\}, \mathsf{Tx})$ and $(\texttt{CREATED}, \gamma.\texttt{id}) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$. Else stop.

Wait for Q: Else wait if $(\texttt{CREATE}, \mathsf{id}_{\mathsf{ch}}) \xleftarrow{\tau \leq \tau_0 + T} Q$ (in that case "Both agreed" option is executed). If such message is not received, stop.

Upon $(\texttt{UPDATE}, \mathsf{id}_{\mathsf{ch}}, \vec{\theta}, t_{\mathsf{stp}}, \texttt{AUpdate}) \xleftarrow{\tau_0} P$, parse $(\{\gamma\}, \mathsf{Tx}) := \Gamma(\mathsf{id}_{\mathsf{ch}})$, set $\gamma' := \gamma$, $\gamma'.\mathsf{st} := \vec{\theta}$:

1) In round $\tau_1 \leq \tau_0 + T$, let $\mathcal{S}$ define $\vec{tid}$ s.t. $|\vec{tid}| = k$. Then $(\texttt{UPDATE–REQ}, \mathsf{id}_{\mathsf{ch}}, \vec{\theta}, t_{\mathsf{stp}}, \vec{tid}, \texttt{AUpdate}) \xhookrightarrow{\tau_1} Q$ and $(\texttt{SETUP}, \mathsf{id}_{\mathsf{ch}}, \vec{tid}) \xhookrightarrow{\tau_1} P$.
2) If $(\texttt{SETUP–OK}, \mathsf{id}_{\mathsf{ch}}, \texttt{AUpdate}) \xleftarrow{\tau_2 \leq \tau_1 + t_{\mathsf{stp}}} P$, then $(\texttt{SETUP–OK}, \mathsf{id}_{\mathsf{ch}}, \texttt{AUpdate}) \xrightarrow{\tau_3 \leq \tau_2 + T} Q$. Else stop.
3) If $(\texttt{UPDATE–OK}, \mathsf{id}_{\mathsf{ch}}, \texttt{AUpdate}) \xleftarrow{\tau_3} Q$, then $(\texttt{UPDATE–OK}, \mathsf{id}_{\mathsf{ch}}, \texttt{AUpdate}) \xrightarrow{\tau_4 \leq \tau_3 + T} P$. Else distinguish:
   - If Q honest or if instructed by $\mathcal{S}$, stop *(reject)*.
   - Else set $\Gamma(\mathsf{id}_{\mathsf{ch}}) := (\{\gamma, \gamma'\}, \mathsf{Tx})$, run $\texttt{ForceClose}(\mathsf{id}_{\mathsf{ch}})$ and stop.
4) If $(\texttt{REVOKE}, \mathsf{id}_{\mathsf{ch}}, \texttt{AUpdate}) \xleftarrow{\tau_4} P$, send $(\texttt{REVOKE–REQ}, \mathsf{id}_{\mathsf{ch}}, \texttt{AUpdate}) \xrightarrow{\tau_5 \leq \tau_4 + T} Q$.
   Else set $\Gamma(\mathsf{id}_{\mathsf{ch}}) := (\{\gamma, \gamma'\}, \mathsf{Tx})$, run $\texttt{ForceClose}(\mathsf{id}_{\mathsf{ch}})$ and stop.
5) If $(\texttt{REVOKE}, \mathsf{id}_{\mathsf{ch}}, \texttt{AUpdate}) \xleftarrow{\tau_5} Q$, $\Gamma(\mathsf{id}_{\mathsf{ch}}) := (\{\gamma'\}, \mathsf{Tx})$, send $(\texttt{UPDATED}, \mathsf{id}_{\mathsf{ch}}, \vec{\theta}) \xrightarrow{\tau_6 \leq \tau_5 + T} \gamma.\mathsf{users}$ and stop *(accept)*. Else set $\Gamma(\mathsf{id}_{\mathsf{ch}}) := (\{\gamma, \gamma'\}, \mathsf{Tx})$, run $\texttt{ForceClose}(\mathsf{id}_{\mathsf{ch}})$ and stop.

Upon $(\texttt{CLOSE}, \mathsf{id}_{\mathsf{ch}}) \xleftarrow{\tau_0} P$, distinguish: **Both agreed:** If already received $(\texttt{CLOSE}, \mathsf{id}_{\mathsf{ch}}) \xleftarrow{\tau} Q$, where $\tau_0 - \tau \leq T$, run $\texttt{ForceClose}(\mathsf{id}_{\mathsf{ch}})$ unless both parties are honest. In this case let $(\{\gamma\}, \mathsf{Tx}) := \Gamma(\mathsf{id}_{\mathsf{ch}})$ and distinguish:

- If Tx', with $\mathsf{Tx}'.\mathsf{In} = \mathsf{Tx}.\mathsf{txid}$ and $\mathsf{Tx}'.\mathsf{Out} = \gamma.\mathsf{st}$ appears on $\mathcal{L}$ in round $\tau_1 \leq \tau_0 + \Delta$, set $\Gamma(\mathsf{id}_{\mathsf{ch}}) := \perp$, send $(\texttt{CLOSED}, \mathsf{id}_{\mathsf{ch}}) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$ and stop.
- Else output $(\texttt{ERROR}) \xrightarrow{\tau_0 + \Delta} \gamma.\mathsf{users}$ and stop.

Wait for Q: Else wait if $(\texttt{CLOSE}, \mathsf{id}_{\mathsf{ch}}) \xleftarrow{\tau \leq \tau_0 + T} Q$ (in that case "Both agreed" option is executed). If such message is not received, run $\texttt{ForceClose}(\mathsf{id}_{\mathsf{ch}})$ in round $\tau_0 + T$.

At the end of every round τ_0: For each $\mathsf{id}_{\mathsf{ch}} \in \{0,1\}^*$ s.t. $(\{\gamma\}, \mathsf{Tx}) := \Gamma(\mathsf{id}_{\mathsf{ch}}) \neq \perp$, do the following. For the last message received for $\mathsf{id}_{\mathsf{ch}}$ that contained the boolean flag $\texttt{AUpdate}$, set $\gamma.\mathsf{Aid} := \texttt{AUpdate}$. Further, check if $\mathcal{L}$ contains Tx' with $\mathsf{Tx}'.\mathsf{In} = \mathsf{Tx}.\mathsf{txid}$. If yes, then define $S := \{\gamma.\mathsf{st} \mid \gamma \in X\}$, $\tau := \tau_0 + 2\Delta$ and distinguish: **Close:** If Tx'' s.t. $\mathsf{Tx}''.\mathsf{In} = \mathsf{Tx}'.\mathsf{txid}$ and $\mathsf{Tx}''.\mathsf{Out} \in S$ appears on $\mathcal{L}$ in round $\tau_1 \leq \tau$, set $\Gamma(\mathsf{id}_{\mathsf{ch}}) := \perp$ and $(\texttt{CLOSED}, \mathsf{id}_{\mathsf{ch}}) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$ if not sent yet.
Punish: If Tx'' s.t. $\mathsf{Tx}''.\mathsf{In} = \mathsf{Tx}'.\mathsf{txid}$ and $\mathsf{Tx}''.\mathsf{Out} = (\gamma.\mathsf{cash}, \texttt{One–Sig}_{\mathsf{pk}_P})$ appears on $\mathcal{L}$ in round $\tau_1 \leq \tau$, for P honest, set $\Gamma(\mathsf{id}_{\mathsf{ch}}) := \perp$, $(\texttt{PUNISHED}, \mathsf{id}_{\mathsf{ch}}) \xhookrightarrow{\tau_1} P$ and stop.
Error: Else $(\texttt{ERROR}) \xhookrightarrow{\tau} \gamma.\mathsf{users}$.
$\texttt{ForceClose}(\mathsf{id}_{\mathsf{ch}})$: Let τ_0 be the current round and $(X, \mathsf{Tx}) := \Gamma(\mathsf{id}_{\mathsf{ch}})$. If within Δ rounds Tx is still unspent on $\mathcal{L}$, then $(\texttt{ERROR}) \xrightarrow{\tau_0 + \Delta} \gamma.\mathsf{users}$ and stop. *Note that otherwise,message $m \in \{\texttt{CLOSED}, \texttt{PUNISHED}, \texttt{ERROR}\}$ is output latest in round $\tau_0 + 3 \cdot \Delta$.*

Functionality wrapper

Upon the ideal functionality executing subprocedure $\texttt{ForceClose}(\mathsf{id}_{\mathsf{ch}})$, first read γ from $\Gamma(\mathsf{id}_{\mathsf{ch}})$ and distinguish

between the two cases:
- If γ.Aid is empty then run $\texttt{ForceClose}(\mathsf{id_{ch}})$
- Else, if γ.Aid is not empty, set $\gamma.\mathsf{idle} \leftarrow True$ (and update γ in $\Gamma(\mathsf{id_{ch}})$).At time $\gamma.\mathsf{ADeadline}$ run $\texttt{ForceClose}(\mathsf{id_{ch}})$ and stop.

4) Lightning Channels: Since we use the functionality for generalized channels in our UC model and want to show that we are compatible with Lightning channels, we below give a formal protocol definition of Lightning channels, $\Pi_{LN}^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}$, along with a simulator proving that Lightning channels UC-realize that functionality. The protocol is instantiated with a ledger $\mathcal{L}(\Delta, \Sigma, \mathcal{V})$. To be as close to the definition in [20], we say there is a hard relation R, the statement witness pairs of which correspond to the public/secret key of Σ. Indeed, there is a function ToKey that takes as input a statement $Y \in L_R$ and outputs a public key pk. The function is s.t. $(\mathsf{ToKey}(Y), y)$, for $(Y, y) \leftarrow \mathsf{GenR}$ has the same distribution as the key generation function of Σ. We only model the *update* part of the Lightning channel protocol, since the opening and closing phase is, essentially, the same as [20].

Lightning Channel Protocol $\Pi_{LN}^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}$

Below, we abbreviate $V := \gamma.\mathsf{otherParty}(P)$ for $P \in \gamma.\mathsf{users}$.

Update

▷ If at any moment $\gamma.\mathsf{idle} = True$, then none of this protocol is run and parties wait for the Alba to end and then close the channel

Party P upon $(\texttt{UPDATE}, \mathsf{id_{ch}}, \vec{\theta}, t_{\mathsf{stp}}, \texttt{AUpdate}) \overset{t_0}{\hookleftarrow} \mathcal{Z}$

1) Generate $(R_P, r_P) \leftarrow \mathsf{GenR}$ and send $(\text{updateReq}, \mathsf{id_{ch}}, \vec{\theta}, t_{\mathsf{stp}}, R_P, \texttt{AUpdate}) \overset{t_0}{\hookrightarrow} V$.

Party V upon $(\text{updateReq}, \mathsf{id_{ch}}, \vec{\theta}, t_{\mathsf{stp}}, R_P, \texttt{AUpdate}) \overset{\tau_0}{\hookleftarrow} P$

2) Generate $(R_V, r_V) \leftarrow \mathsf{GenR}$.
3) Extract $\texttt{TX}_\mathtt{f}$ and γ from $\Gamma^V(\mathsf{id_{ch}})$; if $\texttt{AUpdate}$ is $False$ then:

$$[\texttt{TX}_1^\mathtt{P}] := \texttt{GenCom}^V((\mathsf{pk}_P, R_P), (\mathsf{pk}_V, _), _, \vec{\theta}, \gamma.\mathsf{Aid}, \gamma.\mathsf{ADeadline}, \texttt{AUpdate})$$

Else if $\texttt{AUpdate}$ is $True$ then:

$$[\texttt{TX}_1^\mathtt{P}] := \texttt{GenCom}^V((\mathsf{pk}_P, R_P), (\mathsf{pk}_V, R_V), \bar{R}_P, \vec{\theta}, \gamma.\mathsf{Aid}, \gamma.\mathsf{ADeadline}, \texttt{AUpdate})$$

4) Send $(\text{updateInfo}, \mathsf{id_{ch}}, R_V) \overset{\tau_0}{\hookrightarrow} P$, $(\texttt{UPDATE–REQ}, \mathsf{id_{ch}}, \vec{\theta}, t_{\mathsf{stp}}, \texttt{TX}_1^\mathtt{P}.\mathsf{txid}, \texttt{AUpdate}) \overset{\tau_0+1}{\hookrightarrow} \mathcal{Z}$.

Party P at time $t_0 + 2$

5) If received $(\text{updateInfo}, \mathsf{id_{ch}}, R_V) \overset{t_0+2}{\hookleftarrow} V$, Extract $\texttt{TX}_\mathtt{f}$ and γ from $\Gamma^P(\mathsf{id_{ch}})$ and

$$[\texttt{TX}_1^\mathtt{V}] := \texttt{GenCom}^P((\mathsf{pk}_P, _), ((\mathsf{pk}_V, R_V), _, \vec{\theta}, \gamma.\mathsf{Aid}, \gamma.\mathsf{ADeadline}, \gamma.\texttt{AUpdate})$$

Send $(\texttt{SETUP}, \mathsf{id_{ch}}, \texttt{TX}_1^\mathtt{V}.\mathsf{txid}, \texttt{AUpdate}) \overset{t_0+2}{\hookrightarrow} \mathcal{Z}$. If $(\texttt{SETUP–OK}, \mathsf{id_{ch}}, \texttt{AUpdate}) \overset{t_1 \le t_0+2+t_{\mathsf{stp}}}{\hookleftarrow} \mathcal{Z}$, compute $\sigma_P([\texttt{TX}_1^\mathtt{V}]) \leftarrow \mathsf{Sign}_{\mathsf{sk}_P}([\texttt{TX}_1^\mathtt{V}])$ and send $(\text{updateComP}, \mathsf{id_{ch}}, [\texttt{TX}_1^\mathtt{V}], \sigma_P([\texttt{TX}_1^\mathtt{V}])) \overset{t_1}{\hookrightarrow} V$. Else stop.

Party V

6) If $(\text{updateComP}, \mathsf{id_{ch}}, [\texttt{TX}_1^\mathtt{V}], \sigma_P([\texttt{TX}_1^\mathtt{V}])) \overset{\tau_1 \le \tau_0+2+t_{\mathsf{stp}}}{\hookleftarrow} P$, s.t. $\mathsf{Vrfy}_{\mathsf{pk}_P}([\texttt{TX}_1^\mathtt{V}]; \sigma_P([\texttt{TX}_1^\mathtt{V}])) = 1$ output $(\texttt{SETUP–OK}, \mathsf{id_{ch}}, \texttt{AUpdate}) \overset{\tau_1}{\hookrightarrow} \mathcal{Z}$. Else stop. ▷ Here, V doesn't go to idle mode because the update was just interrupted before exchanging signatures.
7) If $(\texttt{UPDATE–OK}, \mathsf{id_{ch}}, \texttt{AUpdate}) \overset{\tau_1}{\hookleftarrow} \mathcal{Z}$ Then sign $\sigma_V([\texttt{TX}_1^\mathtt{P}]) \leftarrow \mathsf{Sign}([\texttt{TX}_1^\mathtt{P}])$ and send $(\text{updateComV}, \mathsf{id_{ch}}, [\texttt{TX}_1^\mathtt{P}], \sigma_V([\texttt{TX}_1^\mathtt{P}])) \overset{\tau_1}{\hookrightarrow} P$. Else if you did not receive the $\texttt{UPDATE–OK}$ message then send $(\text{updateNotOk}, \mathsf{id_{ch}}, r_V) \overset{\tau_1}{\hookrightarrow} P$ and stop.

Party P

8) In round $t_1 + 2$ distinguish the following cases:
   - If $(\text{updateComV}, \mathsf{id_{ch}}, [\texttt{TX}_1^\mathtt{P}], \sigma_V([\texttt{TX}_1^\mathtt{P}])) \overset{t_1+2}{\hookleftarrow} V$, s.t. $\mathsf{Vrfy}_{\mathsf{pk}_V}([\texttt{TX}_1^\mathtt{P}]; \sigma_V([\texttt{TX}_1^\mathtt{P}])) = 1$, and if $\texttt{AlbaUpdate}$ is $False$ output $(\texttt{UPDATE–OK}, \mathsf{id_{ch}}, \texttt{AUpdate}) \overset{t_1+2}{\hookrightarrow} \mathcal{Z}$, otherwise if $\texttt{AlbaUpdate}$ is $True$, check the $\texttt{OP_RETURN}$ output of $[\texttt{TX}_1^\mathtt{P}]$ and verify that it is well-formed (is built using $\texttt{GenCom}^V$ function) and only then send $(\texttt{UPDATE–OK}, \mathsf{id_{ch}}, \texttt{AUpdate}) \overset{t_1+2}{\hookrightarrow} \mathcal{Z}$, otherwise if $[\texttt{TX}_1^\mathtt{P}]$ is not well-formed execute the procedure $\texttt{ForceClose}^P(\mathsf{id_{ch}})$ and stop.
   - If $(\text{updateNotOk}, \mathsf{id_{ch}}, r_V) \overset{t_1+2}{\hookleftarrow} V$, s.t. $(R_V, r_V) \in R$, add $\Theta^P(\mathsf{id_{ch}}) := \Theta^P(\mathsf{id_{ch}}) \cup ([\texttt{TX}_1^\mathtt{P}], r_P, r_V, \sigma_P([\texttt{TX}_1^\mathtt{V}]))$ and stop.
   - Else, execute the procedure $\texttt{ForceClose}^P(\mathsf{id_{ch}})$ and stop.
9) If $(\texttt{REVOKE}, \mathsf{id_{ch}}, \texttt{AUpdate}) \overset{t_1+2}{\hookleftarrow} \mathcal{Z}$, parse $\Gamma^P(\mathsf{id_{ch}})$ as $(\gamma, \texttt{TX}_\mathtt{f}, (\overline{\texttt{TX}}_1^\mathtt{V}, \overline{\texttt{TX}}_1^\mathtt{P} \bar{r}_P, \bar{R}_V, _))$ where _ denotes a wildcard and update the channel space as $\Gamma^P(\mathsf{id_{ch}}) := (\gamma, \texttt{TX}_\mathtt{f}, (\texttt{TX}_1^\mathtt{P}, \texttt{TX}_1^\mathtt{V}, r_P, R_V, \bar{r}_P))$ for $\texttt{TX}_1^\mathtt{P} := ([\texttt{TX}_1^\mathtt{P}], \{\mathsf{Sign}_{\mathsf{sk}_P}([\texttt{TX}_1^\mathtt{P}]), \sigma_V([\texttt{TX}_1^\mathtt{P}])\})$, and $\texttt{TX}_1^\mathtt{V} := ([\texttt{TX}_1^\mathtt{V}], \sigma_P([\texttt{TX}_1^\mathtt{V}]))$, send $(\text{revokeP}, \mathsf{id_{ch}}, \bar{r}_P) \overset{t_1+2}{\hookrightarrow} V$. Else, execute $\texttt{ForceClose}^P(\mathsf{id_{ch}})$ and stop.

Party V

10) Parse $\Gamma^V(\mathsf{id_{ch}})$ as $(\gamma, \texttt{TX}_\mathtt{f}, (\overline{\texttt{TX}}_1^\mathtt{P}, \overline{\texttt{TX}}_1^\mathtt{V}, \bar{r}_V, \bar{R}_P, _))$. If $(\text{revokeP}, \mathsf{id_{ch}}, \bar{r}_P) \overset{\tau_1+2}{\hookleftarrow} P$, s.t. $(\bar{R}_P, \bar{r}_P) \in R$, $(\texttt{REVOKE–REQ}, \mathsf{id_{ch}}, \texttt{AUpdate}) \overset{\tau_1+2}{\hookrightarrow} \mathcal{Z}$. Else execute $\texttt{ForceClose}^V(\mathsf{id_{ch}})$ and stop.
11) If $(\texttt{REVOKE}, \mathsf{id_{ch}}, \texttt{AUpdate}) \overset{\tau_1+2}{\hookleftarrow} \mathcal{Z}$ as a reply, set

$$\Theta^V(\mathsf{id_{ch}}) := \Theta^V(\mathsf{id_{ch}}) \cup ([\overline{\texttt{TX}}_1^\mathtt{V}], \bar{r}_P, \bar{r}_V, \sigma_V([\overline{\texttt{TX}}_\mathtt{c}^\mathtt{P}]))$$
$$\Gamma^V(\mathsf{id_{ch}}) := (\gamma, \texttt{TX}_\mathtt{f}, (\texttt{TX}_1^\mathtt{P}, \texttt{TX}_1^\mathtt{V}, r_V, R_P, \bar{r}_V)),$$

for $\texttt{TX}_1^\mathtt{V} := ([\texttt{TX}_1^\mathtt{V}], \{\mathsf{Sign}_{\mathsf{sk}_V}([\texttt{TX}_1^\mathtt{V}]), \sigma_P([\texttt{TX}_1^\mathtt{V}])\})$, and $\texttt{TX}_1^\mathtt{P} := ([\texttt{TX}_1^\mathtt{P}], \sigma_V([\texttt{TX}_1^\mathtt{P}]))$, then send $(\text{revokeV}, \mathsf{id_{ch}}, \bar{r}_V) \overset{\tau_1+2}{\hookrightarrow} P$. In the next round $(\texttt{UPDATED}, \mathsf{id_{ch}}) \overset{\tau_1+3}{\hookrightarrow} \mathcal{Z}$ and stop. Else, in round $\tau_1 + 2$, execute $\texttt{ForceClose}^V(\mathsf{id_{ch}})$ and stop.

Party P

12) If $(\text{revokeV}, \mathsf{id_{ch}}, \bar{r}_V) \overset{t_1+4}{\hookleftarrow} V$ s.t. $(\bar{R}_V, \bar{r}_V) \in R$, then set $\Theta^P(\mathsf{id_{ch}}) := \Theta^P(\mathsf{id_{ch}}) \cup ([\overline{\texttt{TX}}_1^\mathtt{P}], \bar{r}_P, \bar{r}_V, \sigma_P([\overline{\texttt{TX}}_\mathtt{c}^\mathtt{V}]))$ and

$(\texttt{UPDATED}, \mathsf{id_{ch}}) \xhookrightarrow{t_1+4} \mathcal{Z}$. Else execute $\texttt{ForceClose}^P(\mathsf{id_{ch}})$ and stop.

Subprocedures

$\texttt{GenFund}(\vec{tid}, \gamma)$:
Return $[\mathsf{Tx}]$, where $\mathsf{Tx.In} := \vec{tid}$ and $\mathsf{Tx.Out} := \{(\gamma.\mathsf{cash}, \texttt{Multi-Sig}_{\gamma.\mathsf{users}})\}$.

$\texttt{GenCom}^A([\texttt{TX}_\mathsf{f}], (\mathsf{pk}_B, R_B), (\mathsf{pk}_A, R_A), \bar{R}_P, \vec{\theta}, \mathsf{Aid}, \mathsf{ADeadline}, \texttt{AUpdate})$:
▷ Called by party $A \in \gamma.\mathsf{users}$, $B := \gamma.\mathsf{otherParty}(A)$. This function produces $\texttt{TX}_1^\mathsf{B}$

1) Let $(c, \texttt{Multi-Sig}_{\mathsf{pk}_A, \mathsf{pk}_B}) := \texttt{TX}_\mathsf{f}.\mathsf{Out}[1]$, and denote
   $\varphi_B := \texttt{Multi-Sig}_{\mathsf{ToKey}(R_B), \mathsf{pk}_A} \vee \{\theta_B.\varphi \wedge \texttt{CheckRelative}_\Delta\}$
   For $i \in [1, |\vec{\theta}|-2]$: ▷ Indices start from 0
   $\varphi_i := \texttt{Multi-Sig}_{\mathsf{ToKey}(R_B), \mathsf{pk}_A} \vee \{\theta_i.\varphi \wedge \texttt{CheckRelative}_\Delta\}$
2) Create $[\mathsf{Tx}]$, where $\mathsf{Tx.In} = \texttt{TX}_\mathsf{f}.\mathsf{txid}\|1$, $\mathsf{Tx.Out} := (\theta_A, \{\theta_B.\mathsf{cash}, \varphi_B\}, \{\theta_i.\mathsf{cash}, \varphi_i\}^{i\in[1,|\vec{\theta}|-2]})$ and $\mathsf{Tx.timelock} = _$.
3) If Aid is not empty and $\texttt{AUpdate} == False$, then set $\mathsf{Tx.timelock} = \mathsf{ADeadline}$.
4) Else If Aid is not empty and $\texttt{AUpdate} == True$ and the function is called by V, then add $\{\varepsilon, \{\texttt{OP_RETURN}\ \bar{R}_P\ R_V\ \mathsf{Aid}\}\}$ to $\mathsf{Tx.Out}$.
5) Return $[\mathsf{Tx}]$

$\texttt{ForceClose}^X(\mathsf{id_{ch}})$:
Let t_0 be the current round.
1) Extract $\texttt{TX}_1^\mathtt{X}$ from $\Gamma(\mathsf{id_{ch}})$ and send $(\mathrm{post}, \texttt{TX}_1^\mathtt{X}) \xhookrightarrow{t_0} \mathcal{L}$.
2) Let $t_1 \leq t_0 + \Delta$ be the round in which $\texttt{TX}_1^\mathtt{X}$ is accepted by the blockchain, set $\Theta^X(\mathsf{id}) := \perp$ and $\Gamma^X(\mathsf{id}) := \perp$ and output $(\texttt{CLOSED}, \mathsf{id}) \xhookrightarrow{t_1} \mathcal{Z}$.

Similarly to the ideal functionality, we present the code here which corresponds to the functionality wrapper and is responsible for delaying the finality in case of a dispute in Alba.

Protocol wrapper: $\mathcal{W}_{checks}$

Party $P \in \mathcal{P}$ proceeds as follows:
ForceClose: If $\texttt{ForceClose}(\mathsf{id_{ch}})$ function is called, first read γ from $\Gamma^P(\mathsf{id_{ch}})$ and distinguish between the two cases:
- If $\gamma.\mathsf{Aid}$ is empty then run $\texttt{ForceClose}(\mathsf{id_{ch}})$
- Else, if γ is not empty, set $\gamma.\mathsf{idle} \leftarrow True$ (and update $\gamma.\mathsf{idle}$ in $\Gamma^P(\mathsf{id_{ch}})$) and execute the following code at every timeslot until time $\gamma.\mathsf{ADeadline}$:
  - Send $(\mathrm{read}, address = \mathsf{Addr}, variable = \mathsf{State}, t_{validProof}) \xhookrightarrow{now} \mathcal{L}_D$, save the output in State.
  - If $\mathsf{State} == \texttt{Valid_Proof}$, run $\texttt{ForceClose}(\mathsf{id_{ch}})$ and stop.

  At time $\gamma.\mathsf{ADeadline}$ run $\texttt{ForceClose}(\mathsf{id_{ch}})$ and stop.

We proceed to give the code for the simulator $\mathcal{S}^{pc}$, which has access to functionalities $\mathcal{L}$ and $\mathcal{F}^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}(T,k)$ (or abbreviated as $\mathcal{PC}$) and simulates the Lightning channels protocol $\Pi_{LN}^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}$ in the ideal world. The main challenge usually in these simulation proofs is dealing with secret inputs, to which the simulator does not have access normally. However, we do not make any claims about privacy and have all inputs forwarded to $\mathcal{S}^{pc}$. The main challenge is recreating the protocol transcripts and effects on the ledger in the same rounds when parties are behaving maliciously. We do not need to provide the simulator code for the case where both channel users are honest, since this can be trivially simulated by running the according code sections of the protocol. Similarly, we do not care about the case where two users are malicious, since this is merely the adversary talking to herself. Since this is not the main focus of the paper, we content ourselves with only providing the simulator code and note that one can simply follow the executed code in the ideal and real world on either receiving or not receiving some input from the environment/counterparty and compare the resulting transcript seen by the environment. Doing so, one can see that this transcript, also sometimes referred to as execution ensemble, is identical. Thus, we state the following theorem.

Theorem 4. *For a given ledger $\mathcal{L}(\Delta,\Sigma,\mathcal{V})$, $k = 2$ and $T = 6 + t_{\mathsf{stp}}$, the protocol $\Pi_{LN}^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}$ UC-realizes $\mathcal{F}^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}(T,k)$.*

Simulator for updating generalized channels

Let $T_1 = 2$ and $T_2 = 1$ and let $|\vec{tid}| = 1$.

Case P is honest and V is corrupted

Upon P sending $(\texttt{UPDATE}, \mathsf{id_{ch}}, \vec{\theta}, t_{\mathsf{stp}}, \texttt{AUpdate}) \xhookrightarrow{\tau_0^P} \mathcal{F}$, proceed as follows:
1) Generate new revocation public/secret pair $(R_P, r_P) \leftarrow \mathsf{GenR}$ and send $(\mathrm{updateReq}, \mathsf{id_{ch}}, \vec{\theta}, t_{\mathsf{stp}}, R_P, \texttt{AUpdate}) \xhookrightarrow{\tau_0^P} V$.
2) Upon $(\mathrm{updateInfo}, \mathsf{id_{ch}}, R_V) \xhookleftarrow{\tau_0^P+2} V$, Extract $\texttt{TX}_\mathsf{f}$ and γ from $\Gamma^P(\mathsf{id_{ch}})$ and
   $[\texttt{TX}_1^\mathtt{V}] := \texttt{GenCom}^P((\mathsf{pk}_P, _), ((\mathsf{pk}_V, R_V), _, \vec{\theta}, \gamma.\mathsf{Aid}, \gamma.\mathsf{ADeadline}, \gamma.\texttt{AUpdate})$
   Send $(\texttt{SETUP}, \mathsf{id_{ch}}, \texttt{TX}_1^\mathtt{V}.\mathsf{txid}, \texttt{AUpdate}) \xhookrightarrow{\tau_0^P+2} \mathcal{Z}$.
3) If P sends $(\texttt{SETUP-OK}, \mathsf{id_{ch}}, \texttt{AUpdate}) \xhookrightarrow{\tau_1^P \leq \tau_0^P+2+t_{\mathsf{stp}}} \mathcal{F}$, compute $\sigma_P([\texttt{TX}_1^\mathtt{V}]) \leftarrow \mathsf{Sign}_{\mathsf{sk}_P}([\texttt{TX}_1^\mathtt{V}])$, and send $(\mathrm{updateComP}, \mathsf{id_{ch}}, [\texttt{TX}_1^\mathtt{V}], \sigma_P([\texttt{TX}_1^\mathtt{V}])) \xhookrightarrow{\tau_1^P} V$. Else stop.
4) In round $\tau_1^P + 2$ distinguish the following cases:
   - If you receive $(\mathrm{updateComV}, \mathsf{id_{ch}}, [\texttt{TX}_1^\mathtt{P}], \sigma_V([\texttt{TX}_1^\mathtt{P}])) \xhookleftarrow{\tau_1^P+2} V$, s.t. $\mathsf{Vrfy}_{\mathsf{pk}_V}([\texttt{TX}_1^\mathtt{P}]; \sigma_V([\texttt{TX}_1^\mathtt{P}])) = 1$, and if $\texttt{AlbaUpdate}$ is $False$, or, if $\texttt{AlbaUpdate}$ is $True$ and the $\texttt{OP_RETURN}$ output of $[\texttt{TX}_1^\mathtt{P}]$ is well-formed (is built using $\texttt{GenCom}_{Alba}^V$ function) and if V has not sent $(\texttt{UPDATE-OK}, \mathsf{id_{ch}}, \texttt{AUpdate}) \xhookrightarrow{\tau_1^P+1} \mathcal{F}$, then send $(\texttt{UPDATE-OK}, \mathsf{id_{ch}}, \texttt{AUpdate}) \xhookrightarrow{\tau_1^P+1} \mathcal{F}$ on behalf of V.
   - If you receive $(\mathrm{updateNotOk}, \mathsf{id_{ch}}, r_V) \xhookleftarrow{\tau_2^P+2} V$, where $(R_V, r_V) \in R$, add $\Theta^P(\mathsf{id_{ch}}) := \Theta^P(\mathsf{id_{ch}}) \cup ([\texttt{TX}_1^\mathtt{P}], r_P, r_V, \sigma_P([\texttt{TX}_1^\mathtt{V}]))$, and stop.

- Else, execute the simulator code for the procedure $\texttt{ForceClose}^P(\mathsf{id})$ and stop.

5) If P sends $(\texttt{REVOKE}, \mathsf{id_{ch}}, \texttt{AUpdate}) \xhookrightarrow{\tau_1^P+2} \mathcal{F}$, then parse $\Gamma^P(\mathsf{id_{ch}})$ as $(\gamma, \texttt{TX}_\mathsf{f}, (\overline{\texttt{TX}_1^\mathsf{V}}, \overline{\texttt{TX}_1^\mathsf{P}}\bar{r}_P, \bar{R}_V, _))$ where _ denotes a wildcard and update the channel space as $\Gamma^P(\mathsf{id_{ch}}) := (\gamma, \texttt{TX}_\mathsf{f}, (\texttt{TX}_1^\mathsf{P}, \texttt{TX}_1^\mathsf{V}, r_P, R_V, \bar{r}_P))$ for $\texttt{TX}_1^\mathsf{P} := ([\texttt{TX}_1^\mathsf{P}], \{\mathsf{Sign}_{\mathsf{sk}_P}([\texttt{TX}_1^\mathsf{P}]), \sigma_V([\texttt{TX}_1^\mathsf{P}])\})$, and $\texttt{TX}_1^\mathsf{V} := ([\texttt{TX}_1^\mathsf{V}], \sigma_P([\texttt{TX}_1^\mathsf{V}]))$. Then send $(\text{revokeP}, \mathsf{id_{ch}}, \bar{r}_P) \xhookrightarrow{\tau_1^P+2} V$. Else, execute the simulator code for the procedure $\texttt{ForceClose}^P(\mathsf{id})$ and stop.

6) If you receive $(\text{revokeV}, \mathsf{id_{ch}}, \bar{r}_V) \xhookleftarrow{\tau_1^P+4} V$ and if V has not sent $(\texttt{REVOKE}, \mathsf{id_{ch}}) \xhookrightarrow{\tau_1^V+2} \mathcal{F}$, then send $(\texttt{REVOKE}, \mathsf{id_{ch}}) \xhookrightarrow{\tau_1^V+2} \mathcal{F}$ on behalf of V. Check if $(\bar{R}_V, \bar{r}_V) \in R$, then set

$$\Theta^P(\mathsf{id_{ch}}) := \Theta^P(\mathsf{id_{ch}}) \cup ([\overline{\texttt{TX}_1^\mathsf{P}}], \bar{r}_P, \bar{r}_V, \sigma_P([\overline{\texttt{TX}_c^\mathsf{V}}]))$$

Else execute the simulator code for the procedure $\texttt{ForceClose}^P(\mathsf{id})$ and stop.

Case V is honest and P is corrupted

Upon P sending $(\text{updateReq}, \mathsf{id_{ch}}, \vec{\theta}, t_{\mathsf{stp}}, R_P, \texttt{AUpdate}) \xhookrightarrow{\tau_0} V$, send $(\texttt{UPDATE}, \mathsf{id_{ch}}, \vec{\theta}, t_{\mathsf{stp}}, \texttt{AUpdate}) \xhookrightarrow{\tau_0} \mathcal{F}$ on behalf of P, if P has not already sent this message. Proceed as follows:

1) Upon $(\text{updateReq}, \mathsf{id_{ch}}, \vec{\theta}, t_{\mathsf{stp}}, R_P, \texttt{AUpdate}) \xhookleftarrow{\tau_0^V} P$, generate $(R_V, r_V) \leftarrow \mathsf{GenR}$.
2) Extract $\texttt{TX}_\mathsf{f}$ and γ from $\Gamma^V(\mathsf{id_{ch}})$; if $\texttt{AUpdate}$ is $False$ then:

$$[\texttt{TX}_1^\mathsf{P}] := \texttt{GenCom}^V((\mathsf{pk}_P, R_P), (\mathsf{pk}_V, _), _, \vec{\theta}, \gamma.\mathsf{Aid}, \gamma.\mathsf{ADeadline}, \texttt{AUpdate})$$

Else if $\texttt{AUpdate}$ is $True$ then:

$$[\texttt{TX}_1^\mathsf{P}] := \texttt{GenCom}^V((\mathsf{pk}_P, R_P), (\mathsf{pk}_V, R_V), \bar{R}_P, \vec{\theta}, \gamma.\mathsf{Aid}, \gamma.\mathsf{ADeadline}, \texttt{AUpdate})$$

3) Send $(\text{updateInfo}, \mathsf{id_{ch}}, R_V) \xhookrightarrow{\tau_0^V} P$
4) If you $(\text{updateComP}, \mathsf{id_{ch}}, [\texttt{TX}_1^\mathsf{V}], \sigma_P([\texttt{TX}_1^\mathsf{V}])) \xhookleftarrow{\tau_1^V \le \tau_0^V+2+t_{\mathsf{stp}}} P$ then send $(\texttt{SETUP–OK}, \mathsf{id_{ch}}, \texttt{AUpdate}) \xhookrightarrow{\tau_1^V} \mathcal{F}$ on behalf of P, if P has not sent this message.
5) Check if $\mathsf{Vrfy}_{\mathsf{pk}_P}([\texttt{TX}_1^\mathsf{V}]; \sigma_P([\texttt{TX}_1^\mathsf{V}])) = 1$
6) If V sends $(\texttt{UPDATE–OK}, \mathsf{id}) \xhookrightarrow{\tau_1^V} \mathcal{F}$, sign $\sigma_V([\texttt{TX}_1^\mathsf{P}]) \leftarrow \mathsf{Sign}([\texttt{TX}_1^\mathsf{P}])$ and send $(\text{updateComV}, \mathsf{id_{ch}}, [\texttt{TX}_1^\mathsf{P}], \sigma_V([\texttt{TX}_1^\mathsf{P}])) \xhookrightarrow{\tau_1^V} P$. Else send $(\text{updateNotOk}, \mathsf{id_{ch}}, r_V) \xhookrightarrow{\tau_1^V} P$ and stop.
7) Parse $\Gamma^V(\mathsf{id})$ as $(\gamma, \texttt{TX}_\mathsf{f}, (\overline{\texttt{TX}_1^\mathsf{P}}, \overline{\texttt{TX}_1^\mathsf{V}}, \bar{r}_V, \bar{R}_P, _))$. If you $(\text{revokeP}, \mathsf{id_{ch}}, \bar{r}_P) \xhookleftarrow{\tau_1^V+2} P$, send $(\texttt{REVOKE}, \mathsf{id_{ch}}, \texttt{AUpdate}) \xhookrightarrow{\tau_1^V+2} \mathcal{F}$ on behalf of P, if P has not sent this message.

   Else if you do not receive $(\text{revokeP}, \mathsf{id_{ch}}, \bar{r}_P) \xhookleftarrow{\tau_1^V+2} P$ or if $(\bar{R}_P, \bar{r}_P) \notin R$, execute the simulator code of the procedure $\texttt{ForceClose}^V(\mathsf{id})$ and stop.
8) If V sends $(\texttt{REVOKE}, \mathsf{id_{ch}}, \texttt{AUpdate}) \xhookrightarrow{\tau_1^V+2} \mathcal{F}$, then set

$$\Theta^V(\mathsf{id_{ch}}) := \Theta^V(\mathsf{id_{ch}}) \cup ([\overline{\texttt{TX}_1^\mathsf{V}}], \bar{r}_P, \bar{r}_V, \sigma_V([\overline{\texttt{TX}_c^\mathsf{P}}]))$$
$$\Gamma^V(\mathsf{id_{ch}}) := (\gamma, \texttt{TX}_\mathsf{f}, (\texttt{TX}_1^\mathsf{P}, \texttt{TX}_1^\mathsf{V}, r_V, R_P, \bar{r}_V)),$$
$$\texttt{TX}_1^\mathsf{V} := ([\texttt{TX}_1^\mathsf{V}], \{\mathsf{Sign}_{\mathsf{sk}_V}([\texttt{TX}_1^\mathsf{V}]), \sigma_P([\texttt{TX}_1^\mathsf{V}])\}), \text{ and } \texttt{TX}_1^\mathsf{P} := ([\texttt{TX}_1^\mathsf{P}], \sigma_V([\texttt{TX}_1^\mathsf{P}])).$$

Then $(\text{revokeV}, \mathsf{id_{ch}}, \bar{r}_V) \xhookrightarrow{\tau_1^V+2} P$ and stop. Else, in round $\tau_1^V + 2$, execute the simulator code of the procedure $\texttt{ForceClose}^B(\mathsf{id})$ and stop.

Simulator for $\texttt{ForceClose}^X(\mathsf{id})$

Let τ_0 be the current round
1) Extract $\texttt{TX}_1^\mathsf{X}$ from $\Gamma(\mathsf{id_{ch}})$.
2) Send $(\text{post}, \texttt{TX}_1^\mathsf{X}) \xhookrightarrow{\tau_0} \mathcal{L}$.
3) Let $\tau_1 \le \tau_0 + \Delta$ be the round in which $\texttt{TX}_1^\mathsf{X}$ is accepted by the blockchain; set $\Theta^X(\mathsf{id}) = \perp$ and $\Gamma^X(\mathsf{id}) = \perp$.

Simulator for wrapper

ForceClose: Upon invoking the simulator code $\texttt{ForceClose}^X(\mathsf{id_{ch}})$ for an honest party X, first read γ from $\Gamma^X(\mathsf{id_{ch}})$ and distinguish between the two cases:
- If $\gamma.\mathsf{Aid}$ is empty then run $\texttt{ForceClose}(\mathsf{id_{ch}})$
- Else, if $\gamma.\mathsf{Aid}$ is not empty, set $\gamma.\mathsf{idle} \leftarrow True$ (and update γ in $\Gamma^X(\mathsf{id_{ch}})$) and execute the following code at every timeslot until time $\gamma.\mathsf{ADeadline}$:
  - Send $(\text{read}, address = \mathsf{Addr}, variable = \mathsf{State}, t_{validProof}) \xhookrightarrow{now} \mathcal{L}_D$, save the output in State.
  - If $\mathsf{State} == \texttt{Valid_Proof}$, run $\texttt{ForceClose}(\mathsf{id_{ch}})$ and stop.

  At time $\gamma.\mathsf{ADeadline}$ run $\texttt{ForceClose}(\mathsf{id_{ch}})$ and stop.

C. Alba Ideal Functionality

We start by (re-) introducing the following variables. Let α be the total money locked in the contract, $r \subseteq \mathcal{S} \times \mathcal{S}$ is the (application-dependent) reflexive state transition relation, where $(s_1, s_2) \in r$ indicates a valid transition from s_1 of the set of all valid states $\mathcal{S}$ of the payment channel in $\mathcal{PC}$ between the two users $\{P, V\}$ to another valid state s_2, and $f : \mathcal{S} \to \mathcal{O}$ is the outcome mapping, which assigns a balance for a valid (final) state to each user (i.e., a tuple giving each user a non-negative balance, in total α coins). As a special case, we note that here (and in the protocol description later), an outcome mapping can map a state to an intermediate state on-chain on $\mathcal{L}_D$, which needs to be resolved via on-chain transactions on $\mathcal{L}_D$. An example of this is a chess game, where a party enforces a state that still needs to be played out.

The security property we aim to achieve with our Alba protocol is **atomicity**. On a high level, atomicity says that given the latest valid state s on $\mathcal{PC}$, an honest user $U \in \{P, V\}$ is ensured that (i) the according outcome $f(s)$ can be enforced by P on $\mathcal{L}_D$ and (ii) no other outcome can be enforced on $\mathcal{L}_D$ as long as s is the latest valid state, or else U gets all the money α on $\mathcal{L}_D$. For a formal definition of atomicity, we define an ideal functionality, $\mathcal{F}_{\mathsf{Alba}}$, that specifies the expected input/output behavior as well as the side effects on the ledger(s) and payment channel functionalities. The functionality proceeds in the following phases. During the *setup*, $\mathcal{F}_{\mathsf{Alba}}$ receives a request from V to start an instance of Alba for some unique identifier id. This request is forwarded to P, and, if P (or rather the environment interacting via this

dummy party) agrees and an according contract appears on $\mathcal{L}_D$ on some address Addr (which is given by the simulator), the setup is considered complete and the functionality moves to the next phase.

This next phase, the *channel update and proof* phase, is the core of the ideal functionality and formally defines its **atomicity (with punish)** property. For each instance (identified by id), the functionality defines a variable tx which is initially $\perp$. This variable tracks which transaction the functionality expects to appear on $\mathcal{L}_D$. If set, i.e., $tx \neq \perp$, the functionality expects any transaction that spends the money stored in Addr to be tx. If not set, i.e., $tx = \perp$, the functionality expects either $\mathbf{TX}_{payout}$, which is the correct payout according to the latest valid state of the channel, if the current time is before T_2, or $\mathbf{TX}_V$, which is giving all money to V, if the current time is after T_2. This ensures part (ii) of the atomicity property, i.e., that no other outcome can be enforced.

For part (i), the functionality will try to determine which tx should appear, which can be upon receiving a message from an honest P either $\mathbf{TX}_{payout}$, $\mathbf{TX}_P$, or $\perp$. This depends on whether the request was for a state s_2, such that given the current state s_1, $(s_1, s_2) \in r$ and with enough time to perform the necessary steps (outcome $\mathbf{TX}_{payout}$), an invalid request or not enough time (outcome $\perp$), or the V not cooperating (outcome $\mathbf{TX}_P$). This is determined by the subprocedures `DetermineReceiver` and `HandleClaim`. For a dishonest P, the functionality only cares about whether P made an invalid claim, in which case the outcome is $\mathbf{TX}_V$. Finally, an honest V can enforce the current tx as well at time T_2, which is $\mathbf{TX}_V$ if $tx = \perp$ by then. Note that this formalization also captures the fact that a dishonest P can submit a valid proof, in this case $tx = \perp$, but $\mathbf{TX}_{payout}$ appears.

We note that we can modify the functionality (and the protocol later), such that both parties can take the role of P and V. For this, the times T_0 and T_2 are set to ∞ initially, i.e., both parties can close $Alba$ at any time. Once they do the initial call, T_0 is set to now, and to some time $T_2 > T_0 + 2\Delta_D$. For simplicity, we omit the formalization of this. We also do not make any claims about the privacy of our protocol and assume that messages sent or received by $\mathcal{F}_{\mathsf{Alba}}$ are implicitly forwarded to $\mathcal{S}$. In Figure 5 we showcase on a high level of the behaviour of the Alba ideal functionality presented in the following. We use the names dispute and claim interchangeably.

Ideal functionality of the Alba protocol $\mathcal{F}_{\mathsf{Alba}}$

Parameters:
$\mathcal{L}_{\mathcal{D}} \dots$ an instance of $\mathcal{G}_{Ledger}$ representing the destination blockchain.
$\mathcal{PC} \dots$ an instance of the payment channel ideal functionality which itself is parameterized by a ledger $\mathcal{L}_S$. We have access to the internal channel space storage of $\mathcal{PC}$ and we denote the channel space of channel $\mathsf{id}_{\mathsf{ch}}$ by $\Gamma(\mathsf{id}_{\mathsf{ch}})$,
$\Delta_D \dots$ the blockchain delay of $\mathcal{L}_{\mathcal{D}}$.

Variables:
$\phi(.) \dots$ a mapping of id to $(\mathsf{id}_{\mathsf{Ch}}, T_0, T_2, \alpha, \alpha', \mathsf{Addr})$, where $\mathsf{id} \in \{0,1\}^*$ is an identifier unique to the pair P and V. $\mathsf{id}_{\mathsf{ch}}$ is the id of the payment channel. Further $T_0, T_2 \in \mathbb{N}$, $\alpha' = (r, f)$, where r is the state transition relation and f the outcome mapping, as defined at the beginning of this section. Addr is the address of the instance of the Alba contract.
$t_{updateDelay} \dots$ The time it takes to perform a channel update, given by $\mathcal{PC}$

Setup
1) Upon $(\texttt{A–SETUP}, \mathsf{id}, \mathsf{id}_{\mathsf{ch}}, P, T_0, T_2, \alpha, \alpha') \overset{\tau}{\hookleftarrow} V$, read $\Gamma(\mathsf{id}_{\mathsf{ch}})$ from the storage of $\mathcal{PC}$ and parse the output as $(\{\gamma\}, \texttt{TX}_{\mathtt{f}}, \dots)$, then check the followings:
   - $\gamma.\mathsf{otherParty}(V) = P$
   - $\gamma.\mathsf{cash} \geq \alpha$
   - $T_0 + 2\Delta_D < T_2$ holds and T_0, T_2 are times in the future.
2) Send $(\texttt{A–CREATED}, \mathsf{id}, \mathsf{id}_{\mathsf{ch}}, T_0, T_2, \alpha, \alpha') \overset{\tau+1}{\hookrightarrow} P$.
3) At round $\tau_1 \leq \tau + 1 + \Delta_D$ if received $(\texttt{CONTRACT–INCLUDED}, \mathsf{Addr}) \overset{\tau_1}{\hookleftarrow} \mathcal{S}$, store $(\mathsf{id}_{\mathsf{Ch}}, T_0, T_2, \alpha, \alpha', \mathsf{Addr})$ in $\phi(\mathsf{id})$ and continue. Otherwise, stop.
4) Send $(\texttt{A–CREATED}, \mathsf{id}, \mathsf{id}_{\mathsf{ch}}, T_0, T_2, \alpha, \alpha') \overset{\tau_1}{\hookrightarrow} V$.
5) Set $\gamma.\mathsf{Aid} \leftarrow \mathsf{id}$.
6) Run the code below "Channel Update and Proof" for id

Channel Update and Proof (executed in every round, for id*)*

Variables

- $\mathbf{TX}_{payout}$: Let st be the current state of the channel γ between P and V in $\mathcal{PC}$. Define transaction TX_{payout} as a transaction that distributes the α coins from the smart contract with address Addr according to $\alpha'.f(\gamma.\mathsf{st})$ on $\mathcal{L}_{\mathcal{D}}$.
- $\mathbf{TX}_U$: Define transaction $\mathbf{TX}_U$ that transfers α from the smart contract with address Addr to $U \in \{P, V\}$ on the chain $\mathcal{L}_{\mathcal{D}}$.
- tx specifies the transaction that should appear on $\mathcal{L}_D$, initially $tx := \perp$.

Atomicity

Let t_0 be the current round. Monitor $\mathcal{L}_D$, if a transaction that spends the money of the contract on Addr appears, either this transaction has to be tx if $tx \neq \perp$, or $\mathbf{TX}_{payout}$ if $tx = \perp \wedge t_0 < T_2$, or $\mathbf{TX}_V$ if $tx = \perp \wedge t_0 \geq T_2$. Else output ERROR.

Execute the following steps:
1) If P is honest, upon receiving $(\texttt{A–INITIATE–PAYMENT}, \mathsf{id}, \vec{\theta}) \overset{t_0}{\hookleftarrow} P$, run `DetermineReceiver`(id) and wait for it to return $\mathbf{TX} \in \{\mathbf{TX}_{payout}, \mathbf{TX}_P, \perp\}$ in round t_1. If $\mathbf{TX} \neq \perp$, set $tx := \mathbf{TX}$. Go to step 5.
2) Else if P is not honest, upon receiving (InvalidClaim) from $\mathcal{S}$, define $t_1 := t_0$ and set $tx := \mathbf{TX}_V$. Go to step 5.
3) Else if V is honest and $t_0 = T_2$, define $t_1 := t_0$ and go to step 5.
4) Else (if none of the above happened), go idle for this round.
5) Distinguish the following cases.
   a) If $tx \neq \perp$, the transaction tx should appear on $\mathcal{L}_D$ at time $t \leq t_1 + \Delta_D$. Else, output ERROR.
   b) If $tx = \perp \wedge t_1 \geq T_2$, the transaction $\mathbf{TX}_V$ should appear on $\mathcal{L}_D$ at time $t \leq t_1 + \Delta_D$. Else, output ERROR.
   c) If $tx = \perp \wedge t_1 < T_2$, go idle.

Subprocedures

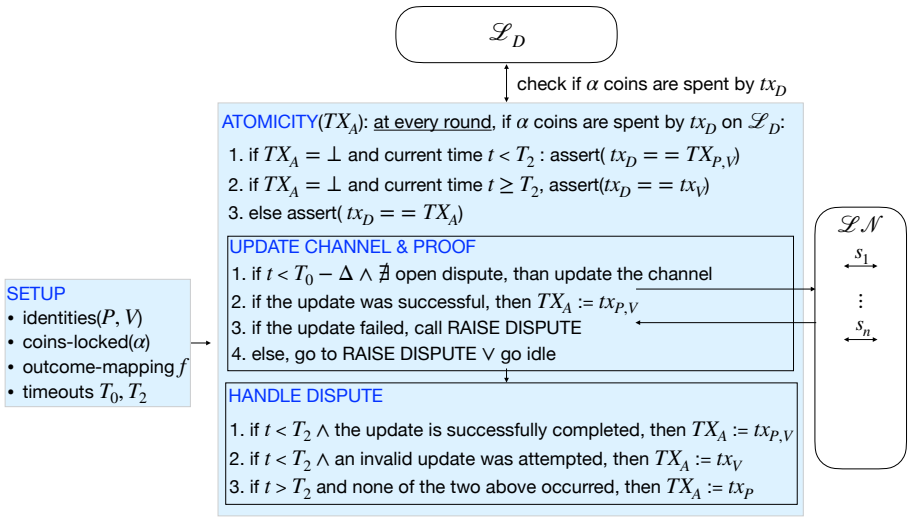

Fig. 5: High-level illustration of the behavior of the Alba ideal functionality $\mathcal{F}_{\mathsf{Alba}}$ defined in Section A-C. We use the names dispute and claim interchangeably.

`DetermineReceiver`(id): ▷ returns **TX**

1) Read ϕ(id) from the storage and parse it as $(\mathsf{id}_{\mathsf{ch}}, T_0, T_2, \alpha, \alpha', \mathsf{Addr})$ and then read $\Gamma(\mathsf{id}_{\mathsf{ch}}) = (\gamma, \mathtt{TX_f}, _)$ from the storage of $\mathcal{PC}$. Let $\gamma.\mathsf{st}$ be the current state of the channel. If $\vec{\theta} \neq \alpha'.r(\gamma.\mathsf{st})$, **Return**($\perp$).
2) If $(\gamma.\mathsf{idle} == True)$, distinguish the time. If $(now > T_0 - \Delta_D)$, **Return**($\perp$), otherwise if $(now \leq T_0 - \Delta_D)$, **Return**(`HandleClaim`(id))
3) If $(\gamma.\mathsf{idle} == False)$ and $(now > T_0 - t_{updateDelay} - \Delta_D)$ **Return**($\perp$), otherwise if $(now \leq T_0 - t_{updateDelay} - \Delta_D)$ continue.
4) Send $(\mathtt{UPDATE}, \mathsf{id}_{\mathsf{ch}}, \vec{\theta}, True) \xleftrightarrow{now} \mathcal{PC}$ to initiate the channel update and proceed with the update until it is either successful (outputs `UPDATED`) or stops unsuccessfully.
5) If the update was successful, **Return**($\mathbf{TX}_{payout}$).
6) Else, **Return**(`HandleClaim`(id)).

`HandleClaim`(id) : ▷ Returns **TX** in the case of a claim.
Let t be the round in which this function is called. Send message (Claim) to $\mathcal{S}$ and wait for $\mathcal{S}$ to reply with one of the following messages:

- If $\mathcal{S}$ sends a message (ClaimOk) in round t_1 where $t \leq t_1 \leq t + 3\Delta_D$, **Return**($\mathbf{TX}_{payout}$).
- If $\mathcal{S}$ sends a message (NoResponse) in round T_2, **Return**($\mathbf{TX}_P$).
- If neither of these messages is received by time T_2, output (`ERROR`).

Appendix B
Alba Protocol (Smart Contract)

While the ideal functionality $\mathcal{F}_{\mathsf{Alba}}$ directly defines which outcomes are enforceable on $\mathcal{L}_D$, we need to formally define the smart contract for the real world *Alba* protocol. This is the contract deployed on $\mathcal{L}_D$ and with which the parties interact with. As mentioned in Section A-B, we model contracts as programs written in pseudocode, with callable functions, which are deployed on $\mathcal{L}_D$ and can send, receive and hold coins. Again, we denote $\alpha' = (r, f)$, where r is the state transition relation and f the outcome mapping. Note that the smart contract uses the signature scheme of $\mathcal{L}_S$ when calling Vrfy.

The smart contract consists of a constructor, which initializes the contract, and 5 functions, which we go over now on a high level.

1) **SubmitProof** allows to resolve the contract in the honest case. For this, the caller will provide the unlocked and signed commitment transactions of both parties, i.e., $[\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}], \sigma_V([\mathtt{TX}^{\mathtt{P}}_{\mathtt{unl}}]), [\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}], \sigma_P([\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}])$. The contract ensures, that both transactions represent the same state and that the conditional outputs of $[\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}]$ are locked with the revocation public key specified in the `OP_RETURN` output of $[\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}]$. We require that always the party that acts as P initiates the update first, and this is to ensure that P locks the conditional outputs with something specified by V and, e.g., not something that P chooses and can immediately take V's money. If all checks pass, the state of the contract is set to `Valid_Proof`.
2) **Claim** initiates allows the caller P to force a valid update and the exchange of the unlocked commitment transaction. For this, the caller provides the last valid state $[\overline{\mathtt{Tx}^{\mathtt{P}}_{1}}]$ (P's version signed by V) along with the proposed, unlocked new state $[\overline{\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}}]$ (V's version signed by P). If some checks verifying the well-formedness of these transactions pass, the state of the contract is set to `Dispute`.
3) **RespondClaim** allows the other party, V, to respond to a previously initiated claim. There are two possibilities. If it was an invalid claim, i.e., the transaction $[\overline{\mathtt{Tx}^{\mathtt{P}}_{1}}]$ that was provided in the claim function is not the latest state. In this case, V knows the revocation secret $\bar{r}$, which will set the state of the contract to `Invalid_Claim`. If it was a valid claim, V can provide P's version of the commitment transaction $[\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}]$ along with V's signature.

If this transaction is well-formed and uses the revocation public key specified by P in the claim function, the state of the contract changes to `Dispute_Resolved`. In case V does not act, P gets all the money.

4) **FinalizeProof** allows P to finalize the proof after executing RespondClaim. P needs to provide V's version of the unlocked commitment transaction $[\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}]$ along with P's signature. The conditional outputs need to be locked with the revocation public key that V specified in $[\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}]$. If all checks pass, the state of the contract is set to `Valid_Proof`. If P does not reply timely, V can get all the money.
5) **Settle** allows a party to conduct a payout, depending on what is the current state of the contract.

Below we give the formal definition of the Alba smart contract, which we also denote as Script_G, and which is to be deployed on $\mathcal{L}_D$ (see Section C for the formal protocol description Π). We use the names dispute and claim interchangeably.

Alba Smart Contract Pseudocode

Variables:
- State
- ζ_{init}
- $sp\ldots$ initially $\perp$, holds $([\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}], \sigma_V([\mathtt{TX}^{\mathtt{P}}_{\mathtt{unl}}]), [\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}], \sigma_P([\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}]))$ after SubmitProof() was successful
- $cl\ldots$ initially $\perp$, holds $([\overline{\mathtt{Tx}^{\mathtt{P}}_{1}}], \sigma_V([\overline{\mathtt{Tx}^{\mathtt{P}}_{1}}]), [\overline{\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}}], R_P)$ after Claim() was successful
- $rc\ldots$ initially $\perp$, holds $([\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}], \sigma_P([\mathtt{TX}^{\mathtt{V}}_{\mathtt{unl}}]))$ after RespondClaim was successful
- $fp\ldots$ initially $\perp$, holds $([\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}], \sigma_P([\mathtt{TX}^{\mathtt{V}}_{\mathtt{unl}}]))$ after FinalizeProof() was successful

Constructor($\zeta_{init} := \{\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{id}, \mathtt{TX}_{\mathtt{f}}, \mathsf{id}_{\mathsf{ch}}, T_0, T_2, \alpha, \alpha'\}$ $, \sigma_V(\zeta_{init}))$:
Set $T_1 := \lfloor \frac{T_0+T_2}{2} \rfloor$. If (i) $T_0 < T_1 < T_2$ are times in the future (ii) $msg.value() = \alpha$ (iii) $msg.sender = P$ and (iv) $\mathsf{Vrfy}_{\mathsf{pk}_V}(\zeta_{init}; \sigma_V(\zeta_{init})) = 1$ then:
- $\mathsf{State} \leftarrow$ `Init`.
- Save ζ_{init} in the state.

SubmitProof($[\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}], \sigma_V([\mathtt{TX}^{\mathtt{P}}_{\mathtt{unl}}]), [\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}], \sigma_P([\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}])$):
Parse ζ_{init} as $\{\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{id}, \mathtt{TX}_{\mathtt{f}}, \mathsf{id}_{\mathsf{ch}}, T_0, T_2, \alpha, \alpha'\}$ and Check the following:
- $now < T_0$.
- $\mathsf{State} ==$ `Init`
- $\mathsf{Vrfy}_{\mathsf{pk}_V}([\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}]; \sigma_V([\mathtt{TX}^{\mathtt{P}}_{\mathtt{unl}}])) = 1 \wedge \mathsf{Vrfy}_{\mathsf{pk}_P}([\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}]; \sigma_P([\mathtt{TX}^{\mathtt{V}}_{\mathtt{unl}}])) = 1$
- $[\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}].\mathsf{In} = [\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}].\mathsf{In} = \mathtt{TX}_{\mathtt{f}}.\mathsf{txid} \| 1$
- $[\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}]$ and $[\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}]$ have the same balance distribution in their output, and both have no transaction level timelock.
- The revocation public key of $[\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}]$ is R_V, specified in the `OP_RETURN` output of $[\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}]$.
- Additionally, id is specified in the `OP_RETURN` of $[\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}]$.

If all checks passed, set $sp := ([\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}], \sigma_V([\mathtt{TX}^{\mathtt{P}}_{\mathtt{unl}}]), [\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}], \sigma_P([\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}]))$ and set $\mathsf{State} \leftarrow$ `Valid_Proof`. Otherwise abort.

Claim($[\overline{\mathtt{Tx}^{\mathtt{P}}_{1}}], \sigma_V([\overline{\mathtt{Tx}^{\mathtt{P}}_{1}}]), [\overline{\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}}], R_P$)
Parse ζ_{init} as $\{\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{id}, \mathtt{TX}_{\mathtt{f}}, \mathsf{id}_{\mathsf{ch}}, T_0, T_2, \alpha, \alpha'\}$ and check the following:
- $now < T_0$
- $\mathsf{State} ==$ `Init`
- $[\overline{\mathtt{Tx}^{\mathtt{P}}_{1}}].\mathsf{In} = [\overline{\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}}].\mathsf{In} = \mathtt{TX}_{\mathtt{f}}.\mathsf{txid} \| 1$
- $\mathsf{Vrfy}_{\mathsf{pk}_V}([\overline{\mathtt{Tx}^{\mathtt{P}}_{1}}]; \sigma_V([\overline{\mathtt{Tx}^{\mathtt{P}}_{1}}])) = 1$
- $[\overline{\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}}]$ has no transaction level timelock
- $([\overline{\mathtt{Tx}^{\mathtt{P}}_{1}}], [\overline{\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}}]) \in \alpha'.r$ ▷ check if the new proposed state is a valid transition

If all checks passed, set $cl := ([\overline{\mathtt{Tx}^{\mathtt{P}}_{1}}], \sigma_V([\overline{\mathtt{Tx}^{\mathtt{P}}_{1}}]), [\overline{\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}}], R_P)$ in the state and $\mathsf{State} \leftarrow$ `Dispute`, otherwise abort.

RespondClaim($\{\bar{r}_P\}$ OR $\{[\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}], \sigma_V([\mathtt{TX}^{\mathtt{P}}_{\mathtt{unl}}])\}$)
If $\mathsf{State} ==$ `Dispute` and $now < T_1$, retrieve $[\overline{\mathtt{Tx}^{\mathtt{P}}_{1}}], \sigma_V([\overline{\mathtt{Tx}^{\mathtt{P}}_{1}}]), [\overline{\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}}], R_P$ from cl and do the following, otherwise abort:
- If $\bar{r}_P$ is provided, retrieve $\bar{R}_P$ from $[\overline{\mathtt{Tx}^{\mathtt{P}}_{1}}].\mathsf{Out}$ and check if $(\bar{R}_P, \bar{r}_P) \in R$. If this is the case, then $\mathsf{State} \leftarrow$ `Invalid_Claim`, otherwise abort.
- Else, if $\{[\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}], \sigma_V([\mathtt{TX}^{\mathtt{P}}_{\mathtt{unl}}])\}$ is provided then check the followings:
  - $[\overline{\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}}]$ is the same as $[\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}]$, except for the `OP_RETURN` output, where $[\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}]$ contains R_V
  - $\mathsf{Vrfy}_{\mathsf{pk}_V}([\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}]; \sigma_V([\mathtt{TX}^{\mathtt{P}}_{\mathtt{unl}}])) = 1$
  - $[\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}].\mathsf{In} = \mathtt{TX}_{\mathtt{f}}.\mathsf{txid} \| 1$
  - Conditional outputs of $[\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}]$ are locked with R_P
  - $[\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}].\mathsf{Out}$ should be well-formed based on the format specified in the $\mathtt{GenCom}^V$ function (defined in Section A-B4)

  If all of these checks passed, save the new, unlocked, and signed channel state $rc := ([\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}], \sigma_P([\mathtt{TX}^{\mathtt{V}}_{\mathtt{unl}}]))$ and set $\mathsf{State} \leftarrow$ `Dispute_Resolved` otherwise abort.

FinalizeProof($[\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}], \sigma_P([\mathtt{TX}^{\mathtt{V}}_{\mathtt{unl}}])$)
Retrieve $([\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}], \sigma_P([\mathtt{TX}^{\mathtt{V}}_{\mathtt{unl}}]))$ from rc and check the followings:
- $now < T_2$
- $\mathsf{State} ==$ `Dispute_Resolved`
- $\mathsf{Vrfy}_{\mathsf{pk}_P}([\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}]; \sigma_P([\mathtt{TX}^{\mathtt{V}}_{\mathtt{unl}}])) = 1$
- $[\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}].\mathsf{In} = \mathtt{TX}_{\mathtt{f}}.\mathsf{txid} \| 1$
- $[\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}].\mathsf{Out}$ should be well-formed based on the format specified in the $\mathtt{GenCom}^P$ function
- $[\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}]$ should have no transaction level timelock
- $[\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}]$ and $[\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}]$ have the same balance distribution in their Out.
- The revocation public key of the conditional outputs of $[\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}]$ is R_V, which is specified in the `OP_RETURN` output of $[\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}}]$.

If all checks passed, then set $fp := ([\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}], \sigma_P([\mathtt{TX}^{\mathtt{V}}_{\mathtt{unl}}]))$ and set $\mathsf{State} \leftarrow$ `Valid_Proof`, otherwise abort.

Settle()

- If $\mathsf{State} ==$ `Valid_Proof` then read $\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}}$ from sp or fp (whichever is not $\perp$) and transfer coins to P and verifier according to $\alpha'.f(\mathtt{Tx}^{\mathtt{V}}_{\mathtt{unl}})$. Set $\mathsf{State} \leftarrow$ `Transferred`.
- Else if $now \geq T_2$ Check the following otherwise abort.
  - If $\mathsf{State} ==$ `Claim`, transfer α to P and set $\mathsf{State} \leftarrow$ `Transferred_P`
  - Else If $\mathsf{State} ==$ `Invalid_Claim`, transfer α to V and set $\mathsf{State} \leftarrow$ `Transferred_V`
  - Else if $\mathsf{State} ==$ `Init` or $\mathsf{State} ==$ `Dispute_Resolved`, transfer α to V and set $\mathsf{State} \leftarrow$ `Transferred_V`.

APPENDIX C
ALBA PROTOCOL

We denote Π as a *hybrid* protocol that has access to the ideal functionalities $\mathcal{F}_{\mathsf{prelim}}$ consisting of $\mathcal{F}_{Channel}$, $\mathcal{G}_{Ledger}$, $\mathcal{F}_{GDC}$ and $\mathcal{G}_{clock}$. The protocol is shown as pseudocode.

We note that even though our functionality is generic any generalized payment channel functionality $\mathcal{PC}$, we are using specifically lightning channels in our protocol, which is why parties are running the lightning channel simulator internally. Both the channel functionality and simulator are presented in Section A-B. Intuitively, honest parties are ensured not to lose funds on their channel in $\mathcal{PC}$, since if there is a problem with the update, the channel γ is set to idle, which means that after the contract is resolved on $\mathcal{L}_D$, the channel is force-closed.

Similar to the functionality, the protocol proceeds in different phases, which we delineate on a high level now. In the *setup* phase, the parties perform some checks, set up and deploy the smart contract, with code Script_G as defined in Section B, on $\mathcal{L}_D$. Once successful, in the *channel update and proof* phase, P can try to initiate an update on $\mathcal{PC}$ to a valid and unlocked state (by unlocked we mean a commitment without a transaction level timelock), if instructed by $\mathcal{Z}$ to do so. In case the update is successful, P extracts the necessary information and calls the SubmitProof function of the contract. In case something goes wrong, P will run `CallClaimFunc`, which extracts the necessary information and calls the Claim function of the contract.

The subprocedure $\texttt{MonitorDestLedger}_P$ is run after P calls the Claim function and checks how V responds. In case V resolves the claim, P will call FinalizeProof, and finally settle the contract. Otherwise, if V does not respond to the proof, P will settle the contract after T_2. This, along with the way the contract is defined, ensures that an honest P can always enforce the outcome associated with the latest valid state.

The subprocedure $\texttt{MonitorDestLedger}_V$ is run by V after the setup phase. In it, V checks if the Claim function of the contract was called. In case the claim is invalid, V extracts the revocation secret and calls the RespondClaim function. If the claim was valid, V generates the necessary commitment transaction and calls RespondClaim. V updates the necessary payment channel information after P has called FinalizeProof, and force-closes the channel. Finally, V settles the contract. This shields an honest V from P trying to settle with an invalid or old state.

Alba Protocol: Parties Interacting with the Smart Contract

Parameters:
$\mathcal{L}_\mathcal{S}, \mathcal{L}_\mathcal{D} \ldots$ two instances of $\mathcal{G}_{Ledger}$ representing the source and destination blockchain.
$\mathcal{PC}_\mathcal{S} \ldots$ an instance of the payment channel ideal functionality.
$\mathsf{Script}_G \ldots$ the smart contract code as defined in Section B

Variables:
$\phi(.) \ldots$ a mapping of Alba ID id to $(\mathsf{pk}_P, \mathsf{pk}_V, \mathtt{TX_f}, \mathsf{id_{Ch}}, T_0, T_2, \mathsf{Addr})$ where $\mathsf{id} \in \{0,1\}^*$ is an identifier unique to the pair P and V. $\mathtt{TX_f}$ is the funding transaction of the channel and $\mathsf{id_{ch}}$ is the id of the payment channel. Further $T_0, T_2 \in \mathbb{N}$. Addr is the address of the instance of the Alba contract. Each party stored their own ϕ.
$t_{updateDelay} \ldots$: The time it takes to perform a channel update, given by $\mathcal{PC}$.

Setup

Party V upon $(\texttt{A–SETUP}, \mathsf{id}, \mathsf{id_{ch}}, P, T_0, T_2, \alpha, \alpha') \overset{\tau}{\hookleftarrow} \mathcal{Z}$

1) Read $\Gamma^V(\mathsf{id_{ch}})$ and parse the output as $(\{\gamma\}, \mathtt{TX_f}, \ldots)$, if $\Gamma^V(\mathsf{id})$ returned $\perp$ go idle otherwise check the followings:
   - $\gamma.\mathsf{otherParty}(V) = P$
   - $\gamma.\mathsf{cash} \geq \alpha$
   - $T_0 + 2\Delta_D < T_2$ holds and T_0, T_2 are times in the future.
2) If all of the above checks hold, define $\zeta_{init} := \{\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{id}, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_2, \alpha, \alpha'\}$, create a signature of it $\sigma_V(\zeta_{init}) := \mathsf{Sign}_{\mathsf{sk}_V}(\zeta_{init})$, and then send $(\text{GSetupInfo}, \zeta_{init}, \sigma_V(\zeta_{init})) \overset{\tau}{\hookrightarrow} P$.

Party P upon $(\text{GSetupInfo}, \zeta_{init}, \sigma_V(\zeta_{init})) \overset{t}{\hookleftarrow} V$

3) Parse ζ_{init} as $\{\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{id}, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_2, \alpha, \alpha'\}$. Read $\Gamma^P(\mathsf{id_{ch}})$ and parse the output as $(\{\gamma\}, \mathtt{TX_f}, \ldots)$, if $\Gamma^P(\mathsf{id_{ch}})$ returned $\perp$ go idle otherwise check the followings:
   - $\gamma.\mathsf{otherParty}(P) = V$
   - $\zeta_{init}.\mathtt{TX_f} = \gamma.\mathtt{TX_f}$
   - $\gamma.\mathsf{cash} \geq \alpha$
   - $T_0 + 2\Delta_D < T_2$ holds and T_0, T_2 are times in the future.
   - $\mathsf{Vrfy}_{\mathsf{pk}_V}(\zeta_{init}; \sigma_V(\zeta_{init})) = 1$
4) If the above checks passed, send $(\texttt{A–CREATED}, \mathsf{id}, \mathsf{id_{ch}}, T_0, T_2, \alpha, \alpha') \overset{t}{\hookrightarrow} \mathcal{Z}$ and send $(\text{InitiateContract}, code = \mathsf{Script}_G, function = \text{Constructor}, args = \{\zeta_{init}, \sigma_V(\zeta_{init})\}) \overset{t}{\hookrightarrow} \mathcal{L}_D$,
5) Wait until the deployment transaction is accepted by $\mathcal{L}_D$ and save its address in variable Addr.
6) Then store $\phi(\mathsf{id}) = (\mathsf{pk}_P, \mathsf{pk}_V, \theta_{P,V}, \mathsf{id_{ch}}, T_0, T_2, \alpha, \alpha', \mathsf{Addr})$, set $\gamma.\mathsf{Aid} \leftarrow \mathsf{id}, \gamma.\mathsf{ADeadline} \leftarrow T_2$.

Party V at time $\tau + 1$

7) Constantly observe $\mathcal{L}_D$. If a smart contract with code Script_G appears by time $\tau_1 \leq \tau + 1 + \Delta_V$ and data ζ_{init} saved in it, then store $\phi(\mathsf{id}) = (\mathsf{pk}_P, \mathsf{pk}_V, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_2, \alpha, \alpha', \mathsf{Addr})$ and continue otherwise stop.
8) Send $(\texttt{A–CREATED}, \mathsf{id}, \mathsf{id_{ch}}, T_0, T_2, \alpha, \alpha') \overset{\tau_1}{\hookrightarrow} \mathcal{Z}$
9) Set $\gamma.\mathsf{Aid} \leftarrow \mathsf{id}$ and $\gamma.\mathsf{ADeadline} \leftarrow T_2$ and run $\texttt{MonitorDestLedger}_V(\mathsf{id})$.

Channel Update and Proof

Party P upon $(\texttt{A–INITIATE–PAYMENT}, \mathsf{id}, \vec{\theta}) \overset{t_0}{\hookleftarrow} \mathcal{Z}$

1) If this phase is already being executed, go idle.
2) Read $\phi(\mathsf{id})$ from the storage and parse it as $(\mathsf{pk}_P, \mathsf{pk}_V, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_2, \alpha, \alpha', \mathsf{Addr})$ and then read $\Gamma(\mathsf{id_{ch}})$ as $(\gamma, \mathtt{TX_f})$ from $\mathcal{PC}_\mathcal{S}$. If $\vec{\theta} \neq \alpha'.r(\gamma.\mathsf{st})$, go idle.
3) If $(\gamma.\mathsf{idle} == True)$, distinguish the time. If $(t_0 > T_0 - \Delta_D)$, stop, otherwise if $(t_0 \leq T_0 - \Delta_D)$ run $\texttt{CallClaimFunc}(\mathsf{id}, \vec{\theta})$, and then stop.
4) If $(\gamma.\mathsf{idle} == False)$ and $(t_0 > T_0 - t_{updateDelay} - \Delta_D)$ stop, otherwise if $(t_0 \leq T_0 - t_{updateDelay} - \Delta_D)$ continue.
5) Send $(\texttt{UPDATE}, \mathsf{id_{ch}}, \vec{\theta}, True) \overset{t_0}{\hookrightarrow} \mathcal{PC}$ to initiate the channel update and proceed with the update until it is either successful (outputs `UPDATED`) or stops unsuccessfully.
6) If the update was unsuccessful, execute $\texttt{CallClaimFunc}(\mathsf{id}, \vec{\theta})$ and stop. Else continue.
7) Read $\Gamma^P(\mathsf{id_{ch}})$ from the $\mathcal{S}_{\mathcal{PC}}$'s storage and parse it as $(\gamma, \mathtt{TX_f}, (\mathtt{TX}_1^\mathtt{P}, \mathtt{TX}_1^\mathtt{V}, r_P, R_V, \bar{r}_P))$; then parse $\mathtt{TX}_1^\mathtt{P}$ as $([\mathtt{TX}_1^\mathtt{P}],$

$\{\sigma_P(\texttt{TX}_1^{\texttt{P}}), \sigma_V([\texttt{TX}_1^{\texttt{P}}])\}$) and $\texttt{TX}_1^{\texttt{V}}$ as $([\texttt{TX}_1^{\texttt{V}}], \sigma_P([\texttt{TX}_1^{\texttt{V}}]))$. Send (CallFunction, $function$ = SubmitProof, $args$ = $\{[\texttt{TX}_1^{\texttt{P}}], \sigma_V([\texttt{TX}_1^{\texttt{P}}]), [\texttt{TX}_1^{\texttt{V}}], \sigma_P([\texttt{TX}_1^{\texttt{V}}])\}$) $\overset{t}{\hookrightarrow} \mathcal{L}_D$.

Subprocedures

$\texttt{CallClaimFunc}(\mathsf{id}, \vec{\theta})$:

- Parse $\phi(\mathsf{id})$ as $(\mathsf{pk}_P, \mathsf{pk}_V, \texttt{TX}_\texttt{f}, \mathsf{id}_{\mathsf{ch}}, T_0, T_2, \alpha, \alpha', \mathsf{Addr})$.
- Read $\Gamma^P(\mathsf{id}_{\mathsf{ch}})$ from the storage of $\mathcal{S}_{\mathcal{PC}}$ and parse it as $(\gamma, \texttt{TX}_\texttt{f}, (\overline{\texttt{TX}_1^{\texttt{V}}}, \overline{\texttt{TX}_1^{\texttt{P}}}, \ldots, \bar{R}_P, \ldots))$, moreover, parse $\overline{\texttt{TX}_1^{\texttt{P}}}$ as $([\overline{\texttt{TX}_1^{\texttt{P}}}], \{\mathsf{Sign}_{\mathsf{sk}_P}([\overline{\texttt{TX}_1^{\texttt{P}}}]), \sigma_V([\overline{\texttt{TX}_1^{\texttt{P}}}])\})$
- Generate $(R_P, r_P) \leftarrow \mathsf{GenR}$. ▷ Even if we already created a revocation key before and exchanged commitment transactions with it, we don't care because if this function is called, P does not have any signed (valid) commitment transaction from V.
- Create a new transaction $[\overline{\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}}] := \texttt{GenCom}^V([\texttt{TX}_\texttt{f}], (\mathsf{pk}_P, R_P), (\mathsf{pk}_V, _), \bar{R}_P, \vec{\theta}, \mathsf{id}, \bot, True)$, using the function defined in Section A-B4
- Send message (CallFunction, $function$ = Claim, $args$ = $\{[\overline{\texttt{Tx}_1^{\texttt{P}}}], \sigma_V([\overline{\texttt{TX}_1^{\texttt{P}}}]), [\overline{\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}}], R_P\}$) $\overset{now}{\hookrightarrow} \mathcal{L}_D$
- Run the $\texttt{MonitorDestLedger}_P(\mathsf{id})$ function.

$\texttt{MonitorDestLedger}_P(\mathsf{id})$:▷ Only called if V does not cooperate. Executed every round once called.

1) Parse $\phi(\mathsf{id})$ as $(\mathsf{pk}_P, \mathsf{pk}_V, \texttt{TX}_\texttt{f}, \mathsf{id}_{\mathsf{ch}}, T_0, T_2, \alpha, \alpha', \mathsf{Addr})$.
2) If $now \leq T_2$:
   - Send (read, $address$ = Addr, $variable$ = State) $\overset{now}{\hookrightarrow} \mathcal{L}_D$, save the output in State.
   - If State == Dispute_Resolved, then send (read, $address$ = Addr, $variable$ = rc := $([\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}], \sigma_P([\texttt{TX}_{\texttt{unl}}^{\texttt{V}}]))$ $\overset{now}{\hookrightarrow} \mathcal{L}_D$, save the corresponding part of the output in $[\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}]$ and do the followings:
     - Read $\Gamma^P(\mathsf{id}_{\mathsf{ch}})$ from the storage of $\mathcal{S}_{\mathcal{PC}}$ and extract $\bar{r}_P$ and $\bar{R}_V$ from it.▷ corresponding to the same commitment transaction that was used to call the Claim function.
     - Extract $\vec{\theta}$, R_P and R_V from $[\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}]$.Out.
     - Create $[\texttt{Tx}_{\texttt{unl}}^{\texttt{V}}]$:= $\texttt{GenCom}^P([\texttt{TX}_\texttt{f}], (\mathsf{pk}_V, \bar{R}_V), (\mathsf{pk}_P, R_P]), R_P, \vec{\theta}, \mathsf{id}, \bot, True)$, using the function defined in Section A-B4.
     - Sign the transaction $\sigma_P([\texttt{Tx}_{\texttt{unl}}^{\texttt{V}}]) := \mathsf{Sign}_{\mathsf{sk}_P}([\texttt{Tx}_{\texttt{unl}}^{\texttt{V}}])$.
     - Send (CallFunction, $function$ = FinalizeProof, $args$ = $\{[\texttt{Tx}_{\texttt{unl}}^{\texttt{V}}], \sigma_P([\texttt{Tx}_{\texttt{unl}}^{\texttt{V}}])\}$) $\overset{now}{\hookrightarrow} \mathcal{L}_D$.
     - Read $\Gamma^P(\mathsf{id}_{\mathsf{ch}})$ from the storage of $\mathcal{S}_{\mathcal{PC}}$ and Parse it as $(\gamma, \texttt{TX}_\texttt{f}, (\overline{\texttt{TX}_1^{\texttt{V}}}, \overline{\texttt{TX}_1^{\texttt{P}}} \bar{r}_P, \bar{R}_V, \ldots))$. Update the $\mathcal{S}_{\mathcal{PC}}$'s channel space as $\Gamma^P(\mathsf{id}_{\mathsf{ch}})$:= $(\gamma, \texttt{TX}_\texttt{f}, (\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}, \texttt{Tx}_{\texttt{unl}}^{\texttt{V}}, r_P, R_V, \bar{r}_P))$ and run subprocedure $\texttt{ForceClose}^P(\mathsf{id}_{\mathsf{ch}})$ in $\mathcal{PC}$. ▷ In this case, P is in the idle mode and doesn't update the channel state in the Payment channel protocol therefore we need to update the channel space.
   - Else if (State == Valid_Proof) and if the contract is not already settled send (CallFunction, $function$ = Settle) $\overset{now}{\hookrightarrow} \mathcal{L}_D$. Return.
   - Else if $now == T_2$ and the contract is not already settled send (CallFunction, $function$ = Settle) $\overset{now}{\hookrightarrow} \mathcal{L}_D$. Return.

$\texttt{MonitorDestLedger}_V(\mathsf{id})$:▷ Called after the setup phase. Executed every round once called.

1) Parse $\phi(\mathsf{id})$ as $(\mathsf{pk}_P, \mathsf{pk}_V, \texttt{TX}_\texttt{f}, \mathsf{id}_{\mathsf{ch}}, T_0, T_2, \alpha, \alpha', \mathsf{Addr})$.
2) Set $UpdateChanSpace \leftarrow False$.
3) While $now \leq T_2$: ▷ The proof might be exactly posted at T_2 and then Settle function can be called after T_2.
   - Send (read, $address$ = Addr, $variable$ = State) $\overset{now}{\hookrightarrow} \mathcal{L}_D$, save the output in State.
   - If State == Dispute, set $\gamma.\mathsf{idle} = True$, then send (read, $address$ = Addr, $variable$ = cl := $([\overline{\texttt{Tx}_1^{\texttt{P}}}], \sigma_V([\overline{\texttt{Tx}_1^{\texttt{P}}}]), [\overline{\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}}], R_P)$) $\overset{now}{\hookrightarrow} \mathcal{L}_D$, save the output in $[\overline{\texttt{TX}_1^{\texttt{P}}}]$, R_P and do the followings: ▷ The fact that the claim function has been called means that P did not complete the Alba update because V is honest and P has no reason to call the claim function if the update was complete.
     - If $\overline{\texttt{TX}_1^{\texttt{P}}}$ existed in the channel history $\Theta^V(\mathsf{id}_{\mathsf{ch}})$ of $\mathcal{S}_{\mathcal{PC}}$ as $([\overline{\texttt{TX}_1^{\texttt{P}}}], \bar{r}_P, \bar{r}_V, \bar{s}_{\mathsf{c}}^P)$, then send (CallFunction, $function$ = RespondClaim, $args = \{\bar{r}_P\}$) $\overset{now}{\hookrightarrow} \mathcal{L}_D$.
     - Else, extract $\vec{\theta}$ and $\bar{R}_P$ from $[\overline{\texttt{TX}_1^{\texttt{P}}}]$.Out; Change $\vec{\theta}$ as follows, $\theta_P.\mathsf{cash} = \theta_P.\mathsf{cash} - \alpha$, $\theta_V.\mathsf{cash} = \theta_V.\mathsf{cash} + \alpha$. Generate $(R_V, r_V) \leftarrow \mathsf{GenR}$ and save it in the local storage. ▷ It might be the case that P and V already exchanged signatures on the same update but the malicious P still Claimed, in this situation the latest channel update remains unrevoked and in fact, a new channel update is initiated on the chain and V is in Idle mode.
       Then generate $[\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}] := \texttt{GenCom}^V([\texttt{TX}_\texttt{f}], (\mathsf{pk}_P, R_P), (\mathsf{pk}_V, R_V), (\bar{R}_P), \vec{\theta}, \mathsf{id})$. $[\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}]$ should look exactly the same as $[\overline{\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}}]$, except for having R_V in the OP_RETURN output. ▷ $[\overline{\texttt{TX}_1^{\texttt{P}}}]$ submitted by P is either a legitimate latest state which is saved in $\Gamma^V(\mathsf{id}_{\mathsf{ch}})$ or is a channel update which has V's signature on it but didn't go through, the fact that transaction did not go through means that the honest V is in the idle mode. In this case, there are two valid states at the same time
       Sign the transaction $\sigma_V([\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}]) := \mathsf{Sign}_{\mathsf{sk}_V}([\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}])$. Then send (CallFunction, $function$ = RespondClaim, $args = \{[\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}], \sigma_V([\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}])\}$) $\overset{now}{\hookrightarrow} \mathcal{L}_D$ and set $updateChanSpace \leftarrow True$.
   - Else if (State == Valid_Proof) do the following:
     - If ($updateChanSpace == True$) then Send (read, $address$ = Addr, $variable$ = fp := $([\texttt{Tx}_{\texttt{unl}}^{\texttt{V}}], \sigma_P([\texttt{TX}_{\texttt{unl}}^{\texttt{V}}]))$) $\overset{now}{\hookrightarrow} \mathcal{L}_D$. Retrieve r_V and $\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}$ from the local storage (as created in the previous step), retrieve $\bar{r}_V$ from $\Theta^V(\mathsf{id}_{\mathsf{ch}})$ stored in $\mathcal{S}_{\mathcal{PC}}$ and update $\Gamma^V(\mathsf{id}_{\mathsf{ch}})$:= $(\gamma, \texttt{TX}_\texttt{f}, (\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}, \texttt{Tx}_{\texttt{unl}}^{\texttt{V}}, r_V, R_P, \bar{r}_V))$. Then set $updateChanSpace \leftarrow False$ and run $\texttt{ForceClose}^V(\mathsf{id}_{\mathsf{ch}})$ in $\mathcal{PC}$. ▷ In the case of an On-chain update, the channel space should be updated here because probably V is in the idle mode.
   - Else if $now == T_2$ or State == Invalid_Claim and if the contract is not settled yet, send (CallFunction, $function$ = Settle) $\overset{now}{\hookrightarrow} \mathcal{L}_D$.

APPENDIX D
UC PROOF

For the proof, we start by presenting the simulator $\mathcal{S}$, which interacts with $\mathcal{Z}$ and $\mathcal{F}_{\mathsf{Alba}}$. Then, we will walk through the simulation and prove that the protocol Π is a GUC-realization of $\mathcal{F}_{\mathsf{Alba}}$.

Variables

$\phi(.)\ldots$ a mapping of Alba ID id to $(\mathsf{pk}_P, \mathsf{pk}_V, \mathtt{TX_f}, \mathsf{id_{Ch}}, T_0, T_2, \mathsf{Addr})$ where $\mathsf{id} \in \{0,1\}^*$ is an identifier unique to the pair P and V. $\mathtt{TX_f}$ is the funding transaction of the channel and $\mathsf{id_{ch}}$ is the id of the payment channel. Further $T_0, T_2 \in \mathbb{N}$. Addr is the address of the instance of the Alba contract.
$t_{updateDelay}\ldots$: The time it takes to perform a channel update, given by $\mathcal{PC}$.
$\mathsf{Script}_G\ldots$ the smart contract code as defined in Section B

Setup

Case P honest, V dishonest

1) Upon V sending $(\text{GSetupInfo}, \zeta_{init}, \sigma_V(\zeta_{init})) \overset{\tau}{\hookrightarrow} P$, send $(\mathtt{A\text{–}SETUP}, \mathsf{id}, \mathsf{id_{ch}}, P, T_0, T_2, \alpha, \alpha') \overset{\tau}{\hookrightarrow} \mathcal{F}_{\mathsf{Alba}}$ on behalf of V and do the following.
2) Parse ζ_{init} as $\{\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{id}, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_2, \alpha, \alpha'\}$. Read $\Gamma^P(\mathsf{id_{ch}})$ and parse the output as $(\{\gamma\}, \mathtt{TX_f}, \ldots)$, if $\Gamma^P(\mathsf{id_{ch}})$ returned $\perp$ go idle otherwise check the followings:
   - $\gamma.\mathsf{otherParty}(P) = V$
   - $\zeta_{init}.\mathtt{TX_f} = \gamma.\mathtt{TX_f}$
   - $\gamma.\mathsf{cash} \geq \alpha$
   - $T_0 + 2\Delta_D < T_2$ holds and T_0, T_2 are times in the future.
   - $\mathsf{Vrfy}_{\mathsf{pk}_V}(\zeta_{init}; \sigma_V(\zeta_{init})) = 1$
3) If the above checks passed, send $(\text{InitiateContract}, code = \mathsf{Script}_G, function = \text{Constructor}, args = \{\zeta_{init}, \sigma_V(\zeta_{init})\}) \overset{t:=\tau+1}{\hookrightarrow} \mathcal{L}_D$,
4) Wait until the deployment transaction is accepted by $\mathcal{L}_D$ at round $t_1 \leq t + \Delta_D$ and save its address in variable Addr.
5) Send $(\mathtt{CONTRACT\text{–}INCLUDED}, \mathsf{Addr}) \overset{t_1}{\hookrightarrow} \mathcal{F}_{\mathsf{Alba}}$ write its address to Addr and add $(\mathsf{pk}_P, \mathsf{pk}_V, \mathtt{TX_f}, \mathsf{id_{Ch}}, T_0, T_2, \mathsf{Addr})$ in ϕ and continue otherwise stop.
6) Then, store $\phi(\mathsf{id}) = (\mathsf{pk}_P, \mathsf{pk}_V, \theta_{P,V}, \mathsf{id_{ch}}, T_0, T_2, \alpha, \alpha', \mathsf{Addr})$, and set $\gamma.\mathsf{Aid} \leftarrow \mathsf{id}, \gamma.\mathsf{ADeadline} \leftarrow T_2$.

Case V honest, P dishonest

1) Upon V sending $(\mathtt{A\text{–}SETUP}, \mathsf{id}, \mathsf{id_{ch}}, P, T_0, T_2, \alpha, \alpha') \overset{\tau}{\hookrightarrow} \mathcal{F}_{\mathsf{Alba}}$
2) Read $\Gamma^V(\mathsf{id_{ch}})$ and parse the output as $(\{\gamma\}, \mathtt{TX_f}, \ldots)$, if $\Gamma^V(\mathsf{id})$ returned $\perp$ go idle otherwise check the followings:
   - $\gamma.\mathsf{otherParty}(V) = P$
   - $\gamma.\mathsf{cash} \geq \alpha$
   - $T_0 + 2\Delta_D < T_2$ holds and T_0, T_2 are times in the future.
3) If all of the above checks hold, define $\zeta_{init} := \{\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{id}, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_2, \alpha, \alpha'\}$, sign it $\sigma_V(\zeta_{init}) := \mathsf{Sign}_{\mathsf{sk}_V}(\zeta_{init})$, and then send message $(\text{GSetupInfo}, \zeta_{init}, \sigma_V(\zeta_{init})) \overset{\tau}{\hookrightarrow} P$.
4) If a smart contract with code Script_G and data ζ_{init} saved in it appears on $\mathcal{L}_D$ at time $t_1 \leq t + \Delta_D$, then store $\phi(\mathsf{id}) = (\mathsf{pk}_P, \mathsf{pk}_V, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_2, \alpha, \alpha', \mathsf{Addr})$ and continue otherwise stop.
5) Send $(\mathtt{CONTRACT\text{–}INCLUDED}, \mathsf{Addr}) \overset{t_1}{\hookrightarrow} \mathcal{F}_{\mathsf{Alba}}$
6) Set $\gamma.\mathsf{Aid} \leftarrow \mathsf{id}$ and $\gamma.\mathsf{ADeadline} \leftarrow T_2$ and run $\mathtt{MonitorDestLedger}^V(\mathsf{id})$.

Channel Update and Proof

Case P honest

Upon $\mathcal{F}_{\mathsf{Alba}}$ sending $(\mathtt{UPDATE}, \mathsf{id_{ch}}, \vec{\theta}, True) \overset{now}{\longrightarrow} \mathcal{PC}$ to initiate the channel update, wait to see whether the update is successful (outputs $\mathtt{UPDATED}$) or stops unsuccessfully. In case it is successful, read $\Gamma^P(\mathsf{id_{ch}})$ from the $\mathcal{S}_{\mathcal{PC}}$'s storage and parse it as $(\gamma, \mathtt{TX_f}, (\mathtt{TX}_1^{\mathtt{P}}, \mathtt{TX}_1^{\mathtt{V}}, r_P, R_V, \bar{r}_P))$; then parse $\mathtt{TX}_1^{\mathtt{P}}$ as $([\mathtt{TX}_1^{\mathtt{P}}], \{\sigma_P(\mathtt{TX}_1^{\mathtt{P}}), \sigma_V([\mathtt{TX}_1^{\mathtt{P}}])\})$ and $\mathtt{TX}_1^{\mathtt{V}}$ as $([\mathtt{TX}_1^{\mathtt{V}}], \sigma_P([\mathtt{TX}_1^{\mathtt{V}}]))$. Send $(\text{CallFunction}, function = \text{SubmitProof}, args = \{[\mathtt{TX}_1^{\mathtt{P}}], \sigma_V([\mathtt{TX}_1^{\mathtt{P}}]), [\mathtt{TX}_1^{\mathtt{V}}], \sigma_P([\mathtt{TX}_1^{\mathtt{V}}]), \bar{r}_P\}) \overset{t}{\hookrightarrow} \mathcal{L}_D$.

Simulator subprocedures

$\mathtt{CallClaimFunc/MonitorDestLedger}_P$:
Upon receiving a message (Claim) from $\mathcal{F}_{\mathsf{Alba}}$ in round t, execute the code specified in the protocol under subprocedure $\mathtt{MonitorDestLedger}_P(\mathsf{id})$. If therein, in step 2 the read state is $\mathtt{Valid_Proof}$ in round t_1, send (ClaimOk) to $\mathcal{F}_{\mathsf{Alba}}$. Else, if in step 2 the time is T_2 and the contract is not settled, send (NoResponse) to $\mathcal{F}_{\mathsf{Alba}}$.

$\mathtt{MonitorDestLedger}_V(\mathsf{id})$:▷ Called after the setup phase.
Execute the code specified in the protocol code under subprocedure $\mathtt{MonitorDestLedger}_V(\mathsf{id})$. If therein, in step 3 the read state is $\mathtt{Dispute}$ in round t_1 and the read state exists in the channel history such that the function RespondClaim of the contract is called, send (InvalidClaim) to $\mathcal{F}_{\mathsf{Alba}}$.

We will now walk through the simulation and show computationally indistinguishability for an environment $\mathcal{Z}$ on whether or not it is interacting with the real-world protocol Π or the idealized protocol $\phi_{\mathcal{F}_{Channel}}$ based on the functionality $\mathcal{F}_{Channel}$. More concretely, we look at the execution ensembles $\text{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}}$ and $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}}$ are the same for $\mathcal{Z}$, i.e., $\mathcal{Z}$ sees the same messages in the same rounds in both worlds. As mentioned, our protocol focuses on lightning channels, which is why, even though our ideal functionality is generic (for the payment channel functionality $\mathcal{PC}$ of [20], which can be found in Section A-B), we have our protocol parties and simulator $\mathcal{S}$ run the simulator code for lightning channels, which is presented Section A-B.

For simplicity, we denote observing message m in round τ as $m[\tau]$. Moreover, there is interaction with other ideal functionalities $\mathcal{F}_{\mathsf{prelim}}$, which in turn interact with other parties, with the environment, or there might be an impact on publicly observable variables, e.g., transactions appearing on a ledger. For simplicity, we denote the set of observable effects of sending a message m to functionality $\mathcal{F}$ in round τ as function $\mathsf{obsSet}(m, \mathcal{F}, \tau)$. Note that these effects depend on the internal state of $\mathcal{F}$. However, if two sets of identical messages are sent to a functionality $\mathcal{F}$ in the same rounds will trivially produce the same internal state.

We split this proof into lemmas on the different phases. For each phase, we distinguish the following corruption cases: (i) case P honest, V dishonest, (ii) case V honest, P dishonest. We emphasize that we do not need to simulate both parties being dishonest, as this is essentially the adversary talking to itself. One of the challenges in UC simulation comes from the simulator not having access to the secret inputs of the parties. We do not make any claims about the privacy of our protocol and assume that messages sent or received by $\mathcal{F}_{\mathsf{Alba}}$ are implicitly forwarded to $\mathcal{S}$. Thus, in our case there are no such secret inputs, instead, the environment sends merely commands. This means the main challenge comes from

handling the behavior of malicious parties. We therefore omit the simulation in case both parties are honest and note that this can be handled trivially by the simulator running essentially the protocol code.

We also emphasize that we are interested only in protocols that realize the functionality without ever outputting (ERROR), as otherwise all security guarantees are lost. Thus, (ERROR) should never be observed in any execution ensemble, which trivially holds if the protocol never mentions this message and GUC-realizes the functionality. The proofs argue how this holds accordingly.

The following lemmas and theorem hold assuming our signature scheme to be EUF-CMA secure, and our hash functions are modeled as random oracles.

Lemma 1. *The* setup *phase of the protocol* Π *GUC-realizes the* setup *phase of the functionality* $\mathcal{F}_{\mathsf{Alba}}$.

Proof. For convenience, we name the following messages:

- $m_0 := (\texttt{A–SETUP}, \mathsf{id}, \mathsf{id}_{\mathsf{ch}}, P, T_0, T_2, \alpha, \alpha')$
- $m_1 := (\text{GSetupInfo}, \zeta_{init}, \sigma_V(\zeta_{init}))$
- $m_2 := (\texttt{A–CREATED}, \mathsf{id}, \mathsf{id}_{\mathsf{ch}}, T_0, T_2, \alpha, \alpha')$
- $m_3 := (\text{InitiateContract}, code = \mathsf{Script}_G, function = \text{Constructor}, args = \{\zeta_{init}, \sigma_V(\zeta_{init})\})$

We consider the following cases:

1. P honest, V dishonest

1a. Real world: In the real world, P acts upon receiving m_1 in round t executing its code as defined in Π. If the checks pass, P sends m_2 to $\mathcal{Z}$ and m_3 to $\mathcal{L}_D$ in round t. This results in the ensemble $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \{m_2[t]\} \cup \mathsf{obsSet}(m_3, \mathcal{L}_D, t)$ or $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \emptyset$, depending on whether or not the checks pass.

1b. Ideal world: In the ideal world, $\mathcal{Z}$ can either instruct V to send m_1 in some round τ or not. $\mathcal{S}$ sees this message and sends m_0 to $\mathcal{F}_{\mathsf{Alba}}$ in round τ. Both $\mathcal{S}$ and $\mathcal{F}_{\mathsf{Alba}}$ perform the same checks on this message. If the checks pass, $\mathcal{F}_{\mathsf{Alba}}$ sends m_2 to $\mathcal{Z}$ (via dummy party P in round $t := \tau + 1$, and $\mathcal{S}$ sends m_3 to $\mathcal{L}_D$, also in round t. This results in the ensemble $\texttt{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}} := \{m_2[t]\} \cup \mathsf{obsSet}(m_3, \mathcal{L}_D, t)$ or $\texttt{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}} := \emptyset$, depending on whether or not the checks pass.

2. V honest, P dishonest

2a. Real world: V receives message m_0 in round τ from $\mathcal{Z}$. After performing the checks defined in the protocol code, (if they succeed) V sends m_1 to P in τ. $\mathcal{Z}$ controls $\mathcal{A}$, and thus P, who observes m_1 in round $\tau + 1$. V proceeds to observe $\mathcal{L}_D$, and if a smart contract with Script_G appears by some time $\tau_1 \leq \tau + 1 + \Delta_D$, sends m_2 to $\mathcal{Z}$. This results in execution ensemble of $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \{m_1[\tau+1], m_2[\tau_1]\}$ or $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \{m_1[\tau+1]$, depending on whether or not the smart contract with Script_G is posted by P.

2b. Ideal world: After $\mathcal{F}_{\mathsf{Alba}}$ receives m_0 in round τ, $\mathcal{F}_{\mathsf{Alba}}$ executes its code as defined for the setup phase. As P is under adversarial control, $\mathcal{Z}$ will not see any messages sent by $\mathcal{F}_{\mathsf{Alba}}$ to dummy party P, such as m_2. However, the simulator meanwhile sends message message m_1 to P in round τ, which is observed in round $\tau + 1$. The simulator proceeds to monitor $\mathcal{L}_D$, and if a smart contract with Script_G appears by some time $\tau_1 \leq \tau + 1 + \Delta_D$, $\mathcal{S}$ informs $\mathcal{F}_{\mathsf{Alba}}$, which in turn will send m_2 to $\mathcal{Z}$ in round τ_1. This results in execution ensemble of $\texttt{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}} := \{m_1[\tau+1], m_2[\tau_1]\}$ or $\texttt{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}} := \{m_1[\tau+1]$, depending on whether or not the smart contract with Script_G is posted by P.

We observe that the execution ensembles are identical and thus, computationally indistinguishable for $\mathcal{Z}$. □

Lemma 2. *The* channel update and proof *phase of the protocol* Π *GUC-realizes the* channel update and proof *phase of the functionality* $\mathcal{F}_{\mathsf{Alba}}$.

Proof. For convenience, we name the following messages:

- $m_4 := (\texttt{A–INITIATE–PAYMENT}, \mathsf{id}, \vec{\theta})$
- $m_5 := (\texttt{UPDATE}, \mathsf{id}_{\mathsf{ch}}, \vec{\theta}, True)$
- $m_6 := (\text{CallFunction}, function = \text{SubmitProof}, args = \{[\texttt{TX}^{\texttt{P}}_1], \sigma_V([\texttt{TX}^{\texttt{P}}_1]), [\texttt{TX}^{\texttt{V}}_1], \sigma_P([\texttt{TX}^{\texttt{V}}_1])\})$

Since this phase only involves P, we only need to consider P being honest. The proof of this lemma is straightforward, since there is no interaction, and there is essentially the same code executed in the protocol and the ideal functionality.

1. P honest

1a. Real world: In the real world, P acts upon receiving m_4 in round t_0. P reads the channel tuple γ from $\mathcal{PC}_{\mathcal{S}}$. We now distinguish some cases. If $\gamma.\mathsf{idle} = True$ (which means there has been a not-yet-resolved dispute in the payment channel), then P stops if $t_0 > T_0 - \Delta_D$, i.e., there is not enough time to claim the funds (case i). Also, even if $\gamma.\mathsf{idle} = False$, but time $t_0 > T_0 - t_{updateDelay} - \Delta_D$, P stops (same as case i). In this case, there is not enough time to perform an update, but still enough time to claim the funds. Or P runs CallClaimFunc if $t_0 \leq T_0 - \Delta_D$ (case ii). If this is not the case, i.e., ($\gamma.\mathsf{idle} == False$) and there is enough time to perform an update, P does so by sending m_5 to $\mathcal{PC}$. If the update does not go through, will force close the channel and run CallClaimFunc in the round t_1 in which $\mathcal{PC}$ responds unsuccessfully (case iii). Otherwise, if the update is successful in round t_2, P reads $\Gamma^P(\mathsf{id}_{\mathsf{ch}})$ from the $\mathcal{S}_{\mathcal{PC}}$'s storage and (after performing some steps) sends m_6, i.e., the proof, to $\mathcal{L}_D$ (case iv).

We now look at the execution ensembles for these different cases and note that running CallClaimFunc in a round t is captured as $\mathsf{obsSet}(\texttt{CallClaimFunc}, P, t)$ and we defer showing indistinguishability for this subprocedure to the next lemma.

- Case i: $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \emptyset$
- Case ii: $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \mathsf{obsSet}(\texttt{CallClaimFunc}, P, t_0)$
- Case iii: $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \mathsf{obsSet}(m_5, \mathcal{PC}, t_0) \cup \mathsf{obsSet}(\texttt{ForceClose}^P(\mathsf{id}_{\mathsf{ch}}), \mathcal{PC}, t_1) \cup \mathsf{obsSet}(\texttt{CallClaimFunc}, P, t_1)$
- Case iv: $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \mathsf{obsSet}(m_5, \mathcal{PC}, t_0) \cup \mathsf{obsSet}(m_6, \mathcal{PC}, t_2)$

1b. Ideal world: In the ideal world, $\mathcal{F}_{\mathsf{Alba}}$ performs exactly the same functionality as the protocol in the real world. This phase is (almost) not interactive, which makes it trivial to simulate. Upon m_4, functionality runs subprocedure `DetermineReceiver` on id. Similar to the real world, we distinguish some cases. If $\gamma.\mathsf{idle} = True$ (which means there has been a not-yet-resolved dispute in the payment channel), the subprocedure returns $\perp$ if $t_0 > T_0 - \Delta_D$, i.e., there is not enough time to claim the funds (case i) or `HandleClaim` if $t \leq T_0 - \Delta_D$ (case ii). `HandleClaim` is the ideal world pendant to `CallClaimFunc`. Also, even if $\gamma.\mathsf{idle} = False$, but time $t_0 > T_0 - t_{updateDelay} - \Delta_D$, the subprocedure runs and returns `HandleClaim` (same as case i). In this case, there is not enough time to perform an update, but still enough time to claim the funds. Similarly, even if $\gamma.\mathsf{idle} = False$, the functionality stops in case the time $t_0 > T_{updateDelay} - \Delta_D$ (same as case i). As in the real world, this corresponds to the scenario where there is not enough time to perform an update, but enough time to claim the funds with another message. In case that $t_0 \leq T_0 - \Delta_D$, the functionality runs `HandleClaim` (case ii). In case there is enough time for an update and $(\gamma.\mathsf{idle} == False)$, the functionality sends m_5 to $\mathcal{PC}$. If the update does not go through, i.e. $\mathcal{PC}$ responds unsuccessfully in round t_1, then the functionality will invoke $\texttt{ForceClose}(\mathsf{id}_{\mathsf{ch}})$ of $\mathcal{PC}$ and run `HandleClaim` (case iii). The main difference to the real world is in case the update was successful. Let t_2 be the round in which $\mathcal{PC}$ responds successfully, the simulator $\mathcal{S}$ reads $\Gamma^P(\mathsf{id}_{\mathsf{ch}})$ from the $\mathcal{S}_{\mathcal{PC}}$'s storage and (after performing some steps) sends m_6 to $\mathcal{L}_D$ (case iv).

- Case i: $\texttt{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}} := \emptyset$
- Case ii: $\texttt{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}} := \mathsf{obsSet}(\texttt{HandleClaim}, P, t_0)$
- Case iii: $\texttt{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}} := \mathsf{obsSet}(m_5, \mathcal{PC}, t_0) \cup \mathsf{obsSet}(\texttt{ForceClose}^P(\mathsf{id}_{\mathsf{ch}}), \mathcal{PC}, t_1) \cup \mathsf{obsSet}(\texttt{HandleClaim}, P, t_1)$
- Case iv: $\texttt{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}} := \mathsf{obsSet}(m_5, \mathcal{PC}, t_0) \cup \mathsf{obsSet}(m_6, \mathcal{PC}, t_2)$

2. V honest

As this phase does not require interaction from V, aside from the channel updates which are handled by the environment via $\mathcal{PC}$, the real and ideal world are trivially indistinguishable. □

Lemma 3. *The* subprocedure *phase of the protocol* Π *GUC-realizes the* subprocedure *phase of the functionality* $\mathcal{F}_{\mathsf{Alba}}$.

Proof. For convenience, we name the following messages:

- $m_7 := (\text{CallFunction}, function = \text{Claim}, args = \{[\overline{\texttt{Tx}_1^{\texttt{P}}}], \sigma_V([\overline{\texttt{TX}_1^{\texttt{P}}}]), [\overline{\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}}], R_P\})$
- $m_8 := (\text{CallFunction}, function = \text{FinalizeProof}, args = \{[\texttt{Tx}_{\texttt{unl}}^{\texttt{V}}], \sigma_P([\texttt{Tx}_{\texttt{unl}}^{\texttt{V}}])$
- $m_9 := (\text{CallFunction}, function = \text{Settle})$
- $m_{10} := (\text{CallFunction}, function = \text{RespondClaim}, args = \{\bar{r}_P\})$
- $m_{11} := (\text{CallFunction}, function = \text{RespondClaim}, args = \{[\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}], \sigma_V([\texttt{Tx}_{\texttt{unl}}^{\texttt{P}}])\})$

1. P honest, V dishonest

1a. Real world: Let t be the round in which this subprotocol is started. P sends m_7 to $\mathcal{L}_D$ and runs $\texttt{MonitorDestLedger}_P$. This subprocedure checks $\mathcal{L}_D$, reading the state of the contract every round. If the state is `Dispute_Resolved` in round t_1, i.e., V has responded correctly to the claim, then P will create the unlocked commitment transaction, update the storage of the payment channel simulator $\mathcal{S}_{\mathcal{PC}}$ accordingly and send it via message m_8 to the contract in $\mathcal{L}_D$. Finally, in round t_2, if the state of the contract is `Valid_Proof`, message m_9 is sent to the smart contract. This will settle the contract and distribute the funds according to the proven state. However, if by round T_2, the state has not changed yet to `Valid_Proof`, P will also send m_9, as this means that V has not replied to the claim, and P will get the money. In this dispute case, P will call $\texttt{ForceClose}^P(\mathsf{id}_{\mathsf{ch}})$ in $\mathcal{PC}$, after settling the contract. This yields the following execution ensembles, depending on the case.

- $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \mathsf{obsSet}(m_7, \mathcal{L}_D, t) \cup \mathsf{obsSet}(m_8\mathcal{L}_D, t_1) \cup \mathsf{obsSet}(m_9, \mathcal{L}_D, t_2)$
- $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \mathsf{obsSet}(m_7, \mathcal{L}_D, t) \cup \mathsf{obsSet}(m_9, \mathcal{L}_D, T_2)$

1b. Ideal world: In the ideal world, the functionality will at this point expect either a message (ClaimOk) or a message (NoResponse) from the simulator. In the first case, this means that V has resolved the claim successfully, which means the functionality will expect $\mathbf{TX}_{payout}$. In the second case, which means there was no response to the claim, the functionality will expect $\mathbf{TX}_P$.

Note that the functionality will notify the simulator with a message (Claim), and the simulator executes the exact same code as the protocol code, in the same round. The messages mentioned before (ClaimOk) and (NoResponse) are to notify the functionality of the outcome. It is thus easy to see, that the execution ensembles are the same. The execution ensembles are as follows, depending on the case.

- $\texttt{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}} := \mathsf{obsSet}(m_7, \mathcal{L}_D, t) \cup \mathsf{obsSet}(m_8\mathcal{L}_D, t_1) \cup \mathsf{obsSet}(m_9, \mathcal{L}_D, t_2)$
- $\texttt{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}} := \mathsf{obsSet}(m_7, \mathcal{L}_D, t) \cup \mathsf{obsSet}(m_9, \mathcal{L}_D, T_2)$, depending on the case.

Perhaps more interesting in this case is to show that there will never be an $(\texttt{ERROR})$ output by the functionality. We can see in the section **Atomicity** of the functionality, that an honest sender initiating a message `A–INITIATE–PAYMENT` containing a valid update and received in time (the cases that do not return $\perp$), can only result in the outcome $\mathbf{TX}_{payout}$ or $\mathbf{TX}_P$. In the real world, for this to happen, it would mean that the P makes a valid claim to the smart contract, but the verifier can convince the smart contract that it was invalid (via the RespondClaim function). However, since this requires the adversary to compute the preimage of a hash (which we model as random oracle) or forge a signature (we consider our signature scheme to be EUF-CMA secure), this cannot happen and thus there will be no error that is output.

2. V honest, P dishonest

2a. Real world: In the real world, the verifier V runs the subprocedure $\texttt{MonitorDestLedger}_V$ after the setup phase, which is then executed every round. If in round t the state of the contract is `Dispute`, either the dispute is invalid, in which case V responds by sending m_{10} to $\mathcal{L}_D$, followed by m_9 in round $t_1 \leq t + \Delta_D$ (after the state has changed to `Invalid_Claim`) (case i). Or, in case the claim was valid, V sends m_{11} to $\mathcal{L}_D$ in round t, and then once the state of the contract changes to `Valid_Proof`, updates the storage of the payment channel simulator $\mathcal{S}_{\mathcal{PC}}$ accordingly (case ii). In this dispute case, V will call $\texttt{ForceClose}^V(\mathsf{id}_{\mathsf{ch}})$ in $\mathcal{PC}$, after the contract is settled. Finally, if nothing happens by time T_2, V sends m_9 to $\mathcal{L}_D$ (case iii). This results in the following execution ensembles.

- Case i: $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \mathsf{obsSet}(m_{10}, \mathcal{L}_D, t) \cup \mathsf{obsSet}(m_9, \mathcal{L}_D, t_1)$
- Case ii: $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \mathsf{obsSet}(m_{11}, \mathcal{L}_D, t)$
- Case iii: $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \mathsf{obsSet}(m_9, \mathcal{L}_D, T_2)$

2b. Ideal world: In the ideal world, the simulator runs the exact same code after the setup phase. The simulator informs the functionality, in case an invalid claim is made, in which case the functionality expects $\mathbf{TX}_V$ to appear. Because the code is the same, however, it is trivial to see that the execution ensemble is the same, for the same cases.

- Case i: $\texttt{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}} := \mathsf{obsSet}(m_{10}, \mathcal{L}_D, t) \cup \mathsf{obsSet}(m_9, \mathcal{L}_D, t_1)$
- Case ii: $\texttt{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}} := \mathsf{obsSet}(m_{11}, \mathcal{L}_D, t)$
- Case iii: $\texttt{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}} := \mathsf{obsSet}(m_9, \mathcal{L}_D, T_2)$

□

Theorem 5. *The protocol* Π *GUC-realizes the functionality* $\mathcal{F}_{\mathsf{Alba}}$.

Proof. This follows directly from Lemmas 1 to 3. □

APPENDIX E
GAME THEORETIC ANALYSIS

Our UC-based analysis shows that our protocol is secure in the presence of at least one honest party. We now shift our model to incorporate selfish players who may deviate from the correct protocol execution if they are to gain from it. To that end, we conduct a game-theoretic analysis assuming all participants are rational utility-maximizing agents. This rational model aims to capture better the behaviors of participants who generally do not engage in irrational decisions that will cost them money (Byzantine adversaries) nor blindly follow a protocol when they are aware of a more profitable strategy (honest parties assumption).

Perfect Information Extensive Form Game. We leverage the sequential nature of the decision-making process of our protocol and employ *extensive form games* [57], [55] to methodically define the set of players in the game, their set of possible actions, and their respective payoffs. Every participant in our protocol is fully aware of the game's structure, including the payoffs corresponding to each outcome – a characteristic of perfect information. When faced with a decision, each player is perfectly informed about all preceding events. With this approach, we capture all the different strategies players can adopt at any point in time and are thereby able to determine which one yields the largest payout.

We simplify our analysis to the interactions within Alba's smart contract, given that the channel cannot be closed while the Alba instance remains active on the contract. This constraint allows us to infer the rational actions parties may take in the payment channel based on the smart contract execution. Specifically, deviations from correct protocol execution on the payment channel side are limited to two scenarios: (i) a party refuses to participate in updates, or (ii) a party stops cooperating during an initiated update. In case (i), if a dispute arises on the contract, it can be resolved; even if initiated by a malicious party, the game's correct outcome is enforceable across both the channel and contract, ensuring cross-ledger atomicity of operations and fulfilling Definition 2. In case (ii), the correct outcome is also enforceable, whether by obtaining the unlocked transaction either from reading the proof from the contract or by opening a dispute. The key distinction in the execution of the protocol in the presence of a dishonest party is that the channel gets closed.

Game Tree. In extensive form games, all possible executions of a protocol can be represented using a tree, whose terminal nodes, or leaves, are tuples $(\mathsf{coins}_P, \mathsf{coins}_V)$ holding the payoffs for each player (in our case (P, V)), at the end of the protocol. Non-terminal nodes, instead, are tuples (Party, TimeCondition) indicating who can take action in the protocol, and at which intermediate stage of the protocol. Submitting invalid data to the contract translates into going idle, i.e., state variables of the contract do not change until valid inputs are provided.

Let us now consider the case where P and V have a payment channel on the LN, and they lock in a smart contract c_P, $c_V \geq 0$ coins, respectively. They update their channel from its current state s_1 to a new state $s_2 = s_1 + \Delta s$, and then, before time T, P proves to the smart contract that the state transition Δs was applied to the channel. As a result, the coins locked in the contract are distributed between P and V accordingly. Should no proof be submitted to the contract, and no dispute brought forth prior to T, the collateral will be returned to the parties in the same proportion as their initial contributions Every state has a monetary mapping for P and V. We denote with $u(s_i) := (u_P(s_i), u_V(s_i))$ the utility function of parties when the payment channel is in state s_i, and with $f(s_i) := (f_P(s_i), f_V(s_i))$ the utility function of parties within the smart contract upon the channel being in state s_i. We also denote with s_2' the state of the channel after applying Δs to an old, revoked state s_r of the channel, i.e., $s_2' = s_r + \Delta s$.

Finally, we observe that the dispute mechanism in our Alba protocol results in keeping coins locked in the contract for a longer time, yielding some hidden costs y: an opportunity cost for having money locked in the contract for a longer time, and higher transaction fees for the dispute mechanism.

Figure 4 depicts the game tree of our smart contract.

Subgame Perfect Nash Equilibrium (SPNE). In game theory, a *player's strategy* refers to the actions they choose to take in a game, out of all the possible actions. A *strategy profile*, fully specifies all actions in a game from all its players, i.e., it is the set of chosen strategies of all players. A strategy profile is a *subgame perfect Nash equilibrium* (SPNE) if it represents a Nash equilibrium of every subgame in the game; subgame being any subset of the game starting from one initial node. In other words, *an SPNE is the strategy profile from which active, utility-maximizing parties do not deviate at any point in the game, regardless of what happened before*.

To identify the SPNE in our game, we apply *backward induction* to the tree of Figure 4: Starting from terminal nodes going upwards toward the root of the tree, we select the action with the highest payoff for the decision-making player, based on the future decisions that are already determined (as they are lower in the depth of the tree).

We recall that any party in the protocol can interact with the contract, and whoever does it first (by either submitting a proof or opening a dispute) takes the role of prover P; the other party takes the role of verifier V. We observe that for at least one party in the protocol it is more profitable to submit a proof, rather than to go idle. In other words, for at least one party the following holds: $c_P < (f_P + u_P)(s_2)$. We call this party *prover* P.

We show that our protocol has one SPNE, corresponding to the case in which rational P and V cooperate to submit a valid proof to the smart contract (action (i) in the tree). In Theorem 6, we prove that under active utility-maximizing agents, a valid proof will be submitted to the smart contract.

Theorem 6. *Consider the tree in Figure 4 reflecting the game between two parties* (P, V). *For* at least one of two parties*, say* P*, the following always holds:* $c_P < (f_P + u_P)(s_2)$. *The following strategy profile* Σ *is the SPNE of the game:*

$$\Sigma(P, V) = \{[\textit{(i)}]_P, [\textit{(v)}, \textit{(viii)}]_V\}.$$

Proof. By backward induction, let us consider terminal nodes of the game tree in Figure 4.

We start by looking at the terminal nodes at the bottom of the tree resulting from V's action, i.e., actions *(v)*, *(vi)*, *(vii)*, *(viii)*, and *(ix)*. V will choose action *(v)* over *(vi)*, as it gives him a higher payoff. Similarly, V will choose action *(viii)* over *(vii)* and *(ix)*. While it is straightforward to see that *(viii)* gives V a higher payoff than *(ix)*, it is not trivial to argue about *(vii)* and *(viii)*. In *(vii)*, V is at loss: since parties are rational, if P opens a dispute claiming the channel is in an old, revoked state s_r, it means that s_2' gives P a higher payoff, taking away some of V's money. Thus, V chooses *(viii)* over *(vii)*, thus reclaiming the maximum he can get, i.e., all the money in the contract, $(c_P + c_V)$, minus some opportunity costs y. It follows that V's strategy in the SPNE is to take either action *(v)* or *(viii)*.

Consider now P's possible actions, i.e., *(i)*, *(ii)*, *(iii)*, or *(iv)*. Given that V will choose *(v)* or *(viii)*, and given that $c_P < (f_P + u_P)(s_2)$ holds, the strategy yielding the highest payoff for P is *(i)*. □

APPENDIX F
ARTIFACT APPENDIX

We introduce the bridge Alba (based on our novel Pay2Chain concept), a new 2-party protocol that enables interoperability between layer-1 and layer-2 of different blockchains by allowing verification and enforcement of an update of a payment channel on a smart contract. Given the mutual distrust between the two parties, Alba leverages an on-chain smart contract to verify the state of the channel and issue a corresponding transaction on the blockchain, but also to settle any dispute that may arise between the users, thereby enforcing on-chain the outcome matching the relevant event in the payment channel.

We claim that the Alba smart contract can be *efficiently* used by users of any Ethereum-like blockchains. While in the optimistic case, i.e., when users follow the protocol, the fees users incur are 2 times lower than the ones of other standard bridges the costs of Alba are on par with other standard bridges in the pessimitic case. *Our Alba smart contract implementation aims to evaluate the costs of calls to Alba functions*, both in an optimistic and in a pessimistic execution.

A. Description & Requirements

1) How to access: The artifact is available at this Digital Object Identifier (DOI): 10.5281/zenodo.14249988.

2) Hardware dependencies: The artifact can be evaluated on commodity hardware.

3) Software dependencies: The evaluation of the artifact requires to have Docker installed.

4) Benchmarks: None.

B. Artifact Installation & Configuration

To have a consistent environment that runs all components without requiring additional configuration or dependencies, please ensure that Docker is installed on your commodity laptop. You can download Docker from here.

C. Major Claims

In the blockchain world, any on-chain action requires issuing transactions. By design, validators and miners of the blockchains are motivated to include transactions in blocks because they are compensated by transaction fees. In Ethereum, transaction fees are measured in *gas* units, that provide a cost for the on-chain storage and for the amount of computational work. Our artifact measures the gas cost incurred by users when interacting with the Alba contract. In the paper, we make the following claims:

- C1: The Alba-specific logic can be implemented in an on-chain smart contract *efficiently*, with a call to each function incurring the costs presented in Table II of the paper. To ease the evaluation of the artifact, we copy-paste such table in Table I of this document.
- C2: In the optimistic case, the on-chain costs of Alba are lower than the ones of standard bridges such as zkBridge (230k gas), Glimpse (290k gas), and SPV-based bridges (205k gas). In the pessimistic case, the costs of Alba are on-par with the ones of the aforementioned bridges, if not lower, as zk techniques require very high off-chain computational costs, and SPV necessitates to relay and verify all block headers, thus incurring continuous high maintenance costs.

	Gas Cost
Setup	393401
SubmitProof	253566
OptimisticProof	48027
Dispute	515860
ResolveInvalidDispute	168046
ResolveValidDispute	37333
Settle	49814

TABLE I
ON-CHAIN GAS COSTS EVALUATION FOR EVM-BASED BLOCKCHAINS.

D. Evaluation

The Alba contract internally verifies the well-formedness and the correctness of Lightning Network transactions. Therefore, we created some of these transactions, with the purpose of emulating both an optimistic and pessimistic execution of Alba, as well as testing it. The Lightning Network transactions have been created by using the python-bitcoin-utils library, and they are provided in Alba-Bridge/data/jsonTestData.json.

We implemented the on-chain component of Alba in Solidity and evaluated its gas costs using Hardhat, a development environment designed for building, debugging, and testing Ethereum-based smart contracts. The gas computed by Hardhat faithfully mimics the gas in Ethereum, as one can see from comparing the costs in Hardhat/revm with, e.g., the ones in the Ethereum yellowpaper and in the Go implementation of Ethereum. We use the hardhat-gas-reporter plugin to obtain detailed gas usage reports for function calls to Ethereum smart contracts.

1) Experiment: [Gas costs evaluation] [approx 2 minutes]: this experiment computes the gas consumed by calls to each Alba function.

[How to] Once Docker is installed and the Alba repository is downloaded from here, open a terminal and navigate to the project folder by typing:

```
cd /path/to/the/cloned/repository
```

[Preparation] Build the Docker image by typing the following command within the Alba-Bridge folder:

```
docker build --no-cache -t alba .
```

[Execution] Run the gas cost evaluation of Alba within Docker by executing the following command in your terminal:

```
docker run alba
```

[Results] The experiment outputs a table similar to the one in Fig. 1 showcasing a gas report per call to the Alba function. In our evaluation, we are interested in the average costs (Avg column). For a quantitative comparison, we recall that the simplest transaction in Ethereum requires 21k gas.

Assessing claim C1: The on-chain costs of each Alba function (Fig. 1) correspond to the ones claimed in the paper (Table I).

Assessing claim C2: In the optimistic case, Alba is more cost effective than the standard bridges mentioned in Section F-C: the OptimisticProof function combined with the Settle function require approx 100k gas, while other bridges approx 250k gas.

In the pessimistic case, Alba requires one party to call the Dispute function and the counter party to close it by either calling ResolveInvalidDispute or ResolveValidDispute. While opening a dispute is quite costly (500k gas), closing it is rather cheap (168k gas or 37k gas). However, we note that these are one-time costs, while other bridges have to bear additional computational or maintenance costs. We further note that a pessimistic execution never takes place in the presence of rational users, i.e., users maximizing their profit, as motivated in the paper in Section V.C.

The function SubmitProof and its associated costs showcase how expensive it is to verify on-chain on Ethereum the correctness of a Lightning Network transaction; this function however, is superseded by OptimisticProof.

Fig. 1. Per-function gas report of Alba.

Methods

Contract	Method	Min	Max	Avg	# calls	chf (avg)
ALBA	dispute	38360	579527	515860	17	–
ALBA	optimisticSubmitProof	46286	49188	48027	5	–
ALBA	resolveInvalidDispute	35254	38165	37333	7	–
ALBA	resolveValidDispute	36030	234054	168046	6	–
ALBA	settle	39979	64979	49814	9	–
ALBA	setup	393399	393445	393401	39	–
ALBA	submitProof	38239	397118	253566	5	–

Deployments

			% of limit	
ALBA	–	–	4610531	15.4 %	–
ParseBitcoinRawTx	–	–	2486068	8.3 %	–