Alba: The Dawn of Scalable Bridges for Blockchains

Abstract—**Over the past decade, cryptocurrencies have received attention from academia and industry alike, which has led to a diverse blockchain ecosystem and novel applications. The inception of bridge protocols has enabled enhanced interoperability, allowing assets to be transferred between distinct blockchain networks, thus capitalizing on their unique features. Despite the surge in popularity and the emergence of Decentralized Finance (DeFi), blockchains continue to face challenges with low throughput and scalability. Various off-chain solutions, such as payment channels, state channels, and rollups, have been proposed to address these issues. However, these solutions lack interoperability both among themselves and with other blockchains. On the other hand, existing bridges are incompatible with off-chain solutions and depend heavily on on-chain transactions, resulting in poor scalability.**

This paper introduces the concept of *scalable bridges* as a solution to these limitations. We focus primarily on their application in payment channels, a prominent and effective decentralized off-chain protocol. Our proposed scalable 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. We demonstrate how Alba enables new applications in *DeFi*, *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. Our empirical evaluation of Alba demonstrates its compatibility and efficiency in terms of communication complexity and on-chain costs, with its optimistic case incurring only twice as much as the cheapest Ethereum transaction.

1. Introduction

Fourteen years have passed since the mining of Bitcoin's genesis block [1] and, while Bitcoin is still the first blockchain by market cap, many others have emerged attracting users due to their complementary design goals in terms of privacy (e.g., Monero and ZCash), high throughput (e.g., Algorand), DeFi support (e.g., Ethereum), and technical features such as their scripting language, consensus protocol, and cryptographic mechanisms. As a result, several bridging solutions have been proposed to enable cross-chain applications like atomic swaps, cross-chain lending, and many more. In general, bridges can be divided into two categories: (i) centralized solutions based on trusted exchanges, which have often led to catastrophic financial losses due to hacks [2], [3], [4], fraudulent activities or bankruptcy [5]; (ii) decentralized solutions based on cryptographic protocols or smart contracts. Examples of these are atomic swaps [6], [7], [8], [9], which are efficient but do not go beyond token swaps, light clients implemented in smart contracts [10], [11], which are expressive but also expensive, and more recently, optimized protocols like Glimpse [12], achieving efficiency and expressiveness at the same time, while also retaining compatibility with blockchains featuring limited scripting languages.

All existing bridging solutions, however, share a fundamental limitation: they require transactions to be posted on-chain, which hinders scalability, besides causing transaction fees. This stays at odds with the recent developments in the blockchain landscape that focus on the design of Layer-2 (L2) solutions to minimize the number of on-chain transactions and thus enhance scalability. For instance, the Lightning Network [13] (LN), the most widely deployed scalability solution in Bitcoin with more than \$200M total value locked, enables parties to transact with each other securely without publishing any information on Bitcoin's underlying Layer-1 (L1) chain. Nevertheless, *existing bridges do not support the LN, nor any other off-chain protocol*: in other words, it is currently not possible to prove on a target chain $\mathcal{L}_D$ that an off-chain transaction occurred on any L2 network, regardless of the L2 solution being a payment channel, a state channel, or a rollup.

It is an open question whether or not such *scalable bridge* protocols 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 an off-chain bridge should account for the subtle game-theoretic arguments underlying the security of off-chain protocols.

Regardless of these challenges, conditioning the execution between off-chain and on-chain transactions would finally unleash the full potential of off-chain solutions, removing one more reason for transactions to be on-chain and paving the way for interoperability between L2-L1, as well as between L2-L2. Users would ultimately be able to use some DeFi protocols off-chain, transact tokens off-chain, and optimistically offload expensive computation off-chain. The L1 would then only be used as a settlement layer, thereby further relieving blockchains from congestion and users from expensive fees, all the while enhancing throughput and boosting the application ecosystem.

While rollups are gaining traction and popularity as a viable off-chain solution, they are still in their infancy, centralized [14], and only compatible with quasi-Turing complete chains. Rollups are also still dealing with the data availability problem, currently solved by periodically storing *on-chain* the state of the rollup along with a compressed history of all the transactions executed off-chain [15], [16], [17]. This results in rollups effectively offloading computation but not significantly scaling in terms of on-chain storage. For all these reasons, in this work, we focus on payment channels, the most well-established, decentralized off-chain solution, effectively offloading storage and computation. Furthermore, they are virtually compatible with all sorts of chains [18], [19].

Our Contributions. In this work, we present Alba, the first bridge that supports off-chain transactions. At its core, Alba encodes in a contract on the target chain $\mathcal{L}_D$, e.g., Ethereum, the logic to verify that a specific off-chain transaction occurred in a payment channel network, i.e., on a L2 solution on top of a source chain $\mathcal{L}_S$ such as Bitcoin.

This way, Alba enables, for the first time, a bridge that verifies transactions that are kept off-chain while introducing several important advantages compared to existing on-chain bridges. First, in the spirit of designing trustless solutions, light clients are not necessary anymore, as the information relayed from L2 to $\mathcal{L}_D$ is now agnostic to the consensus of the underlying blockchain $\mathcal{L}_S$. This, in particular, reduces the computational and storage overhead of existing light client-based bridges, since the consensus of $\mathcal{L}_S$ does not need to be verified within a smart contract on $\mathcal{L}_D$. Moreover, bridging off-chain transactions means that one can inherit important security and efficiency properties from the L2 protocol, e.g., instant finality and compatibility as in the case of payment channels.[1] Lastly, since miners (or validators) do not play any role in executing off-chain transactions, such a bridge is secure against temporary attacks on the liveness of the source blockchain $\mathcal{L}_S$, and against bribery attacks where miners (or validators) are bribed with the Total Value Locked (TVL) in the bridge to forge fake proofs of transaction inclusion. These two attacks are particularly dangerous and have been carried out in the cross-chain MEV domain [24], or in any protocol relying on timelocks [25]. We summarize the advantages of Alba with respect to existing light client and bridge solutions in Figure 1.

Besides, Alba introduces compelling new classes of applications that improve the scalability, interoperability, and enhance functionalities of both the payment channel and the destination chain it operates between. For instance, Alba enables *DeFi applications* between L2-L1, such as lending protocols. This relieves the underlying blockchain from dealing with a high volume of transactions competing for limited block space [26], but also reduces user fees. Furthermore, Alba enhances the *functionalities of existing payment channels* by serving as a trustless protocol for bringing backed assets into, for example, the LN. This has two benefits: Ethereum users can leverage the LN and transact off-chain virtual representations of their tokens, also known as *wrapped tokens*, thus reducing fees and forgoing storage costs. Finally, Alba optimistically brings *arbitrary computation to scriptless blockchains*: users can perform arbitrary computations off-chain, store the result into their L2 channel, while security and correctness of the result can always be enforced on the target L1 ($\mathcal{L}_D$).

Our contributions are summarized as follows:

- We present the key building blocks and an overview of Alba protocol (Section 2), the first scalable bridge supporting verification of off-chain transactions.
- We identify three novel off-chain protocol classes enabled by Alba (Section 3) and we demonstrate how they improve the state-of-the-art of both the payment channel (L2) and the destination chain (L1) they operate between. In particular, we describe how to use Alba for DeFi applications, trustless backed assets in payment channels, and optimistically offload arbitrary computation to the L2 while still enforcing the correctness of the result.
- We give an instantiation of Alba between LN and Ethereum (Section 4), describing the rationale behind our design choices.
- We prove the security of Alba in the UC framework [27] (Section 5.1) showing that balance security is guaranteed for honest users even against byzantine adversaries. We then prove via a game-theoretic analysis that rational parties follow the honest protocol flow of Alba (Section 5.2).
- We conduct a performance evaluation (Section 6), characterizing Alba's communication complexity, its on-chain costs on EVM-based chains, and demonstrate its practicality: in the optimistic case, Alba only consumes twice as much as the simplest Ethereum transaction.

2. Key Building Blocks & Protocol Overview

Payment Channels. Payment channels are, so far, the most well-established, decentralized, and effective solution to the scalability problem of blockchains. In addition, payment channels are virtually compatible with all existing chains, even ones that have limited scripting capabilities, like Bitcoin and Monero [18]. Users transacting via payment channels can carry out an arbitrary number of off-chain payments and only publish two on-chain transactions, one to open the channel and one to close it. Abstractly, as shown in Figure 3, a payment channel consists of three phases: open, update, and close.

First, in the *opening* phase, two users post a single, funding transaction $\mathtt{Tx_f}$ on the blockchain, where they jointly lock up some coins in a shared output that can only be spent if both users agree (i.e., sign a transaction that spends the shared output). Before posting $\mathtt{Tx_f}$ on-chain,

1. In payment channels, honest parties do not lose money even in the presence of Byzantine adversaries. Instant finality is guaranteed in the rational setting by parties being live in the protocol.

	LC [10], [20]	SLC [21], [22]	zkBridge [23]	Glimpse [12]	**Alba**
Support for off-chain Tx:	No	No	No	No	Yes
Information relayed wrt $\mathcal{L}_S$ length:	Linear	Polylog	Linear	Constant	Constant
Storage overhead wrt $\mathcal{L}_S$ length:	Linear	Polylog	Constant	Constant	Constant
Instant finality:	No	No	No	No	Yes
Backward compatibility with $\mathcal{L}_S$:	Yes	No	Yes	Yes	Yes
Secure against liveness attacks on $\mathcal{L}_S$:	No	No	No	No	Yes
Unbounded TVL even if miners are rational:	No	No	No	No	Yes

Figure 1: Comparison of Alba to other SPV-based light clients (LC), super-light clients (SLC) such as NiPoPoW and FlyClient, and bridge protocols like zkBridge and Glimpse.

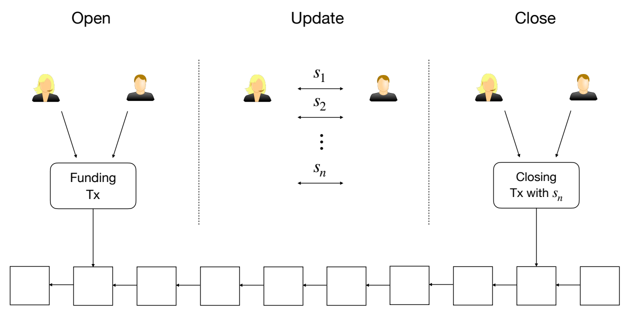


Figure 2: The three phases of a payment channel protocol.

however, the two users exchange signed transactions that spend the shared output of $\mathtt{Tx_f}$ and return to each user their initial funds. This way, users ensure their counterparty cannot hold their coins hostage in the channel. After $\mathtt{Tx_f}$ is published on-chain, the *update* phase begins: Users may now perform arbitrarily many payments by exchanging signed transactions that spend the shared output of $\mathtt{Tx_f}$ and assign to each user a new balance distribution. These transactions are referred to as channel *states* or *commitment transactions*. Finally, in the *closing* phase, users can close the channel by posting on-chain the last state of the channel either in collaboration or unilaterally. In the former case, users can claim their balances immediately. If the counterparty does not react, a user can close the channel unilaterally by publishing a signed commitment transaction, which incurs a time delay, the so-called dispute period. In the LN, the dispute period enables the counterparty to claim the total capacity of the channel in case the published state is not the latest one agreed in the channel.

Bridges. At its core, a bridge is a protocol that facilitates communication between different blockchains by transferring information and assets. To do so, a bridge protocol must enable users to condition the execution of a transaction on a destination chain $\mathcal{L}_D$ to the execution of a transaction on a source chain $\mathcal{L}_S$. An ideal bridge is as secure as the least secure of the two blockchains it bridges. There are two key challenges when designing robust bridges: (i) make $\mathcal{L}_D$ aware of transactions permanently included on $\mathcal{L}_S$, by reliably and efficiently transferring state information from $\mathcal{L}_S$ to $\mathcal{L}_D$; (ii) execute transactions atomically, i.e., transactions to be published on $\mathcal{L}_S$ and $\mathcal{L}_D$ are either both included in their respective chains, or neither of them is, thus ensuring the security and consistency of the bridge.

Existing bridges address these two challenges in different ways. For instance, (i) can be solved by trusting central authorities such as exchanges, by relying on a committee of validators [28], [29], or by forgoing any trust assumption by using light clients [22], [30], [31], [32], [33], [34]. We observe that the most secure bridges employ light client functionalities within smart contracts, where each bridged chain follows the other chain's consensus. As for (ii), atomicity is guaranteed either cryptographically via hash locks [7], [9] and adaptor signatures [19], or programmatically, as in the case of chain relays [10], [11], [35].

Regardless of their specific techniques, existing bridges require transactions to be posted on the blockchain. Exchanges, validators, or light clients need a footprint on $\mathcal{L}_S$ proving that the chain's consensus executed a particular transaction, i.e., it irrevocably updated the state of the blockchain. Such evidence is necessary to convince $\mathcal{L}_D$ to update its state accordingly. The need for transactions to be on-chain has become a major limiting factor in a multi-chain world, where cross-chain protocols flourish thanks to novel chains and applications offering different, appealing features to users. By requiring transactions to be on-chain, bridges hinder not only scalability but also interoperability between L2-L1 and L2-L2. Time has come to have *scalable bridges*, which unlock interoperability not only between chains, but also between layers. Off-chain solutions hold a significant, increasing share of the total value locked in blockchains, approximately \$25B as of December 2023: bridging off-chain transactions would enable a plethora of *more scalable* applications, increasing the volume of transaction processed while relieving users from high fees.

Advantages of Scalable Bridges. This new paradigm of *scalable bridges* presents new, remarkable features in terms of performance and security. Since transactions are kept off-chain, trustless and scalable bridges do not have to deal with the consensus of the underlying source chain, which is hard to verify in resource-constrained environments such as blockchains. Instead, one can leverage the more lightweight transaction execution on L2 solutions, thus minimizing the amount of data to be transferred, verified, and stored on $\mathcal{L}_D$.

Moreover, scalable bridges are secure against those attacks specifically targeting blockchains. Examples of these are censorship attacks, where miners (or validators) are bribed to censor transactions temporarily [12], [25], and block forgery attacks, where bribed miners (or validators) deviate from the correct consensus protocol and, e.g., equivocate signing some multiple blocks at the same height (PoS protocols), or mine fake or orphaned blocks (PoW protocols). Since the money held by the bridge is the one used for the bribe, bridges usually impose an upper bound on their TVL so that security holds [12]. Scalable bridges would be inherently safe against these two kinds of attacks, thus removing the strict limit on the TVL.

Furthermore, scalable bridges can inherit beneficial properties from the specific L2 considered. For instance, a protocol bridging the LN inherits instant finality guarantees, as opposed to standard bridges that have to wait hours for transactions to be considered final on Bitcoin. Payment channels guarantee balance security: honest parties can always enforce the latest state or, alternatively, redeem the full capacity of the channel, even in the presence of Byzantine adversaries. This yields, in turn, instant finality guarantees in the rational setting.

Lastly, some off-chain solutions can be built relying on a minimal set of assumptions. For instance, payment channels are virtually compatible with all existing blockchains, as long as they support transaction authorization, i.e., a signature scheme [18], [19].

Challenges of Scalable Bridges. Designing bridges that condition the execution of an on-chain transaction on the execution of an off-chain transaction is far from trivial, with specific challenges varying with the particular off-chain solution considered. Let us look at payment channels again: first, the protocol must be opt-in and flexible, i.e., it should be executed on top of already opened channels, unlike standard bridges that require users to close the channel and go on-chain.

Second, the protocol must protect honest users from losing money: any desired channel update must always be enforced atomically, even in the presence of Byzantine adversaries.

Third, while users can issue on-chain transactions independently, payment channel updates require both parties' cooperation. This is a subtle distinction with important consequences. The protocol must protect users from the case where one party refuses to complete the update upon receiving the channel update from the counterparty and withholds some data. This puts the malicious party in a condition of advantage, unilaterally holding a valid update and being able to trigger the conditional transaction on $\mathcal{L}_D$. On the other hand, the honest party neither holds a valid update, nor has the chance to go on $\mathcal{L}_D$. This can be thought of as a data availability problem.

Alba: A Novel, Scalable Bridge. We propose Alba, a new, *scalable bridge* which enables users to *efficiently*, *securely*, and *trustlessly* condition the execution of a transaction in the destination chain $\mathcal{L}_D$ on a specific transaction occurring in the payment channel.

As depicted in Figure 3, Alba comprises three main phases: (1) users set up a contract on $\mathcal{L}_D$, lock some coins in it, and condition the execution of a transaction to a specific update of the channel. Then, (2) users perform arbitrarily many channel updates, until reaching the update

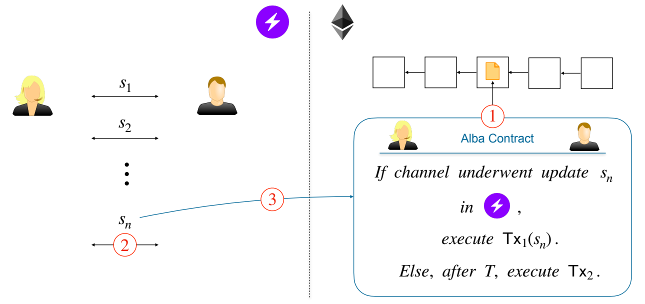

s_1
s_2
s_n
2
3
1
Alba Contract
If channel underwent update s_n
in ,
execute $\mathsf{Tx}_1(s_n)$.
Else, after T, execute Tx_2.

Figure 3: Abstractly, the flow of our Alba protocol, with a LN payment channel on the left and Ethereum on the right.

that conditions the transaction on $\mathcal{L}_D$. Finally, (3) a user relays the update to the contract on $\mathcal{L}_D$, which verifies its validity and triggers the corresponding transaction.

To achieve the same security properties of payment channels, the smart contract implements a punishing mechanism that ensures that honest parties will not lose money (*balance security*) against any Byzantine adversary. As we will see in Section 5.2, when all participants in the protocol want to maximize their profit, i.e., they are *rational agents*, their best strategy is to follow the protocol.

Alba composes with existing payment channel solutions and can be used opt-in: it only requires one additional round of communication between parties, which can take place either on the payment channel system itself or it can be executed concurrently.

Users of Alba directly relay information about the state of their channel to $\mathcal{L}_D$, by disclosing commitment transactions created *ad hoc*. This solves the aforementioned data availability challenge, while retaining efficiency, as only two transactions are relayed, verified, and stored on $\mathcal{L}_D$. These two transactions suffice: payment channel users run a two-party full consensus or, in other words, a two-party Byzantine fault safe state machine replication protocol [36], with instant finality guarantees when all parties are rational. This allows to efficiently verify the state of the channel, as opposed to standard, on-chain bridges, which have to relay and verify the state of a, e.g., permissionless, dynamically available, longest chain protocol with probabilistic finality guarantees, or a BFT-like protocol with multiple nodes.

3. Applications of Alba

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

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. For now, we avoid discussing the dispute mechanism that protects honest parties from Byzantine adversaries: We analyze this particular case in detail in Section 4.

3.1. 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 an overwhelming amount of transactions being broadcast to the network and competing for a limited block space [26]. Shifting the DeFi ecosystem partially off-chain would benefit users and the security of the chain itself threatened by MEV [37]. 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*.

In many cases, users take loans because they want to invest the money in profitable trades and have some gain. By allowing (collateralized) loans to be granted off-chain, users can move these trades off-chain and pay back the loan off-chain as well, thus minimizing the number of on-chain transactions. Furthermore, the collateral in the contract could be used for *flash loans*, thus mitigating opportunity costs.

Setting Up the Lending. Assume Alice and Bob are both on Ethereum and have a payment channel on the LN. Alice wishes to borrow 5 BTC on Lightning from Bob to participate in some online games [38]. 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 designed so that Alice can get back her collateral only if she returns the loan to Bob within a dedicated channel update. If she defaults on the loan, Bob can keep the collateral after a certain time.

Granting the Loan. Upon Alice locking her collateral in the contract, Alice and Bob prepare and finalize a channel update where Bob lends 5 BTC to Alice. If such an update aborts, Alice can get back her collateral. Now, Alice is free to use the 5 BTC she borrowed from Bob in the way she desires.

Paying Back the Loan. Finally, when Alice pays back the loan to Bob, she can unlock the collateral by submitting the commitment transactions of the payback to the smart contract. If she defaults on paying back the loan, after a certain time, Bob keeps the collateral.

3.2. Multi-Asset Payment Channels

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

Recent developments such as Taproot Assets [39] and RGB [40] 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 or 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 issue and transfer on Bitcoin and on the LN *wrapped tokens*, e.g., token representations of existing tokens on Ethereum, exchange them back and forth as many times as they want, and then 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 token, while avoiding on-chain fees.

Issuing Non-Native Tokens on the LN. Consider two users, Alice and Bob, who have a channel on the LN and wish to top it up with some, e.g., 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 and, for instance, naively representing WETH in their channel within a dedicated output[2] in their commitment transactions. More sophisticated solutions also exist [39], [40]. At this point, Alice and Bob can transact WETH 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 the latest state of their channel and call the contract function that pays them out according to the WETH distribution in the channel.

Payment Channel Networks. We conjecture that in case WETH are used in multiple channels of the LN, they can be sent across these channels via intermediaries. Otherwise, if WETH exists in a payment A-B-C in channel A-B but not in B-C, B can offer an ad-hoc exchange of ETH to BTC. We leave it to future work to explore this extension in more detail.

3.3. 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 [41], [42], trusted executioners, or honest majority of a quorum [43]. 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 a LN channel, while in case of misbehavior, the correct execution enforced by the smart contract on another chain.

Playing Chess on LN. A simple but nevertheless interesting example of stateful computation is the game of chess, where the state of the game (position of all non-captured pieces

2. For instance, it can be used an `OP_RETURN` output, i.e., a Bitcoin script opcode that marks a transaction output as unspendable and can be used to embed up to 80 bytes in a transaction.

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 $\geq$ 10, inbound capacity $\geq$ 5) and some collateral $\geq$ 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 [44]. 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 smart contract 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 winner of its coins in LN, the winner can still cash in in Ethereum via the smart contract.

4. Protocol Design

In this section, we introduce the system model and assumptions and we give an overview of channel updates on the LN. Then, we present the construction of Alba, motivating its design choices and instantiating Alba for the DeFi lending protocol described in Section 3.1.

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.

System Model and Assumptions. 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.

Channel Updates on the LN. From Section 2, we recall that payment channels composes of three phases: open, update, and close. In particular, on the LN, a channel update consists of the following steps: first, parties exchange commitments to their new revocation keys, i.e., Bob gives $R_B := \mathcal{H}(r_B)$ to Alice and Alice gives $R_A := \mathcal{H}(r_A)$ to Bob; second, parties exchange signed commitment transactions, i.e., Bob gives Alice $\mathtt{Tx}_1^\mathtt{A}$ with $\sigma_B(\mathtt{Tx}_1^\mathtt{A})$ and Alice gives to Bob $\mathtt{Tx}_1^\mathtt{B}$ with $\sigma_A(\mathtt{Tx}_1^\mathtt{B})$; third, they exchange their old revocation keys, i.e., Bob gives r_B to Alice and Alice gives r_A to Bob.

For each update $\mathtt{Tx}_1^\mathtt{A}$ and $\mathtt{Tx}_1^\mathtt{B}$ are asymmetric [13], [45]. $\mathtt{Tx}^\mathtt{A}$ gives Alice the right to redeem her coins only after a timeout, while Bob can redeem his coins immediately. Similarly, $\mathtt{Tx}^\mathtt{B}$ gives Bob the right to redeem his coins only after a timeout, while Alice can redeem her coins immediately. The timeout on $\mathtt{Tx}_1^\mathtt{A}$ ($\mathtt{Tx}_1^\mathtt{B}$) is a protection mechanism for Bob (Alice): if Alice (Bob) cheats and publishes on-chain an old, revoked state of the channel, i.e., $\overline{\mathtt{Tx}}_1^{\mathtt{A}}$, Bob (Alice) can steal Alice's (Bob's) coins by revealing $\bar{r}_A$ ($\bar{r}_B$, i.e., the old revocation secret of Alice (Bob) for that transaction. In this way, LN channels guarantee that honest users are always able to enforce on-chain either the latest state of the channel, or a state where they get all the money.

A Naive Construction. We recall from Section 3.1 that Bob is willing 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 time T, 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.

In a preliminary setup phase, Alice and Bob make the contract aware of their channel by storing information on the funding transaction $\mathtt{Tx_f}$, of T, and of some other protocol-specific information, as shown in Figure 4.

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 key 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 (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. In Bitcoin, and therefore in the LN as well, such an output θ would look as follows: $\theta = (0, \mathtt{OP\_RETURN}(\mathsf{id}, R_B))$.

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

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, un-revoked* state. In normal channel operations, the latest state of the channel is an un-revoked state. However, due to the fair exchange problem, during channel updates, one party will receive the counter party's revocation key before revealing its own. 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. We notice that, on the other hand, it is fine if a cheating party goes on-chain and closes the channel with an old, revoked state, as the counter party will punish it and steal all its money.

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 between the creation of Alba and its closure. 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 commitments to the new revocation keys; second, parties exchange signed *locked* commitment transactions ($\mathtt{Tx}_1^{\mathtt{A}}$ and $\mathtt{Tx}_1^{\mathtt{B}}$); third, they exchange their old revocation keys. Finally, when they reach the update they want to verify via Alba, i.e., $A \xrightarrow{5} B$, there is one additional communication round where parties also exchange signed *unlocked* commitment transactions ($\mathtt{Tx}_{\mathtt{unl}}^{\mathtt{A}}$ and $\mathtt{Tx}_{\mathtt{unl}}^{\mathtt{B}}$). The transaction-level timelock expires after the contract has been closed, i.e., after T, and prevents malicious parties from going on-chain and close the channel while Alba is still active. Since the Alba instance is closed upon revealing unlocked transactions, as soon as the Alba is settled, standard LN updates and closure are restored.

To summarize, as shown in Figure 4, the Alba proof given as input to the SubmitProof function of the contract is $\pi := (\mathtt{Tx}_{\mathtt{unl}}^{\mathtt{A}}, \sigma_B(\mathtt{Tx}_{\mathtt{unl}}^{\mathtt{A}}), \mathtt{Tx}_{\mathtt{unl}}^{\mathtt{B}}, \sigma_A(\mathtt{Tx}_{\mathtt{unl}}^{\mathtt{B}}))$, with $\mathtt{Tx}_{\mathtt{unl}}^{\mathtt{A}}$ embedding id and R_B.

One more point now needs to be addressed: if one party does not cooperate in an update, e.g., by refusing to send the commitment transaction or the old revocation key, the counter party 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 to the money he lent on the LN. In this situation, Alice is at the mercy of Bob, as she is unable to update the channel, unable to close the channel, and also unable to redeem the coins in the contract.

Incentivize Cooperation for 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 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, Alice can *open a dispute* by calling the Dispute function of the contract (see Figure 4). 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. Bob has now time until $T + T_D$ 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 $\overline{\mathtt{Tx}}_1^{\mathtt{A}}$.

We now analyze an important, subtle case with adversarial Alice: (i) during the channel update, adversarial Alice does not share with Bob the revocation secret for the previous state: since for Bob this update is not complete, Bob will not share with Alice the unlocked transaction; (ii) Alice opens the dispute on $\mathcal{L}_D$, reclaiming the unlocked transaction. Our protocol still guarantees balance security: indeed, Alice cannot go on-chain on Bitcoin and close the channel with the old state because its commitment transaction is locked until time T; additionally, Bob can obtain the *unlocked* transaction $\mathtt{Tx}_{\mathtt{unl}}^{\mathtt{B}}$ from the contract, close the channel (since Alice proved to be malicious) and disclose $\mathtt{Tx}_{\mathtt{unl}}^{\mathtt{A}}$. As a result, the state the channel closes in, correctly reflects the outcome on $\mathcal{L}_D$.

Optional Timelock T. Our smart contract is parameterized with an *optional* T. If Alice and Bob omit to specify T in the setup phase, there is no time limit for parties to submit a proof: in case of disputes, T_D is a relative timelock that starts counting from the moment the dispute is raised. In this way, the contract is more flexible, and hostage situations are avoided in the rational setting.

Functionalities and Remarks. Our 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: (ii) is a subcase of (i) where the channel is updated with the identity function.

We note that Alba, as described above, is a unidirectional 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 chains supporting the scripting capabilities outlined in Figure 4. We also observe that, generally, both Alice and Bob can submit the proof to the smart contract: whoever submits the proof first or, alternatively, opens a dispute, it acts as *prover* P. The other party, instead, acts as *verifier* V.

As we have seen in Section 3, our protocol can serve many purposes, from reacting to simple updates in DeFi applications, to more general and sophisticated applications, such as topping up channels with wrapped tokens and optimistically offloading computation. In these more complex cases, commitment transactions store the state of the application, and the smart contract needs to be able to verify that the data stored correspond to a valid transition from the previous state of the application. Moreover, by interacting with the contract, parties need to be able to reach the final state of the application, if they are not there yet. For instance, consider the case where Alice and Bob are playing chess (Section 3.3) 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. For this reason, when designing the contract for complex applications, we recommend extracting the application logic from the protocol-specific functions.

5. Analysis

In this section, we analyze the security of our Alba protocol. Our security goal, similarly to the LN, is to achieve *balance security*, a property that guarantees that honest parties do not lose money by participating in the protocol. We show that when parties are active (i.e., online and responsive) and rational, the correct execution of the protocol is guaranteed.

```
Smart Contract Pseudocode

Setup($\mathsf{pk}_A, \mathsf{pk}_B, \mathsf{id}, \mathtt{Tx_f}, T, T_D, f, \mathsf{s_{init}}$)

  • $\mathsf{pk}_A, \mathsf{pk}_B$ are public keys of A and B respectively
  • id is a unique identifier (prevents replay attacks)
  • $\mathtt{Tx_f}$ is the funding transaction of the channel
  • $T$ (optional) is an absolute timelock and $T_D$ is a relative
    timelock (both in the future)
  • $f : \mathcal{S} \to \mathcal{O}$ maps the state space to the outcome space
  • $\mathsf{s_{init}}$ (optional) is a valid initial state of the payment channel

SubmitProof($[\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}}], \sigma_B([\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}}]), [\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}], \sigma_A([\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}])$)[a]

If the current time $t > T$ or a dispute has been opened, abort.
Otherwise, check that:
  • $\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}$ and $\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}$ are well-formed and have same balance
    distribution
  • $\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}}$ has an output $\theta = (0, \mathtt{OP\_RETURN}(\mathsf{id}, R_B))$, where
    $R_B$ is the revocation secret of $V$ for $\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}$
  • $\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}$ and $\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}}$ don't have any transaction-level timelock
  • $\sigma_B([\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}}])$, $\sigma_A([\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}])$ are valid sigs on $[\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}}]$, $[\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}]$
If all checks passed, extract $\mathsf{s_{latest}}$ from $[\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}}\ ]$, $[\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}\ ]$, and
distributes funds as per $f(\mathsf{s_{latest}})$. Otherwise, abort.

Dispute($[\overline{\mathtt{Tx}}^{\mathtt{A}}_{\mathtt{1}}], \sigma_B([\overline{\mathtt{Tx}}^{\mathtt{A}}_{\mathtt{1}}]), [\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}], \sigma_A([\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}])$)[b]

If the current time $t > T$ or if a valid proof has been submitted,
abort. Otherwise, check that:
  • $\mathtt{Tx}^{\mathtt{A}}_{\mathtt{1}}$ and $\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}$ are well-formed
  • $\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}$ results from applying $A \xrightarrow{5} B$ to $\overline{\mathtt{Tx}}^{\mathtt{A}}_{\mathtt{1}}$
  • $\sigma_B([\mathtt{Tx}^{\mathtt{A}}_{\mathtt{1}}])$, $\sigma_A([\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}])$ are valid sigs on $[\mathtt{Tx}^{\mathtt{A}}_{\mathtt{1}}]$, $[\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}]$
  • $[\overline{\mathtt{Tx}}^{\mathtt{A}}_{\mathtt{1}}]$ has a transaction-level timelock until $T$ ▷ If $T$ unde-
    fined, timelock is far in the future
  • $[\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}]$ has no transaction-level timelock
If all checks passed, store $[\overline{\mathtt{Tx}}^{\mathtt{A}}_{\mathtt{1}}]$, $[\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}]$, extract from $[\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}]$ the
state $\mathsf{s_{latest}}$ and store it. If $T$ was undefined, set $T$ as current time
$t$. Otherwise, abort.

ResolveDispute($\{\bar{r}_A\} \vee \{[\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}}], \sigma_B([\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}})]\}$)

If Dispute function was called and current time $t > T + T_D$,
abort. Otherwise, check that:
  • If $\bar{r}_A$ is provided, retrieve the stored $[\overline{\mathtt{Tx}}^{\mathtt{A}}_{\mathtt{1}}]$ and check that its
    revocation key is $\bar{r}_A$; if this is the case, send all the money
    to Bob
  • If $\{[\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}}], \sigma_V([\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}})]\}$ are provided, check that:
      – $[\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}}]$ is well-formed and does not have any transaction-
        level timelock
      – $\sigma_B([\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}})]$ is a valid signature of Bob over $[\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}})]$
      – $[\mathtt{Tx}^{\mathtt{A}}_{\mathtt{unl}}]$ matches the balance distribution in stored $[\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}]$
    If all checks passed, distributes funds as per $f(\mathsf{s_{latest}})$. Oth-
    erwise, abort.

Settle($\cdot$)

If no valid proof has been submitted, not dispute has been opened
before $T$, distribute the coins locked in the contract to $P$ and $V$
according to their initial contribution, i.e., $f(\mathsf{s_{init}})$.
```

[a]. 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.

[b]. 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, $\overline{\mathtt{Tx}}^{\mathtt{A}}_{\mathtt{1}}$ refers to the current latest state whereas $\mathtt{Tx}^{\mathtt{B}}_{\mathtt{unl}}$ refers to the new update.

Figure 4: Smart contract pseudocode.

To the best of our knowledge, no practical framework integrating Byzantine adversaries and rational agents yields meaningful results for complex protocols such as ours, instantiated across multiple blockchains. Therefore, we split our proof technique into two phases: we prove all protocol executions are captured by an ideal functionality (standard UC proof, Section 5.1), and then perform a game theoretic analysis on the ideal functionality itself (Section 5.2).

5.1. Security in the UC Framework

To formally model our construction, we use the global universal composability (GUC) framework [46]. Our analysis follows [19], [47], [48], [49], [50]. We use the ideal functionality of generalized channels [19] to capture the functionality of payment channels. Exactly as in the model of generalized channels, we model synchrony with a global clock [51], authenticated communication with guaranteed delivery [48], and, finally, the ledger as an idealized append-only data structure keeping track of all published transactions [19]. 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 Appendix A.3 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 destination chain, or in case of a cheating party posting either an old state or refusing to perform a valid update, all the money goes to the non-cheating party (punish). This, in turn, gives us *balance security*, i.e., no honest party loses funds.

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 $\mathcal{F}_{\mathsf{Alba}}$ (Appendix A.3), Π (Appendix B), GUC security, and the proof of the following main security theorem (Appendix D).

Theorem 1. *The Alba protocol* Π *GUC-realizes the ideal functionality* $\mathcal{F}_{\mathsf{Alba}}$.

5.2. 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* [52], [53] 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.

Without loss of generality, we focus on analyzing the game played on the smart contract side of our protocol: since the channel cannot be closed until the Alba instance on the smart contract is active, the set of actions rational parties can take on the payment channel side can be inferred from the smart contract execution. More explicitly, on the payment channel side, there are only two possible deviations from the correct protocol execution: (i) one party completely refuses to update, or (ii) it stops cooperating in the middle of an initiated update. When case (i) occurs, a dispute on the contract can be opened; if the dispute was raised by a cheating party, the correct outcome of the game can still be enforced on both sides (channel and contract), thereby guaranteeing balance security. On the other hand, when case (ii) occurs, the correct outcome can again be enforced, by obtaining the unlocked transaction either from reading the proof from the contract or by opening a dispute. Essentially, on the payment channel side, what changes between a correct execution of the protocol and an execution in the presence of a dishonest party, is that, in the second case, 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 $(\mathsf{Party}, \mathsf{TimeCondition})$ 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.

For the sake of generality, 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 update Δs was applied to the channel. As a result, the coins locked in the contract are distributed between P and V accordingly. In no proof is submitted to the contract, nor dispute is raised before T, the collateral goes back to the parties with the same distribution they contributed with. We denote with $u(\Delta s)$ the money transfer taking place within the channel update Δs, and with $f(\Delta s) := (f_P(\Delta s), f_V(\Delta s))$ the utility function of parties within the smart contract upon proving the channel underwent Δs. We also denote with $\Delta s'$ an update applied on an old, revoked state $s < s_1$ of the channel.

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 5 depicts the game tree of our smart contract.

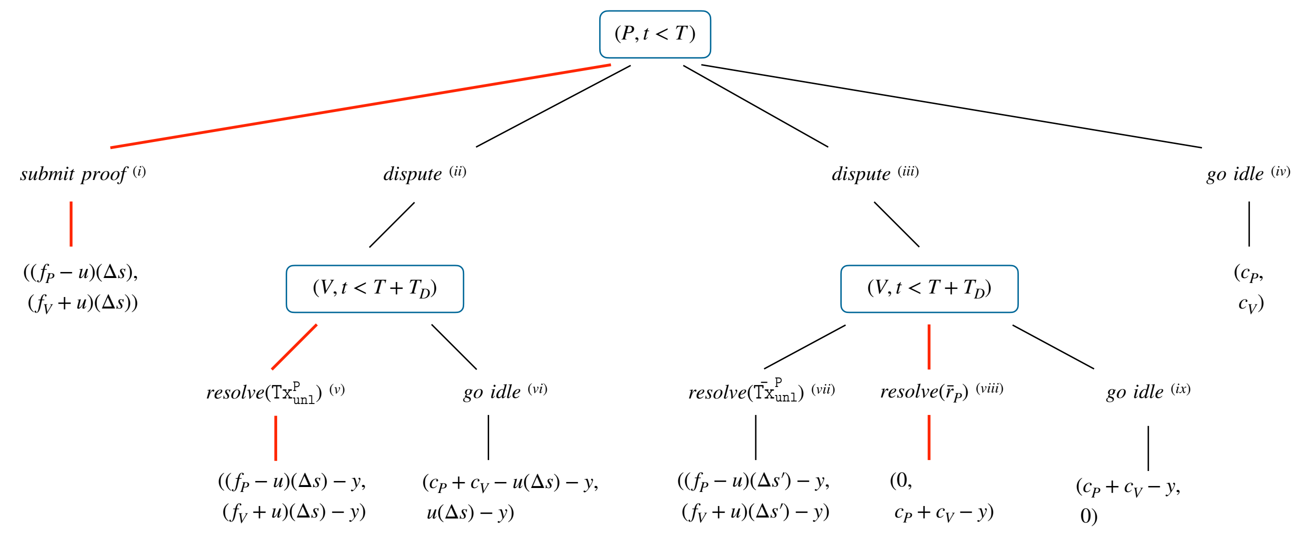
Subgame Perfect Nash Equilibrium (SPNE). In game theory, a *player's strategy* refers to the actions they choose to take in a game, out of all the possible actions. A *strategy profile*, fully specifies all actions in a game from all its players, i.e., it is the set of chosen strategies of all players. 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 look at the tree in Figure 5 and use the *backward induction* technique [54]: 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: $c_P < (f_P - u)(\Delta s)$. 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 2, we prove that under active utility-maximizing agents, a valid proof will be submitted to the smart contract.

Theorem 2. *Consider the tree in Figure 5 reflecting the game between two parties (P, V). For* at least



$(P, t < T)$
submit proof (i)
dispute (ii)
dispute (iii)
go idle (iv)
$((f_P - u)(\Delta s), (f_V + u)(\Delta s))$
$(V, t < T + T_D)$
$(V, t < T + T_D)$
(c_P, c_V)
resolve$(\mathtt{Tx}^{\mathtt{P}}_{\mathtt{unl}})$ (v)
go idle (vi)
resolve$(\bar{\mathtt{Tx}}^{\mathtt{P}}_{\mathtt{unl}})$ (vii)
resolve$(\bar{r}_P)$ (viii)
go idle (ix)
$((f_P - u)(\Delta s) - y, (f_V + u)(\Delta s) - y)$
$(c_P + c_V - u(\Delta s) - y, u(\Delta s) - y)$
$((f_P - u)(\Delta s') - y, (f_V + u)(\Delta s') - y)$
$(0, c_P + c_V - y)$
$(c_P + c_V - y, 0)$

Figure 5: Game tree. In *(i)*, P submits a valid proof. In *(ii)*, P opens a valid dispute by presenting the correct latest state of the channel. In *(iii)*, P opens an invalid dispute by presenting an old, revoked state of the channel. In *(iv)*, P goes idle. Upon P opening a valid dispute, V can resolve it by responding either with the unlocked transaction (action *(v)*) or by going idle (action *(vi)*). Upon P opening an invalid dispute, V can resolve it by responding either with the corresponding old, revoked state (action *(vii)*) (thus reflecting on $\mathcal{L}_D$ an old, revoked state of the payment channel which reverts some updated), with the corresponding revocation secret $\bar{r}_P$ (action *(viii)*), or by going idle (action *(ix)*. With y we capture opportunity and fee costs. The red lines highlight the SPNE of the game.

one of two parties, *say P, the following always holds:* $c_P < (f_P - u)(\Delta s)$. *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 5.

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 presenting an update $\Delta s'$ built on an old, revoked state, it means that $\Delta s'$ gives P a higher payoff, taking away some of V's money. In *(vii)*, V can at most gain what he would get from P choosing *(i)*, 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 $c_P < (f_P - u)(\Delta s)$ holds, the strategy yielding the highest payoff for P is *(i)*.

□

6. Evaluation

In this section, we evaluate the protocol performance regarding communication and on-chain costs. We further discuss possible optimizations.

LN Transaction Size. We recall from Section 4 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 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 [55], broadcast it to the Bitcoin testnet [56], and compared its size to LN transactions. Adding the two `OP_RETURN` outputs results in a 668-byte transaction, i.e., 108 bytes larger than the 560-byte standard ones, whereas including the hash of the revocation secret alone results in a 646-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

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

TABLE 1: On-chain gas costs evaluation for EVM-based blockchains. For more details, see our implementation and evaluation [57].

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 to deploy, run, and test the smart contract. A proof-of-concept implementation and evaluation of the ALBA contract is publicly available at [57]. 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 1.

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 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. While on-chain costs of the Alba contract are in line with those of other cross-chain protocols [10], [11], [12], [58], any optimizations that may bring the cost down closer to the 21k gas of the simplest Ethereum transaction are desirable. Towards this end, a natural optimization for Alba occurs by observing that both parties are honest and collaborate in the *optimistic* case. Thus, 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. In this case, the smart contract shall only verify two signatures (consuming 48k gas, see OptimisticSubmitProof in Table 1), before unlocking the coins, significantly reducing the cost of Alba. Additional cost optimizations may be feasible in a production-level implementation as our smart contract in [57] is a research prototype.

7. Conclusion

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

In particular, we presented Alba, a new, *scalable 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., lending protocols), but also towards *multi-asset payment channels*, and *optimistic off-chain computation*.

We analyzed security aspects of Alba when faced with Byzantine adversaries within the UC framework. Furthermore, we completed the analysis by incorporating a game-theoretic study, capturing the case where all 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.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009, http://bitcoin.org/bitcoin.pdf.

[2] "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/

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

[4] 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.

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

[6] "Submarine swap in lightning network," https://wiki.ion.radar.tech/tech/research/submarine-swap, 2021.

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

[8] M. Herlihy, "Atomic cross-chain swaps," *CoRR*, vol. abs/1801.09515, 2018. [Online]. Available: http://arxiv.org/abs/1801.09515

[9] 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.

[10] 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.

[11] M. Westerkamp and J. Eberhardt, "zkRelay: Facilitating Sidechains using zkSNARK-based Chain-Relays," in *IEEE European Symposium on Security and Privacy Workshops*, 2020.

[12] 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

[13] 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.

[14] S. Sharma, "Ethereum's Rollups are Centralized. A Look Into Decentralized Sequencers," 2023, https://shorturl.at/dfuM0.

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

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

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

[18] 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

[19] 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.

[20] "BTC Relay," https://github.com/ethereum/btcrelay,http://btcrelay.org/, 2016.

[21] A. Kiayias, A. Miller, and D. Zindros, "Non-interactive proofs of proof-of-work," in *Financial Cryptography and Data Security FC*, 2020.

[22] 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

[23] 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.

[24] A. Obadia, A. Salles, L. Sankar, T. Chitra, V. Chellani, and P. Daian, "Unity is strength: A formalization of cross-domain maximal extractable value," *CoRR*, 2021. [Online]. Available: https://arxiv.org/abs/2112.01472

[25] T. Nadahalli, M. Khabbazian, and R. Wattenhofer, "Timelocked bribing." Berlin, Heidelberg: Springer-Verlag, 2021. [Online]. Available: https://doi.org/10.1007/978-3-662-64322-8_3

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

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

[28] M. Sober, G. Scaffino, and C. S. S. Schulte, "A voting-based blockchain interoperability oracle," in *IEEE International Conference on Blockchain*, 2021.

[29] "Gravity bridge," https://github.com/Gravity-Bridge/Gravity-Docs, 2022.

[30] A. Kiayias, N. Lamprou, and A.-P. Stouka, "Proofs of proofs of work with sublinear complexity," in *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 2016.

[31] A. Kiayias, A. Miller, and D. Zindros, "Non-interactive Proofs of Proof-of-Work," in *Financial Cryptography and Data Security*. Springer International Publishing, 2020.

[32] K. Karantias, A. Kiayias, and D. Zindros, "Compact storage of superblocks for nipopow applications," in *Mathematical Research for Blockchain Economy*, P. Pardalos, I. Kotsireas, Y. Guo, and W. Knottenbelt, Eds. Springer International Publishing, 2020.

[33] S. Agrawal, J. Neu, E. N. Tas, and D. Zindros, "Proofs of Proof-Of-Stake with Sublinear Complexity," in *5th Conference on Advances in Financial Technologies, AFT*, J. Bonneau and S. M. Weinberg, Eds. Schloss Dagstuhl, 2023.

[34] O. Ciobotaru, F. Shirazi, A. Stewart, and S. Vasilyev, "Accountable light client systems for pos blockchains," *IACR Cryptol. ePrint Arch.*, p. 1205, 2022. [Online]. Available: https://eprint.iacr.org/2022/1205

[35] M. Westerkamp and M. Diez, "Verilay: A verifiable proof of stake chain relay," in *IEEE International Conference on Blockchain and Cryptocurrency, ICBC*, 2022.

[36] C Ittai Abraham, "A Payment Channel is a two person BFS-SMR system," 2019, https://shorturl.at/ctvzJ.

[37] 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

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

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

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

[41] 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.

[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] 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

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

[45] E. Mouton, "LN Things Part 2: Updating State," https://ellemouton.com/posts/updating-state/, 2021.

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

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

[48] 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.

[49] 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.

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

[51] J. Katz, U. Maurer, B. Tackmann, and V. Zikas, “Universally composable synchronous computation,” in *Theory of Cryptography*, 2013.

[52] R. Selten, “Multistage game models and delay supergames,” *Nobel lecture*, 1994. [Online]. Available: https://www.nobelprize.org/uploads/2018/06/selten-lecture.pdf

[53] 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

[54] M. M. Kaminski, “Generalized backward induction: Justification for a folk algorithm,” *Games*, 2019. [Online]. Available: https://www.mdpi.com/2073-4336/10/3/34

[55] “python-bitcoin-utils library,” https://github.com/karask/python-bitcoin-utils, 2016.

[56] “Prover’s commitment transaction,” https://shorturl.at/prsBC.

[57] “ALBA Proof of Concept and Evaluation,” https://github.com/ALBA-blockchain/ALBA-Protocol.

[58] M. Sober, M. Kobelt, G. Scaffino, D. Kaaser, and S. Schulte, “Distributed Key Generation with Smart Contracts Using Zk-SNARKs,” in *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. Association for Computing Machinery, 2023.

[59] 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.

[60] 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.

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

[62] 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.

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

[64] J. A. Garay, A. Kiayias, and N. Leonardos, “The bitcoin backbone protocol: Analysis and applications,” in *Advances in Cryptology - EUROCRYPT*. Springer, 2015.

Appendix A.

A.1. 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 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.

A.2. Modeling in the UC-Framework

To analyze the security of Alba, we make use of the global universal composability (GUC) framework [46], i.e., an extension to the original UC framework [27]. Our analysis closely follows [19], [47], [48], [49], [50], [59], [60], [61], [62].

A.2.1. 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 [51]), 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 [48]) 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 [19]. 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., [63], [64]. These functionalities introduce a lot of additional complexity. To increase readability, we opt for this simplified ledger functionality.

Ledgers can support smart contracts. Following [47], [48], 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 instances of this functionality. 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 in the functionality we define in Section A.2.3. 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 [19]. We proceed now to give the API of the $\mathcal{G}_{Ledger}$ functionality.

Interface of $\mathcal{G}_{Ledger}(\Delta, \Sigma, \mathcal{V})$ [19]

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.

A.2.2. 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 $\texttt{EXEC}^{\mathcal{F}_{aux}}_{\Pi,\mathcal{A},\mathcal{Z}}(\lambda, z)$ 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 $\texttt{EXEC}^{\mathcal{F}_{aux}}_{\phi\mathcal{F},\mathcal{A},\mathcal{Z}}(\lambda, z)$. 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 1. 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

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

where $\approx^c$ denotes computational indistinguishability.

A.2.3. Payment Channels

We reuse the functionality introduced in [19]. 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 $\gamma := (\mathsf{id}, \mathsf{users}, \mathsf{cash}, \mathsf{st}, \mathsf{Aid}, \mathsf{ADeadline}, \mathsf{id_{ch}})$. $\gamma.\mathsf{id} \in \{0,1\}^*$ 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 [19] 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 $(\texttt{CREATE}, \gamma, tid_P) \xleftrightarrow{\tau_0} P$, distinguish:

Both agreed: If already received $(\texttt{CREATE}, \gamma, tid_Q) \xleftrightarrow{\tau} Q$, 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 $\Gamma(\gamma.\mathtt{id}) := (\{\gamma\}, \mathsf{Tx})$ and $(\texttt{CREATED}, \gamma.\mathtt{id}) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$. Else stop.

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

Upon $(\texttt{UPDATE}, \mathsf{id_{ch}}, \vec{\theta}, t_{\mathsf{stp}}, \texttt{AUpdate}) \xleftrightarrow{\tau_0} P$, parse $(\{\gamma\}, \mathsf{Tx}) := \Gamma(\mathsf{id_{ch}})$, 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 $(\texttt{UPDATE-REQ}, \mathsf{id_{ch}}, \vec{\theta}, t_{\mathsf{stp}}, \vec{tid}, \texttt{AUpdate}) \xhookrightarrow{\tau_1} Q$ and $(\texttt{SETUP}, \mathsf{id_{ch}}, \vec{tid}) \xhookrightarrow{\tau_1} P$.
2) If $(\texttt{SETUP-OK}, \mathsf{id_{ch}}, \texttt{AUpdate}) \xleftarrow{\tau_2 \leq \tau_1 + t_{\mathsf{stp}}} P$, then $(\texttt{SETUP-OK}, \mathsf{id_{ch}}, \texttt{AUpdate}) \xhookrightarrow{\tau_3 \leq \tau_2 + T} Q$. Else stop.
3) If $(\texttt{UPDATE-OK}, \mathsf{id_{ch}}, \texttt{AUpdate}) \xleftrightarrow{\tau_3} Q$, then $(\texttt{UPDATE-OK}, \mathsf{id_{ch}}, \texttt{AUpdate}) \xhookrightarrow{\tau_4 \leq \tau_3 + T} P$. Else distinguish:
   - If Q honest or if instructed by $\mathcal{S}$, stop *(reject)*.
   - Else set $\Gamma(\mathsf{id_{ch}}) := (\{\gamma, \gamma'\}, \mathsf{Tx})$, run $\texttt{ForceClose}(\mathsf{id_{ch}})$ and stop.
4) If $(\texttt{REVOKE}, \mathsf{id_{ch}}, \texttt{AUpdate}) \xleftrightarrow{\tau_4} P$, send $(\texttt{REVOKE-REQ}, \mathsf{id_{ch}}, \texttt{AUpdate}) \xhookrightarrow{\tau_5 \leq \tau_4 + T} Q$.
   Else set $\Gamma(\mathsf{id_{ch}}) := (\{\gamma, \gamma'\}, \mathsf{Tx})$, run $\texttt{ForceClose}(\mathsf{id_{ch}})$ and stop.
5) If $(\texttt{REVOKE}, \mathsf{id_{ch}}, \texttt{AUpdate}) \xleftrightarrow{\tau_5} Q$, $\Gamma(\mathsf{id_{ch}}) := (\{\gamma'\}, \mathsf{Tx})$, send $(\texttt{UPDATED}, \mathsf{id_{ch}}, \vec{\theta}) \xhookrightarrow{\tau_6 \leq \tau_5 + T} \gamma.\mathsf{users}$ and stop *(accept)*.
   Else set $\Gamma(\mathsf{id_{ch}}) := (\{\gamma, \gamma'\}, \mathsf{Tx})$, run $\texttt{ForceClose}(\mathsf{id_{ch}})$ and stop.

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

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

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

At the end of every round τ_0: For each $\mathsf{id_{ch}} \in \{0,1\}^*$ s.t. $(\{\gamma\}, \mathsf{Tx}) := \Gamma(\mathsf{id_{ch}}) \neq \perp$, do the following. For the last message received for $\mathsf{id_{ch}}$ that contained the boolean flag $\texttt{AUpdate}$, set $\gamma.\mathsf{Aid} := \texttt{AUpdate}$. Further, check if $\mathcal{L}$ contains Tx' with $\mathsf{Tx}'.\mathsf{In} = \mathsf{Tx.txid}$. 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 $\Gamma(\mathsf{id_{ch}}) := \perp$ and $(\texttt{CLOSED}, \mathsf{id_{ch}}) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$ 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 $\Gamma(\mathsf{id_{ch}}) := \perp$, $(\texttt{PUNISHED}, \mathsf{id_{ch}}) \xhookrightarrow{\tau_1} P$ and stop.
Error: Else $(\texttt{ERROR}) \xhookrightarrow{\tau} \gamma.\mathsf{users}$.

$\texttt{ForceClose}(\mathsf{id_{ch}})$: Let τ_0 be the current round and $(X, \mathsf{Tx}) := \Gamma(\mathsf{id_{ch}})$. If within Δ rounds Tx is still unspent on $\mathcal{L}$, then $(\texttt{ERROR}) \xhookrightarrow{\tau_0 + \Delta} \gamma.\mathsf{users}$ 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 $\texttt{ForceClose}(\mathsf{id_{ch}})$, first read γ from $\Gamma(\mathsf{id_{ch}})$ 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 $\Gamma(\mathsf{id_{ch}})$).At time $\gamma.\mathsf{ADeadline}$ run $\texttt{ForceClose}(\mathsf{id_{ch}})$ and stop.

A.2.4. 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 [19], 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 [19].

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 $(\mathtt{UPDATE}, \mathsf{id_{ch}}, \vec{\theta}, t_{\mathsf{stp}}, \mathtt{AUpdate}) \xhookleftarrow{t_0} \mathcal{Z}$

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

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

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

$$[\mathtt{TX_1^P}] := \mathtt{GenCom}^V((\mathsf{pk}_P, R_P), (\mathsf{pk}_V, \_), \_, \vec{\theta}, \gamma.\mathsf{Aid}, \gamma.\mathsf{ADeadline}, \mathtt{AUpdate})$$

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

$$[\mathtt{TX_1^P}] := \mathtt{GenCom}^V((\mathsf{pk}_P, R_P), (\mathsf{pk}_V, R_V), \bar{R}_P, \vec{\theta}, \gamma.\mathsf{Aid}, \gamma.\mathsf{ADeadline}, \mathtt{AUpdate})$$

4) Send $(\text{updateInfo}, \mathsf{id_{ch}}, R_V) \xhookrightarrow{\tau_0} P$, $(\mathtt{UPDATE\text{–}REQ}, \mathsf{id_{ch}}, \vec{\theta}, t_{\mathsf{stp}}, \mathtt{TX_1^P}.\mathsf{txid}, \mathtt{AUpdate}) \xhookrightarrow{\tau_0+1} \mathcal{Z}$.

Party P at time $t_0 + 2$

5) If received $(\text{updateInfo}, \mathsf{id_{ch}}, R_V) \xhookleftarrow{t_0+2} V$, Extract $\mathtt{TX_f}$ and γ from $\Gamma^P(\mathsf{id_{ch}})$ and

$$[\mathtt{TX_1^V}] := \mathtt{GenCom}^P((\mathsf{pk}_P, \_), ((\mathsf{pk}_V, R_V), \_, \vec{\theta}, \gamma.\mathsf{Aid}, \gamma.\mathsf{ADeadline}, \gamma.\mathtt{AUpdate})$$

Send $(\mathtt{SETUP}, \mathsf{id_{ch}}, \mathtt{TX_1^V}.\mathsf{txid}, \mathtt{AUpdate}) \xhookrightarrow{t_0+2} \mathcal{Z}$. If $(\mathtt{SETUP\text{–}OK}, \mathsf{id_{ch}}, \mathtt{AUpdate}) \xhookleftarrow{t_1 \leq t_0+2+t_{\mathsf{stp}}} \mathcal{Z}$, compute $\sigma_P([\mathtt{TX_c^V}]) \leftarrow \mathsf{Sign}_{\mathsf{sk}_P}([\mathtt{TX_1^V}])$ and send $(\text{updateComP}, \mathsf{id_{ch}}, [\mathtt{TX_1^V}], \sigma_P([\mathtt{TX_c^V}])) \xhookrightarrow{t_1} V$. Else stop.

Party V

6) If $(\text{updateComP}, \mathsf{id_{ch}}, [\mathtt{TX_1^V}], \sigma_P([\mathtt{TX_c^V}])) \xhookleftarrow{\tau_1 \leq \tau_0+2+t_{\mathsf{stp}}} P$, s.t. $\mathsf{Vrfy}_{\mathsf{pk}_P}([\mathtt{TX_1^V}]; \sigma_P([\mathtt{TX_c^V}])) = 1$ output $(\mathtt{SETUP\text{–}OK}, \mathsf{id_{ch}}, \mathtt{AUpdate}) \xhookrightarrow{\tau_1} \mathcal{Z}$. Else stop. ▷ Here, V doesn't go to idle mode because the update was just interrupted before exchanging signatures.
7) If $(\mathtt{UPDATE\text{–}OK}, \mathsf{id_{ch}}, \mathtt{AUpdate}) \xhookleftarrow{\tau_1} \mathcal{Z}$ Then sign $\sigma_V([\mathtt{TX_c^P}]) \leftarrow \mathsf{Sign}([\mathtt{TX_1^P}])$ and send $(\text{updateComV}, \mathsf{id_{ch}}, [\mathtt{TX_1^P}], \sigma_V([\mathtt{TX_c^P}])) \xhookrightarrow{\tau_1} P$. Else if you did not receive the $\mathtt{UPDATE\text{–}OK}$ message then send $(\text{updateNotOk}, \mathsf{id_{ch}}, r_V) \xhookrightarrow{\tau_1} P$ and stop.

Party P

8) In round $t_1 + 2$ distinguish the following cases:
   - If $(\text{updateComV}, \mathsf{id_{ch}}, [\mathtt{TX_1^P}], \sigma_V([\mathtt{TX_c^P}])) \xhookleftarrow{t_1+2} V$, s.t. $\mathsf{Vrfy}_{\mathsf{pk}_V}([\mathtt{TX_1^P}]; \sigma_V([\mathtt{TX_c^P}])) = 1$, and if $\mathtt{AlbaUpdate}$ is $False$ output $(\mathtt{UPDATE\text{–}OK}, \mathsf{id_{ch}}, \mathtt{AUpdate}) \xhookrightarrow{t_1+2} \mathcal{Z}$, otherwise if $\mathtt{AlbaUpdate}$ is $True$, check the $\mathtt{OP\_RETURN}$ output of $[\mathtt{TX_1^P}]$ and verify that it is well-formed (is built using $\mathtt{GenCom}^V_{Alba}$ function) and only then send $(\mathtt{UPDATE\text{–}OK}, \mathsf{id_{ch}}, \mathtt{AUpdate}) \xhookrightarrow{t_1+2} \mathcal{Z}$, otherwise if $[\mathtt{TX_1^P}]$ is not well-formed execute the procedure $\mathtt{ForceClose}^P(\mathsf{id_{ch}})$ and stop.
   - If $(\text{updateNotOk}, \mathsf{id_{ch}}, r_V) \xhookleftarrow{t_1+2} V$, s.t. $(R_V, r_V) \in R$, add $\Theta^P(\mathsf{id_{ch}}) := \Theta^P(\mathsf{id_{ch}}) \cup ([\mathtt{TX_1^P}], r_P, r_V, \sigma_P([\mathtt{TX_c^V}]))$ and stop.
   - Else, execute the procedure $\mathtt{ForceClose}^P(\mathsf{id_{ch}})$ and stop.
9) If $(\mathtt{REVOKE}, \mathsf{id_{ch}}, \mathtt{AUpdate}) \xhookleftarrow{t_1+2} \mathcal{Z}$, parse $\Gamma^P(\mathsf{id_{ch}})$ as $(\gamma, \mathtt{TX_f}, (\overline{\mathtt{TX}}_1^{\mathtt{V}}, \overline{\mathtt{TX}}_1^{\mathtt{P}} \bar{r}_P, \bar{R}_V, \_))$ where _ denotes a wildcard and update the channel space as $\Gamma^P(\mathsf{id_{ch}}) := (\gamma, \mathtt{TX_f}, (\mathtt{TX_1^P}, \mathtt{TX_1^V}, r_P, R_V, \bar{r}_P))$ for $\mathtt{TX_1^P} := ([\mathtt{TX_1^P}], \{\mathsf{Sign}_{\mathsf{sk}_P}([\mathtt{TX_1^P}]), \sigma_V([\mathtt{TX_c^P}])\})$, and $\mathtt{TX_1^V} := ([\mathtt{TX_1^V}], \sigma_P([\mathtt{TX_c^V}]))$, send $(\text{revokeP}, \mathsf{id_{ch}}, \bar{r}_P) \xhookrightarrow{t_1+2} V$. Else, execute $\mathtt{ForceClose}^P(\mathsf{id_{ch}})$ and stop.

Party V

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

$$\Theta^V(\mathsf{id_{ch}}) := \Theta^V(\mathsf{id_{ch}}) \cup ([\overline{\mathtt{TX}}_1^{\mathtt{V}}], \bar{r}_P, \bar{r}_V, \sigma_V([\overline{\mathtt{TX}}_c^{\mathtt{P}}]))$$
$$\Gamma^V(\mathsf{id_{ch}}) := (\gamma, \mathtt{TX_f}, (\mathtt{TX_1^P}, \mathtt{TX_1^V}, r_V, R_P, \bar{r}_V)),$$

for $\mathtt{TX_1^V} := ([\mathtt{TX_1^V}], \{\mathsf{Sign}_{\mathsf{sk}_V}([\mathtt{TX_1^V}]), \sigma_P([\mathtt{TX_c^V}])\})$, and $\mathtt{TX_1^P} := ([\mathtt{TX_1^P}], \sigma_V([\mathtt{TX_c^P}]))$, then send $(\text{revokeV}, \mathsf{id_{ch}}, \bar{r}_V) \xhookrightarrow{\tau_1+2} P$. In the next round $(\mathtt{UPDATED}, \mathsf{id_{ch}}) \xhookrightarrow{\tau_1+3} \mathcal{Z}$ and stop. Else, in round $\tau_1 + 2$, execute $\mathtt{ForceClose}^V(\mathsf{id_{ch}})$ and stop.

Party P

12) If $(\text{revokeV}, \mathsf{id_{ch}}, \bar{r}_V) \xhookleftarrow{t_1+4} V$ s.t. $(\bar{R}_V, \bar{r}_V) \in R$, then set $\Theta^P(\mathsf{id_{ch}}) := \Theta^P(\mathsf{id_{ch}}) \cup ([\overline{\mathtt{TX}}_1^{\mathtt{P}}], \bar{r}_P, \bar{r}_V, \sigma_P([\overline{\mathtt{TX}}_c^{\mathtt{V}}]))$ and $(\mathtt{UPDATED}, \mathsf{id_{ch}}) \xhookrightarrow{t_1+4} \mathcal{Z}$. Else execute $\mathtt{ForceClose}^P(\mathsf{id_{ch}})$ and stop.

Subprocedures

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

$\mathtt{GenCom}^A([\mathtt{TX_f}], (\mathsf{pk}_B, R_B), (\mathsf{pk}_A, R_A), \bar{R}_P, \vec{\theta}, \mathsf{Aid}, \mathsf{ADeadline}$

```
$, \texttt{AUpdate}):$
$\triangleright$ Called by party $A \in \gamma.\mathsf{users}$, $B := \gamma.\mathsf{otherParty}(A)$. This function produces $\texttt{TX}_1^{\texttt{B}}$
1) Let $(c, \texttt{Multi-Sig}_{\mathsf{pk}_A, \mathsf{pk}_B}) := \texttt{TX}_{\texttt{f}}.\mathsf{Out}[1]$ , and denote
     $\varphi_B := \texttt{Multi-Sig}_{\mathsf{ToKey}(R_B), \mathsf{pk}_A} \vee \{\theta_B.\varphi \wedge \texttt{CheckRelative}_\Delta\}$
     For $i \in [1, |\vec{\theta}|-2]$: $\triangleright$ 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 $\mathsf{Tx.In} = \texttt{TX}_{\texttt{f}}.\mathsf{txid}\|1$, $\mathsf{Tx.Out} := (\theta_A, \{\theta_B.\mathsf{cash}, \varphi_B\}, \{\theta_i.\mathsf{cash}, \varphi_i\}^{i \in [1, |\vec{\theta}|-2]})$ and $\mathsf{Tx.timelock} = \_$.
3) If $\mathsf{Aid}$ is not empty and $\texttt{AlbaUpdate} == False$, then set $\mathsf{Tx.timelock} = \mathsf{ADeadline}$.
4) Else If $\mathsf{Aid}$ is not empty and $\texttt{AlbaUpdate} == True$ 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 $\texttt{TX}_1^{\texttt{X}}$ from $\Gamma(\mathsf{id_{ch}})$ and send $(\mathrm{post}, \texttt{TX}_1^{\texttt{X}}) \xhookrightarrow{t_0} \mathcal{L}$.
2) Let $t_1 \leq t_0 + \Delta$ be the round in which $\texttt{TX}_1^{\texttt{X}}$ 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 $\gamma$ 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 $\gamma$ 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 3. *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 $\texttt{TX}_{\texttt{f}}$ and $\gamma$ from $\Gamma^P(\mathsf{id_{ch}})$ and
        $[\texttt{TX}_1^{\texttt{V}}] := \texttt{GenCom}^P((\mathsf{pk}_P, \_), ((\mathsf{pk}_V, R_V), \_, \vec{\theta}, \gamma.\mathsf{Aid}, \gamma.\mathsf{ADeadline}, \gamma.\texttt{AUpdate})$

   Send $(\texttt{SETUP}, \mathsf{id_{ch}}, \texttt{TX}_1^{\texttt{V}}.\mathsf{txid}, \texttt{AUpdate}) \xhookrightarrow{\tau_0^P+2} \mathcal{Z}$.
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 $\sigma_P([\texttt{TX}_{\texttt{c}}^{\texttt{V}}]) \leftarrow \mathsf{Sign}_{\mathsf{sk}_P}([\texttt{TX}_1^{\texttt{V}}])$, and send $(\mathrm{updateComP}, \mathsf{id_{ch}}, [\texttt{TX}_1^{\texttt{V}}], \sigma_P([\texttt{TX}_{\texttt{c}}^{\texttt{V}}])) \xhookrightarrow{\tau_1^P} V$. Else stop.
4) In round $\tau_1^P + 2$ distinguish the following cases:
   • If you receive $(\mathrm{updateComV}, \mathsf{id_{ch}}, [\texttt{TX}_1^{\texttt{P}}], \sigma_V([\texttt{TX}_{\texttt{c}}^{\texttt{P}}])) \xhookleftarrow{\tau_1^P+2} V$, s.t. $\mathsf{Vrfy}_{\mathsf{pk}_V}([\texttt{TX}_1^{\texttt{P}}]; \sigma_V([\texttt{TX}_{\texttt{c}}^{\texttt{P}}])) = 1$, and if $\texttt{AlbaUpdate}$ is $False$, or, if $\texttt{AlbaUpdate}$ is $True$ and the $\texttt{OP\_RETURN}$ output of $[\texttt{TX}_1^{\texttt{P}}]$ is well-formed (is built using $\texttt{GenCom}^V_{Alba}$ 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 $\Theta^P(\mathsf{id_{ch}}) := \Theta^P(\mathsf{id_{ch}}) \cup ([\texttt{TX}_1^{\texttt{P}}], r_P, r_V, \sigma_P([\texttt{TX}_{\texttt{c}}^{\texttt{V}}]))$, 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 $(\gamma, \texttt{TX}_{\texttt{f}}, (\overline{\texttt{TX}}_1^{\texttt{V}}, \overline{\texttt{TX}}_1^{\texttt{P}} \bar{r}_P, \bar{R}_V, \_))$ where $\_$ denotes a wildcard and update the channel space as $\Gamma^P(\mathsf{id_{ch}}) := (\gamma, \texttt{TX}_{\texttt{f}}, (\texttt{TX}_1^{\texttt{P}}, \texttt{TX}_1^{\texttt{V}}, r_P, R_V, \bar{r}_P))$ for $\texttt{TX}_1^{\texttt{P}} := ([\texttt{TX}_1^{\texttt{P}}], \{\mathsf{Sign}_{\mathsf{sk}_P}([\texttt{TX}_1^{\texttt{P}}]), \sigma_V([\texttt{TX}_{\texttt{c}}^{\texttt{P}}])\})$, and $\texttt{TX}_1^{\texttt{V}} := ([\texttt{TX}_1^{\texttt{V}}], \sigma_P([\texttt{TX}_{\texttt{c}}^{\texttt{V}}]))$. Then send $(\mathrm{revokeP}, \mathsf{id_{ch}}, \bar{r}_P) \xhookrightarrow{\tau_1^P+2} V$.
```

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

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

$$\Theta^P(\mathsf{id}_{\mathsf{ch}}) := \Theta^P(\mathsf{id}_{\mathsf{ch}}) \cup ([\overline{\mathtt{TX}}_{\mathtt{1}}^{\mathtt{P}}], \bar{r}_P, \bar{r}_V, \sigma_P([\overline{\mathtt{TX}}_{\mathtt{c}}^{\mathtt{V}}]))$$

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

<u>Case V is honest and P is corrupted</u>

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

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

$$[\mathtt{TX}_{\mathtt{1}}^{\mathtt{P}}] := \mathtt{GenCom}^V((\mathsf{pk}_P, R_P), (\mathsf{pk}_V, \_), \_, \vec{\theta}, \gamma.\mathsf{Aid}, \gamma.\mathsf{ADeadline}, \mathtt{AUpdate})$$

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

$$[\mathtt{TX}_{\mathtt{1}}^{\mathtt{P}}] := \mathtt{GenCom}^V((\mathsf{pk}_P, R_P), (\mathsf{pk}_V, R_V), \bar{R}_P, \vec{\theta}, \gamma.\mathsf{Aid}, \gamma.\mathsf{ADeadline}, \mathtt{AUpdate})$$

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

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

$$\Theta^V(\mathsf{id}_{\mathsf{ch}}) := \Theta^V(\mathsf{id}_{\mathsf{ch}}) \cup ([\overline{\mathtt{TX}}_{\mathtt{1}}^{\mathtt{V}}], \bar{r}_P, \bar{r}_V, \sigma_V([\overline{\mathtt{TX}}_{\mathtt{c}}^{\mathtt{P}}]))$$
$$\Gamma^V(\mathsf{id}_{\mathsf{ch}}) := (\gamma, \mathtt{TX}_{\mathtt{f}}, (\mathtt{TX}_{\mathtt{1}}^{\mathtt{P}}, \mathtt{TX}_{\mathtt{1}}^{\mathtt{V}}, r_V, R_P, \bar{r}_V)),$$

   $\mathtt{TX}_{\mathtt{1}}^{\mathtt{V}} := ([\mathtt{TX}_{\mathtt{1}}^{\mathtt{V}}], \{\mathsf{Sign}_{\mathsf{sk}_V}([\mathtt{TX}_{\mathtt{1}}^{\mathtt{V}}]), \sigma_P([\mathtt{TX}_{\mathtt{c}}^{\mathtt{V}}])\})$, and $\mathtt{TX}_{\mathtt{1}}^{\mathtt{P}} := ([\mathtt{TX}_{\mathtt{1}}^{\mathtt{P}}], \sigma_V([\mathtt{TX}_{\mathtt{c}}^{\mathtt{P}}]))$. Then $(\text{revokeV}, \mathsf{id}_{\mathsf{ch}}, \bar{r}_V) \xhookrightarrow{\tau_1^V+2} P$ and stop. Else, in round $\tau_1^V + 2$, execute the simulator code of the procedure $\mathtt{ForceClose}^B(\mathsf{id})$ and stop.

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

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

Simulator for wrapper

<u>ForceClose:</u> Upon invoking the simulator code $\mathtt{ForceClose}^X(\mathsf{id}_{\mathsf{ch}})$ for an honest party X, first read γ from $\Gamma^X(\mathsf{id}_{\mathsf{ch}})$ and distinguish between the two cases:
- If $\gamma.\mathsf{Aid}$ is empty then run $\mathtt{ForceClose}(\mathsf{id}_{\mathsf{ch}})$
- Else, if $\gamma.\mathsf{Aid}$ is not empty, set $\gamma.\mathsf{idle} \leftarrow True$ (and update γ in $\Gamma^X(\mathsf{id}_{\mathsf{ch}})$) 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 $\mathtt{State} == \mathtt{Valid\_Proof}$, run $\mathtt{ForceClose}(\mathsf{id}_{\mathsf{ch}})$ and stop.

  At time $\gamma.\mathsf{ADeadline}$ run $\mathtt{ForceClose}(\mathsf{id}_{\mathsf{ch}})$ and stop.

A.3. Alba Ideal Functionality

We proceed to the formalization of the security properties we aim to achieve with our Alba protocol. For this, 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. 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 $\mathcal{L}_D$, the setup is considered complete and the functionality moves to the next phase.

In the *channel update and proof* phase, the ideal functionality determines the receiver after the Alba execution, as well as the outcome. This is done via the $\mathtt{DetermineReceiver}$ and $\mathtt{HandleClaim}$ subprocedures. In essence, according to the messages received by P, V as well as the state of the channel functionality $\mathcal{PC}$, $\mathcal{F}_{\mathsf{Alba}}$ can determine what should be the outcome of the $Alba$ instance and the time t_{settle} by which it needs to be enforced. The outcomes can be either the correct payout $\mathbf{TX}_{payout}$, or the whole amount going to either party $U \in \{P, V\}$. If in time, either party can enforce this outcome on-chain. If no one does so in time t_{settle}, parties can additionally enforce the initial state $\mathbf{TX}_{init}$ afterwards.

The property that this functionality achieves is *atomicity*, and is formally defined by the functionality below. It is easy to see on a high level, what this property does, however. Indeed, it says that after determining the correct outcome of running the Alba instance, (i) if an outcome is enforced, it has to be the determined one, (ii) if a user wishes to enforce this outcome, she can do so, and (iii) if no user has enforced an outcome by the time t_{settle}, the outcome corresponding to the initial state can be enforced as well.

We (re-)introduce the following variables. Let α be the total money locked in the contract. init is the initial valid state of the contract, $r : \mathcal{S} \rightarrow \mathcal{S}$ is the state transition function, mapping a valid state to another valid state, and $f : \mathcal{S} \rightarrow \mathcal{O}$ is the outcome mapping, which assigns a balance for a valid (final) state to each user. We say that $\alpha' = (r, f, \mathsf{init})$ is a tuple holding this variables. Again, 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 $\mathcal{S}$. We now present the functionality.

Ideal functionality of the Alba protocol

Parameters:

$\mathcal{L_S}, \mathcal{L_D} \dots$ two instances of $\mathcal{G}_{Ledger}$ representing the source and destination blockchain.
$\mathcal{PC} \dots$ an instance of the payment channel ideal functionality. We have access to the internal channel space storage of $\mathcal{PC}$ and we denote the channel space of channel $\mathsf{id_{ch}}$ by $\Gamma(\mathsf{id_{ch}})$,
$\Delta_D \dots$ the blockchain delay of $\mathcal{L_D}$.

Variables:

$\phi(.) \dots$ a mapping of Alba ID id to $(\mathsf{pk}_P, \mathsf{pk}_V, \mathtt{TX_f}, \mathsf{id_{Ch}}, T_0, T_1, T_2, \mathsf{Addr})$ 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 $T_0 < T_1 < T_2, \in \mathbb{N}$. Addr is the address of the instance of the Alba contract.
$t_{updateDelay} \dots$ The time it takes for P to initiate the update and receive the required data to generate the proof. For Lightning Network based Alba, $t_{updateDelay}$ is $4 + t_{\mathsf{stp}}$.

Setup

1) Upon $(\mathtt{A\text{–}SETUP}, \mathsf{id}, \mathsf{id_{ch}}, P, T_0, T_1, T_2, \alpha, \alpha') \xhookleftarrow{\tau} V$, read $\Gamma(\mathsf{id_{ch}})$ from the storage of $\mathcal{PC}$ and parse the output as $(\{\gamma\}, \mathtt{TX_f}, \dots)$, then check the followings:
   - $\gamma.\mathsf{otherParty}(V) = P$
   - $\gamma.\mathsf{cash} \geq \alpha$
   - $T_0 < T_1 < T_2$ are times in the future.
2) Send $(\mathtt{A\text{–}CREATED}, \mathsf{id}, \mathsf{id_{ch}}, T_0, T_1, T_2, \alpha, \alpha') \xhookrightarrow{\tau+1} P$.
3) At round $\tau_1 \leq \tau + 1 + \Delta_D$ if received $(\mathtt{CONTRACT\text{–}INCLUDED}, \mathsf{Addr}) \xhookleftarrow{\tau_1} \mathcal{S}$ write its address to Addr and add $(\mathsf{pk}_P, \mathsf{pk}_V, \mathtt{TX_f}, \mathsf{id_{Ch}}, T_0, T_1, T_2, \mathsf{Addr})$ in ϕ and continue otherwise stop.
4) Send $(\mathtt{A\text{–}CREATED}, \mathsf{id}, \mathsf{id_{ch}}, T_0, T_1, T_2, \alpha, \alpha') \xhookrightarrow{\tau_1} V$.
5) Set $\gamma.\mathsf{Aid} \leftarrow \mathsf{id}$.
6) Run the code below "Channel Update and Proof"

Channel Update and Proof

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 $f(\mathsf{st})$ on $\mathcal{L_D}$.
- $\mathbf{TX}_U$: Define transaction $\mathbf{TX}_U$ that transfers α from the smart contract with address Addr to $U \in \{P, V\}$ on the chain $\mathcal{L_D}$.
- $\mathbf{TX}_{init}$: Define transaction $\mathbf{TX}_{init}$ that distributes the α from the smart contract with address Addr according to the initial state init on the chain $\mathcal{L_D}$.
- variable t_{settle} specifies the time that we expect to see the transaction $\mathbf{TX}_{\mathsf{receiver}}$, $\mathsf{receiver} \in \gamma.\mathsf{users}$ with a maximum delay of Δ_D on $\mathcal{L}_D$.

Atomicity

1) Run $\mathtt{DetermineReceiver(id)}$ and wait for it to output $(\mathbf{TX}, t_{settle})$, where $\mathbf{TX} \in \{\mathbf{TX}_{payout, \mathbf{TX}_U}\}$, for $U \in \{P, V\}$.
2) Monitor $\mathcal{L}_D$, if a transaction which spends the money of the contract on Addr appears before t_{settle}, it has to be **TX**. Else, output $\mathtt{ERROR}$.
3) After receiving a message $(\mathtt{SETTLE}, \mathsf{id}) \xhookleftarrow{t < t_{settle} - \Delta_D} U$ from $U \in \{P, V\}$, the transaction **TX** should appear on $\mathcal{L}_D$ at time $t \leq t_{settle} + \Delta_D$, if this was not the case output $\mathtt{ERROR}$.
4) Else, after no such message and after $< t_{settle}$, a transaction $\mathbf{TX}_{init}$ may appear instead of **TX**.

Subprocedures

$\mathtt{DetermineReceiver(id)}$: ▷ This function returns receiver and t_{settle}

- **If** received $(\mathtt{A\text{–}INITIATE\text{–}PAYMENT}, \mathsf{id}, \vec{\theta}) \xhookleftarrow{t_0} P$ such that $t_0 < T_0 - \Delta_D$ do the followings: ▷ This means that P has initiated the update.
  Read $\phi(\mathsf{id})$ from the storage and parse it as $(\mathsf{pk}_P, \mathsf{pk}_V, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_1, T_2, \alpha, \alpha', \mathsf{Script}_G, \mathsf{Addr})$ and then read $\Gamma(\mathsf{id_{ch}}) = (\gamma, \mathtt{TX_f}, \_)$ from the storage of $\mathcal{PC}$. Let $\gamma.\mathsf{st}$ be the current state of the channel.
  1) Initiate-Payment message was late: If $(\gamma.\mathsf{idle} == False) \wedge (t_0 > T_0 - t_{updateDelay} - \Delta_D)$ then $\mathbf{Return}(\mathbf{TX}_V :, T_2)$.
  2) Channel is in the idle mode or no update: Else, if $\gamma.\mathsf{idle} == True$ then $\mathbf{Return}(\mathtt{HandleClaim(id)})$
  3) Channel is not in the idle mode Else, if $\gamma.\mathsf{idle} == False$ distinguish the following cases:▷ Check whether the Claim function was called or the proof was submitted.
     - SETUP message missing: If you received no message $(\mathtt{SETUP}, \mathsf{id_{ch}}, \mathsf{txid}, True) \xhookleftarrow{t_b} \mathcal{PC}$ for $t_b \leq t_0 + 2T$, then $\mathbf{Return}(\mathtt{HandleClaim(id)})$.
     - Proof was successfully submitted: Else, if you received a message $(\mathtt{UPDATE\text{–}OK}, \mathsf{id_{ch}}, True) \xhookleftarrow{t_c} \mathcal{PC}$ in the $\mathcal{PC}$ history for $t_c \leq t_0 + t_{updateDelay}$, then send $(\mathtt{SUBMIT\text{–}PROOF}, \mathsf{id}) \xhookrightarrow{t_c} P$, if received $(\mathtt{SUBMIT\text{–}PROOF}, \mathsf{id}) \xhookleftarrow{t_c} P$ then $\mathbf{Return}(\mathbf{TX}_{payout}, t_c)$ otherwise $\mathbf{Return}(\mathbf{TX}_V, T_2)$.
     - UPDATE-OK message missing: Else, if there was no message $(\mathtt{UPDATE\text{–}OK}, \mathsf{id_{ch}}, True) \xhookleftarrow{t_c} \mathcal{PC}$ for $t_c \leq t_0 + t_{updateDelay}$ then $\mathbf{Return}(\mathtt{HandleClaim(id)})$.
- **Else, if** no $\mathtt{A\text{–}INITIATE\text{–}PAYMENT}$ message was sent, before $t - \Delta_D$ then $\mathbf{Return}(\mathbf{TX}_V, T_2)$.

$\mathtt{HandleClaim(id)}$:▷ Returns receiver and t_{settle} in the case of a claim.
Send $(\mathtt{CALL\text{–}CLAIM}, \mathsf{id}) \xhookrightarrow{now} P$ and distinguish the following cases:

- **If** you received $(\mathtt{CALL\text{–}CLAIM\text{–}OK}, \mathsf{id}) \xhookleftarrow{now} P$: send $(\mathtt{RESPOND\text{–}CLAIM}, \mathsf{id}) \xhookrightarrow{\tau_a \leq now + \Delta_D} V$ and proceed as follows:
  1) If received $(\mathtt{RESPOND\text{–}CLAIM}, \mathsf{id}) \xhookleftarrow{\tau_a} V$ continue otherwise $\mathbf{Return}(\mathbf{TX}_P, T_2)$.▷ V being unresponsive
  2) Send $(\mathtt{FINALIZE\text{–}PROOF}, \mathsf{id}) \xhookrightarrow{t_d \leq \tau_a + \Delta_D} P$, if received $(\mathtt{FINALIZE\text{–}PROOF\text{–}OK}, \mathsf{id}) \xhookrightarrow{t_d} P$ $\mathbf{Return}(\mathbf{TX}_{payout}, t_d)$ otherwise $\mathbf{Return}(\mathbf{TX}_V, T_2)$
- **Else if** you received $(\mathtt{CALL\text{–}CLAIM\text{–}INVALID}, \mathsf{id}) \xhookleftarrow{t_b} P$ Send $(\mathtt{RESPOND\text{–}CLAIM\text{–}PUNISH}, \mathsf{id}) \xhookrightarrow{\tau_a \leq now + \Delta_D} V$, if received $(\mathtt{RESPOND\text{–}CLAIM\text{–}PUNISH\text{–}OK}, \mathsf{id}) \xhookleftarrow{\tau_a} V$, then $\mathbf{Return}(\mathbf{TX}_V, T_2)$ otherwise $\mathbf{Return}(\mathbf{TX}_P, T_2)$.▷ P submitted an invalid claim
- **Else if** you did not receive any message $\mathbf{Return}(\mathbf{TX}_V, T_2)$.▷ P did not claim.

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 Appendix A.2, 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.

Alba Smart Contract Pseudocode

```
Variables:
    • State
    • $\zeta_{init}$
    • $[\overline{\texttt{Tx}}_1^{\texttt{P}}], \sigma_V([\overline{\texttt{TX}}_{\texttt{c}}^{\texttt{P}}]), R_P$ ▷ Need to be stored if Claim() function was called.
    • $\sigma_P([\texttt{TX}_1^{\texttt{V}}])$
Constructor($\zeta_{init} := \{\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{id}, \texttt{TX}_{\texttt{f}}, \mathsf{id}_{\mathsf{ch}}, T_0, T_1, T_2, \alpha, \alpha'\}, \sigma_V(\zeta_{init})$):
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:
• State ← $\texttt{Init}$.
• Save $\zeta_{init}$ in the state.

SubmitProof($[\texttt{TX}_1^{\texttt{P}}], \sigma_V([\texttt{TX}_{\texttt{c}}^{\texttt{P}}]), [\texttt{TX}_1^{\texttt{V}}], \sigma_P([\texttt{TX}_1^{\texttt{V}}]), \bar{r}_P$):
Parse $\zeta_{init}$ as $\{\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{id}, \texttt{TX}_{\texttt{f}}, \mathsf{id}_{\mathsf{ch}}, T_0, T_1, T_2, \alpha, \alpha'\}$ and Check the following:
• $now < T_2$.
• State == $\texttt{Init}$
• $\mathsf{Vrfy}_{\mathsf{pk}_V}([\texttt{TX}_1^{\texttt{P}}]; \sigma_V([\texttt{TX}_{\texttt{c}}^{\texttt{P}}])) = 1$ and $\mathsf{Vrfy}_{\mathsf{pk}_P}([\texttt{TX}_1^{\texttt{V}}]; \sigma_P([\texttt{TX}_{\texttt{c}}^{\texttt{V}}])) = 1$
• $[\texttt{Tx}_1^{\texttt{V}}].\mathsf{In} = [\texttt{Tx}_1^{\texttt{P}}].\mathsf{In} = \texttt{TX}_{\texttt{f}}.\mathsf{txid} \| 1$
• $[\texttt{Tx}_1^{\texttt{V}}]$ and $[\texttt{Tx}_1^{\texttt{P}}]$ have the same balance distribution in their output. ▷ I.e., $[\texttt{Tx}_1^{\texttt{V}}]$ and $[\texttt{Tx}_1^{\texttt{P}}]$ are the outputs of functions $\texttt{GenCom}^P(.)$ and $\texttt{GenCom}^V_{Alba}(.)$ with the same $\vec{\theta}$.
• The revocation public key of $\texttt{Tx}_1^{\texttt{V}}$ is $R_V$, specified in the OP_RETURN output of $\texttt{Tx}_1^{\texttt{P}}$.
• $(\bar{R}_P, \bar{r}_P) \in R$ for $\bar{R}_P$ specified in the OP_RETURN output of $\texttt{Tx}_1^{\texttt{P}}$.
If all checks passed, then save $[\texttt{Tx}_1^{\texttt{P}}], [\texttt{Tx}_1^{\texttt{V}}]$ in the storage and set $t_{validProof} \leftarrow now$, State ← $\texttt{Valid\_Proof}$ otherwise abort.

Claim($[\overline{\texttt{Tx}}_1^{\texttt{P}}], \sigma_V([\overline{\texttt{TX}}_{\texttt{c}}^{\texttt{P}}]), R_P$)
Parse $\zeta_{init}$ as $\{\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{id}, \texttt{TX}_{\texttt{f}}, \mathsf{id}_{\mathsf{ch}}, T_0, T_1, T_2, \alpha, \alpha'\}$ and check the following:
• $now < T_0$
• State == $\texttt{Init}$
• $[\overline{\texttt{Tx}}_1^{\texttt{P}}].\mathsf{In} = \texttt{TX}_{\texttt{f}}.\mathsf{txid} \| 1$
• $\mathsf{Vrfy}_{\mathsf{pk}_V}([\overline{\texttt{Tx}}_1^{\texttt{P}}]; \sigma_V([\overline{\texttt{TX}}_{\texttt{c}}^{\texttt{P}}])) = 1$
• Parse $[\overline{\texttt{Tx}}_1^{\texttt{P}}].\mathsf{Out}$ as $(\theta_V, \{\theta_P.\mathsf{cash}, \varphi_P\}, \dots)$ and check whether $\theta_P.\mathsf{cash} \geq \alpha$.
If all checks passed, then save $\{[\overline{\texttt{Tx}}_1^{\texttt{P}}], \sigma_V([\overline{\texttt{TX}}_{\texttt{c}}^{\texttt{P}}]), R_P\}$ in the state and State ← $\texttt{Dispute}$, otherwise abort.

RespondClaim($\{\bar{r}_P\}$ OR $\{[\texttt{Tx}_1^{\texttt{P}}], \sigma_V([\texttt{TX}_{\texttt{c}}^{\texttt{P}}])\}$)
If State == $\texttt{Dispute}$ and $now < T_1$, retrieve $[\overline{\texttt{Tx}}_1^{\texttt{P}}]$ from the storage and do the following, otherwise abort:
• If $\bar{r}_P$ is provided, retrieve $\bar{R}_V, R_P$ from $[\overline{\texttt{Tx}}_1^{\texttt{P}}].\mathsf{Out}$ and check whether $(\bar{R}_P, \bar{r}_P) \in R$. If this is the case, then State ← $\texttt{Invalid\_Claim}$, otherwise abort.
• Else, if $\{\texttt{Tx}_1^{\texttt{P}}, \sigma_V([\texttt{TX}_{\texttt{c}}^{\texttt{P}}])\}$ is provided then check the followings:
    – $\mathsf{Vrfy}_{\mathsf{pk}_V}([\texttt{Tx}_1^{\texttt{P}}]; \sigma_V([\texttt{TX}_{\texttt{c}}^{\texttt{P}}])) = 1$
    – $[\texttt{Tx}_1^{\texttt{P}}].\mathsf{In} = \texttt{TX}_{\texttt{f}}.\mathsf{txid} \| 1$
    – Parse $[\overline{\texttt{Tx}}_1^{\texttt{P}}].\mathsf{Out}$ and $[\texttt{Tx}_1^{\texttt{P}}].\mathsf{Out}$ respectively as $(\theta_V^0, \{\theta_P^0.\mathsf{cash}, \varphi_P\}, \dots)$, $(\theta_V^1, \{\theta_P^1.\mathsf{cash}, \varphi_P\}, \dots)$ and check whether $\theta_P^1.\mathsf{cash} = \theta_P^0.\mathsf{cash} - \alpha$ and $\theta_V^1.\mathsf{cash} = \theta_V^0.\mathsf{cash} + \alpha$.
    – Conditional outputs of $\texttt{Tx}_1^{\texttt{P}}$ are locked with $R_P$ saved in the storage.
    – $\texttt{Tx}_1^{\texttt{P}}.\mathsf{Out}$ should be well-formed based on the format specified in the $\texttt{GenCom}^V$ function where $\texttt{AUpdate} = True, \mathsf{ADeadline} = T_2$
  If all checks passed, save $\{[\texttt{Tx}_1^{\texttt{P}}], \sigma_P([\texttt{TX}_{\texttt{c}}^{\texttt{V}}])\}$ and set State ← $\texttt{Dispute\_Resolved}$ otherwise abort.

FinalizeProof($\bar{r}_P, [\texttt{Tx}_1^{\texttt{V}}], \sigma_P([\texttt{TX}_{\texttt{c}}^{\texttt{V}}])$)
Retrieve $[\texttt{Tx}_1^{\texttt{P}}]$ from the storage and check the followings:
• $now < T_2$
• State == $\texttt{Dispute\_Resolved}$
• $\mathsf{Vrfy}_{\mathsf{pk}_P}([\texttt{TX}_1^{\texttt{V}}]; \sigma_P([\texttt{TX}_{\texttt{c}}^{\texttt{V}}])) = 1$
• $[\texttt{Tx}_1^{\texttt{V}}].\mathsf{In} = \texttt{TX}_{\texttt{f}}.\mathsf{txid} \| 1$
• $\texttt{Tx}_1^{\texttt{V}}$ and $\texttt{Tx}_1^{\texttt{P}}$ have the same balance distribution in their Out.
• The revocation public key of $\texttt{Tx}_1^{\texttt{V}}$ is $R_V$, specified in the OP_RETURN output of $\texttt{Tx}_1^{\texttt{P}}$.
• $(\bar{R}_P, \bar{r}_P) \in R$ for $\bar{R}_P$ that $\overline{\texttt{Tx}}_1^{\texttt{P}}.\mathsf{Out}$ is locked with.
If all checks passed, then save $[\texttt{Tx}_1^{\texttt{V}}]$ in the storage and set $t_{validProof} \leftarrow now$, State ← $\texttt{Valid\_Proof}$, otherwise abort.

Settle()
• If State == $\texttt{Valid\_Proof}$ then transfer $\alpha'$ to $P$ and set State ← $\texttt{Transferred\_P}$.
• Else if $now \geq T_2$ Check the following otherwise abort.
    – If State == $\texttt{Claim}$ then transfer $\alpha'$ to $P$ and set State ← $\texttt{Transferred\_P}$
    – Else If State == $\texttt{Invalid\_Claim}$ then transfer $\alpha$ to $V$ and set State ← $\texttt{Transferred\_V}$
    – Else if State == $\texttt{Init}$ or State == $\texttt{Claim\_Resolved}$ then transfer $\alpha'$ to $V$ and set State ← $\texttt{Transferred\_V}$.
```

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 $\mathcal{G}_{clock}$.

Alba Protocol: Parties Interacting with the Smart Contract

```
Parameters:
$\mathcal{L_S}, \mathcal{L_D}$ ... two instances of $\mathcal{G}_{Ledger}$ representing the source and destination blockchain.
$\mathcal{PC_S}$ ... an instance of the payment channel ideal functionality.

Variables:
$\phi(.)$ ... a mapping of Alba ID id to $(\mathsf{pk}_P, \mathsf{pk}_V, \texttt{TX}_{\texttt{f}}, \mathsf{id}_{\mathsf{Ch}}, T_0, T_1, T_2, t_{wait}, \mathsf{Script}_G, \mathsf{Addr})$ where id $\in \{0,1\}^*$ is an identifier unique to the pair $P$ and $V$. $\texttt{TX}_{\texttt{f}}$ is the funding transaction of the channel and $\mathsf{id}_{\mathsf{ch}}$ is the id of the payment channel. Further $T_0 < T_1 < T_2, t_{wait} \in \mathbb{N}$. $\mathsf{Script}_G$ is the code of the Alba contract and Addr is the address of the instance of the Alba contract.

                                   Setup

Party $V$ upon $(\texttt{A-SETUP}, \mathsf{id}, \mathsf{id}_{\mathsf{ch}}, P, T_0, T_1, T_2, \alpha, \alpha') \overset{\tau}{\hookleftarrow} \mathcal{Z}$
1) Read $\Gamma^V(\mathsf{id}_{\mathsf{ch}})$ and parse the output as $(\{\gamma\}, \texttt{TX}_{\texttt{f}}, \dots)$, if $\Gamma^V(\mathsf{id})$ returned $\bot$ go idle otherwise check the followings:
    • $\gamma.\mathsf{otherParty}(V) = P$
    • $\gamma.\mathsf{cash} \geq \alpha$
    • $T_0 < T_1 < T_2$ are times in the future.
```

2) If all of the above checks hold, define $\zeta_{init} := \{\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{id}, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_1, T_2, t_{wait}, \alpha, \alpha'\}$, sign it $\sigma_V(\zeta_{init}) := \mathsf{Sign}_{\mathsf{sk}_V}(\zeta_{init})$, and then send (GSetupInfo, $\zeta_{init}, \sigma_V(\zeta_{init})) \overset{\tau}{\hookrightarrow} P$.

<u>Party P upon (GSetupInfo, $\zeta_{init}, \sigma_V(\zeta_{init})) \overset{t}{\hookleftarrow} V$</u>

3) Parse ζ_{init} as $\{\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{id}, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_1, T_2, t_{wait}, \alpha, \alpha'\}$. Read $\Gamma^P(\mathsf{id_{ch}})$ and parse the output as $(\{\gamma\}, \mathtt{TX_f}, \dots)$, if $\Gamma^P(\mathsf{id_{ch}})$ returned $\perp$ go idle otherwise check the followings:
   - $\gamma.\mathsf{otherParty}(P) = V$
   - $\zeta_{init}.\mathtt{TX_f} = \gamma.\mathtt{TX_f}$
   - $\gamma.\mathsf{cash} \geq \alpha$
   - $T_0 < T_1 < T_2$ are times in the future.
   - $\mathsf{Vrfy}_{\mathsf{pk}_V}(\zeta_{init}; \sigma_V(\zeta_{init})) = 1$
4) If the above checks passed, send $(\mathtt{A\text{–}CREATED}, \mathsf{id}, \mathsf{id_{ch}}, T_0, T_1, T_2, t_{wait}, \alpha, \alpha') \overset{t}{\hookrightarrow} \mathcal{Z}$ and send (InitiateContract, $code$ = Script_G, $function$ = 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 $\phi(\mathsf{id}) = (\mathsf{pk}_P, \mathsf{pk}_V, \theta_{P,V}, \mathsf{id_{ch}}, T_0, T_1, T_2, t_{wait}, \alpha, \alpha', \mathsf{Script}_G, \mathsf{Addr})$, set $\gamma.\mathsf{Aid} \leftarrow \mathsf{id}, \gamma.\mathsf{ADeadline} \leftarrow T_2$.

<u>Party V at time $\tau + 1$</u>

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 $\phi(\mathsf{id}) = (\mathsf{pk}_P, \mathsf{pk}_V, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_1, T_2, \alpha, \alpha', \mathsf{Script}_G, \mathsf{Addr})$ and continue otherwise stop.
8) Send $(\mathtt{A\text{–}CREATED}, \mathsf{id}, \mathsf{id_{ch}}, T_0, T_1, T_2, t_{wait}, \alpha, \alpha') \overset{\tau_1}{\hookrightarrow} \mathcal{Z}$
9) Set $\gamma.\mathsf{Aid} \leftarrow \mathsf{id}$ and $\gamma.\mathsf{ADeadline} \leftarrow T_2$ and run $\mathtt{MonitorDestLedger}^V(\mathsf{id})$.

Channel Update and Proof

<u>Party P upon $(\mathtt{A\text{–}INITIATE\text{–}PAYMENT}, \mathsf{id}, \vec{\theta}) \overset{t_0}{\hookleftarrow} \mathcal{Z}$</u>

1) Read $\phi(\mathsf{id})$ from the storage and parse it as $(\mathsf{pk}_P, \mathsf{pk}_V, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_1, T_2, t_{wait}, \alpha, \alpha', \mathsf{Script}_G, \mathsf{Addr})$ and then read $\Gamma(\mathsf{id_{ch}})$ as $(\gamma, \mathtt{TX_f})$ from $\mathcal{PC_S}$.
2) 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 $\mathtt{CallClaimFunc}(\mathsf{id})$, and then stop.
3) 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.
4) Send $(\mathtt{UPDATE}, \mathsf{id_{ch}}, \vec{\theta}, True) \overset{t_0}{\hookrightarrow} \mathcal{PC}$.
5) If you received $(\mathtt{SETUP}, \mathsf{id_{ch}}, \mathsf{txid}, True) \overset{t_0+2}{\hookleftarrow} \mathcal{PC}$ send $(\mathtt{SETUP\text{–}OK}, \mathsf{id_{ch}}, True) \xhookrightarrow{t_1 \leq t_0+2+t_{\mathsf{stp}}} \mathcal{PC}$. Otherwise, if you did not receive the $\mathtt{SETUP}$ message, run $\mathtt{ForceClose}^P(\mathsf{id_{ch}})$. Run $\mathtt{CallClaimFunc}(\mathsf{id})$ and stop.

<u>Party V upon receiving $(\mathtt{SETUP\text{–}OK}, \mathsf{id_{ch}}, True) \overset{\tau_1}{\hookleftarrow} \mathcal{PC}$</u>

6) Send $(\mathtt{UPDATE\text{–}OK}, \mathsf{id_{ch}}, True) \overset{\tau_1}{\hookrightarrow} \mathcal{PC}$

<u>Party P at time $t_2 = t_1 + 2$</u>

7) If received $(\mathtt{UPDATE\text{–}OK}, \mathsf{id_{ch}}, True) \overset{t_2}{\hookleftarrow} \mathcal{PC}$, then :
   - Send $(\mathtt{REVOKE}, \mathsf{id_{ch}}, True) \overset{t_2}{\hookrightarrow} \mathcal{PC}$.
   - Read $\Gamma^P(\mathsf{id_{ch}})$ from the $\mathcal{S_{PC}}$'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 $([\mathtt{TX}_1^{\mathtt{P}}], \{\sigma_P(\mathtt{TX}_1^{\mathtt{P}}), \sigma_V([\mathtt{TX}_\mathtt{c}^{\mathtt{P}}])\})$ and $\mathtt{TX}_1^{\mathtt{V}}$ as $([\mathtt{TX}_1^{\mathtt{V}}], \sigma_P([\mathtt{TX}_\mathtt{c}^{\mathtt{V}}]))$.
   - Send $(\mathtt{SUBMIT\text{–}PROOF}, \mathsf{id}) \overset{t_2}{\hookrightarrow} \mathcal{Z}$, if you received $(\mathtt{SUBMIT\text{–}PROOF\text{–}OK}, \mathsf{id}) \overset{t_2}{\hookleftarrow} \mathcal{Z}$ continue otherwise go idle.
   - Send (CallFunction, $function$ = SubmitProof, $args$ = $\{[\mathtt{TX}_1^{\mathtt{P}}], \sigma_V([\mathtt{TX}_\mathtt{c}^{\mathtt{P}}]), [\mathtt{TX}_1^{\mathtt{V}}], \sigma_P([\mathtt{TX}_\mathtt{c}^{\mathtt{V}}]), \bar{r}_P\}) \overset{t_2}{\hookrightarrow} \mathcal{L}_D$.
8) Else if did not receive the $\mathtt{UPDATE\text{–}OK}$ message then run $\mathtt{ForceClose}^P(\mathsf{id_{ch}})$. Run $\mathtt{CallClaimFunc}(\mathsf{id})$ and stop. ▷ In this case P has not received the V's commitment transaction in time. By calling Force Close, we command P_{pc} to go idle in the $\Pi_{\mathcal{PC}}$ and wait for the lifetime of the Alba to end and then close the channel.

<u>Party V Upon $(\mathtt{REVOKE\text{–}REQ}, \mathsf{id_{ch}}, True) \overset{\tau_1+2}{\hookleftarrow} \mathcal{PC}$</u>

9) Send $(\mathtt{REVOKE}, \mathsf{id_{ch}}, True) \overset{\tau_1+2}{\hookrightarrow} \mathcal{PC}$

<u>Party P at time $t_2 = t_1 + 2 + \Delta_D + t_{wait}$</u>

10) If the $Settle()$ function of the contract is not already called, post (CallFunction, $function$ = Settle) $\overset{t_2}{\hookrightarrow} \mathcal{L}_D$.

Subprocedures

$\mathtt{CallClaimFunc}(\mathsf{id})$:

- Send $(\mathtt{CALL\text{–}CLAIM}, \mathsf{id}) \overset{now}{\hookrightarrow} \mathcal{Z}$, if you received $(\mathtt{CALL\text{–}CLAIM\text{–}OK}, \mathsf{id}) \overset{now}{\hookleftarrow} \mathcal{Z}$ continue otherwise go idle.
- Parse $\phi(\mathsf{id})$ as $(\mathsf{pk}_P, \mathsf{pk}_V, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_1, T_2, t_{wait}, \alpha, \alpha', \mathsf{Script}_G, \mathsf{Addr})$.
- Read $\Gamma^P(\mathsf{id_{ch}})$ from the storage of $\mathcal{S_{PC}}$ and parse it as $(\gamma, \mathtt{TX_f}, (\overline{\mathtt{TX}}_1^{\mathtt{V}}, \overline{\mathtt{TX}}_1^{\mathtt{P}}, \dots))$, moreover, parse $\overline{\mathtt{TX}}_1^{\mathtt{P}}$ as $([\overline{\mathtt{TX}}_1^{\mathtt{P}}], \{\mathsf{Sign}_{\mathsf{sk}_P}([\overline{\mathtt{TX}}_1^{\mathtt{P}}]), \sigma_V([\overline{\mathtt{TX}}_\mathtt{c}^{\mathtt{P}}])\})$
- 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.
- Send (CallFunction, $function$ = Claim, $args$ = $\{[\overline{\mathtt{Tx}}_1^{\mathtt{P}}], \sigma_V([\overline{\mathtt{TX}}_\mathtt{c}^{\mathtt{P}}]), R_P\}) \overset{now}{\hookrightarrow} \mathcal{L}_D$
- Run the $\mathtt{MonitorDestLedger}_P(\mathsf{id})$ function.

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

1) Parse $\phi(\mathsf{id})$ as $(\mathsf{pk}_P, \mathsf{pk}_V, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_1, T_2, t_{wait}, \alpha, \alpha', \mathsf{Addr})$.
2) If $now \leq T_2$: ▷ The proof might be exactly posted at T_2 and then Settle function can be called after $T_2 + t_{wait}$.
   - Send (read, $address$ = Addr, $variable$ = State) $\overset{now}{\hookrightarrow} \mathcal{L}_D$, save the output in State.
   - If State == $\mathtt{Dispute\_Resolved}$, then send (read, $address$ = Addr, $variable$ = $[\mathtt{TX}_1^{\mathtt{P}}], R_P) \overset{now}{\hookrightarrow} \mathcal{L}_D$, save the output in $[\mathtt{TX}_1^{\mathtt{P}}]$ and do the followings:
     - Send $(\mathtt{FINALIZE\text{–}PROOF}, \mathsf{id}) \overset{now}{\hookrightarrow} \mathcal{Z}$, if you received $(\mathtt{FINALIZE\text{–}PROOF\text{–}OK}, \mathsf{id}) \overset{now}{\hookleftarrow} \mathcal{Z}$ continue otherwise go idle.
     - Read $\Gamma^P(\mathsf{id_{ch}})$ from the storage of $\mathcal{S_{PC}}$ and extract $\bar{r}_P$ from it.▷ corresponding to the same commitment transaction that was used to call the Claim function.
     - Extract $\vec{\theta}$ and R_V from $\mathtt{TX}_1^{\mathtt{P}}.\mathsf{Out}$.
     - Create $[\mathtt{TX}_1^{\mathtt{V}}] := \mathtt{GenCom}^P([\mathtt{TX_f}], (\mathsf{pk}_P), (\mathsf{pk}_V, R_V), \vec{\theta})$.
     - Sign the transaction $\sigma_P([\mathtt{TX}_\mathtt{c}^{\mathtt{V}}]) := \mathsf{Sign}_{\mathsf{sk}_P}([\mathtt{TX}_1^{\mathtt{V}}])$.
     - Send (CallFunction, $function$ = FinalizeProof, $args$ = $\{\bar{r}_P, [\mathtt{Tx}_1^{\mathtt{V}}], \sigma_P([\mathtt{TX}_\mathtt{c}^{\mathtt{V}}])\}) \overset{now}{\hookrightarrow} \mathcal{L}_D$.
     - Read $\Gamma^P(\mathsf{id_{ch}})$ from the storage of $\mathcal{S_{PC}}$ and Parse it as $(\gamma, \mathtt{TX_f}, (\overline{\mathtt{TX}}_1^{\mathtt{V}}, \overline{\mathtt{TX}}_1^{\mathtt{P}} \bar{r}_P, \bar{R}_V, \dots))$. Update the $\mathcal{S_{PC}}$'s channel space as $\Gamma^P(\mathsf{id_{ch}}) := (\gamma, \mathtt{TX_f}, (\mathtt{TX}_1^{\mathtt{P}}, \mathtt{TX}_1^{\mathtt{V}}, r_P, R_V, \bar{r}_P))$▷ 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 ($\mathsf{State} == \mathtt{Valid\_Proof}$) and if the contract is not already settled send (CallFunction, $function$ = Settle) $\xhookrightarrow{now} \mathcal{L}_D$. Return.
- Else if $now == T_2$ and the contract is not already settled send (CallFunction, $function$ = Settle) $\xhookrightarrow{now} \mathcal{L}_D$. Return.

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

1) Parse $\phi(\mathsf{id})$ as $(\mathsf{pk}_P, \mathsf{pk}_V, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_1, T_2, t_{wait}, \alpha, \alpha', \mathsf{Addr})$.
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) $\xhookrightarrow{now} \mathcal{L}_D$, save the output in State.
   - If $\mathsf{State} == \mathtt{Dispute}$, then send (read, $address$ = Addr, $variable = [\overline{\mathtt{TX}}_1^{\mathtt{P}}], R_P$) $\xhookrightarrow{now} \mathcal{L}_D$, save the output in $[\overline{\mathtt{TX}}_1^{\mathtt{P}}], R_P$ 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 $\overline{\mathtt{TX}}_1^{\mathtt{P}}$ existed in the channel history $\Theta^V(\mathsf{id_{ch}})$ of $\mathcal{S}_{\mathcal{PC}}$ as $([\overline{\mathtt{TX}}_1^{\mathtt{P}}], \bar{r}_P, \bar{r}_V, \bar{s}_c^P)$, then first send $(\mathtt{RESPOND{-}CLAIM{-}PUNISH}, \mathsf{id}) \xhookrightarrow{now} \mathcal{Z}$, if you received $(\mathtt{RESPOND{-}CLAIM{-}PUNISH{-}OK}, \mathsf{id}) \xhookleftarrow{now} \mathcal{Z}$ continue otherwise stop. Send (CallFunction, $function$ = RespondClaim, $args = \{\bar{r}_P\}$) $\xhookrightarrow{now} \mathcal{L}_D$.
     - Else, extract $\vec{\theta}$ and $\bar{R}_P$ from $[\overline{\mathtt{TX}}_1^{\mathtt{P}}]$.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 $[\mathtt{TX}_1^{\mathtt{P}}] := \mathtt{GenCom}^V_{Alba}([\mathtt{TX_f}], (\mathsf{pk}_P, R_P), (\mathsf{pk}_V, R_V), (\bar{R}_P), \vec{\theta}, \mathsf{id})$ ▷ $[\overline{\mathtt{TX}}_1^{\mathtt{P}}]$ submitted by P is either a legitimate latest state which is saved in $\Gamma^V(\mathsf{id_{ch}})$ 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 Send then first send $(\mathtt{RESPOND{-}CLAIM}, \mathsf{id}) \xhookrightarrow{now} \mathcal{Z}$, if you received $(\mathtt{RESPOND{-}CLAIM}, \mathsf{id}) \xhookleftarrow{now} \mathcal{Z}$ continue otherwise stop. Sign the transaction $\sigma_V([\mathtt{TX}_c^{\mathtt{P}}]) := \mathsf{Sign}_{\mathsf{sk}_V}([\mathtt{TX}_1^{\mathtt{P}}])$. Then send (CallFunction, $function$ = RespondClaim, $args = \{[\mathtt{TX}_1^{\mathtt{P}}], \sigma_V([\mathtt{TX}_c^{\mathtt{P}}])\}$) $\xhookrightarrow{now} \mathcal{L}_D$ and set $updateChanSpace \leftarrow True$.
   - Else if ($\mathsf{State} == \mathtt{Valid\_Proof}$) do the following:
     - If ($updateChanSpace == True$) then Send (read, $address$ = Addr, $variable = [\mathtt{TX}_1^{\mathtt{V}}], [\mathtt{TX}_1^{\mathtt{P}}], \sigma_P([\mathtt{TX}_c^{\mathtt{V}}])$) $\xhookrightarrow{now} \mathcal{L}_D$. Retrieve r_V from the local storage, reprieve $\bar{r}_V$ from $\Theta^V(\mathsf{id_{ch}})$ stored in $\mathcal{S}_{\mathcal{PC}}$ and update $\Gamma^V(\mathsf{id_{ch}}) := (\gamma, \mathtt{TX_f}, (\mathtt{TX}_1^{\mathtt{P}}, \mathtt{TX}_1^{\mathtt{V}}, r_V, R_P, \bar{r}_V))$. Then set $updateChanSpace \leftarrow False$. ▷ In the case of an On-chain update, the channel space should be updated here because probably V is in the idle mode.
     - If the contract is not settled yet, send (CallFunction, $function$ = Settle) $\xhookrightarrow{now} \mathcal{L}_D$.
   - Else if $now == T_2$ and if the contract is not settled yet, send (CallFunction, $function$ = Settle) $\xhookrightarrow{now} \mathcal{L}_D$.

Appendix D. UC Proof

Setup

Case P honest, V dishonest

1) Upon V sending (GSetupInfo, $\zeta_{init}, \sigma_V(\zeta_{init})$) $\xhookrightarrow{\tau} P$, send $(\mathtt{A{-}SETUP}, \mathsf{id}, \mathsf{id_{ch}}, P, T_0, T_1, T_2, \alpha, \alpha') \xhookrightarrow{\tau} \mathcal{F}_{\mathsf{Alba}}$ on behalf of V and do the following.
2) Parse ζ_{init} as $\{\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{id}, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_1, T_2, t_{wait}, \alpha, \alpha'\}$. Read $\Gamma^P(\mathsf{id_{ch}})$ and parse the output as $(\{\gamma\}, \mathtt{TX_f}, \dots)$, if $\Gamma^P(\mathsf{id_{ch}})$ returned $\bot$ go idle otherwise check the followings:
   - $\gamma.\mathsf{otherParty}(P) = V$
   - $\zeta_{init}.\mathtt{TX_f} = \gamma.\mathtt{TX_f}$
   - $\gamma.\mathsf{cash} \geq \alpha$
   - $T_0 < T_1 < T_2$ are times in the future.
   - $\mathsf{Vrfy}_{\mathsf{pk}_V}(\zeta_{init}; \sigma_V(\zeta_{init})) = 1$
3) If the above checks passed, send (InitiateContract, $code$ = Script_G, $function$ = Constructor, $args$ = $\{\zeta_{init}, \sigma_V(\zeta_{init})\}$) $\xhookrightarrow{t:=\tau+1} \mathcal{L}_D$,
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 $(\mathtt{CONTRACT{-}INCLUDED}, \mathsf{Addr}) \xhookrightarrow{t_1} \mathcal{F}_{\mathsf{Alba}}$ write its address to Addr and add $(\mathsf{pk}_P, \mathsf{pk}_V, \mathtt{TX_f}, \mathsf{id_{Ch}}, T_0, T_1, T_2, \mathsf{Addr})$ in ϕ and continue otherwise stop.
6) Then store $\phi(\mathsf{id}) = (\mathsf{pk}_P, \mathsf{pk}_V, \theta_{P,V}, \mathsf{id_{ch}}, T_0, T_1, T_2, t_{wait}, \alpha, \alpha', \mathsf{Script}_G, \mathsf{Addr})$, set $\gamma.\mathsf{Aid} \leftarrow \mathsf{id}$, $\gamma.\mathsf{ADeadline} \leftarrow T_2$.

Case V honest, P dishonest

1) Upon V sending $(\mathtt{A{-}SETUP}, \mathsf{id}, \mathsf{id_{ch}}, P, T_0, T_1, T_2, \alpha, \alpha') \xhookrightarrow{\tau} \mathcal{F}_{\mathsf{Alba}}$
2) Read $\Gamma^V(\mathsf{id_{ch}})$ and parse the output as $(\{\gamma\}, \mathtt{TX_f}, \dots)$, if $\Gamma^V(\mathsf{id})$ returned $\bot$ go idle otherwise check the followings:
   - $\gamma.\mathsf{otherParty}(V) = P$
   - $\gamma.\mathsf{cash} \geq \alpha$
   - $T_0 < T_1 < T_2$ are times in the future.
3) If all of the above checks hold, define $\zeta_{init} := \{\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{id}, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_1, T_2, t_{wait}, \alpha, \alpha'\}$, sign it $\sigma_V(\zeta_{init}) := \mathsf{Sign}_{\mathsf{sk}_V}(\zeta_{init})$, and then send (GSetupInfo, $\zeta_{init}, \sigma_V(\zeta_{init})$) $\xhookrightarrow{\tau} P$.
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 $\phi(\mathsf{id}) = (\mathsf{pk}_P, \mathsf{pk}_V, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_1, T_2, \alpha, \alpha', \mathsf{Script}_G, \mathsf{Addr})$ and continue otherwise stop.
5) Send $(\mathtt{CONTRACT{-}INCLUDED}, \mathsf{Addr}) \xhookrightarrow{t_1} \mathcal{F}_{\mathsf{Alba}}$
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, V dishonest

1) Upon P sending $(\mathtt{A{-}INITIATE{-}PAYMENT}, \mathsf{id}) \xhookrightarrow{t_0} \mathcal{F}_{\mathsf{Alba}}$
   a) Read $\phi(\mathsf{id})$ from the storage and parse it as $(\mathsf{pk}_P, \mathsf{pk}_V, \mathtt{TX_f}, \mathsf{id_{ch}}, T_0, T_1, T_2, t_{wait}, \alpha, \alpha', \mathsf{Script}_G, \mathsf{Addr})$ and then read $\Gamma^P(\mathsf{id_{ch}}) = (\gamma, \mathtt{TX_f}, \dots)$.
   b) Define vector $\{bal_P, bal_V\} := \gamma.\mathsf{bal}$; If $bal_P < \alpha$ go idle; Otherwise continue.
   c) 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 $\mathtt{CallClaimFunc}(\mathsf{id})$, and then stop.
   d) If ($\gamma.\mathsf{idle} == False$) and ($t_0 > T_0 - \Delta_D - (4 + t_{\mathsf{stp}})$) stop, otherwise if ($t_0 \leq T_0 - \Delta_D - (4 + t_{\mathsf{stp}})$) continue.
   e) Define $\vec{\theta} := (\{bal_P - \alpha, \mathtt{One{-}Sig}_{\mathsf{pk}_P}\}, \{bal_V + \alpha, \mathtt{One{-}Sig}_{\mathsf{pk}_V}\})$ and send $(\mathtt{UPDATE}, \mathsf{id_{ch}}, \vec{\theta}, True) \xhookrightarrow{t_0} \mathcal{PC}$.
   f) If you received $(\mathtt{SETUP}, \mathsf{id_{ch}}, \mathsf{txid}, True) \xhookleftarrow{t_0+2} \mathcal{PC}$ send $(\mathtt{SETUP{-}OK}, \mathsf{id_{ch}}, True) \xhookrightarrow{t_1 \leq t_0+2+t_{\mathsf{stp}}} \mathcal{PC}$. Other-

wise, if you did not receive the $\texttt{SETUP}$ message, run $\texttt{ForceClose}^P(\mathsf{id_{ch}})$. Run $\texttt{CallClaimFunc}(\mathsf{id})$ and stop. $\triangleright$ In this case, P has not received the updateInfo message in time.

2) Else, if P does not send a message $(\texttt{A-INITIATE-PAYMENT}, \mathsf{id}) \xhookrightarrow{t_0} \mathcal{F}_{\mathsf{Alba}}$ by time $t_0 \leq t - \Delta_D$.

Case V honest, P dishonest

1) Upon V receiving $(\texttt{SETUP-OK}, \mathsf{id_{ch}}, True) \xhookleftarrow{\tau_1} \mathcal{PC}$, send $(\texttt{UPDATE-OK}, \mathsf{id_{ch}}, True) \xhookrightarrow{\tau_1} \mathcal{PC}$
2) Upon V receiving $(\texttt{REVOKE-REQ}, \mathsf{id_{ch}}, True) \xhookleftarrow{\tau_1+2} \mathcal{PC}$, send $(\texttt{REVOKE}, \mathsf{id_{ch}}, True) \xhookrightarrow{\tau_1+2} \mathcal{PC}$

Simulator subprocedures

$\texttt{CallClaimFunc}(\mathsf{id})$:

- Upon P sending $(\texttt{CALL-CLAIM-OK}, \mathsf{id}) \xhookrightarrow{\tau} \mathcal{F}_{\mathsf{Alba}}$, do the following.
- Parse $\phi(\mathsf{id})$ as $(\mathsf{pk}_P, \mathsf{pk}_V, \texttt{TX}_\texttt{f}, \mathsf{id_{ch}}, T_0, T_1, T_2, t_{wait}, \alpha, \alpha', \mathsf{Script}_G, \mathsf{Addr})$.
- Read $\Gamma^P(\mathsf{id_{ch}})$ from the storage of $\mathcal{S}_{\mathcal{PC}}$ and parse it as $(\gamma, \texttt{TX}_\texttt{f}, (\overline{\texttt{TX}}^{\texttt{V}}_{\texttt{1}}, \overline{\texttt{TX}}^{\texttt{P}}_{\texttt{1}}, \dots))$, moreover, parse $\overline{\texttt{TX}}^{\texttt{P}}_{\texttt{1}}$ as $([\overline{\texttt{TX}}^{\texttt{P}}_{\texttt{1}}], \{\mathsf{Sign}_{\mathsf{sk}_P}([\overline{\texttt{TX}}^{\texttt{P}}_{\texttt{1}}]), \sigma_V([\overline{\texttt{TX}}^{\texttt{P}}_{\texttt{c}}])\})$
- Generate $(R_P, r_P) \leftarrow \mathsf{GenR}$. $\triangleright$ Even if we already created a revocation key before and exchanged commitment transactions with it, we don't care about it anymore because if this function is called, P does not have any signed (valid) commitment transaction from V.
- Send $(\text{CallFunction}, function = \text{Claim}, args = \{[\overline{\texttt{Tx}}^{\texttt{P}}_{\texttt{1}}], \sigma_V([\overline{\texttt{TX}}^{\texttt{P}}_{\texttt{c}}]), R_P\}) \xhookrightarrow{\tau} \mathcal{L}_D$
- Run the $\texttt{MonitorDestLedger}_P(\mathsf{id})$ function.

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

1) Let now be the current round.
2) Parse $\phi(\mathsf{id})$ as $(\mathsf{pk}_P, \mathsf{pk}_V, \texttt{TX}_\texttt{f}, \mathsf{id_{ch}}, T_0, T_1, T_2, t_{wait}, \alpha, \alpha', \mathsf{Addr})$.
3) If $now \leq T_2$: $\triangleright$ The proof might be exactly posted at T_2 and then Settle function can be called after $T_2 + t_{wait}$.
   - Send $(\text{read}, address = \mathsf{Addr}, variable = \mathsf{State}) \xhookrightarrow{now} \mathcal{L}_D$, save the output in State.
   - If $\mathsf{State} == \texttt{Dispute\_Resolved}$, then send $(\text{read}, address = \mathsf{Addr}, variable = [\texttt{TX}^{\texttt{P}}_{\texttt{1}}], R_P) \xhookrightarrow{now} \mathcal{L}_D$, save the output in $[\texttt{TX}^{\texttt{P}}_{\texttt{1}}]$ and do the followings:
     - If P sends $(\texttt{FINALIZE-PROOF-OK}, \mathsf{id}) \xhookrightarrow{now} \mathcal{F}_{\mathsf{Alba}}$ continue, otherwise go idle.
     - Read $\Gamma^P(\mathsf{id_{ch}})$ from the storage of $\mathcal{S}_{\mathcal{PC}}$ and extract $\bar{r}_P$ from it.$\triangleright$ corresponding to the same commitment transaction that was used to call the Claim function.
     - Extract $\vec{\theta}$ and R_V from $\texttt{TX}^{\texttt{P}}_{\texttt{1}}.\mathsf{Out}$.
     - Generate $[\texttt{TX}^{\texttt{V}}_{\texttt{1}}] := \texttt{GenCom}^P([\texttt{TX}_\texttt{f}], (\mathsf{pk}_P), (\mathsf{pk}_V, R_V), \vec{\theta})$.
     - Sign the transaction $\sigma_P([\texttt{TX}^{\texttt{V}}_{\texttt{c}}]) := \mathsf{Sign}_{\mathsf{sk}_P}([\texttt{TX}^{\texttt{V}}_{\texttt{1}}])$.
     - Send $(\text{CallFunction}, function = \text{FinalizeProof}, args = \{\bar{r}_P, [\texttt{Tx}^{\texttt{V}}_{\texttt{1}}], \sigma_P([\texttt{TX}^{\texttt{V}}_{\texttt{c}}])\}) \xhookrightarrow{now} \mathcal{L}_D$.
     - Read $\Gamma^P(\mathsf{id_{ch}})$ from the storage of $\mathcal{S}_{\mathcal{PC}}$ and Parse it as $(\gamma, \texttt{TX}_\texttt{f}, (\overline{\texttt{TX}}^{\texttt{V}}_{\texttt{1}}, \overline{\texttt{TX}}^{\texttt{P}}_{\texttt{1}} \bar{r}_P, \bar{R}_V, \dots))$. Update the $\mathcal{S}_{\mathcal{PC}}$'s channel space as $\Gamma^P(\mathsf{id_{ch}}) := (\gamma, \texttt{TX}_\texttt{f}, (\texttt{TX}^{\texttt{P}}_{\texttt{1}}, \texttt{TX}^{\texttt{V}}_{\texttt{1}}, r_P, R_V, \bar{r}_P))$$\triangleright$ 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 $(\mathsf{State} == \texttt{Valid\_Proof})$ and if the contract is not already settled send $(\text{CallFunction}, function = \text{Settle}) \xhookrightarrow{now} \mathcal{L}_D$. Return.
   - Else if $now == T_2$ and the contract is not already settled send $(\text{CallFunction}, function = \text{Settle}) \xhookrightarrow{now} \mathcal{L}_D$. Return.

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

1) Parse $\phi(\mathsf{id})$ as $(\mathsf{pk}_P, \mathsf{pk}_V, \texttt{TX}_\texttt{f}, \mathsf{id_{ch}}, T_0, T_1, T_2, t_{wait}, \alpha, \alpha', \mathsf{Addr})$.
2) Set $UpdateChanSpace \leftarrow False$.
3) While $now \leq T_2$:$\triangleright$ The proof might be exactly posted at T_2 and then Settle function can be called after T_2.
   - Send $(\text{read}, address = \mathsf{Addr}, variable = \mathsf{State}) \xhookrightarrow{now} \mathcal{L}_D$, save the output in State.
   - If $\mathsf{State} == \texttt{Dispute}$, then send $(\text{read}, address = \mathsf{Addr}, variable = [\overline{\texttt{TX}}^{\texttt{P}}_{\texttt{1}}], R_P) \xhookrightarrow{now} \mathcal{L}_D$, save the output in $[\overline{\texttt{TX}}^{\texttt{P}}_{\texttt{1}}], R_P$ and do the followings: $\triangleright$ 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 $\overline{\texttt{TX}}^{\texttt{P}}_{\texttt{1}}$ existed in the channel history $\Theta^V(\mathsf{id_{ch}})$ of $\mathcal{S}_{\mathcal{PC}}$ as $([\overline{\texttt{TX}}^{\texttt{P}}_{\texttt{1}}], \bar{r}_P, \bar{r}_V, \bar{s}^P_{\mathsf{c}})$, then first send $(\texttt{RESPOND-CLAIM-PUNISH}, \mathsf{id}) \xhookrightarrow{now} \mathcal{Z}$, if you received $(\texttt{RESPOND-CLAIM-PUNISH-OK}, \mathsf{id}) \xhookleftarrow{now} \mathcal{Z}$ continue otherwise stop. Send $(\text{CallFunction}, function = \text{RespondClaim}, args = \{\bar{r}_P\}) \xhookrightarrow{now} \mathcal{L}_D$.
     - Else, extract $\vec{\theta}$ and $\bar{R}_P$ from $[\overline{\texttt{TX}}^{\texttt{P}}_{\texttt{1}}].\mathsf{Out}$; 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. $\triangleright$ 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 the Idle mode.; Then generate $[\texttt{TX}^{\texttt{P}}_{\texttt{1}}] := \texttt{GenCom}^V_{Alba}([\texttt{TX}_\texttt{f}], (\mathsf{pk}_P, R_P), (\mathsf{pk}_V, R_V), (\bar{R}_P), \vec{\theta}, \mathsf{id})$ $\triangleright$ $[\overline{\texttt{TX}}^{\texttt{P}}_{\texttt{1}}]$ submitted by P is either a legitimate latest state which is saved in $\Gamma^V(\mathsf{id_{ch}})$ 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
       Send then first send $(\texttt{RESPOND-CLAIM}, \mathsf{id}) \xhookrightarrow{now} \mathcal{Z}$, if you received $(\texttt{RESPOND-CLAIM}, \mathsf{id}) \xhookleftarrow{now} \mathcal{Z}$ continue otherwise stop. Sign the transaction $\sigma_V([\texttt{TX}^{\texttt{P}}_{\texttt{c}}]) := \mathsf{Sign}_{\mathsf{sk}_V}([\texttt{TX}^{\texttt{P}}_{\texttt{1}}])$. Then send $(\text{CallFunction}, function = \text{RespondClaim}, args = \{[\texttt{TX}^{\texttt{P}}_{\texttt{1}}], \sigma_V([\texttt{TX}^{\texttt{P}}_{\texttt{c}}])\}) \xhookrightarrow{now} \mathcal{L}_D$ and set $updateChanSpace \leftarrow True$.
   - Else if $(\mathsf{State} == \texttt{Valid\_Proof})$ do the following:
     - If $(updateChanSpace == True)$ then Send $(\text{read}, address = \mathsf{Addr}, variable = [\texttt{TX}^{\texttt{V}}_{\texttt{1}}], [\texttt{TX}^{\texttt{P}}_{\texttt{1}}], \sigma_P([\texttt{TX}^{\texttt{V}}_{\texttt{c}}])) \xhookrightarrow{now} \mathcal{L}_D$. Retrieve r_V from the local storage, reprieve $\bar{r}_V$ from $\Theta^V(\mathsf{id_{ch}})$ stored in $\mathcal{S}_{\mathcal{PC}}$ and update $\Gamma^V(\mathsf{id_{ch}}) := (\gamma, \texttt{TX}_\texttt{f}, (\texttt{TX}^{\texttt{P}}_{\texttt{1}}, \texttt{TX}^{\texttt{V}}_{\texttt{1}}, r_V, R_P, \bar{r}_V))$. Then set $updateChanSpace \leftarrow False$. $\triangleright$ In the case of an On-chain update, the channel space should be updated here because probably V is in the idle mode.
     - If the contract is not settled yet, send $(\text{CallFunction}, function = \text{Settle}) \xhookrightarrow{now} \mathcal{L}_D$.
   - Else if $now == T_2$ and if the contract is not settled yet, send $(\text{CallFunction}, function = \text{Settle}) \xhookrightarrow{now} \mathcal{L}_D$.

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 $\mathrm{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}}$ and $\mathrm{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}}$ are the same for $\mathcal{Z}$, i.e., $\mathcal{Z}$ sees the same messages in the same rounds in both worlds.

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.

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:

- $m_0 := (\texttt{A-SETUP}, \mathsf{id}, \mathsf{id}_{\mathsf{ch}}, P, T_0, T_1, T_2, \alpha, \alpha')$
- $m_1 := (\text{GSetupInfo}, \zeta_{init}, \sigma_V(\zeta_{init}))$
- $m_2 := (\texttt{A-CREATED}, \mathsf{id}, \mathsf{id}_{\mathsf{ch}}, T_0, T_1, T_2, t_{wait}, \alpha, \alpha')$
- $m_3 := (\text{InitiateContract}, code = \mathsf{Script}_G, function = \text{Constructor}, args = \{\zeta_{init}, \sigma_V(\zeta_{init})\})$

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 $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \{m_2[t]\} \cup \mathsf{obsSet}(m_3, \mathcal{L}_D, t)$ or $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \emptyset$, 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 $\mathtt{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}} := \{m_2[t]\} \cup \mathsf{obsSet}(m_3, \mathcal{L}_D, t)$ or $\mathtt{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}} := \emptyset$, 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 $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \{m_1[\tau + 1], m_2[\tau_1]\}$ or $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \{m_1[\tau + 1]$, 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 $\mathtt{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}} := \{m_1[\tau + 1], m_2[\tau_1]\}$ or $\mathtt{EXEC}_{\mathcal{F}_{\mathsf{Alba}},\mathcal{S},\mathcal{Z}} := \{m_1[\tau + 1]$, 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 setup phase of the functionality* $\mathcal{F}_{\mathsf{Alba}}$.

Proof. For convenience, we name the following messages:

- $m_4 := (\texttt{A-INITIATE-PAYMENT}, \mathsf{id}, \vec{\theta})$
- $m_5 := (\texttt{UPDATE}, \mathsf{id}_{\mathsf{ch}}, \vec{\theta}, True)$
- $m_6 := (\texttt{SETUP}, \mathsf{id}_{\mathsf{ch}}, \mathsf{txid}, True)$
- $m_7 := (\texttt{SETUP-OK}, \mathsf{id}_{\mathsf{ch}}, True)$
- $m_8 := (\texttt{UPDATE-OK}, \mathsf{id}_{\mathsf{ch}}, True)$
- $m_9 := (\texttt{REVOKE}, \mathsf{id}_{\mathsf{ch}}, True)$
- $m_{10} := (\texttt{SUBMIT-PROOF}, \mathsf{id})$
- $m_{11} := (\texttt{SUBMIT-PROOF-OK}, \mathsf{id})$

We consider the following cases:

1. P honest, V dishonest

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 $t_0 > T_{updateDelay} - \Delta_D$, 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 (by sending another message). Or P runs $\texttt{CallClaimFunc}(\mathsf{id})$ 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, i.e., P does not receive m_6 in round $t_0 + 2$, then P will force close the channel and run $\texttt{CallClaimFunc}(\mathsf{id})$ (case iii). Otherwise, P continues with the update by sending m_7 to $\mathcal{PC}$ in round $t_1 \leq t_0 + 2 + t_{\mathsf{stp}}$. If P does not receive m_8 by round t_2, P will force close the channel and run $\texttt{CallClaimFunc}(\mathsf{id})$ in round t_2 (case iv). Else, if P receives m_8 in round t_2, it sends m_9 to $\mathcal{PC}$ in round t_2, and also m_{10} to $\mathcal{Z}$ in t_2. Finally, after receiving m_{11} from $\mathcal{Z}$ in round t_2, P sends m_{12} to $\mathcal{L}_D$ (case v).

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

- Case i: $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \emptyset$
- Case ii: $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \mathsf{obsSet}(\texttt{CallClaimFunc}(\mathsf{id}), P, t_0)$
- Case iii: $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \mathsf{obsSet}(m_5, \mathcal{PC}, t_0) \cup \mathsf{obsSet}(\texttt{ForceClose}^P(\mathsf{id}_{\mathsf{ch}}), \mathcal{PC}, t_0) \cup \mathsf{obsSet}(\texttt{CallClaimFunc}(\mathsf{id}), P, t_0)$
- Case iv: $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} := \mathsf{obsSet}(m_5, \mathcal{PC}, t_0) \cup \mathsf{obsSet}(m_7, \mathcal{PC}, t_1) \cup \mathsf{obsSet}(\texttt{ForceClose}^P(\mathsf{id}_{\mathsf{ch}}), \mathcal{PC}, t_0) \cup \mathsf{obsSet}(\texttt{CallClaimFunc}(\mathsf{id}), P, t_0)$
- Case v: $\mathsf{obsSet}(m_5, \mathcal{PC}, t_0) \cup \mathsf{obsSet}(m_7, \mathcal{PC}, t_1) \cup \mathsf{obsSet}(m_9, \mathcal{PC}, t_2) \cup \{m_{10}[t_2]\} \cup \mathsf{obsSet}(m_{12}, \mathcal{L}_D, t_2)$

1b. Ideal world: In the ideal world, $\mathcal{F}_{\mathsf{Alba}}$ receives m_4 in round t_0 and reads the channel tuple from $\mathcal{PC}$. Similar to the real world, we distinguish some cases. In case the time $t_0 > T_0 - \Delta_D$, the functionality does not proceed (case i). Similarly

2. V **honest,** P **dishonest**

2a. Real world:

2b. Ideal world: □

Lemma 3. *The* channel update and proof *phase of the protocol* Π *GUC-realizes the setup phase of the functionality* $\mathcal{F}_{\mathsf{Alba}}$.

Proof. We consider the following cases:

1. P **honest,** V **dishonest**

1a. Real world:

1b. Ideal world:

2. V **honest,** P **dishonest**

2a. Real world:

2b. Ideal world: □

Theorem 4. *The protocol* Π *GUC-realizes the functionality* $\mathcal{F}_{\mathsf{Alba}}$.

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