Zeus: Defending against Fee Stealing and Griefing Attacks in Multi-Hop Payments

Jingyu Liu
jliu514@connect.hkust-gz.edu.cn
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, Guangdong, China

Yingjie Xue
yjxue@ustc.edu.cn
University of Science and Technology of China
Hefei, Anhui, China

Di Wu
wu.di@zju.edu.cn
Zhejiang University
Hangzhou, Zhejiang, China

Jian Liu
liujian2411@zju.edu.cn
Zhejiang University
Hangzhou, Zhejiang, China

Xuechao Wang
xuechaowang@hkust-gz.edu.cn
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, Guangdong, China

Abstract

Payment Channel Networks (PCNs) are the most scalable and trust-minimized solution to Bitcoin's scalability challenges. Within PCNs, connected payer and payee can make arbitrary off-chain transactions through multi-hop payments (MHPs) over payment channel paths, while intermediate relays charge relay fees by providing liquidity. However, current MHP protocols face critical security threats including fee-stealing attacks and griefing attacks. In this paper, we identify new fee-stealing attacks targeting most existing MHP protocols. Second, we prove that eliminating griefing attacks in current MHP protocols is impossible by reducing the problem to fair secret exchange. Finally, we introduce *Zeus*, the first Bitcoin-compatible MHP protocol that is secure against fee-stealing attacks and offers bounded griefing protection against k-cost-sensitive adversaries—those who only launch griefing attacks when the expected damage exceeds a k fraction of their own cost. These guarantees are established through rigorous proofs in the Global Universal Composability (GUC) framework. Our comprehensive evaluation demonstrates that *Zeus* reduces worst-case griefing damage to 28% and 75% compared to MHP schemes such as AMHL (NDSS'19) and Blitz (USENIX SEC'21), respectively. Our results further show that, even under the most adverse configurations within the Lightning Network, *Zeus* imposes costs on adversaries that are at least ten times greater than their potential damage.

CCS Concepts

• **Security and privacy → Distributed systems security**.

Keywords

Payment Channel Networks, Fair Exchange, Griefing Attack, Wormhole Attack

1 Introduction

Payment channel networks (PCNs), exemplified by Lightning Network (LN), are one of the most promising solutions to solve the scalability problem of Bitcoin. Compared to other scalability solutions, such as sidechains, rollups, PCNs stand out by not introducing additional security assumptions [44] (e.g., reliance on external committees), rendering them the trust-minimized solutions for scaling Bitcoin. Since the introduction of LN in 2015, LN now processes more than 15% of total BTC transactions [51].

In PCNs, transactions can be conducted off-chain. Given a payment channel, two users can securely conduct numerous payments, while only burdening the blockchain with two transactions (opening and closing) [43]. Creating direct channels with every potential user is inefficient due to the need to lock BTC on mainnet. Consequently, multi-hop payment (MHP) protocols emerge [8, 32, 33, 43], enabling payments between indirect payer and payee through intermediate relays. Most MHP protocols adopt a lock-resolve paradigm to ensure atomic channel updates [8, 32, 33, 36, 43]. For ease of exposition, we use the most widely adopted Hashed Timelock Contract (HTLC) based MHP[1] to show how to realize multi-hop payments. Other solutions share similar basic structures as HTLC-based MHP.

As illustrated in Fig. 1, HTLC-based MHP follows a lock-resolve paradigm: in the lock phase, starting from the payer U_0, each left user U_i makes a conditional payment to its right neighbor U_{i+1}, with the amount $v_i = x + \sum_{j=i+1}^{n} f_j$ through the intermediate channel γ_i, where x is the payment amount and f_j is the relay fee towards relay U_j. The unlock condition of U_i's payment is: once the preimage of a hash h (denoted by s) is revealed before t_i[2], the locked fund is given to U_{i+1}; otherwise, it is refunded to U_i. Once all funds are locked, payee U_{n+1} enters the resolve phase, gives the preimage s to U_n and asks U_n to update channel γ_n, where the new state directly gives the locked fund v_n to U_{n+1}. If U_n refuses, U_{n+1} can always close channel and redeem v_n on-chain. Subsequently, U_{i+1} redeems the payment from U_i, until U_0's payment gets redeemed.

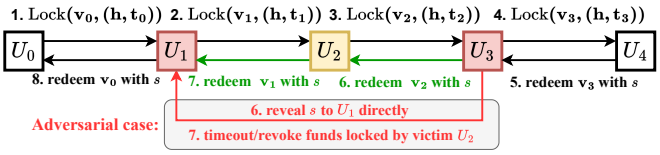


Figure 1: HTLC-based MHP protocol and wormhole attack.

[1]It supports LN with ~ 50M USD TVL as of April 2025, with price ~ 82k USD per BTC. We use this price for all conversions between BTC and USD throughout this paper.
[2]As shown in the figure, $t_0 = 4\delta$, and $t_{i+1} = t_i - \delta$, δ is number of blocks a node has to settle a stalled payment before it could potentially lose money. In LN, δ is at least 18 Bitcoin blocks [1]. For comparison, the block confirmation time Δ is 6 blocks.

1.1 Challenges

The lock-resolve paradigm introduces vulnerabilities for achieving secure and efficient multi-hop payment over payment channel networks, as we demonstrate below.

Fee Stealing Attack. MHPs face *fee-stealing attacks* where the payee receives payments while relays go unpaid, exemplified by the wormhole attack [33]. Incremental-lock protocols (Fulgor [32], AMHL [33], EAMHL+ [54], Astrape [16]) enforce atomicity via a trusted payer setup (secrets generation), assuming payer honesty [33, 46]. However, we observe that the payer can profitably deviate (detailed in Section 3), violating incentive compatibility. These protocols fail to ensure that rational payers behave honestly.

Griefing Attack. Griefing attacks in MHPs allow a malicious payer to initiate doomed payments (including loop attacks [48]), locking relay funds until timeout expiration [37, 41, 47]. These attacks can isolate critical nodes and disrupt network operations. Existing defenses—path restrictions [47], rerouting [19], and reputation/fee systems [28, 50]—remain insufficient. As discussed in Section 1.3, these solutions are either hindered by the permissionless natural of PCNs, or introduce unfairness for honest participants.

Moreover, in most MHP protocols, such as the HTLC-based MHP [43], Fulgor [32], and AMHL [33], the timeout set for funds locked at each channel γ_i must be progressively longer than that of the subsequent relay U_{i+1} to guarantee the balance security [33]. For example, in Fig.1, the timeout t_i increases by one δ at each hop from right to left, where δ is a safe timelock gap ($\geq$ 18 Bitcoin blocks in LN [1]). Given a n-hop payment with value x, an adversary can inflict a disproportionately larger *griefing damage*—defined as the total fund times their locking time for innocent intermediaries—amounting to $\Theta(n^2 \cdot \delta \cdot x)$. Based on the current LN topology, griefing a single 200 USD MHP can equate to immobilizing 11,878 USD-hours of liquidity for relays. Payment rejection by the payee enables an infinite griefing factor (damage-to-cost ratio [21]) at zero cost.

Recent works (Sprites [36], AMCU [17], Blitz [8]) use global on-chain events to enforce payment atomicity, achieving constant timeout per hop, reducing the griefing damage to $\Theta(n \cdot \delta \cdot x)$. Blitz further forces malicious payers to post on-chain transactions, raising attack cost. However, it fails to prevent the zero-cost griefing launched by the payee and remains vulnerable to fee-stealing attacks (Section 3).

Research Question. *Is it possible to design a Bitcoin-compatible MHP protocol that simultaneously achieves both fee-stealing resistance and griefing resistance?*

1.2 Contributions

Identification of Fee Stealing Vulnerabilities within Existing Bitcoin-Compatible MHPs. We identify that, for incremental-lock based MHPs (e.g., Fulgor [32], AMHL [33], and Astrape [16]) requiring a payer-side trusted setup, it is not incentive compatible for payers to stay honest after the setup phase. The key insight is that payers can engage in the illicit sale of setup secrets to malicious relays, a collusive action which increase the payoff for both the payer and the complicit malicious relay. For Blitz, we discover that colluding relays can steal relay fees by exploiting its fast revoke mechanism (see Section 3.2). Our empirical analysis in Section 8 underscores the acute vulnerability of the current LN topology to fee-stealing attacks. Remarkably, it is shown that the engagement of less than 0.6% of total relays in fee-stealing activities allows an adversary to amass roughly 50% of all distributed relay fees.

Impossibility of Griefing Resistant MHPs. We model lock-resolve MHPs and prove that no such MHP with multiple relays can achieve *griefing resistance* against malicious adversaries. Our proof reduces the griefing resistant MHPs to fair secret exchanges, where two parties exchange their secrets atomically. We then prove that such fair secret exchange is impossible within the context of public blockchains. This result provides theoretical insights with direct implications for many blockchain-based protocols, such as atomic swaps, decentralized data markets, and cross-chain bridges, which may be of independent interest.

***Zeus*: A Secure MHP Protocol.** We propose *Zeus*, the first Bitcoin-compatible MHP simultaneously achieves:

- *Fee-Stealing Resistance.* We identify that fast-revoke (design to improve the efficiency) is the root of the fee stealing risks. To prevent this, we remove the fast-revoke procedure and instead introduce a *refundable deposit* that the payer locks before the payment begins, maintaining the same level of efficiency as Blitz. We further ensure deposit fairness: a honest payer can always claim their deposit back, while deposits will be forfeited if the payer attempts to grief the payment.
- *Griefing Resistance against k-Cost-Sensitive Adversaries.* As griefing resistance is impossible within the current MHP paradigm, *Zeus* defends griefing by (i) lowering the griefing damage via constant timelocks for all channel funds (payment-length agnostic) and (ii) increasing the griefing cost for any adversary. For (ii), we first disincentivize payee-side griefing by letting the payee post a *small on-chain stake* (a few USD). This stake is then effectively augmented by off-chain *premiums* locked by the payee, scaled to the potential griefing damage, thereby increasing the payee's griefing cost. Second, to prevent the payer from griefing these premiums, the payer will place a small refundable off-chain deposit to the payee (routed by relays). Should the payer attempt to grief, this deposit is forfeited to compensate the relays and the payee. Last, even in scenarios involving colluding payer and payee, griefing the MHP necessitates at least two distinct on-chain transactions. Our evaluation demonstrates that, even in the worst-case setting in LN, *Zeus* maintains a griefing factor below 0.1. For typical payment scenarios (e.g., payment $\leq$ \$500), the griefing factor drops significantly to below $0.01 \ll 1$.

Formal Security Analysis and Evaluation. We provide a comprehensive security analysis of *Zeus* using the global universal composability (GUC) framework [14]. This rigorous approach proves the security of *Zeus* by defining its ideal functionality capturing all properties, then demonstrates that our construction securely realizes it. We implement *Zeus* within Bitcoinjs-lib [2] and evaluate it using real LN topology captured in March 2025. Our evaluation shows that *Zeus* achieves a griefing factor lower than 0.1 for any griefing adversaries. For griefing damage, compared to the state-of-the-art solution Blitz, *Zeus* reduces griefing damage by 25%, and achieves a 72% reduction relative to widely adopted HTLC and AMHL. At the protocol level, *Zeus* decreases on-chain transaction

cost by 10.3% in dispute resolution scenarios compared to HTLC and AMHL.

1.3 Related Works

Protocols	Fee Security	Collateral	Griefing Factor	Script
HTLC-based [43]	✗	Linear	∞	BTC
Sprites [36]	✗	Constant	∞	SC
Fulgor [32], AMHL [33]	honest payer	Linear	∞	BTC
Payment Tree [26]	honest payer & payee	Logarithmic	∞	BTC
HTLC-GP [34]	honest payee	Linear	∞	SC
Blitz [8]	✗	Constant	∞	BTC
Zeus (this work)	✓	Constant	$\leq 0.1^*$	BTC

Table 1: Comparison of MHP protocols. BTC: Bitcoin script, SC: Smart Contract. For typical payments ranging from \$10 to \$500, griefing factors drop to below $0.01 \ll 1$.

MHPs. Existing MHP protocols have made significant strides, yet each faces distinct limitations, as summarized in Table 1. Beginning with Fulgor [32], several protocols (e.g., AMHL [33], EAMHL+ [54], and Astrape [16]) have emerged to address the *wormhole attack*. These protocols, which we term *incremental-lock* based MHPs, are widely considered in industry as the replacement to vulnerable HTLC-based approaches. They employ different cryptographic primitives (e.g., zero knowledge proofs, homomorphic one-way functions) to ensure *strong atomicity* - guaranteeing that U_i's payment can be redeemed only if all preceding sub-payments have been redeemed. Rain et al. [46] validate that these MHPs can effectively prevent the wormhole attack through game-theoretic analysis. However, these protocols rely on trusted setup performed by payers, and require payers to stay honest during the following execution. Moreover, these MHPs inherit the *linear collateral* inefficiency and griefing risks from HTLC-based MHP. Linear collateral means the timelock along the MHP increases along the payment, thereby leading to a griefing damage with quadratic complexity with respect to the path length. To reduce the total collateral, Sprites [36] pioneered the use of global state through an on-chain preimage manager contract. While innovative, this approach requires smart contract beyond Bitcoin's capabilities. Subsequent efforts to adapt these concepts to Bitcoin's UTXO model include AMCU [17], which was later proven insecure by Payment Tree [26]. Although Payment Tree achieved logarithmic collateral, it remains vulnerable to the *Domino attack* [9], where a party can force close all involved channels on-chain. Blitz [8] first securely achieves constant-collateral but only partially mitigates the griefing attacks. Blitz introduced the "pay-or-revoke" paradigm, which has been widely adopted in protocols like Thora [6], Donner [9] and X-Transfer [31]. Section 3.2 will further demonstrate that Blitz remains vulnerable to fee stealing.

Griefing Mitigation. To mitigate griefing attacks within MHPs, several works [19, 28, 47, 50] introduce different mitigations. Path restrictions [47] and reputation systems [50] are fundamentally limited by the permissionless nature of PCNs, making enforcement and reliable identity tracking infeasible. Fee-based [28] approaches require endpoints to pre-pay non-refundable fees regardless of payment success, introducing unfairness and discouraging honest participation. Rerouting [19] shifts additional cost to victim relays, exacerbating fairness issues and failing to deter determined adversaries. However, these approaches either bring unfairness or require additional assumptions. For instance, HTLC-GP [34] requires an "honest payee" setup and utilized smart contract capabilities to achieve dynamic fee rates.

In atomic swaps [22, 25], a domain closely related to MHPs, Xue and Herlihy [52] have shown that griefing attacks can be reduced to a negligible level through multiple rounds of premiums. However, the dust limit-the minimum payment value in Bitcoin-compatible PCNs-renders this approach infeasible for MHPs, as detailed in Section 5. Grief-free Swap [38] uses a similar premium-based mechanism compatible with Bitcoin scripts and claims to eliminate griefing by binding the premium and principal locking within a single transaction. Nevertheless, one participant can refuse to sign the transaction, thereby implicitly griefing the principal reserved by its counter-party.

2 Background and Model

UTXO Ledger and Payment Channel Networks. Bitcoin is an *Unspent Transaction Outputs* (UTXO) ledger, modeled by an ideal functionality $\mathcal{G}_{\mathrm{L}}$ [7]. A valid transaction submitted to $\mathcal{G}_{\mathrm{L}}$ consumes existing UTXOs and creates new UTXOs. For a transaction $tx := (\mathrm{id}, \mathrm{Ins}, \mathrm{ScriptSigs}, \mathrm{Outs})$, tx.id is the unique identifier, tx.Ins $:= (\mu_1, \ldots, \mu_n)$ is the list of inputs, where each μ_i contains a reference to a UTXO, identified by the source transaction's id μ_i.id and an index μ_i.idx. tx.ScriptSigs $:= (\lambda_1, \ldots, \lambda_n)$ is the list of scripts that unlock inputs, and tx.Outs $:= (\theta_1, \ldots, \theta_m)$ is the list of outputs. For each output θ_i, θ_i.amt is the amount of coins, and $\theta_i.\phi$ is the locking script of this output.

A PCN can be modeled as an ideal functionality $\mathcal{F}_{Channels}$ [7], that supports channel opening, updating, and closing. Each channel γ has a unique identifier γ.id, left and right users ($\gamma.lu$, $\gamma.ru$), a current state γ.state that contains a list of outputs $\theta_1, \ldots, \theta_n$. *Channel splicing* [49] is widely-supported scheme in LN that allows channel owners to partially withdraw/add in-channel funds on-chain without closing the channel. This operation avoids the standard overhead of channel closure and reopening while preserving remaining funds for continued off-chain operations.

MHP Model. Let a n-hop payment be defined as a tuple $\mathcal{M} = (U, \Gamma, x, \mathrm{Fees}, \mathcal{S})$ where $U = (U_0, U_1, ..., U_n, U_{n+1})$ is an ordered sequence of $n+2$ parties with payer U_0, payee U_{n+1}, and intermediate relays U_1 through U_n. $\Gamma = (\gamma_0, ..., \gamma_n)$ denotes the payment channels, where U_i establishes channel γ_i with U_{i+1}. Payer U_0 makes a payment with amount x to the payee U_{n+1}. We denote the payment x as *principal* in the following context. Fees $= (f_1, ..., f_n)$ represents the sequence of relay fees where f_i is the fee charged by the relay U_i. We denote the cost of the transaction on the chain as F. For each γ_i, $\mathcal{S}_i \in \mathcal{S}$, $\mathcal{S}_i$ defines all the possible states derived from an initial `idle` state. Each state specifies a list of token distributions mapped with spending conditions (signatures or specific constraints). In lock-resolve MHPs, each γ_i has three possible states:

$$\begin{aligned} \texttt{locked} &: (\{lb_i - v_i : U_i\}, \{rb_i : U_{i+1}\}, \{v_i : \mathrm{cond}_i\}), \\ \texttt{paid} &: (\{lb_i - v_i : U_i\}, \{rb_i + v_i : U_{i+1}\}), \\ \texttt{unpaid} &: (\{lb_i : U_i\}, \{rb_i : U_{i+1}\}), \end{aligned}$$

where lb_i, rb_i are initial balances of channel γ_i, $v_i = x + \sum_{j=i+1}^{n} f_j$ is the locked amount, and cond_i is the spending condition for the locked funds.

Lock-Resolve MHPs execute in two phases:

1) Lock Phase: From U_0 to U_n, U_i sends *a lock message* m^i_{lock} (e.g., U_i's signature over the `locked` channel state) to the right user U_{i+1}, updating channel γ_i to `locked` state.

2) Resolve Phase: Before a predefined deadline N, participants invoke the resolution process, updating each channel γ_i to `paid` state if the spending condition cond_i met. Otherwise, reset γ_i to `unpaid`.

Communication Model. We assume 1) authenticated, private communication channels $\mathcal{F}_{sec}$ [13] among adjacent participants; 2) a synchronous network model [27], where communication has a bounded 1-round delay; 3) any message sent to the ledger $\mathcal{G}_{\text{L}}$ is public and is processed by the ledger within Δ rounds. Δ refers to the upper bound of the blockchain confirmation time. $(m) \overset{\tau}{\hookleftarrow} Q$ presents a message m sent from party Q is received at round τ.

Minimum Security Requirements. Here we define the well-established minimum security requirements for MHP protocols [8, 18, 32, 33]. Our definition is directly derived from [18]. Let Π_{MHP} be a protocol for $\mathcal{M}$ and $\mathcal{A}$ be a probabilistic polynomial time (PPT) adversary controlling all but honest parties. Any Π_{MHP} must satisfy:

Definition 2.1 (Balance Security)**.** When Π_{MHP} terminates: 1) the balance of any honest relay $U_{i\in[1,n]}$ will not decrease compared with its initial balance, and 2) if payer U_0 and payee U_{n+1} are honest, U_0 pays U_1 in the first channel γ_0 only if U_{n+1} gets paid in γ_n.

Definition 2.2 (Correctness)**.** If all channels have sufficient balances and the parties are honest, then the payment will be settled off-chain, and each relay U_i gets the relay fee f_i.

Definition 2.3 (Coin Availability)**.** For any honest $U_{i\in[0,n+1]}$, its funds will never be locked forever.

These properties provide minimal security, but fail to prevent sophisticated fee stealing attacks like wormhole attacks or the griefing attacks.

Fee Security. So we strengthen MHP security by formally introducing *fee security*. This definition ensures that once a payment is settled off-chain, all honest relays are guaranteed to receive their fees.

Definition 2.4 (Fee Security)**.** Given an MHP, for any honest relay $U_{i\in[1,n]}$, if the payee U_{n+1} receives the payment off-chain, then U_i must receive the relay fee f_i if it has forwarded this payment.

Griefing Resistance. Now we strengthen the MHP security by introducing *griefing resistance*. Compared with fee stealing, griefing attacks are usually performed by endpoints, and the attackers do not care whether payments are successful or not. We first denote the possible griefing coalition as $\mathcal{S}_A$, which includes at least one endpoint (payer or payee). And the adversary can further corrupt the relays. Formally, $\mathcal{S}_A := \mathcal{A}_{\text{endpoints}} \cup \mathcal{A}_{\text{relays}}$, where $\emptyset \neq \mathcal{A}_{\text{endpoints}} \subseteq \{U_0, U_{n+1}\}$ and $\mathcal{A}_{\text{relays}} \subseteq \{U_1, \ldots, U_n\}$. We define the *griefed relay set* as S_G. A relay $U_r \in S_G$ if U_r locks funds but does not get the corresponding relay fee. We formalize griefing resistance in two different settings: 1) *griefing resistance against malicious adversaries* (strong griefing resistance): Guarantees honest relays cannot be grieved against any $\mathcal{S}_A$;

Definition 2.5 (Griefing resistance against malicious adversaries)**.** For any honest $U_{i\in[1,n]}$ and PPT adversary $\mathcal{A}$ controlling any $\mathcal{S}_A$, if U_i locks funds, then U_i gets the corresponding relay fee f_i.

2) *Griefing resistance against cost-sensitive adversaries*: To refine the analysis of griefing attacks, for $U_t \in S_G$, we introduce its *griefing damage* GD_t. GD_t quantifies the liquidity loss from U_t's locked collateral. This is formally $GD_t := r \cdot (FL_{t-1} \cdot TL_{t-1} + FL_t \cdot TL_t)$, where r is the effective interest rate (e.g., opportunity cost of capital), FL_{t-1} and FL_t are the fund amounts locked by U_t in its adjacent channels γ_{t-1} (with U_{t-1}) and γ_t (with U_{t+1}) respectively, and TL_{t-1} and TL_t are the corresponding lock durations. Then the griefing factor λ is defined as $\lambda := (\sum_{U_t \in S_G} GD_t)/c$, where c is the griefing cost incurred by the adversary. Here, we ensures griefing attacks are *economically irrational* by capping the griefing factor λ lower than k, where k be any number smaller than one.

Definition 2.6 (Griefing resistance against k-cost-sensitive adversaries)**.** For a PPT adversary $\mathcal{A}$ controlling any $\mathcal{S}_A$ and $k < 1$, the griefing factor $\lambda \leq k$.

3 Exploiting Bitcoin-Compatible MHPs

In this section, we demonstrate fee stealing attacks exploiting vulnerabilities in MHPs, specifically incremental-lock based MHPs (e.g., AMHL) and Blitz.

3.1 Attacking Incremental-Lock Based MHPs

HTLC-based MHPs are vulnerable to the wormhole attack [33], where colluding relays steal fees from honest intermediaries by revoking locked funds (Fig. 1).

As instanced by AMHL [33], incremental-lock based MHPs prevent this via a trusted setup: the payer generates secrets $s_0, \ldots, s_n$ (relay $U_{i\in[1,n]}$ receives s_i, the payee U_{n+1} gets $\bar{s}_n = \sum_{j=0}^{n} s_j$) and uses a homomorphic one-way function g [29] with $g(\sum s_j) = \prod g(s_j)$. Compared to HTLC, each channel γ_i replaces hashlock with $h_i = \prod_{j=0}^{i} g(s_j)$, requiring $\bar{s}_i = \sum_{j=0}^{i} s_j$ for redemption. Honest payer setup ensures U_i gets paid only if U_{i+1} does, enforcing strong atomicity by induction (Fig. 2).

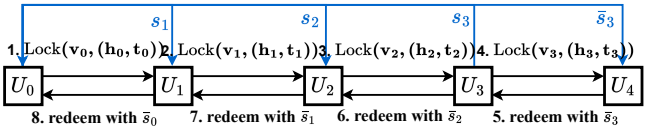


Figure 2: Workflow of AMHL. The blue part highlights the difference between AMHL and HTLC.

Exploiting Incremental-Lock Based MHPs. We start with a simple observation: once the payer U_0 confirms that the last relay U_n has locked funds in the channel γ_n, payee U_{n+1} can always redeem the payment by sending $\bar{s}_n$ to U_n in time. Since U_0 possesses all setup secrets necessary to unlock any locked fund, this creates perverse incentives: U_0 *can minimize costs by selling these secrets to*

malicious relays while still ensuring payment settlement. Such *secret resale attack* works as follow:

Once the payer U_0 confirms that the last relay U_n has locked funds in channel γ_n, U_0 can can publish the identity of the payment (e.g., lock conditions) on various off-chain communication channels to orchestrate fee theft. Upon recognition, malicious relays can negotiate secret acquisition through secure exchange protocols, such as atomic swaps [25]. Once the adversaries obtain the necessary secrets, they can execute the wormhole attack by bypassing honest intermediaries and capturing their intended relay fees.

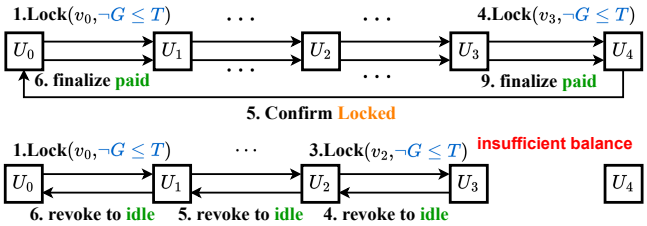


Figure 3: Blitz protocol workflow showing ideal execution (top) and fast revocation path (bottom).

3.2 Attacking Blitz

To achieve constant collateral, Blitz introduces a "pay-or-revoke" mechanism where locked payments automatically default to the *paid* state after a deadline T, unless the payee publishes a revocation transaction G on-chain before T. Since this requires costly on-chain revocations for failed payments — which are frequent due to PCN anonymity constraints and volatile channel balances — Blitz introduces *fast revoke* for efficient off-chain cancellation. Blitz enables two execution paths when participants are honest:

- *Fast Confirmation*: After the payment locked, once U_{n+1} confirms the payment is locked (step 5, top part of Fig. 3), from the payer to the payee, honest participants settle all channels to the `paid` state (step 6 to step 9).
- *Fast Revoke*: During the lock phase, if a relay U_i finds it does not have sufficient balance to continue, U_i quickly revokes the payment with its previous U_{i-1} (step 4, bottom part of Fig. 3). If U_{i-1} receives this revocation within a predefined revoke-period, U_{i-1} accepts and continues the Fast Revoke process, canceling the MHP off-chain.

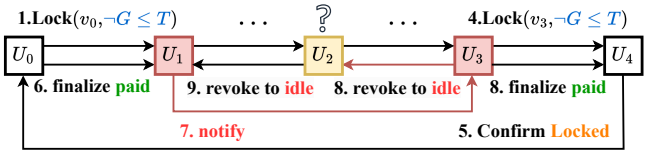


Figure 4: Fee stealing attack over Blitz.

Exploiting Blitz. However, the fast-revoke introduces protocol ambiguity where honest relays cannot distinguish between legitimate liquidity failures and adversarial revocations. This indistinguishability creates an opportunity for colluding parties to simulate revocation and effectively steal fees from honest participants.

Consider the example depicted in Fig. 4, which demonstrates how colluding relays (U_1, U_3) exploit the fast-revoke ambiguity in Blitz. The attack proceeds as follows:

- After all channels are successfully locked, honest U_0 initiates the fast confirmation process (step 6), settling the payment directly off-chain.
- Instead of forwarding the confirmation to U_2, the adversarial U_1 covertly notifies its partner U_3. Once notified, adversarial U_3 initiates a false revocation with U_2 while simultaneously finalizing the payment with U_4 (step 8).
- If this attack is executed within the revoke period, the honest U_2 cannot distinguish this from a legitimate revocation and will accept it, resulting in U_2 being free-ridden despite providing liquidity.

As a result, colluding relays (U_1 and U_3) capture the additional fee f_2 intended for the honest relay U_2. A naive solution to address this attack would be to remove the fast-revoke logic, but this will bring much burden for the payer.

4 Impossibilities of Griefing Resistant MHPs

Before diving into the impossibilities of griefing resistant MHPs, we first provide theoretical background on fair exchange, then reduce griefing resistant lock-resolve MHPs to fair secret exchange problems. Finally, we show that such fair secret exchange is impossible when only public trusted third parties like blockchains are available.

Fair Exchange. A fair exchange (FE) protocol enables two mutually distrusting parties $\mathcal{P}$ and $\mathcal{Q}$ to exchange their items $(i_{\mathcal{P}}, i_{\mathcal{Q}})$ with descriptions $(\text{desc}_{\mathcal{P}}, \text{desc}_{\mathcal{Q}})$ [5, 40, 53]. Fairness is divided into strong fairness and weak fairness in the literature [5, 40]. Here, we adopt strong fairness.

- **Fairness**: If $\mathcal{P}$ does not obtain $i_{\mathcal{Q}}$ such that $\text{desc}(i_{\mathcal{Q}}) = \text{desc}_{\mathcal{Q}}$, then $\mathcal{Q}$ must not obtain $i_{\mathcal{P}}$, and vice versa.
- **Timeliness**: All honest parties eventually terminate before a finite point of time T_{Deadline}.
- **Effectiveness**: If both parties are honest, then when the protocol completes, $\mathcal{P}$ gets the item $i_{\mathcal{Q}}$, and $\mathcal{Q}$ gets $i_{\mathcal{P}}$, while $i_{\mathcal{P}}$, and $i_{\mathcal{Q}}$ match corresponding descriptions.

A special case of fair exchanges is **fair secret exchanges** (FSE). An FSE requires all items in the exchange to be non-revocable secrets. Non-revocable means that once a secret is delivered to the receiver, there is no way to return it. For completeness, Appendix A gives a formal definition of fair secret exchange. We further extend two-party exchange to n-party exchange rings $\mathcal{R} = (\{P_j\}_{j=0}^{n-1}, \{(i_j, \text{desc}_j)\})$ where party P_j sends its inputs i_j to party $P_{(j+1 \mod n)}$.

4.1 Reducing Griefing Resistance MHPs to FSE

We establish the fundamental limits of griefing resistance MHPs through a reduction to fair (secret) exchanges.

4.1.1 One-Relay Case: Link Griefing Resistance with Fair Exchanges. Consider the payment hub model [45]—a MHP with a single relay satisfying minimal security requirements. By our definition, griefing resistance must be guaranteed even when both payer and payee are adversarial. Without loss of generality, we assume the payer

and payee collude as a single entity. For relay U_1, strong griefing resistance means that U_1 locks funds in γ_1 (sends the lock message m^1_{lock}) only if U_1 gets the relay fee in γ_0. In the lock-resolve model, this means U_1 sends m^1_{lock} only if γ_0 ends in the `paid` state. Illustrated in Fig. 5), this creates a fair exchange between relay U_1 and the entity controlling U_1 and U_2. Fortunately, since the payment is revocable, we can instantiate such an exchange using a simple conditional payment within channel γ_0 (detailed in Appendix B.1). *Therefore, in the one-relay case, there exists a strongly griefing resistant and minimally secure MHP protocol.*

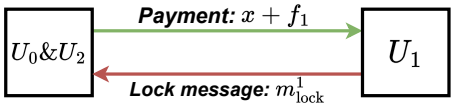


Figure 5: Griefing resistance interpreted as a fair exchange. The green line indicates a revocable input (payment), and the red line indicates a non-revocable secret input (the lock message).

4.1.2 General Case: MHP with More than One Hop. For minimal secure MHPs with at least 2 relays, we establish impossibility through a reduction to fair secret exchange:

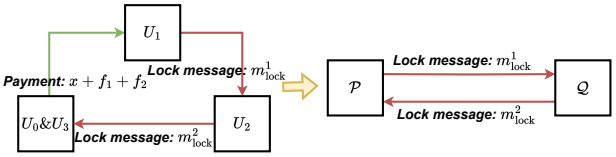


Figure 6: The fair exchange model of two relays.

Lemma 1. For MHP with at least 2 relays, any lock-resolve MHP protocol Π_{MHP} satisfying *minimum security requirements* and *strong griefing resistance* implies existence of a fair secret exchange protocol Π_{FSE}.

Proof. *Step 1: (2-relay MHP ⇒ 3-party FE)* Any strong griefing resistance 2-relay MHP protocol Π_{MHP} implies a 3-party fair exchange protocol Π^3_{FE} (left part of Fig. 6):

- **Fairness:** The strong griefing resistance creates a circular dependency: U_1 sends m^1_{lock} only when guaranteed payment in γ_0, U_2 sends m^2_{lock} only when guaranteed payment in γ_1 (requiring m^1_{lock}), and U_0 pays U_1 only if U_3 receives the payment (requiring U_3 to lock γ_3). Therefore, these three constraints ensure that once any message/payment is settled, then all settled, and vice versa. This meets the definition of fairness.
- **Effectiveness and timeliness:** These follow directly from the correctness and coin availability of Π_{MHP}.

Step 2: (3-party fair exchange ⇒ 2-party fair secret exchange) We prove that any 3-party FE protocol Π^3_{FE} can construct a 2-party FSE protocol Π_{FSE} (the right part of Fig. 6). The construction works as follows: In the constructed Π_{FSE}, we map the roles as follows: party $\mathcal{P}$ plays both the role of relay U_1 and the combined roles of endpoints U_0&U_3 from Π^3_{FE}, while party $\mathcal{Q}$ plays the role of relay U_2. The secret exchange is realized by mapping the lock messages: $\mathcal{P}$'s secret corresponds to m^1_{lock} (from U_1), $\mathcal{Q}$'s secret corresponds to m^2_{lock} (from U_2), while the revocable payment inputs can be set to dummy values since our focus is solely on the non-revocable secret exchange.

This reduction establishes that any strong griefing resistant 2-relay MHP protocol implies the existence of a fair secret exchange protocol. □

4.2 Impossibility Results

While the famous impossibility of fair exchanges [20, 39] shows a FE is impossible without trusted third party (TTP), blockchain-based systems introduce *public TTPs* $\mathcal{T}_{pub}$, where public TTPs holds everything(stored data, received messages) in a public ledger. In this work, we prove even with $\mathcal{T}_{pub}$, fair secret exchange remains impossible:

Theorem 1 (Impossibility of FSE with Public TTP)**.** No FSE protocol Π_{FSE} can simultaneously satisfy *Timeliness*, *Effectiveness*, and *Fairness*, when only $\mathcal{T}_{pub}$ is available.

Proof Sketch. The proof uses a contradiction argument similar to Even and Yacobi [20]. Assume a protocol Π_{FSE} exists that satisfies all three properties with $\mathcal{T}_{pub}$. Consider the first round where one party (say P) can infer the other party's (Q's) secret i_Q, while Q cannot yet infer i_P (denoted by N). For P to gain this knowledge, there must exist an earlier round N' ($N' < N$) when Q transmitted some critical information $m_{N'}$ that enables P to infer i_Q.

Since Π_{FSE} must satisfy fairness and timeliness, at round N', the protocol must guarantee that Q can obtain P's secret by the deadline T_{Deadline} despite any subsequent misbehavior by P. This guarantee must hold even if P stops communicating after round N'. Therefore, using only: 1) the information exchanged up to round N' and 2) the functionality of $\mathcal{T}_{pub}$, party Q must be able to extract i_P by the deadline T_{Deadline}. This implies some extraction function Ext exists where $Ext(\text{view}^{N'}_P, \text{view}^{N'}_{TTP}) = i_P$, where $\text{view}^{N'}_P$ and $\text{view}^{N'}_{TTP}$ are the information owned by P and $\mathcal{T}_{pub}$ up to time N', respectively.

However, since $\mathcal{T}_{pub}$ is public by definition, Q has access to all information in $\mathcal{T}_{pub}$ and to any extraction function Ext that would be used. This creates a strategic problem: Q can simply wait until T_{Deadline} to extract i_P without sending $m_{N'}$ at round N'. Any rational Q would choose this strategy since it allows obtaining P's secret while withholding its own. But if Q never sends $m_{N'}$, then P cannot infer i_Q at round N – contradicting the definition of round N. The formal proof appears in Appendix A. □

Following Lemma 1 and Theorem 1, we naturally derive the impossibility result of lock-resolve MHPs.

Theorem 2 (Impossibility of strong griefing resistant MHPs)**.** Given a MHP with at lease two relays, there is no *minimal secure* lock-resolve MHP protocol satisfying *strong griefing resistance*.

In summary, we have demonstrated that strong griefing resistance is impossible for MHPs with multiple relays, yet achievable for single-relay MHPs. Furthermore, for cost-sensitive adversaries, lock-resolve MHPs present a vulnerability: the payee may incur no loss, as they lock no collateral and their channel balance in γ_n can

be zero, enabling them to grief payments at no personal cost. Consequently, achieving griefing resistance against such adversaries also impossible for lock-resolve MHPs. The subsequent section details how *Zeus* addresses these fundamental limitations through its multi-faceted defense mechanisms.

5 Protocol Overview

In this section, we elaborate on the challenges of defending against fee stealing and griefing attacks and sketch how our proposed protocol addresses them.

5.1 Challenges

Preventing Fee Stealing. As analyzed in Section 3, Blitz's vulnerability to fee stealing stems from its fast-revoke mechanism, which allows relay U_i to accept off-chain revocations from U_{i+1} within a revocation period, even post-payment completion. While enabling efficient off-chain refunds, this exposes relays to fee theft via malicious downstream revocations. A naive solution-removing the fast-revoke mechanism-would reintroduce inefficiency that Blitz originally resolved: locked funds cannot be swiftly reclaimed when a relay lacks liquidity, necessitating costly on-chain transactions. This leads to the first challenge:

- *Challenge 1. How can we achieve fee stealing resistance without forfeiting the efficiency gains of fast revocation?*

A straightforward solution replaces fast revocation with a *probing round*: each node, from the payer to the payee, sequentially sends a probing message requesting that relays reserve funds. While this guarantees payment routing capability, it introduces two critical issues: 1) relays' reserved funds go uncompensated if the payer does not use them, and 2) free probing exposes channel balances, compromising node privacy. To address these problems, we introduce a *deposit round* before locking funds, where the payer and relays deposit to their downstream peers to ensure compensation for any unused reservations. Each relay charge a deposit c_{relay}. However, this approach introduces new problems:

- *Problem 1: Deposit fairness.* The deposit mechanism must satisfy two core requirements: (i) refunds when either the payment completes or a downstream node refuses a liquidity reservation, and (ii) forfeiture to relays whose reserved liquidity remains unused. Achieving this demands reconciling each relay's local channel view—where they only validate adjacent transactions—with the global payment outcome. Malicious intermediaries may strategically exploit this discrepancy between local verification and the system-wide truth, leading to allocation conflicts in off-chain enforcement.
- *Problem 2: Linearly increasing timelock and griefing.* Introducing deposits to Blitz causes timelocks to increase linearly with each hop, resulting in quadratic collateral requirements and linear dispute-resolution times—a significant regression from Blitz's constant timelock design. Moreover, deposits must exceed Bitcoin's dust limit (currently 546 satoshis, $\approx$ \$0.45, denoted as F_{dust}), with cumulative values growing linearly with payment path length [3]. This creates a systemic vulnerability: adversaries can exploit multi-hop paths to maliciously lock up deposits, making griefing prevention imperative.

Mitigating Griefing Attack. As shown in Section 1, existing MHP protocols like Blitz remain vulnerable to griefing attacks, where the payee can stall payments with no cost. Although prior works [22, 38, 52] propose premium mechanisms to compensate liquidity providers who locked funds in atomic swaps, adapting these solutions to Bitcoin-compatible systems presents two key challenges:

- *Challenge 2: How can fair premium distribution be enforced when a revocation is triggered?*

Premium-based solutions face a fundamental conflict when integrated with global revocation mechanisms like those in Blitz. In premium-based schemes, liquidity consumers must compensate providers immediately upon fund reservation to prevent cost-free griefing. In the context of MHP, this requires the payee to lock premiums *before* principal locking, allowing relays to claim compensation once they commit liquidity. However, because global revocation can reclaim principal regardless of premium settlement, combining premiums with global revocation leads to an unfair outcome: if on-chain revocation is triggered after premium settlement, the honest payee ends up paying premiums to relays *without* ever receiving the promised funds.

- *Challenge 3: How can we protect premiums from griefing attacks when fully grief-free is impossible?*

Recent work [52] proposes recursive premiums (locking "premiums-of-premiums") to mitigate griefing of premiums via infinite recursion. However, this approach suffers from two critical limitations: it is incompatible with Bitcoin's dust limits and requires linearly increasing timelocks. In LN, adversaries can grief up to 483 simultaneous 20-hop payments, yielding approximately 140,000 USD-hours of damage. Although our theoretical analysis shows griefing cannot be fully eliminated, carefully designed incentives can deter cost-sensitive attackers by significantly raising the economic barrier to such attacks.

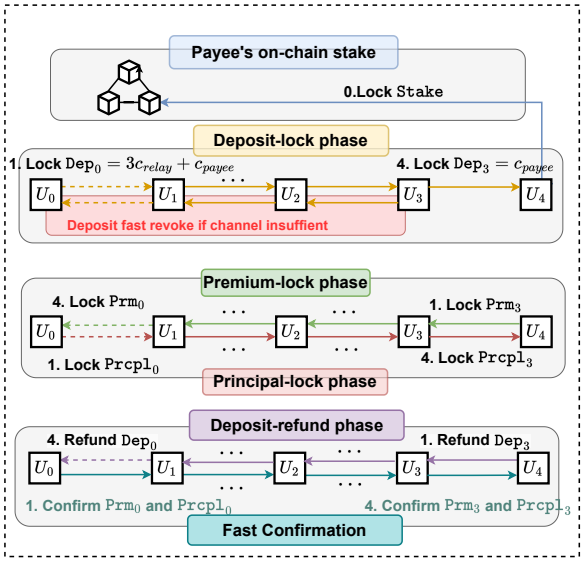


Figure 7: Simplified workflow of *Zeus* in the optimistic case.

5.2 Our Solution

As depicted in Fig. 7, *Zeus* integrates deposits and premiums into Blitz while addressing the above challenges.

To achieve deposit fairness without compromising efficiency (Challenge 1), *Zeus* introduces a three-phase deposit revocation mechanism: (i) During the deposit-lock phase, relays with insufficient liquidity trigger instant off-chain deposit revocation with upstream peers within seconds, preserving Blitz's efficiency; (ii) In the deposit-refund phase, after the fast deposit revocation window closes, deposits can only be refunded once the principal is successfully locked and payment forwarded to enforce deposit commitments; (iii) During the dispute-resolution phase, payers must lock the full principal directly to the payee on-chain before deadline T to revoke deposits, with channel splicing enabling reuse of existing channel γ_0's balance for capital efficiency. This design guarantees two outcomes: First, it deters cost-sensitive griefing by making attackers forfeit deposits or incur substantial on-chain fees; Second, it maintains constant timelocks by making deposit revocation solely rely on on-chain principal locking, eliminating the linear timelock growth of naive solutions. The unified process resolves both deposits and principals in a single on-chain transaction, removing redundant cost while maintaining deposit fairness.

To address premium fairness violations (Challenge 2), we introduce a conditional premium unlocking mechanism based on the payee's actions. Specifically, the premium locked by U_i ($\texttt{Prm}_{i-1}$) is awarded to relay U_{i-1} *unless* payee U_{n+1} redeems the on-chain principal before $T + \Delta$. This ensures: 1) *Premium fairness*: when a global revocation occurs, the payee reclaims premiums while relays receive nothing; 2) *Griefing deterrence*: the payee must either forfeit their premiums-which covers the relays' fees-or pay transaction fees to reclaim them, a cost that typically exceeds any potential griefing gain; 3) *Constant timelock*: global revocation enforces a constant timelock for all premiums.

To prevent premium griefing (Challenge 3), we design mechanisms that counter cost-free griefing attacks by the payer and payee respectively. To deter payer from griefing payee's premium, we require an extra payer-to-payee deposit (c_{payee} in Fig. 7) during the Deposit-Lock phase, in addition to deposits made with relays. This additional deposit is subject to the same unlock conditions as the others, ensuring that any premium-griefing attempt by the payer incurs a direct economic cost.

Now a malicious payee can grief payer deposits by refusing to lock premiums after receiving deposits. To prevent this, we require payees to commit a small, globally slashable on-chain stake during channel creation. Failure to proceed with premium locking after receiving payer deposits triggers slashing of this stake. The stake amount is calibrated so that griefing costs always exceed potential damage, even under concurrent payments, ensuring the griefing factor remains below any chosen k parameter. Fig. 11c illustrates how the on-chain stake value affects the griefing factor under concurrent griefing attacks.

The proposed design still has one inefficiency: relay fees remain locked until the premium timelock $T + \Delta$. To eliminate this delay, we introduce a *fast confirmation* protocol (like Blitz) initiated by the payer, which expedites finalization of both premiums and principals.

Dispute Handling and Incentive Structure. When any payment stalls off-chain, honest payers trigger on-chain principal locking through channel splicing to revoke all locked deposits and principals, while honest payees redeem the principal on-chain to revoke all locked premiums. Our hybrid design creates strong economic incentives for honest execution by **making dispute resolution a strictly worse outcome for any participant**, thereby deterring griefing behavior. When disputes arise, payers suffer either deposit forfeiture or costly on-chain reclamation; payees either forfeit their on-chain stake or pay transaction fees to trigger premium revocation; and relays receive no compensation, contrasting sharply with the relay fees earned through honest execution. This tripartite incentive structure ensures that griefing costs always exceed griefing damage across all attack vectors. Fig. 11a and Fig. 11b show that the griefing factor remains below 0.1, and Table 4 provides a detailed cost–damage analysis under various corruption scenarios.

6 *Zeus* Construction

In this section, we first detail basic primitives, then elaborate on the two-stage revocable channel update protocol, a critical building block for constructing our multi-hop protocol. Finally, we describe the multi-hop protocol.

Digital Signature and Commitment Schemes. A digital signature scheme is a tuple $\Sigma := (\text{KeyGen}, \text{Sign}, \text{Vrfy})$ satisfying EUF-CMA secure [13], where $\text{KeyGen}(\lambda)$ generates a keypair (sk, pk) with security parameter λ. $\sigma := \text{Sign}(sk, m)$ signs a message m with the secret key sk, and $\text{Vrfy}(pk, m, \sigma)$ returns 1 only if the signature σ on message m with the public key pk. For simple notation, we use (sk_i, pk_i) to denote the key-pair of U_i, and $Sig(U_i)$ to denote the signature of U_i. A commitment scheme is a tuple $C := (\text{Com}, \text{Open})$, where $\text{Com}(s)$ commits a secret s, outputting commitment h. $\text{Open}(s, h)$ returns 1 only if the secret s matches h. We use a hiding and binding commitment scheme as a black-box.

PCN Interface. Channel interactions are handled by the functionality $\mathcal{F}_{Channels}$, which is an extension of functionality in [7]. $\mathcal{F}_{Channels}$ allows channel owners to create, update, close, and splice out their payment channels. For simplicity, we highlight following interfaces:

- UPDATE(γ.id, tx_{state}): Update channel γ to the new state tx_{state}, co-signed by $\gamma.lu$ and $\gamma.ru$.
- CLOSE(γ.id): Close γ when called by a user $\in \gamma.lu/ru$. A transaction containing the latest state of γ will be finalize in $\mathcal{G}_{\text{L}}$ within t_{close}.
- UPDATESPLICING(γ.id, tx_{splicing}): Similar to UPDATE, owners can provide a co-signed state-update transaction tx_{splicing} to $\mathcal{F}_{Channels}$, while containing two types of outputs: *splicing outputs* and *funding outputs*. $\mathcal{F}_{Channels}$ will update γ to this splicing state off-chain, and wait for the finalization call-SPLICINGCHANNEL.
- SPLICINGCHANNEL(γ.id): Similar to CLOSE, called by a channel owner, both *splicing outputs* and *funding outputs* of the latest off-chain state will be finalized over $\mathcal{G}_{\text{L}}$ within t_{close}. The *funding outputs* will serve as the new channel state.

Pay-or-Revoke Paradigm. The "pay-or-revoke" paradigm, initially proposed in Blitz, enables atomic UTXO synchronization within Bitcoin's scripting constraints. In this paradigm, a global transaction carries dust outputs that serve as *event triggers*. When

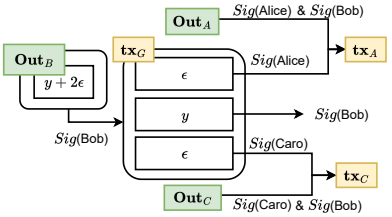


Figure 8: The pay-or-revoke paradigm.

this transaction is finalized on-chain, corresponding participants can update their UTXO states by submitting transactions that consume these dust outputs as inputs. For example, consider two 2-of-2 multi-signature UTXOs controlled by Alice-Bob and Bob-Carol pairs, respectively (Fig. 8). Bob requires that Alice and Carol can spend their respective outputs ($\mathbf{Out}_A$ and $\mathbf{Out}_C$) only after Bob spends output $\mathbf{Out}_B$. To enforce this dependency, Bob's spending transaction tx_G includes two *trigger outputs* carrying negligible amounts (e.g., 1 satoshi). These triggers serve as inputs for Alice's and Carol's spending transactions (tx_A and tx_C). Since Bob only pre-signs these specific spending transactions, outputs $\mathbf{Out}_A$ and $\mathbf{Out}_C$ can only be spent after tx_G is finalized on-chain.

6.1 General Construction for Channel Updates with Two-Stage Revocation

U_i upon $(\texttt{init}, \texttt{Meta}_L)$ from $\tilde{U}_i$:

1) Create$(tx_s, tx^l_{Rvk}, tx^r_{Rvk})$ based on $\texttt{Meta}_L$.
2) $\sigma^i_{rvk-l} := \text{Sign}(sk_i, tx_s.\text{Outs}[0] \rightarrow tx^l_{Rvk})$
3) $\sigma^i_{rvk-r} := \text{Sign}(sk_i, tx_s.\text{Outs}[1] \rightarrow tx^r_{Rvk})$
4) Fill tx^l_{Rvk}, tx^r_{Rvk} with $\sigma^i_s, \sigma^i_{rvk-l}$, and send: $(tx_s, tx^l_{Rvk}, tx^r_{Rvk}) \overset{0}{\hookrightarrow} U_{i+1}$

$(tx_s, tx^l_{Rvk}, tx^r_{Rvk}) \overset{1}{\hookleftarrow} U_i, U_{i+1}$:

5) Check 1) if receiving $\texttt{Meta}_R$ from $\tilde{U}_{i+1}$, and check; 2) if transactions are align with $\texttt{Meta}_R$; 3) if signature are correct. Abort if any check fails.
6) $(\texttt{UPDATE}, tx^r_{Rvk}) \overset{1}{\hookrightarrow} \Pi^i_{\texttt{redeem-handler}}$
7) $\sigma^{i+1}_s := \text{Sign}(sk_{i+1}, tx_{fund}.\text{Outs}[0] \rightarrow tx_s)$
8) $\sigma^{i+1}_{rvk-l} := \text{Sign}(sk_{i+1}, tx_s.\text{Outs}[0] \rightarrow tx^l_{Rvk})$
9) Send $(\sigma^{i+1}_s, \sigma^{i+1}_{rvk-l}) \overset{1}{\hookrightarrow} U_i$

$(\sigma^{i+1}_s, \sigma^{i+1}_{rvk-l}) \overset{2}{\hookleftarrow} U_{i+1}, U_i$:

10) Check signatures, sign tx_s and update channel $(\texttt{UPDATE}, \gamma_i.id, tx_s) \rightarrow \mathcal{G}_L$
11) Upon $\mathcal{G}_L$ returns UPDATED, update splicing handler: $(\texttt{UPDATE}, \mathbf{tx}^l_{Rvk}) \hookrightarrow \Pi^i_{\texttt{splicing-handler}}$
12) Return Updated-Ok.

Upon $\mathcal{G}_L$ returns UPDATED, U_{i+1}: 13) Return Updated-Ok.

Figure 9: Channel update with two stage revocation. $\tilde{U}_i$ denotes the external user invoking interface. External functionalities and sub-procedures are highlighted in red.

As overviewed in Section 5, to ensure atomic global revocation over locked funds thereby achieving a constant timelock, each channel γ_i should support: 1) U_i can reclaim its locked deposit and/or principal (denotes the amount as α) once the payer locks the principal on-chain using channel splicing; 2) U_{i+1} can reclaim its locked premium (denotes the amount as β) if the on-chain redemption occurs; 3) If the splicing event does not occur by a deadline (T_l), U_{i+1} can claim α. Conversely, if the redemption event does not occur by a deadline (T_r), U_i can claim β. To this end, here we demonstrates how to channel owners to update their channel to a state support above functionalities.

Using the pay-or-revoke paradigm, such splicing and redemption events are signaled by dust outputs, denoted as *event triggers* in the following context. We assume that for each γ_i, there exists a *splicing trigger* $tr^i_{splicing}$ and a *redemption trigger* tr^i_{redeem}, associated with the splicing and redemption events, respectively (detailed in Section. 6.2). For brevity, we use $\texttt{Meta} := \{\gamma_i, \alpha, T_l, \beta, T_r, tr^i_{splicing}, tr^i_{redeem}\}$ to denote the parameters governing a state transition. For readability, we define the following transactions:

- tx_s: Transaction representing a two-stage revocable state with two primary outputs (Fig. 15):
  - An output of amount α (funded by U_i), spendable either by U_i (via tx^l_{Rvk}) if $tr^i_{splicing}$ is finalized on $\mathcal{G}_L$, or by U_{i+1} if timelock T_l expires.
  - An output of amount β (funded by U_{i+1}), spendable either by U_{i+1} (via tx^r_{Rvk}) if tr^i_{redeem} is revealed, or by U_i if timelock T_r expires.
- tx^l_{Rvk}: The left revoke transaction. It spends the α-output from tx_s (when $tr^i_{splicing}$ is revealed) and creates a single output locked to U_i's public key pk_i.
- tx^r_{Rvk}: The right revoke transaction. It spends the β-output from tx_s (when tr^i_{redeem} is revealed) and creates a single output locked to U_{i+1}'s public key pk_{i+1}.

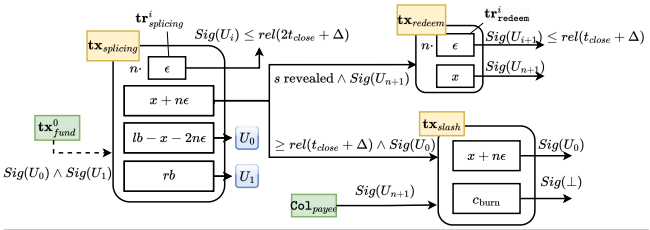
Consider U_i initiating the update (Fig. 9). Initially, each party has its respective metadata set ($\texttt{Meta}_L$ for U_i, $\texttt{Meta}_R$ for U_{i+1}), defining the parameters for the proposed state update of channel γ_i. The first step for U_i is to create these three transactions based on $\texttt{Meta}_L$ and provide its signatures for tx^l_{Rvk} and tx^r_{Rvk}. For instance, $\text{Sign}(sk_i, tx_s.\text{Outs}[0] \rightarrow tx^l_{Rvk})$ indicates U_i approves spending its locked fund (output 0 of tx_s, corresponding to α) via transaction tx^l_{Rvk}. Then U_i sends tx_s and partially-signed tx^l_{Rvk} and tx^r_{Rvk} (i.e., signed by U_i) to U_{i+1}.

Now, U_{i+1} checks if the received transactions align with its view $\texttt{Meta}_R$ and if U_i's signatures are correct. If everything is satisfactory, U_{i+1} signs tx^r_{Rvk} (making it fully signed from U_{i+1}'s perspective for its own use) and push this tx^r_{Rvk} to its redeem handler sub-protocol, $\Pi^i_{\texttt{redeem-handler}}$. $\Pi^i_{\texttt{redeem-handler}}$ maintains a list of such redemption revoke transactions, monitors the channel state, and if tr^i_{redeem} of the latest channel state is confirmed on-chain, it will invoke a sub-procedure to claim the β funds back. Ideally, if U_i cooperates, U_i can directly refund the β back to U_{n+1} off-chain through a new state. If U_i does not cooperate, U_{i+1} will close γ_n and push tx^r_{Rvk} on $\mathcal{G}_L$, reclaiming U_{i+1}'s funds.

Subsequently, U_{i+1} signs tx_s and tx^l_{Rvk}, showing that U_{i+1} agrees with the new state and the left revoke transaction. U_{i+1} then returns these signatures to U_i. Now, U_i possesses a fully signed tx_s and a fully signed tx^l_{Rvk}. U_i update the channel with tx_s and submits tx^l_{Rvk} to its splicing handler protocol, $\Pi^i_{\texttt{splicing-handler}}$. Similar to the redeem handler, this handler monitors for $tr^i_{splicing}$, and once $tr^i_{splicing}$ is confirmed on-chain before T_l, $\Pi_{\texttt{splicing-handler}}$ will invoke in the sub-procedure to reclaim α back.

6.2 Multi-Hop Payment Description

In this section, we elaborate the MHP protocol. The formal specification are specified in Appendix F. Recall a MHP, where a payer U_0 aims to make a payment of amount x to a payee U_{n+1} through an n-relay path $(U_1, \ldots, U_n)$. Each relay U_i charges a relay fee f_i, f_0 is a padding fee to ensure the premium greater than the dust limit. We denote a single on-chain transaction cost as F, deposit and principal timelock as T. *Zeus* consists of six phases: Setup, Lock-deposit, Lock-premium, Lock-principal, Refund-deposit and Fast-confirmation.



U_0 upon $(\texttt{setup}, \mathcal{M}, h, T)$ from $\tilde{U}_0$:

0) $\texttt{Stake} := \text{checkPayeeStake}(U_{n+1}, T)$ Abort if $\texttt{Stake} = \perp$.
1) Create $tx_{\text{splicing}}, tx_{\text{redeem}}, tx_{\text{slash}}$
2) $(tx_{\text{splicing}}, tx_{\text{redeem}}, tx_{\text{slash}}) \overset{0}{\hookrightarrow} U_{n+1}$

$(tx_{\text{splicing}}, tx_{\text{redeem}}, tx_{\text{slash}}) \overset{1}{\hookleftarrow} U_0, U_{n+1}$:

3) Check transactions and initialize the redeem launcher:
   $(\texttt{init}, \gamma_0.id, T, tx_{\text{splicing}}.\text{Outs}[0], tx_{\text{redeem}}) \overset{1}{\hookrightarrow} \Pi_{\text{redeem-launcher}}$
4) $\sigma_{\text{slash}} := \text{Sign}(sk_{n+1}, tx_{splicing}.\text{Outs}[n] \rightarrow tx_{\text{slash}})$,
   $\sigma_{\text{slash}} \overset{1}{\hookrightarrow} U_0$

$\sigma_{\text{slash}} \overset{2}{\hookleftarrow} U_{n+1}, U_0$:

5) Check signature and generate the setup messages.
   $m_{\text{setup}} := \text{genOnion}(\mathcal{M}, tx_{\text{splicing}}, tx_{\text{redeem}})$
   $\forall i \in [1, n+1], m_{\text{setup}}[i] \overset{2}{\hookrightarrow} U_i$
6) $(\texttt{init}, \gamma_0.id, tx_{\text{splicing}}) \overset{2}{\hookrightarrow} \Pi^0_{\text{2pSetupSplicing}}$

$(m_{\text{setup}}[i]) \overset{3}{\hookleftarrow} U_0$:

7) U_1: Extract information and setup $\Pi_{\text{2pSetupSplicing}}$
8) $U_{i\in[2,n+1]}$: Extract information and setup $\Pi^{i-1}_{\text{2pSetupDep}}$

U_0 upon $(\texttt{slicing-Ok})$, then:

9) Enable the splicing and slash launcher.
   - $(\texttt{init}, T, \gamma_0.\text{id}) \hookrightarrow \Pi^0_{\text{splicing-launcher}}$
   - $(\texttt{init}, T, tx_{\text{splicing}}.\text{Outs}[0], tx_{slash}) \hookrightarrow \Pi_{\text{slash-launcher}}$

Figure 10: Setup phase of Π_{Zeus}, where the 0-th to $(n-1)$-th outputs of tx_{splicing} are splicing triggers, and the n-th output is the spliced-out principal.

Setup. U_0 cooperates with U_{n+1} and the U_1 to prepare transactions for splicing, redemption, and slashing (Fig. 10):

- $tx_{splicing}$: the splicing transaction $tx_{splicing}$. $tx_{splicing}$ represents the SPLICED state of channel γ_0, where U_0 splices out a HTLC payment with amount $x + n\epsilon$ in exchange for a preimage s. In *Zeus*, s can be interpreted as the proof of payment provided by U_{n+1}, where U_{n+1} gives s only after receiving the payment. Meanwhile, $tx_{splicing}$ contains n dust outputs with value ϵ, denoted as *splicing triggers* $(tr^1_{\texttt{splicing}}, \ldots, tr^n_{\texttt{splicing}})$. $tr^i_{\texttt{splicing}}$ can be redeemed by U_i's signature, or recollected by U_0 after a relative timelock $rel(2t_{\text{close}} + \Delta)$. This timelock ensures that after the splicing of γ_0 (taking up to t_{close}), U_i still has enough time to close γ_i and revoke its locked deposit or principal if necessary (see Section 6.1).
- tx_{redeem}: the redeem transaction. tx_{redeem} takes the splicing-out output of $tx_{splicing}$ as input along with the preimage s, and contains one output with value x to itself and n *redeem triggers* $(tr^1_{\texttt{redeem}}, \ldots, tr^n_{\texttt{redeem}})$ to signal premium revocation (tx^r_{Rvk} in Section 6.1). $tr^i_{\texttt{redeem}}$ can be spent by U_{i+1}, before a relative timelock $rel(t_{\text{close}} + \Delta)$, leaving enough time for revoking premiums.
- tx_{slash}: the slash transaction. If U_{n+1} fails redeem the spliced payment with s within $rel(t_{\text{close}} + \Delta)$, U_0 can submit tx_{slash} to reclaim its on-chain payment, simultaneously burning U_{n+1}'s stake, Stake. This timelock ensures that even splicing takes up to t_{close} time, U_{n+1} still has Δ time to redeem it.

Now let's dive into how to construct these transaction. *Step 1. Make payee accountable.* At beginning, U_0 first locates and verifies whether U_{n+1} has locked a stake Stake using function checkPayeeStake. Then U_0 requests U_{n+1} to sign off on tx_{slash}. After this step, U_{n+1} will be slashed if it refuses to settle the payment with s. To prevent this slash, honest U_{n+1} will launch a redeem launcher sub-protocol $\Pi_{\text{redeem-launcher}}$ to finalize the tx_{redeem} once $tx_{splicing}$ is confirmed on-chain.

Step 2. Prepare the channel splicing. Now U_0 sends the corresponding setup metadata $m_{\text{setup}}[i]$ to each U_i (e.g., payment amounts and timelocks, triggers in channels γ_i and γ_{i+1}) via standard onion routing. After receiving this metadata, U_i verifies that splicing and redemption triggers are correctly bound within the same transaction and prepares for the subsequent channel update. Specifically, U_1 initializes the splicing sub-protocol $\Pi_{\texttt{2pSetupSplicing}}$, preparing channel γ_0 for transition to the SPLICED state. After setup message broadcast, U_0 initialize the splicing of γ_0 from IDLE state to SPLICED state by invoking $\Pi_{\text{2pSetupSplicing}}$. In $\Pi_{\text{2pSetupSplicing}}$, U_{i+1} and U_i will co-sign $tx_{splicing}$, and splicing γ_0 by calling the UPDATESPLICING interface of $\mathcal{F}_{Channels}$. Thereafter, both U_0 and U_1 maintain the off-chain splicing state. Note to avoid race conditions with potential channel splicing in γ_0, U_0 actually will not participate in the deposit, premium and principal locking.

Forward Deposit Locking. The protocol proceeds sequentially from U_1 to U_n to lock deposits. Each U_i engages with the next participant U_{i+1} using a channel update sub-protocol $\Pi^i_{\text{2pSetupDep}}$ to lock deposit $\texttt{Dep}_i$, where $\texttt{Dep}_i := c_{\text{payee}} + (n-i) \cdot c_{\text{relay}})$, where c_{payee} is the deposit required by the payee, and c_{relay} is the deposit charged by one relay. The value of these deposits is specified in Appendix C.2. $\Pi^i_{\text{2pSetupDep}}$ can be considered as an degenerated instance of the two-stage revocable channel update protocol detailed in Section 6.1 with $\alpha := \texttt{Dep}_i$, $\beta := \perp$, $T_l := T$ and $T_r := \perp$, updating their shared channel γ_i from IDLE to the LOCKED-Dep state.

Backward Premium Locking. After the deposit locking completes up to γ_n, *Zeus* proceeds backward from the last channel towards the first. For each $\gamma_{i\in[1,n]}$, U_{i+1} now initiates the sub-protocol $\Pi^i_{\text{2pSetupPrm}}$ with U_i. This updates γ_i further to the LOCKED-Prm state, committing a premium $\texttt{Prm}_i$ from U_{i+1} to U_i, where $\texttt{Prm}_i := \sum_{j=0}^{i} f_j$. Similarly, $\Pi^i_{\text{2pSetupPrm}}$ can be considered as a reversed instance (where U_{i+1} initializing) of the two-stage revocable update, with $\alpha := \texttt{Dep}_i$, $\beta := \texttt{Prm}_i$, $T_l := T$, and $T_r := T + \Delta$.

Principal Locking. Once U_1 confirms γ_1 has updated to the `LOCKED-Prm` state, starting from γ_1 to γ_n, each U_i and U_{i+1} invokes $\Pi^{i}_{\text{2pSetupPrcpl}}$ to cooperatively update γ_i to the `LOCKED-Prcpl` state, locking the principal amount $\texttt{Prcpl}_i := x + \sum_{j=0}^{n} f_j$ in channel γ_i, containing a base payment x plus all relay fees. Similar to previous setting, but α will be modified to $\alpha := \texttt{Dep}_i + \texttt{Prcpl}$.

Deposit Refund. Start from U_{n+1} to U_2, U_{i+1} refunds $\texttt{Dep}_n$ with a new `REFUNDED-Dep` state through an off-chain sub-protocol $\Pi^{n}_{\text{2pRevokeDep}}$. This step is almost identical to the Lock-premium stage, but reduces U_n's locked amount α from $\texttt{Dep}_i + \texttt{Prm}$ to $\texttt{Prm}$. Simultaneously, U_{n+1} sends the proof of payment s to U_0 privately.

Fast Confirmation. Once U_0 receives s and U_1 confirms all deposits are revoked, to speed up the finalization of the off-chain payment, starting from U_0, each U_i invokes the Π_{2pSettle} sub-protocol to update γ_i to the `PAID` state, where U_i pays an amount of $x + \sum_{j=i}^{n} f_j$ to U_{i+1}.

Handling Stalled Payments. As illustrated in Fig. 16, if the payment has not been completed by $T - 2t_{close} - \Delta$, either U_0 or U_1 can submit $tx_{splicing}$ to initiate dispute resolution. By $T - t_{close} - \Delta$, $tx_{splicing}$ containing n splicing triggers finalizes on $\mathcal{G}_{\text{L}}$. Once the splicing handlers observe this finalization, they execute the deposit/principal revocation process, ensuring that each U_i can reclaim its locked funds by T. Concurrently, when $tx_{splicing}$ finalizes on $\mathcal{G}_{\text{L}}$, the redeem launcher $\Pi_{\texttt{redeem-launcher}}$ executed by U_{n+1} submits tx_{redeem} (taking up to Δ). Following the same revocation mechanism as the deposit/principal recovery, each U_{i+1} can always reclaim its locked premium by $T + \Delta$.

7 Security Analysis

We model *Zeus* in a global universal composability framework, following similar approaches to recent PCN protocols [7–9]. The complete security analysis appears in Appendix H. For each channel update sub-protocol covered in Section 6.2 and dispute resolution sub-protocols summarized in Fig. 17, we provide corresponding ideal functionality and prove it with standard simulation paradigm. Here, we denote all these sub-functionalities as $\mathcal{F}_{\texttt{sub}}$, and global ideal functionalities(e.g., $\mathcal{G}_{\text{L}}$, $\mathcal{F}_{Channels}$) as $\mathcal{F}_{\texttt{prelim}}$. Then, we provide the main functionality $\mathcal{F}_{Zeus}$ in the $\{\mathcal{F}_{\texttt{prelim}}, \mathcal{F}_{\texttt{sub}}\}$ hybrid world, and prove:

Theorem 3. Let Σ be an EUF-CMA secure [13] signature scheme and C be a hiding and binding commitment scheme, then Π_{Zeus} described in Section 6.2 UC-realizes $\mathcal{F}_{Zeus}$.

Due to space constraints, we informally argue here how *Zeus* achieves *balance security* (Def. 2.1), *fee security* (Def. 2.4), and *griefing resistance* (Def. 2.6). Formal proofs are provided in Appendix H.

Balance Security: We first show payers pay only if the payee gets the payment. As shown in Section 6.2, if payee does not receive the money off-chain, payer can always push $tx_{splicing}$ on-chain, revoking all their locked funds and paying payee directly on-chain. For relays, we discuss U_1 and other relays $U_{i\in[2,n]}$ separately. For U_1, *Zeus* ensures U_1 can initiate on-chain principal lock with splicing before U_1 locks any funds in γ_1. The transaction cost for splicing is covered by U_0's balance in γ_0. $\Pi^{1}_{\texttt{splicing-launcher}}$ ensures this splicing occurs timely if payments stalled. Concurrently, $\Pi^{1}_{\texttt{splicing-handler}}$ guarantees that U_1's funds locked in γ_1 can be reclaimed. For any honest $U_{i\in[2,n]}$, *Zeus* ensures atomicity for fund revocation across adjacent channels γ_{i-1} and γ_i through linked triggers. Specifically, U_{i-1} can revoke its funds locked for U_i in γ_{i-1} only if the corresponding splicing event occurs, consuming $tr^{i-1}_{splicing}$. Since $tr^{i-1}_{splicing}$ (for γ_{i-1}) and $tr^{i}_{splicing}$ (for γ_i) are bound within $tx_{splicing}$, the occurrence of the former implies the latter is available, preventing U_i from losing funds but U_{i+1} does not.

Fee Security: For an honest relay $U_{i\in[1,n]}$, once U_i locks any funds, there are two possible cases:

- *$tx_{splicing}$ does not finalize on $\mathcal{G}_L$ in time:* If U_i has locked payer's deposit, it is compensated with c_{relay}. If U_i has locked principal, it can claim its relay fee, from the premium locked by U_{i+1}.
- *$tx_{splicing}$ finalized in $\mathcal{G}_L$ in time.* The off-chain principal will be revoked, so the payment will not settle off-chain. Thereby, the fee security is always secure.

Griefing Resistance against k-Cost-Sensitive Adversaries: For simplicity, we only show the griefing damage and cost under the most dangerous cases. Fig. 4 demonstrates the griefing damage and cost cross MHP protocols, under different adversary setting.

- *Payee griefs multiple deposits by forfeiting payee's on-chain stakes:* If `Stake` significantly exceeds the cumulative (and typically small) liquidity inconvenience across all these minimal deposit locks, the attack is economically irrational. Fig. 11c shows a 2.5 USD on-chain stakes can make the griefing factor lower than 0.1.
- *Payee griefs a single payment right after principals fully locked:* At this time, the griefing damage reach the peak as all funds are locked and non are refunded, payee could begin griefing this MHP by triggering the on-chain dispute, and reclaim its premium back at last minute by submitting tx_{redeem}. Doing so costs the adversary an on-chain fee F while incurring a damage of cumulative liquidity inconvenience across channels. Fig. 11a shows even in the worst LN configuration, the griefing factor $\lambda < 0.1$, while $\lambda \leq 0.01$ for payments lower than \$500.

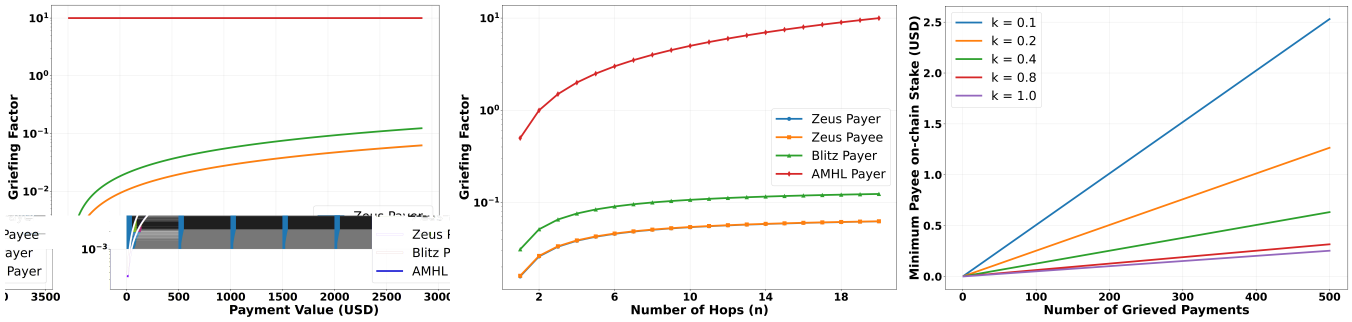
8 Evaluation

Implementation. We implemented *Zeus*[3] using Bitcoin scripts, leveraging the Taproot [42] upgrade. A key challenge is that current Bitcoin does not support the slash mechanism for on-chain stakes. While various proposals exist, contemporary approaches often introduce external trust assumptions or significant computational overheads [15, 23]. Given that the intended slash amount is relatively small, we approximate this slash by mandating that payees can only withdraw their collateral with a resource-intensive spending path. The anticipated high transaction fee incurred to execute this path effectively serves as the burned amount (Appendix C.1).

Computation & Communication Overheads. *Zeus* uses only lightweight primitives (i.e., digital signatures and hash functions), resulting in negligible computational overhead—comparable to standard Bitcoin transactions. The requirement to mitigate griefing and fee-stealing attacks compelled Zeus to introduce three additional phases. Consequently, the settlement time of Zeus (~ 11.3s) is 6.4s longer than that of standard HTLC-based MHP (4.9s), under a 200ms network delay.

Transaction Overheads. Ideally, a n-relay payment in *Zeus* settles entirely off-chain with no on-chain costs. Table 2 compares

[3] Code available at https://anonymous.4open.science/r/Zeus-074B



(a) # of hops $n = 20$. **(b) Payment value $x = \$3400$.** **(c) Payee griefs multiple deposits, $n = 20$.**

Figure 11: (a) and (b) evaluate how the griefing factor varies when different adversaries grief a single payment across protocols. (c) shows the minimum on-chain stake required by the payee for a target upper bound on the griefing factor.

transaction sizes across protocols when payments cannot settle off-chain. In Bitcoin, transaction fees are calculated as vBytes×fee rate, where virtual bytes (vBytes) represent the transaction's size, and the fee rate (typically 1-10 satoshis/vByte) varies with network congestion. For context, a standard pay-to-taproot transaction is 111 vBytes, costing approximately 0.33 USD at a moderate fee rate of 3 satoshis/vByte. When an n-relay payment stalls off-chain, *Zeus*, similar to Blitz, allows the payer to revoke the jammed payment with a single transaction, sizing $154 + 43n$ vBytes. Even in a worst-case scenario where a multi-hop payment reaches the LN's upper bound of 20 hops, the resulting revocation transaction size is 914 vBytes. This is substantially below the Bitcoin transaction limit of 10^6 vBytes, and the associated fee remains acceptable at approximately 2.25 USD. Furthermore, *Zeus* mandates that the payee must also redeem the payment on-chain using a transaction also sized at $154 + 43n$ vBytes, thereby increasing the cost for adversarial payers attempting griefing attacks. As with all MHP protocols, if an honest relay's neighbor refuses off-chain collaboration, the honest relay must first update the channel to its latest state on-chain, sizing at most 197 vBytes (see Table 3), then enforce payment settlement for 138 vBytes or payment revocation for 194 vBytes.

Fee Stealing and Griefing Simulation. We created a realistic testing environment based on current LN topology captured in March 2025 from local LN clients. We formalized this topology as a directed graph where edges represent channels with capacities and fee policies (base fee plus fee rate). This graph contains 15,447 nodes and 71,007 edges (channels) with an average capacity of 5.68M satoshis (~0.057 BTC). The block delay Δ is configured as 6 Bitcoin blocks (1 hour). For HTLC, we used the minimum secure lock delta δ of 18 blocks (3 hours), and channel closure upper bound is configured as 1 hour. In LN, the single payment limit is 0.042 BTC(est. \$3400), and the maximum number of hops is 20. To simulate MHPs, we first (uniformly or liquidity-weighted) sampled payer-payee pairs, then found the lowest-fee path between them.

Griefing Attack Analysis. We assess griefing vulnerability by measuring griefing damage and griefing factor for MHPs.

Griefing Damage. Fig. 12 and Fig. 18 illustrate how collateral scales with payment amount across protocols. For a 200 USD payment under uniform sampling, the linear collateral inefficiency of linear collateral protocols like AMHL results in approximately

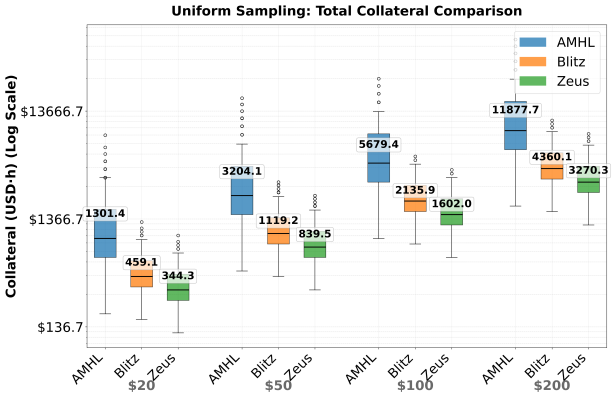


Figure 12: Uniformly sampled collateral with different payment values. We highlight the mean collateral of each test.

11,877 USD·h of locked collateral during an attack. This vulnerability escalates dramatically if a payee griefs multiple payments concurrently; with an upper bound of 483 such payments, potential damage could reach 5.5M USD·h. Both Blitz and *Zeus* significantly reduce griefing damage compared to HTLC/AMHL—by approximately 63% and 72% respectively—by implementing constant timelocks across channels. *Zeus* exhibits 25% lower collateral than Blitz due to its shorter payment timelock ($2t_{close} + \Delta$) versus Blitz's ($t_{close} + 3\Delta$, see [8]'s Fig.10).

Griefing Factor. Zeus enforces a minimal griefing factor (GF), rendering attacks economically irrational by combining liquidity costs (opportunity loss from locked funds) and transaction fees. Unlike Fulgor, AMHL, Astrape, or Blitz—where malicious payees execute cost-free attacks with unbounded GF—*Zeus* caps GF at ≤ 0.1 even at maximum payment values (Fig. 11a), regardless of payer/payee compromise. Fig. 11a further demonstrates that for typical payments less than \$500 USD, GF is less than $0.01 \ll 1$. Corrupting both parties doubles transaction costs, further reducing GF. Assuming a 1% annual interest rate (matching ACINQ's Lightning Network rate), Fig. 11b shows GF's path-length dependence. *Zeus* requires payees to forfeit on-chain stake to grief multiple deposits; Fig. 11c quantifies the minimum required stake to ensure the griefing factor remains below a threshold k. Even when a payee

simultaneously griefs 483 MHPs and the required factor is less than 0.1, the necessary on-chain stake remains only a few USD.

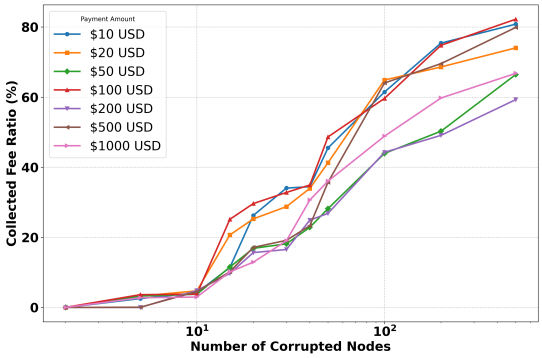


Figure 13: Fee ratio under weighted sampling.

Fee Stealing Attack Analysis. We evaluate fee-stealing attacks by measuring the maximum portion of relay fees that colluding nodes can capture. We designate the top μ liquid nodes as potential colluders. The results demonstrate significant vulnerability. Under weighted sampling (Fig. 13), merely 0.6% of network nodes can capture over 40% of total relay fees. Under uniform sampling (Fig. 19), the same fraction captures approximately 25% of fees. This disparity stems from the LN's hub-and-spoke structure, where high-liquidity nodes naturally attract more routing traffic. Under weighted sampling, payments preferentially traverse these central nodes, amplifying adversarial advantage.

9 Discussion

UTXO Interoperability. Bitcoin's UTXO model lacks native support for direct interaction between arbitrary UTXOs. Protocols like *Zeus* rely on dust outputs to signal state changes, increasing transaction fees and contributing to UTXO set bloat. Exploring alternatives for UTXO interoperability would benefit the broader Bitcoin scaling ecosystem like BitVM or Discreet Log Contracts.
Privacy Concerns. *Zeus* employs channel splicing to improve capital efficiency. However, this potentially exposes payers' identities even when utilizing stealth addresses [8]. A potential mitigation involves payers utilizing untraceable funds to replace splicing.
Applications of FSEs. We prove FSEs are impossible only with the existence of public blockchains. FSE is the foundational model to various applications. For example, [35] frames the pre-confirmation between users and block proposers as FSE. Investigating how our results apply to such contexts is an interesting avenue for future research.

References

[1] Anonymous. 2024. LN BOLT 2: Peer Protocol for Channel Management. https://github.com/lightning/bolts/blob/master/02-peer-protocol.md.
[2] Anonymous. 2025. bitcoinjs-lib - v7.0.0. https://bitcoinjs.github.io/bitcoinjs-lib/.
[3] Anonymous. 2025. LN BOLT 3: Bitcoin Transaction and Script Formats. https://github.com/lightning/bolts/blob/master/03-transactions.md.
[4] Nadarajah Asokan, Victor Shoup, and Michael Waidner. 1998. Asynchronous protocols for optimistic fair exchange. In *Proceedings of 19th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 86–99.
[5] Nadarajah Asokan, Victor Shoup, and Michael Waidner. 1998. Optimistic fair exchange of digital signatures. In *Proceedings of the 16th International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, 591–606.
[6] Lukas Aumayr, Kasra Abbaszadeh, and Matteo Maffei. 2022. Thora: Atomic and Privacy-Preserving Multi-Channel Updates. In *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[7] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. 2021. Generalized channels from limited blockchain scripts and adaptor signatures. In *Proceedings of the 27th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. Springer, 635–664.
[8] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. 2021. Blitz: Secure Multi-Hop Payments Without Two-Phase Commits. In *Proceedings of 30th USENIX Security Symposium (USENIX Security)*.
[9] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. 2023. Breaking and Fixing Virtual Channels: Domino Attack and Donner. In *Proceedings of the 30th Network and Distributed System Security Symposium (NDSS)*.
[10] Lukas Aumayr, Sri AravindaKrishnan Thyagarajan, Giulio Malavolta, Pedro Moreno-Sanchez, and Matteo Maffei. 2022. Sleepy Channels: Bi-directional Payment Channels without Watchtowers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[11] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. 2018. The Wonderful World of Global Random Oracles. Cryptology ePrint Archive.
[12] Jan Camenisch and Anna Lysyanskaya. 2005. A formal treatment of onion routing. In *Proceedings of the 25th Annual International Cryptology Conference (CRYPTO)*. Springer, 169–187.
[13] Ran Canetti. 2004. Universally composable signature, certification, and authentication. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004. (CSFW)*.
[14] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. 2007. Universally Composable Security with Global Setup. In *Proceedings of the 4th Theory of Cryptography (TCC)*.
[15] Xinshu Dong, Orfeas Stefanos Thyfronitis Litos, Ertem Nusret Tas, David Tse, Robin Linus Woll, Lei Yang, and Mingchao Yu. 2025. Remote Staking with Optimal Economic Safety. arXiv:2408.01896 [cs.CR] https://arxiv.org/abs/2408.01896
[16] Yuhao Dong, Ian Goldberg, Sergey Gorbunov, and Raouf Boutaba. 2022. Astrape: Anonymous payment channels with boring cryptography. In *Proceeding of the 20th International Conference on Applied Cryptography and Network Security (ACNS)*. Springer, 748–768.
[17] Christoph Egger, Pedro Moreno-Sanchez, and Matteo Maffei. 2019. Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[18] Oğuzhan Ersoy, Jérémie Decouchant, Satwik Prabhu Kimble, and Stefanie Roos. 2022. SyncPCN/PSyncPCN: Payment Channel Networks without Blockchain Synchrony. In *Proceedings of the 4th ACM Conference on Advances in Financial Technologies (AFT)*. 16–29.
[19] Oğuzhan Ersoy, Pedro Moreno-Sanchez, and Stefanie Roos. 2023. Get Me Out of This Payment! Bailout: An HTLC Re-routing Protocol. In *Proceedings of 27th Financial Cryptography and Data Security (FC)*. Springer, 92–109.
[20] S. Even and Yacov Yacobi. 1980. Relations amoung public key signature systems. *Technion - Isrel Institute of Technology, Technical Report 175*.
[21] William George. 2023. An analysis of griefs and griefing factors. *Frontiers in Blockchain*.
[22] Runchao Han, Haoyu Lin, and Jiangshan Yu. 2019. On the optionality and fairness of atomic swaps. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies (AFT)*. 62–75.
[23] Ethan Heilman, Victor I Kolobov, Avihu M Levy, and Andrew Poelstra. 2024. ColliderScript: Covenants in Bitcoin via 160-Bit Hash Collisions. *Cryptology ePrint Archive*.
[24] Ethan Heilman, Victor I. Kolobov, Avihu M. Levy, and Andrew Poelstra. 2024. ColliderScript: Covenants in Bitcoin via 160-bit hash collisions. Cryptology ePrint Archive, Paper 2024/1802.
[25] Maurice Herlihy. 2018. Atomic cross-chain swaps. In *Proceedings of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*. 245–254.
[26] Maxim Jourenko, Mario Larangeira, and Keisuke Tanaka. 2021. Payment trees: Low collateral payments for payment channel networks. In *Proceedings of the 25th Financial Cryptography and Data Security (FC)*. Springer, 189–208.
[27] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. 2013. Universally composable synchronous computation. In *Proceedings of Theory of Cryptography Conference (TCC)*.
[28] Satwik Prabhu Kumble, Dick Epema, and Stefanie Roos. 2023. Game-Theoretic Analysis of (Non-) Refundable Fees in the Lightning Network. In *Proceedings of the 29th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 645–652.

[29] Yehuda Lindell. 2021. Fast secure two-party ECDSA signing. *Journal of Cryptology* 34, 4, 44.

[30] Robin Linus, Lukas Aumayr, Alexei Zamyati, Andrea Pelosi, Zeta Avarikioti, and Matteo Maffei. 2024. BitVM2: Bridging Bitcoin to Second Layers. https://bitvm.org/bitvm_bridge.pdf.

[31] Aumayr Lukas, Avarikioti Zeta, Salem Iosif, Schmid Stefan, and Yeo Michelle. 2025. X-Transfer: Enabling and Optimizing Cross-PCN Transactions. In *Proceedings of 20th Financial Cryptography and Data Security (FC)*.

[32] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. 2017. Concurrency and Privacy with Payment-Channel Networks. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[33] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. 2019. Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability. In *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS)*.

[34] Subhra Mazumdar, Prabal Banerjee, and Sushmita Ruj. 2020. Time is money: Countering griefing attack in lightning network. In *Proceeding of the 19th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 1036–1043.

[35] Conor McMenamin and Lin Oshitani. 2025. Preconfirmation Fair Exchange. Ethereum Research Forum. https://ethresear.ch/t/preconfirmation-fair-exchange/21891

[36] Andrew Miller, Iddo Bentov, Surya Bakshi, Ranjit Kumaresan, and Patrick McCorry. 2019. Sprites and State Channels: Payment Networks that Go Faster Than Lightning. In *Proceedings of Financial Cryptography and Data Security (FC)*. Springer, 508–526.

[37] Ayelet Mizrahi and Aviv Zohar. 2021. Congestion attacks in payment channel networks. In *Proceedings of the 25th International Conference on Financial Cryptography and Data security (FC)*. Springer, 170–188.

[38] Tejaswi Nadahalli, Majid Khabbazian, and Roger Wattenhofer. 2022. Grief-free Atomic Swaps. In *Proceedings of the 4th IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 1–9.

[39] Henning Pagnia and Felix C. Gartner Darmstadt. 1999. On the Impossibility of Fair Exchange without a Trusted Third Party.

[40] Henning Pagnia, Holger Vogt, and Felix C Gärtner. 2003. Fair exchange. *Comput. J.* 46, 1, 55–75.

[41] Cristina Pérez-Sola, Alejandro Ranchal-Pedrosa, Jordi Herrera-Joancomartí, Guillermo Navarro-Arribas, and Joaquin Garcia-Alfaro. 2020. Lockdown: Balance availability attack against lightning network channels. In *Proceedings of the 24th International conference on financial cryptography and data security (FC)*. Springer, 245–263.

[42] Anthony Towns Pieter Wuille, Jonas Nick. 2020. BIP-341: Taproot: SegWit version 1 spending rules. https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki.

[43] Joseph Poon and Thaddeus Dryja. 2016. The bitcoin lightning network: Scalable off-chain instant payments.

[44] Minfeng Qi, Qin Wang, Zhipeng Wang, Manvir Schneider, Tianqing Zhu, Shiping Chen, William Knottenbelt, and Thomas Hardjono. 2024. SoK: Bitcoin Layer Two (L2). arXiv:2409.02650 [cs.CR] https://arxiv.org/abs/2409.02650

[45] Xianrui Qin, Shimin Pan, Arash Mirzaei, Zhimei Sui, Oğuzhan Ersoy, Amin Sakzad, Muhammed F Esgin, Joseph K Liu, Jiangshan Yu, and Tsz Hon Yuen. 2023. Blindhub: Bitcoin-compatible privacy-preserving payment channel hubs supporting variable amounts. In *Proceedings of the 44th 2023 IEEE symposium on security and privacy (S&P)*. IEEE, 2462–2480.

[46] Sophie Rain, Georgia Avarikioti, Laura Kovács, and Matteo Maffei. 2023. Towards a game-theoretic security analysis of off-chain protocols. In *Proceedings of the 36th IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 107–122.

[47] Elias Rohrer, Julian Malliaris, and Florian Tschorsch. 2019. Discharged payment channels: Quantifying the lightning network's resilience to topology-based attacks. In *Proceeding of the 4th IEEE european symposium on security and privacy workshops (EuroS&PW)*. IEEE, 347–356.

[48] Rusty Russell. 2015. Loop attack with onion routing. https://diyhpl.us/~bryan/irc/bitcoin/bitcoin-dev/linuxfoundation-pipermail/lightning-dev/2015-August/000135.txt.

[49] Rusty russell. 2021. Splicing Proposal. https://github.com/lightning/bolts/pull/863.

[50] Clara Shikhelman and Sergei Tikhomirov. 2022. Unjamming Lightning: A Systematic Approach. Cryptology ePrint Archive, Paper 2022/1454. https://eprint.iacr.org/2022/1454

[51] Lightspark Team. 2025. 15 persent of Bitcoin Transactions on Coinbase Now Move Over Lightning. https://www.lightspark.com/blog/news/coinbase-lightning-network-lightspark.

[52] Yingjie Xue and Maurice Herlihy. 2021. Hedging against sore loser attacks in cross-chain transactions. In *Proceedings of the 40th ACM Symposium on Principles of Distributed Computing (PODC)*. Association for Computing Machinery, 155–164.

[53] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 162–167.

[54] Yi Zhang, Xiaofeng Jia, Bianjing Pan, Jun Shao, Liming Fang, Rongxing Lu, and Guiyi Wei. 2023. Anonymous multi-hop payment for payment channel networks. *IEEE Transactions on Dependable and Secure Computing (TDSC)* 21, 1, 476–485.

A Fair secret exchange with public TTPs

A.1 Modeling Fair secret exchanges

Background. In a *fair secret exchange* with a public blockchain, consider two parties, denoted as $\mathcal{P}$ and $\mathcal{Q}$, who want to exchange their secrets. At beginning, $\mathcal{P}$ possesses a secret item $i_{\mathcal{P}}$, and $\mathcal{Q}$ possesses a secret item $i_{\mathcal{Q}}$ with corresponding descriptions $(\text{desc}_{\mathcal{P}}, \text{desc}_{\mathcal{Q}})$. The descriptions are common knowledge of all parties. A state machine (smart contract) deployed on the public blockchain functions as a **public trusted third party** $\mathcal{T}$ (public TTP). This entity, $\mathcal{T}$, processes requests from external users in each round and publicly responds in the subsequent round, assuming its program is publicly known from round 0. As the blockchain is public, every message sent to $\mathcal{T}$ at round r is considered broadcasted to all participants and external observers.

Assumptions. In our settings, all protocols operate under the standard round-based synchronous network model [27]. In this model, when an honest party transmits a message in round r to an honest recipient, the recipient receives the message at the beginning of round $r+1$. All inter-party communication is conducted through private, authenticated channels, underpinned by modern public-key infrastructure. We consider a static, *probabilistic-polynomial-time* (PPT) adversary $\mathcal{A}$ capable of corrupting either party $\mathcal{P}$ or $\mathcal{Q}$ prior to the protocol's commencement.

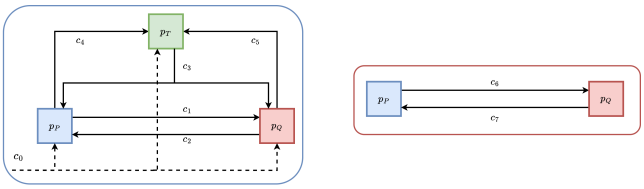


Figure 14: The model of Fair Secret Exchange with/without public blockchain.

Definition of fair secret exchange. We formally define a *Fair Secret Exchange* protocol, denoted as Π_{FSE}, operating with a public blockchain. This definition adheres to the standard structure outlined in [4, 5, 39]. Π_{FSE} comprises three processes: $p_{\mathcal{P}}$, $p_{\mathcal{Q}}$, and $p_{\mathcal{T}}$. Each process is characterized by *input variables* and *output variables*. Communication occurs through designated *channels*.

We use the notation $p_x.\text{input}_i^\tau$ and $p_x.\text{output}_i^\tau$ to denote the input or output variable of party $x \in \{\mathcal{P}, \mathcal{Q}, \mathcal{T}\}$ at round τ within channel c_i. The notation h_x^τ represents the complete input history of party x up to round τ. Within the public blockchain, a *random oracle* [11] is assumed to exist, which generates a public random string $r_t \leftarrow \{0,1\}^*$ as input for each party at round t through channel c_0. Fig. 14 illustrates the model of Π_{FSE} with and without a public blockchain, depicted in blue and red boxes, respectively. In the FSE model with public TTP (blue box in Fig. 14), there are the following *channels*:

- c_0: the public channel for the random oracle.

- c_1: the channel for messages sent from $\mathcal{P}$ to $\mathcal{Q}$.
- c_2: the channel for messages sent from $\mathcal{Q}$ to $\mathcal{P}$.
- c_3: the broadcast channel sent from the public TTP $\mathcal{T}$.
- c_4: the channel for messages sent from $\mathcal{P}$ to $\mathcal{T}$.
- c_5: the channel for messages sent from $\mathcal{Q}$ to $\mathcal{T}$.

At the initial round (0), $p_{\mathcal{P}}$ receives input $(i_{\mathcal{P}}, auxi_{\mathcal{P}})$, $p_{\mathcal{Q}}$ receives input $(i_{\mathcal{Q}}, auxi_{\mathcal{Q}})$, where $auxi_{\mathcal{P}}$ and $auxi_{\mathcal{Q}}$ are auxiliary inputs. The TTP process $p_{\mathcal{T}}$ receives the initial state state_0 as the input.

For each process p_x, where $x \in \{\mathcal{P}, \mathcal{Q}, \mathcal{T}\}$, the fair secret exchange protocol defines a unique state function f_x, which determines the outputs of p_x.

Now we formally define fairness of *fair secret exchange* previously introduced in Section 4. The definition of timeliness and effectiveness of FSE is equivalent to FE.

Definition A.1 (Fairness)**.** For any party $x \in \{\mathcal{P}, \mathcal{Q}\}$, for any PPT adversary $\mathcal{A}$, when process p_x terminates, if x fails to obtain i_{-x} matching desc_{-x}, $\mathcal{A}$ and/or $-x$ must know nothing about x's secret. Formally, this means they can not distinguish the real secret item of x, i_x with any random secret, i'_x, sampled in the same domain:

$$\left|\Pr[\mathcal{A}(\text{view}^N_{-x}, i_x) = 1] - \Pr[\mathcal{A}(\text{view}^N_{-x}, i'_x) = 1]\right| \leq negl(\lambda)$$

where $negl(\lambda)$ is a negligible function, view^{τ}_{-x} represents the view of the counterparty up to the final round round N, i'_x is sampled uniformly at random from the same domain as i_x, and λ is the security parameter.

A.2 Impossibility result of fair secret exchange

In this work, our proof adopts a similar contradiction argument as presented by Even and Yacobi [20], who proving that FE is impossible without a TTP. The central insight of their proof lies in the observation that during a fair secret exchange, the secrets must, at some point, be transmitted over the communication channel between $\mathcal{P}$ and $\mathcal{Q}$. However, within a mutually distrustful environment, neither party is willing to transmit its secret first. Their formalization of this proof employed a contradiction argument.

Proof for Theorem 1. We proceed to prove Theorem 1 by contradiction. Given that *Effectiveness* and *Timeliness* are satisfied, there exists a round $n \leq N$ such that a party x (without loss of generality, let x be $\mathcal{P}$) gains initial information about $i_{\mathcal{Q}}$, while its counterparty $\mathcal{Q}$ remains uninformed about $i_{\mathcal{P}}$. Formally, at round n, there exists a probabilistic polynomial-time (PPT) distinguisher $\mathcal{D}$ satisfying:

$$|\Pr[\mathcal{D}(\text{view}^n_{\mathcal{P}}, i_{\mathcal{Q}}) = 1] - \Pr[\mathcal{D}(\text{view}^n_{\mathcal{P}}, i'_{\mathcal{Q}}) = 1]| > negl(\lambda)$$

for some non-negligible function, where $i'_{\mathcal{Q}}$ is a random secret sampled from the same distribution as $i_{\mathcal{Q}}$, and $\text{view}^n_{\mathcal{P}}$ represents $\mathcal{P}$'s history up to round n.

Conversely, we formally define that $\mathcal{Q}$ knows nothing about $i_{\mathcal{P}}$: for any PPT distinguisher $\mathcal{D}'$

$$|\Pr[\mathcal{D}'(\text{view}^n_{\mathcal{Q}}, i_{\mathcal{P}}) = 1] - \Pr[\mathcal{D}'(\text{view}^n_{\mathcal{Q}}, i'_{\mathcal{P}}) = 1]| \leq negl(\lambda)$$

for some negligible function $negl(\lambda)$, where $i'_{\mathcal{P}}$ is a random secret sampled from the same distribution as $i_{\mathcal{P}}$, and $\text{view}^n_{\mathcal{Q}}$ represents $\mathcal{Q}$'s history up to round n.

Based on the synchronous communication model, there must exist a round n' ($n' < n$) where process $p_{\mathcal{Q}}$ sends a message that helps $\mathcal{P}$ to "learn" $i_{\mathcal{Q}}$ at round n. Now consider the implications of $\mathcal{Q}$'s fairness at round n'. This property necessitates that even if $\mathcal{P}$ terminates the protocol at round n', $\mathcal{Q}$ retains the ability to deterministically recover $i_{\mathcal{P}}$ within the finite round N, aided by $\mathcal{T}$.

As $N > n'$, by definition, at round n', $\mathcal{Q}$ must ensure that $\mathcal{Q}$ and $\mathcal{T}$ possess the capability to recover $i_{\mathcal{P}}$ deterministically by round N, irrespective of $\mathcal{P}$'s outputs from round n' onwards and the random stream from channel r_0. Formally, at round n', $\mathcal{Q}$ can ensure that by round N, there exists a DPT extractor E such that:

$$E(\text{view}^{n'}_{\mathcal{Q}}, \text{view}^{n'}_{\mathcal{T}}, \{r_{n'}, \ldots, r_N\}) = i_{\mathcal{P}}$$

Since $\mathcal{T}$ is a public TTP, we have:

$$\text{view}^{n'}_{\mathcal{T}} \subset \text{view}^{n'+1}_{\mathcal{Q}}$$

As $n' < n \leq N$, $n' + 1 \leq N$, $\text{view}^{n'+1}_{\mathcal{T}} \subseteq \text{view}^N_{\mathcal{Q}}$, so at round N, there exists a DPT extractor E' such that:

$$E'(\text{view}^N_{\mathcal{Q}}, \{r_{n'}, \ldots, r_N\}) = i_{\mathcal{P}}$$

Therefore, at round n', the FSE protocol guarantees that $\mathcal{Q}$ will be capable to extract the secret $i_{\mathcal{P}}$ with probability 1 at round N. So any rational $\mathcal{Q}$ will not send the message to $\mathcal{P}$ at round n'. Therefore, round n will never exist, leading to a contradiction.

Symmetrically, a similar contradiction can be demonstrated in the scenario where $\mathcal{Q}$ initially gains information about $i_{\mathcal{P}}$ while $\mathcal{P}$ remains uninformed about $i_{\mathcal{Q}}$. □

B Extended proof for Griefing Resistant MHPs

B.1 Strong griefing Resistant MHP in Payment Hubs

Here, we construct a simple MHP protocol Π_{MHP} to instance the fair exchange proposed in the payment hub model in Section 4.1.1, thereby achieving a strong griefing resistance under the assumption that payer and payee are controlled by the same entity. Π_{MHP} is a simple lock-resolve MHP with two phases:

- **Lock Setup**: Payee U_2 give a lock message verification function $\text{Ver}^1_{\text{lock}}(\cdot)$ to U_0. $\text{Ver}^1_{\text{lock}}(m)$ returns true only if m is the correct lock message m^1_{lock}, which transfer γ_1 to `locked` state. And this locked condition allows U_2 to unlock the payment once after $\gamma_1 \rightarrow$ `locked`.
- **Lock Phase**: U_0 locks γ_0, with amount $x + f_1$, and lock condition cond_0 returns true if a message m satisfying $\text{Ver}^1_{\text{lock}}$ provided before time T".
- **Redeem Phase**: U_1 sends the lock message m^1_{lock} to U_0 and settles γ_0 to `paid`. Then U_0 sends m^1_{lock} (by mutual trust) to U_2, and finally $\gamma_1 \rightarrow$ `paid`.

Security follows from the immediate settlement of γ_0 upon m^1_{lock} delivery. Strong griefing resistance holds as U_1's fee is atomically redeemed with payment forwarding. Fig. 5 illustrates this protocol in the fair exchange model-U_0 (cooperates with U_2) settles γ_0 to *paid* only if U_1 delivers the correct lock message m^1_{lock}.

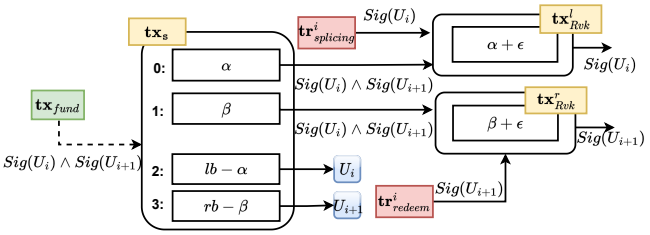


Figure 15: Transaction structure of two-stage revocable channel state.

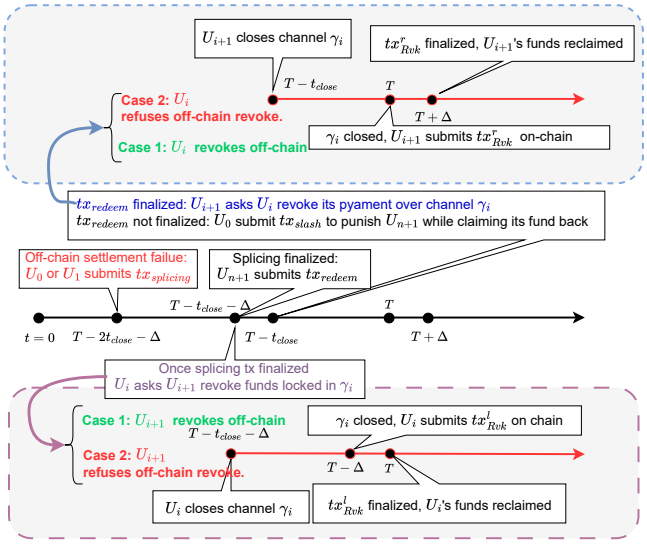


Figure 16: The timeline of the *Zeus* execution. The lower purple dashed box shows how relay U_i reclaim its locked payment over channel γ_i when splicing happens; The upper blue dashed box shows how the relay U_{i+1}reclaim its locked payment(premium) over γ_i when the redeem happens.

(1) $\Pi_{\text{splicing-launcher}}$: Automatically initiates channel splicing at time $T - 2t_{\text{close}} - \Delta$ if payment hasn't settled off-chain.
(2) $\Pi_{\text{redeem-launcher}}$: Monitors the ledger $\mathcal{G}_{\text{L}}$ for confirmation of splicing transaction tx_{splicing} and posts redemption transactions when appropriate, enabling payees to claim funds.
(3) $\Pi_{\text{slash-launcher}}$: Enforces protocol accountability by posting slash transactions when spliced payment remains unspent, penalizing misbehaving parties.
(4) $\Pi^i_{\text{splicing-handler}}$: Maintained by U_i, if the splicing transaction tx_{splicing} is finalized in $\mathcal{G}_{\text{L}}$ before time T, then closes channel γ_i and revokes its locked funds (deposit or principal).
(5) $\Pi^i_{\text{redeem-handler}}$: Maintained by U_{i+1}, if the redemption transaction tx_{redeem} is finalized in $\mathcal{G}_{\text{L}}$ before time $T + \Delta$, then closes channel γ_i and revokes its locked premium.

Figure 17: Sub-protocols of *Zeus* for handling disputes.

Case	HTLC/AMHL		Blitz		*Zeus*	
	# txs	vBytes	# txs	vBytes	# txs	vBytes
Payer splices/revokes	-	-	1	154 + 43n	1	154 + 43n
Payee redeems	-	-	-	-	1	154 + 43n
Relay revokes	1	138	1	194	1	194
Relay redeems	1	154/138	1	138	1	138

Table 2: Transaction cost comparison across protocols.

State	$tx_{\texttt{IDLE}}$	$tx_{\texttt{Lock-Dep}}$	$tx_{\texttt{Lock-Prm}}$	$tx_{\texttt{Lock-Prcpl}}$	$tx_{\texttt{Rfd-Dep}}$	$tx_{\texttt{Paid}}$
vsize	111	154	197	197	197	111

Table 3: Transaction size of *Zeus* in different states. For reference, a standard channel state transaction including a HTLC output is 154 vBytes.

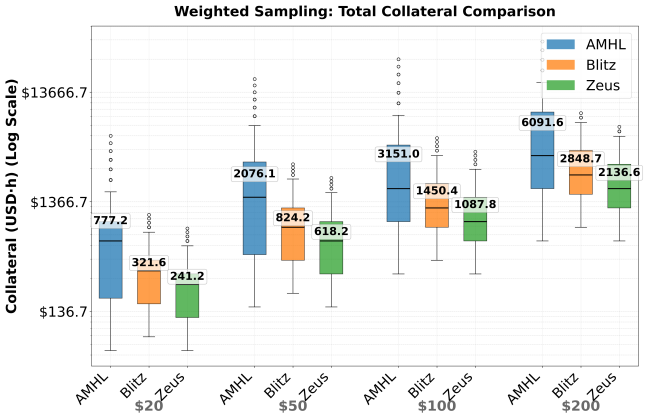


Figure 18: Weighted sampled collateral.

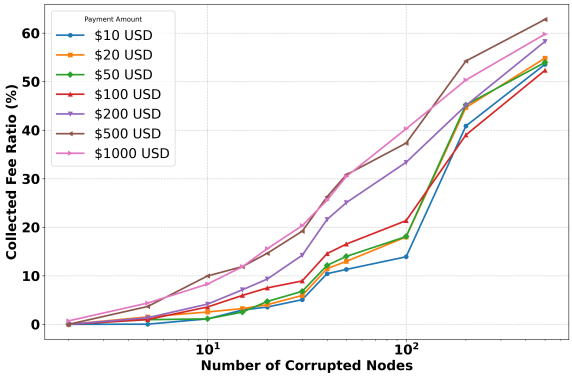


Figure 19: Relay fee collected by corrupted nodes with uniform sampling.

C Supplemental Discussions of Zeus.

C.1 Slash Functionality over Bitcoin Script

Our protocol requires the payee's on-chain stake to implement a slash mechanism with conditional spending rules: 1) early withdrawal before expiry time T_{expire} with penalty c_{burn}, or 2) full withdrawal after T_{expire} without penalty. Implementing such conditional forfeiture in Bitcoin's stateless script environment traditionally requires covenants — proposed opcodes that constrain how UTXOs

Protocol	Griefing Damage	Payer corrupted		Payee corrupted		Both corrupted
		payoff_{adv}	payoff_{payee}	payoff_{adv}	payoff_{payer}	payoff_{adv}
HTLC/AMHL	$ir \cdot p \cdot \Theta(n^2x + n^3f)$	-	-	0	0	0
Blitz	$ir \cdot p \cdot \Theta(nx + n^2f)$	$-rp(154 + 43n)$	0	0	$-43nr$	$-rp(154 + 43n)$
Zeus(case 1)	$ir \cdot \cdot \Theta(nc_{payee} + n^2c_{relay})$	-	-	$-\min\{rp(154 + 43n), c_{\text{burn}}\}$	$-43nr$	$-\min\{rp(154 + 43n), c_{\text{burn}}\}$
Zeus(case 2)	$ir \cdot p \cdot \Theta(n(x + c_{payee}) + n^2(f + c_{relay}))$	$-rp(154 + 43n)$	$-43rpn$	$-rp(154 + 43n)$	$-43nr$	$-2rp(154 + 43n)$

Table 4: Comparison of griefing damage and costs. An adversary controls p in-flight payments with amount x and n relays per payment, each charging a relay fee f, with fee rate r USD/vBytes. ir is the interest rate for griefed funds locked in channels. Case 1: Griefing before the Principal Locking phase; Case 2: Griefing during/after the Principal Locking phase. For on-chain settlement, the payoff for honest parties excludes the idle on-chain transaction costs.

can be spent [4]. While various covenant proposals exist, most require consensus changes. Alternative approaches like multi-signature committees [30] or hash-based constructions [24] avoid soft forks but introduce additional trust assumptions or computational overhead.

We propose a practical alternative that leverages Bitcoin's existing fee market mechanism. Instead of burning c_{burn}, we structure the early withdrawal path to require excessive transaction fees, effectively transferring the penalty to miners. This is implementable using Taproot's script-path spending by constructing a computationally expensive witness script for early withdrawal (e.g., requiring multiple redundant signature verifications or hash computations). The honest payee uses the key-path spending after T_{expire}, incurring only standard transaction costs.

The security of this approach relies on the honestly of Bitcoin miners. Given that c_{burn} represents a small fraction of typical block rewards ($0.25-5 versus $250,000), and Bitcoin's mining ecosystem exhibits strong decentralization, coordination between miners and adversarial payees is both economically irrational and operationally infeasible. The penalty mechanism thus provides equivalent security guarantees to direct burning while maintaining compatibility with current Bitcoin consensus rules.

C.2 Value of Deposits and Stakes

At the beginning of payment, the payee needs to lock a stake `Stake` on-chain to hedge against potential griefing of the payer's off-chain deposit. Recall that the off-chain deposit of the payer in channel γ_i ($i \in [1, n]$), denoted as $\texttt{Dep}_i$, has a value of $c_{\text{payee}} + (n - i) \cdot c_{\text{relay}}$. This off-chain deposit hedges the griefing risk associated with the future premium, which amounts to $f_0 + \sum_{i=1}^{n} f_i$, where f_i is the fee charged by relay node U_i, and f_0 is the padding fee. The padding fee ensures that the relay fee in each hop is sufficient(bigger than the dust limit F_{dust}) to be safely routed through the PCNs:

$$\text{For } \forall k \in [1, n], \quad \sum_{i=0}^{k} f_i \geq F_{\text{dust}} \tag{1}$$

Next, we show that to ensure balance security, the payer's deposit must satisfy

$$c_{\text{payee}} \geq \sum_{i=1}^{n} f_i + F_{\text{dust}} \tag{2}$$

[4] https://bitcoinops.org/en/topics/covenants/

This requirement can be understood by examining the last step of the payer's deposit lock phase (`Lock-Dep`), where the last relay U_n locks payment amount $\texttt{Dep}_n = c_{\text{payee}}$ in channel γ_n to payee U_{n+1}. The payee then locks the premium back in channel γ_n with amount $\texttt{Prm}_n = f_0 + \sum_{i=1}^{n} f_i$. To ensure that even if all other parties go offline, there is no loss for the payee, we require $c_{\text{payee}} \geq f_0 + \sum_{i=1}^{n} f_i$. Since the padding fee f_0 must satisfy $f_0 \geq F_{\text{dust}} - \sum_{i=1}^{n} f_i$ when $\sum_{i=1}^{n} f_i < F_{\text{dust}}$, we can derive Equation 2.

Now we discuss the maximum values of c_{payee} and value of c_{relay}. According to Equation 2, the maximum value of c_{payee} is determined by the maximum total relay fees, which depend on the maximum payment amount and the maximum number of hops. In the LN, the default single payment limit is 0.042 BTC (approximately 3400 USD), and the maximum number of hops is 20. With a typical fee rate of 0.2% (0.01% per hop × 20 hops), the total fees would be around 6.8 USD. Adding the dust limit (typically 546 satoshis, around $0.45), the value of c_{payee} is approximately 7.25 USD (around 9k satoshis). The value of c_{relay} can be considered as the relay fee to route c_{payer}. In *Zeus*, we set c_{relay} to the minimum relay fee of (1 satoshi), as this minimum fee is sufficient to cover the liquidity cost of routing payer's deposit(consider the common fee rate is about 0.01%)

Finally, we calculate the required value of the payee's on-chain stake `Stake`. As shown in Table 4 (case 1), this stake must be sufficient to hedge the griefing risk of p in-flight payer deposits. The worst-case griefing damage(locked amount times locked time) can be calculated as:

$$r \cdot p \cdot (c_{\text{payee}} \cdot n_{\max} + \frac{n_{\max}(n_{\max} + 1)}{2} c_{\text{relay}})(2T_{\text{close}} + \Delta)$$

where:

- $p = 483$ is the maximum number of concurrent MHPs in the Lightning Network.
- $n_{\max} = 20$ is the maximum number of hops.
- $\frac{n_{\max}(n_{\max}+1)}{2} c_{\text{relay}}$ is the total c_{relay} locked in this MHP.
- $(2T_{\text{close}} + \Delta)$ is the timelock and r is the interest rate.

Under the worst case, the total griefing collateral is approximate 221k USD·h. Assuming an interest rate of 1% per year(interest rate charged by ACINQ one of the largest liquidity providers in Lightning Network), the griefing damage derived from payer's collateral is approximately 0.25 USD. If we further constraint the griefing factor lower than 0.1, a $\sim$ 2.5 USD global stake will be sufficient.

D When payer and payee have conflict interests: MHP with credential exchange support

Theorem 2 in the paper demonstrates that no *minimal secure* lock-resolve MHP protocol with more than one relay can satisfy *strong griefing resistance*, even when the payer and payee are assumed to cooperate. We now extend this finding to scenarios where payer and payee may have conflicting interests, specifically by considering MHPs that support *credential exchange*. In this model, upon successful settlement, the payee receives a predefined credential, s (e.g., the preimage of a hash), from the payer. Such a credential can function as a proof of payment (receipt). We denote an MHP augmented with credential exchanges as $\mathcal{M}_{\text{CE}} := (\mathcal{M}, s, h)$, where $\mathcal{M}$ encompasses the standard MHP parameters, s is the secret credential, and h is the commitment to s.

This extension leads to the following corollary:

Corollary 4. *There is no minimal secure lock-resolve MHP with at lease one relay, Π_{MHP}, that simultaneously satisfies strong griefing resistance and supports credential exchange.*

Definition D.1 (Credential exchange support.)**.** If U_0 is honest, γ_0 settled in `paid` only if U_0 get a pre-defined credential s. Also, similarly, U_{n+1} give s to U_0 only if γ_n settles in `paid`.

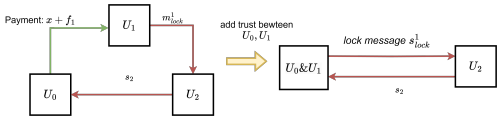


Figure 20: Fair exchange with credential exchange support.

Proof. Credential exchange support (Def. D.1) introduces fairness between U_0 and U_{n+1}. For $|U| = 3$, similar to the general case exchange(Fig. 6)), such MHP protocol can be reduced to a 3-party fair exchange ring, where U_0 gives a revocable payment to U_1, U_1 gives a non-revocable secret(channel lock message of γ_1) to U_2, and U_2 give the secret(credential) back to U_0. Same as the proof of Lemma 1, this is impossible with $\mathcal{T}_{pub}$. □

E *Zeus* Preliminaries

In the GUC framework, a protocol Π is executed between a set of parties $\mathcal{P}$ with an adversary $\mathcal{A}$, who receives as input a security parameter $\lambda \in \mathbb{N}$ along with an auxiliary input $z \in \{0,1\}^*$. In our work, we follow a static corrupt model, where $\mathcal{A}$ can corrupt any party $P_i \in \mathcal{P}$ at the beginning of the protocol execution. Corrupting a party P_i means that $\mathcal{A}$ takes control over P_i and learns all its internal state(e.g., credential, private key). The parties and the adversary $\mathcal{A}$ take their input from the environment $\mathcal{E}$, while $\mathcal{E}$ observes the messages that are output by the parties of the protocol.

E.1 Preliminary ideal functionalities

In our model, we assume a synchronous communication, formalized by a global clock idea functionality [27], in which all parties have access to the current round number. We further assume that there are two types of private and secure communication channel amoung parties: a synchronous channel $\mathcal{F}_{sec}$ [27] and an asynchronous channel $\mathcal{F}_{Onion}$ [12]. In $\mathcal{F}_{sec}$, a party P can privately send a message to another party Q with guranteed delivery after precisely one round. In $\mathcal{F}_{Onion}$, a party P can send a private message to another party Q without guaranteed delivery. $\mathcal{F}_{Onion}$ is used to model the onion messages issued from payment issuer(payer) to all the intermediaries along the path. We denote $m \overset{\tau}{\hookrightarrow} Q$ to represent that some party sends a message m to Q at round τ. $Q \overset{\tau+1}{\longleftarrow} m$ represents that Q receives the message m at round $\tau + 1$. We use $\hookrightarrow_{\mathcal{F}_{Onion}}$ to denote the message sent through $\mathcal{F}_{Onion}$.

We assume there is a perfect hiding and binding commitment scheme $\mathcal{F}_{\text{Commit}}$ with interface (Com, Open), where $h := \text{Com}(s)$ return the commitment to a secret s, and the $\text{Open}(s, h)$ is the functionality that returns 1 iff the secret s matches the commitment h.

A global digital signature ideal functionality $\mathcal{F}_{Sig}$ [13] has a tuple of interfaces (KeyGen, Sign, Vrfy) satisfying EUF-CMA secure.

E.2 Ideal functionalities of ledger and a PCN

Ledger. We use the global ideal functionality $\mathcal{G}_{\text{L}}$ [7] to model a UTXO based blockchain. $\mathcal{G}_{Ledger}$ communicates with a fixed set of parties $\mathcal{P}$. The environment $\mathcal{E}$ first initializes $\mathcal{G}_{Ledger}$ by setting up a key pair $(\text{sk}_P, \text{pk}_P)$ for every party $P \in \mathcal{P}$ and registers it to the ledger by sending $(\texttt{sid}, \texttt{REGISTER}, \text{pk}_P)$ to $\mathcal{G}_{Ledger}$. Any party $P \in \mathcal{P}$ can always post a transaction $\overline{tx}$ on $\mathcal{L}$ via $(\texttt{sid}, \texttt{POST}, \overline{tx})$. If a transaction is valid, it will be appear on $\mathcal{G}_{\text{L}}$ after at most Δ round, the exact number is chosen by the adversary. Recall that a transaction is valid, if all its inputs exist and are unspent, there is a correct witness for each input and a unique id.

$\mathcal{G}_{\text{L}}$

Parameters:
- Δ: the maximum time for a transaction to be included in $\mathcal{G}_{\text{L}}$.
- $\mathcal{F}_{Sig}$: a digital signature ideal functionality.

Local vairables:
- PKI: a public key infrastructure mapping public keys to users.
- TXs: a set of valid transactions.

API
- $(\texttt{sid}, \texttt{REGISTER}, \text{pk}_P) \overset{\tau}{\hookleftarrow} P$: adds an entry $(\text{PKI}[\text{pk}_P] := P)$ to PKI table.
- $(\texttt{sid}, \texttt{POST}, tx) \overset{\tau}{\hookleftarrow} P$: in round $[\tau, \tau + \Delta]$, adds tx to TXs.

Figure 21: Ideal functionality of UTXO Ledger.

PCN. In this work, we model a payment channel network (PCN) by global ideal functionality $\mathcal{F}_{Channels}$ (Fig. 22) based on the standard payment channel ideal functionality widely adopted in $\mathcal{F}_{Channels}$ [6, 7, 9, 10]. Compare with the standard payment channel ideal functionality, We omit the create interface, and assume all channels required in a multi-hop payment is pre-established. For each channel $\bar{\gamma}$, $\mathcal{F}_{Channels}$ provides standard `UPDATE` interfaces, where if both the channel owners $\bar{\gamma}.lu/ru$ agree, can update the channel state to a new state represented by the output of a update transaction tx_{state}. The `CLOSE` interface allows any channel owner to close the channel peacefully or forcefully. `UPDATESPLICING` is almost the same as the `UPDATE` interface, but its new state contains both *splicing outputs* and *funding outputs*. When the `SPLICINGCHANNEL` interface is triggered, $\mathcal{G}_{\text{L}}$ will finalize the latest state of $\bar{\gamma}$ by broadcasting a

$\mathcal{F}_{Channels}$

Local vairables:

- t_{close}: the upper bound for the time to close the channel.
- t_{update}: the upper bound for the time to update the channel.

API

- $(\mathtt{sid}, \mathtt{UPDATE}, \mathrm{id}, tx_{state}) \overset{\tau}{\hookleftarrow} P$: Let $\bar{\gamma}$ be the channel where $\bar{\gamma}.\mathrm{id} = \mathrm{id}$. When invoked by $P \in \bar{\gamma}.luor\bar{\gamma}.ru$ and both parties agree, the channel $\bar{\gamma}$ is updated to the new state represented by the outputs of tx_{state}. $(\mathtt{sid}, \mathtt{UPDATED}, \mathrm{id}, tx_{state}) \xhookrightarrow{[\tau,\tau+t_{update}]} \bar{\gamma}.\mathtt{users}$ is output.
- $(\mathtt{sid}, \mathtt{CLOSE}, \mathrm{id}) \overset{\tau}{\hookleftarrow} P$: close the channel $\bar{\gamma}$, where $\bar{\gamma}.\mathrm{id} = \mathrm{id}$, either peacefully or forcefully. In round $[\tau, \tau + t_{close}]$, a transaction $\mathrm{tx}_{\mathrm{close}}$ with the current state $\bar{\gamma}.\mathtt{state}$ as output ($\mathrm{tx}_{\mathrm{close}}.\mathrm{Outs} := \bar{\gamma}.\mathtt{state}$) appears on $\mathcal{G}_{\mathrm{L}}$. Then $\mathcal{F}_{Channels}$ broadcast $(\mathtt{sid}, \mathtt{CLOSED}, \mathrm{id}, \mathrm{tx}_{\mathrm{close}})$.
- $(\mathtt{sid}, \mathtt{UPDATESPLICING}, \mathrm{id}, tx_{splicing}) \overset{\tau}{\hookleftarrow} P$: Let $\bar{\gamma}$ be the channel where $\bar{\gamma}.\mathrm{id} = \mathrm{id}$ from either channel owner. When invoked by $P = \bar{\gamma}.lu/ru$ and both parties agree, $\bar{\gamma}$ is updated to the new state represented by the outputs of $tx_{splicing}$. $tx_{splicing}$ contains *splicing outputs* $(\hat{\theta}_0, \ldots)$ and *funding outputs* $(\bar{\theta}_0, \ldots)$. $\bar{\gamma}.\mathtt{state} := \{\hat{\theta}_0, \ldots, \bar{\theta}_0, \ldots\}$, $\mathrm{UPDATED\text{-}SPLICING}, \mathrm{id}, tx_{splicing}) \xhookrightarrow{[\tau,\tau+t_{update}]} \bar{\gamma}.lu/ru$ is output.
- $(\mathtt{sid}, \mathtt{SPLICINGCHANNEL}, \mathrm{id}) \overset{\tau}{\hookleftarrow} P$: For $\bar{\gamma}.\mathrm{id} = \mathrm{id}$ and $P = \bar{\gamma}.lu/ru$. In round $\tau' \in [\tau, \tau + t_{close}]$, a transaction $tx_{spliced}$ containing all outputs within $\bar{\gamma}.\mathtt{state}$ appears on $\mathcal{G}_{\mathrm{L}}$. Meanwhile, if the latest state contains *funding outputs*, $\mathcal{G}_{\mathrm{L}}$ will update the channel state: $\bar{\gamma}.\mathtt{state} := \{\bar{\theta}_0, \ldots\}$. $(\mathtt{sid}, \mathtt{SPLICED}, \mathrm{id}, tx_{spliced}) \xhookrightarrow{\tau'} \bar{\gamma}.lu/ru$ is output.

Figure 22: Ideal functionality of payment channel.

transaction containing all outputs of the current state. But compare with the the standard CLOSE interface, the SPLICINGCHANNEL interface will not close te channel, but replace the latest state with the *funding outputs* of the transaction.

E.3 UC definition.

Here, we define the prelimilary ideal functionality set as $\mathcal{F}_{\mathtt{prelim}}$, containing $\{\mathcal{F}_{clk}, \mathcal{F}_{sec}, \mathcal{F}_{Onion}, \mathcal{F}_{Sig}, \mathcal{F}_{\mathrm{Commit}}, \mathcal{G}_{\mathrm{L}}, \mathcal{F}_{Channels}\}$. A hybrid protocol Π with corresponding ideal functionality $\mathcal{F}$ interacts with $\mathcal{F}_{\mathtt{prelim}}$. An environment $\mathcal{E}$ that interacts with Π and an adversary $\mathcal{A}$ will on input a security parameter λ and an auxiliary input z output $\mathrm{EXEC}^{\mathcal{F}_{\mathtt{prelim}}}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, z)$. In the ideal the dummy users simply forward their input to $\mathcal{F}$. It has access to the same functionalities $\mathcal{F}_{\mathrm{prelim}}$. The output of F on input λ and z when interacting with $\mathcal{E}$ and a simulator $\mathcal{S}$ is denoted as $\mathrm{EXEC}^{\mathcal{F}_{\mathtt{prelim}}}_{\mathcal{F},\mathcal{S},\mathcal{E}}(\lambda, z)$.

Definition E.1. A protocol Π *GUC-realizes* an ideal functionality $\mathcal{F}$, w.r.t. $\mathcal{F}_{\mathtt{prelim}}$ if for every adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that for all environments $\mathcal{E}$:

$$\mathrm{EXEC}^{\mathcal{F}_{\mathtt{prelim}}}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, z) \approx_c \mathrm{EXEC}^{\mathcal{F}_{\mathtt{prelim}}}_{\mathcal{F},\mathcal{S},\mathcal{E}}(\lambda, z)$$

where $\approx_c$ denotes computational indistinguishability.

F Formal Protocol Description

Zeus Protocol: Π_{Zeus}

Parameters owned by all parties:

- pidSet: the set of all payment ids. Initially empty.

`Setup phase`

$(\mathtt{sid}, \mathtt{pid}, \mathtt{setup}, x, \bar{T}, s) \overset{0}{\hookleftarrow} \mathcal{E}, U_{n+1}$:

0) If pid $\in$ pidSet, ignore. Otherwise, add pid to pidSet. Save the setup parameters: $x, \bar{T}, s$. Set $h := \mathrm{Commit}(s)$.

$(\mathtt{sid}, \mathtt{pid}, \mathtt{setup}, \mathcal{M}, h, T) \overset{0}{\hookleftarrow} \mathcal{E}, U_0$:

1) If pid $\in$ pidSet, ignore. Otherwise, add pid to pidSet.
2) **Check payee's stake.** $\mathrm{Out}_{\mathtt{Stake}} := \mathrm{checkPayeeStake}(U_{n+1}, T)$ If $\mathrm{Out}_{\mathtt{Stake}} = \perp$, then abort.
3) **Construct setup transactions.**

$(tx_{\mathrm{splicing}}, tx_{\mathrm{redeem}}, tx_{\mathrm{slash}}) := \mathrm{genSplicingTx}(\gamma_0.\mathrm{id}, \mathrm{Out}_{\mathtt{Stake}}, \mathcal{M}, h)$

4) **Request for accountability.** Send:

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{request\text{-}acc}, tx_{\mathrm{splicing}}, tx_{\mathrm{redeem}}, tx_{\mathrm{slash}}) \overset{0}{\hookrightarrow} U_{n+1}$$

$(\mathtt{sid}, \mathtt{pid}, \mathtt{request\text{-}acc}, tx_{\mathrm{splicing}}, tx_{\mathrm{redeem}}, tx_{\mathrm{slash}}) \overset{1}{\hookleftarrow} U_0, U_{n+1}$:

5) **Check the validity.** If $\perp = \mathrm{checkSlashTx}(tx_{\mathrm{splicing}}, tx_{\mathrm{redeem}}, tx_{\mathrm{slash}}, h)$, then abort.
6) **Enable the redeem launcher.**
   - $\sigma_{\mathrm{redeem}} := \mathrm{approveTx}(tx_{\mathrm{redeem}}, 0)$
   - $tx_{\mathrm{redeem}}.\mathrm{ScriptSigs}[0] := (\sigma_{\mathrm{redeem}})$

$(\mathtt{sid}, \mathtt{pid}, \mathtt{init}, \gamma_0.id, T, tx_{\mathrm{splicing}}.\mathrm{Outs}[0], tx_{\mathrm{redeem}}) \overset{1}{\hookrightarrow} \mathcal{F}_{\text{redeem-launcher}}$

7) **Approve accountability.**
   - $\sigma_{\mathrm{slash}} := \mathrm{approveTx}(tx_{\mathrm{slash}}, 1)$
   - $(\mathtt{sid}, \mathtt{pid}, \mathtt{approved\text{-}acc}, \sigma_{\mathrm{slash}}) \overset{1}{\hookrightarrow} U_0$

$(\mathtt{sid}, \mathtt{pid}, \mathtt{approved\text{-}acc}, \sigma_{\mathrm{slash}}) \overset{2}{\hookleftarrow} U_{n+1}, U_0$:

8) **Check signature.** If σ_{slash} is not a valid signature of U_{n+1} over tx_{slash}, abort.
9) **Generate the setup messages.**

$$m_{\mathrm{setup}} := \mathrm{genOnion}(\mathcal{M}, tx_{\mathrm{splicing}}, tx_{\mathrm{redeem}})$$

$$\forall i \in [1, n+1], (\mathtt{sid}, \mathtt{pid}, \mathtt{setup}, m_{\mathrm{setup}}[i]) \overset{2}{\hookrightarrow}_{\mathcal{F}_{Onion}} U_i$$

10) **Splice** γ_0. $(\mathtt{sid}, \mathtt{pid}, \mathtt{init}, \gamma_0.id, tx_{\mathrm{splicing}}) \overset{3}{\hookrightarrow} \mathcal{F}^0_{\text{2pSetupSplicing}}$

$(\mathtt{sid}, \mathtt{pid}, \mathtt{setup}, m_{\mathrm{setup}}[i]) \overset{3}{\hookleftarrow} U_0$:

11) U_1:
   - Extract $\mathtt{Dep}_1$, Prcpl , T, *splicing trigger* $(tr^1_{\mathrm{splicing}})$.
   - Check if the trigger tr^1_{splicing} has the correct lock condition, and the payment lifetime $\tau \ll T$.
   - Setup the splicing state of γ_0:

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{setup}, tr^1_{\mathrm{splicing}}) \overset{3}{\hookrightarrow} \mathcal{F}_{\text{2pSetupSplicing}}$$

11) $U_{i\in[2,n]}$:
   - Extract $\mathtt{Dep}_i$, $\mathtt{Prm}_{i-1}$, Prcpl , T, *splicing triggers* $(tr^i_{\mathrm{splicing}}, tr^{i-1}_{\mathrm{splicing}})$, and *redeem triggers* $(tr^i_{redeem}, tr^{i-1}_{redeem})$ from $m_{\mathrm{setup}}[i]$
   - Check if the deposit amount is correct: $\mathtt{Dep}_i = c_{\mathrm{payee}} + r \cdot c_{relay}$, where $r \in \mathbb{N}^+$; payment lifetime is long enough $\tau \ll T$, and the triggers has the correct lock condition.
   - Check if splicing triggers share the same splicing transaction, and the redeem triggers share the same redeem transaction.
   - Setup the next state of γ_{i-1}:

$(\mathtt{sid}, \mathtt{pid}, tr^{i-1}_{redeem}\mathtt{setup}, \mathtt{Dep}_i + c_{relay}, T, tr^i_{\mathrm{splicing}}.tx) \overset{3}{\hookrightarrow} \mathcal{F}^{i-1}_{\text{2pSetupDep}}$

11) U_{n+1}:
   - Extract $\mathtt{Prm}_n$, Prcpl , T, *redeem trigger* (tr^n_{redeem}).

- Check if $\texttt{Prcpl} - \texttt{Prm}_n \geq x$; timelock is consistent ($T = \bar{T}$); and the redeem trigger has the correct lock condition.
- Setup the next state of γ_n:

$$(\texttt{sid}, \texttt{pid}, \texttt{setup}, c_{\text{payee}}, T, tr^n_{\text{splicing}}.tx) \overset{3}{\hookrightarrow} \mathcal{F}^n_{\text{2pSetupDep}}$$

Then $U_1, \ldots, U_{n+1}$ enter the lock-deposit phase.

U_0 upon $(\texttt{sid}, \texttt{pid}, \texttt{slicing-Ok})$, then:

12) **Enable the splicing and slash launcher.**
- $(\texttt{sid}, \texttt{pid}, \texttt{init}, T, \gamma_0.\text{id}) \overset{\tau}{\hookrightarrow} \mathcal{F}^0_{\text{splicing-launcher}}$
- $(\texttt{sid}, \texttt{pid}, \texttt{init}, T, tx_{\text{splicing}}.\text{Outs}[0], tx_{slash}) \overset{\tau}{\hookrightarrow} \mathcal{F}_{\text{slash-launcher}}$
- Enter the deposit-refund phase.

`Lock-deposit phase`

$(\texttt{sid}, \texttt{pid}, \texttt{slicing-Ok}) \overset{\tau_1}{\hookleftarrow} \mathcal{F}_{\text{2pSetupSplicing}}, U_1$:

1) **Lock payer's deposit within γ_1.** Invoke:

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_1.\text{id}, \texttt{Dep}_1, tr^1_{\text{splicing}}, T) \overset{\tau_1}{\hookrightarrow} \mathcal{F}^1_{\text{2pSetupDep}}$$

2) **Enable splicing launcher and setup the next state of γ_1** If $\mathcal{F}^1_{\text{2pSetupDep}}$ returns $(\texttt{lock-Dep-Ok})$, then:
- $(\texttt{sid}, \texttt{pid}, \texttt{init}, T, \gamma_0.\text{id}) \overset{\tau'_1}{\hookrightarrow} \mathcal{F}^1_{\text{splicing-launcher}}$
- $(\texttt{sid}, \texttt{pid}, tr^1_{\text{splicing}}, \texttt{Prm}_1, T, tr^1_{\text{redeem}}) \overset{\tau'_1}{\hookrightarrow} \mathcal{F}^1_{\text{2pSetupPrm}}$

and enter the lock-premium phase.

$(\texttt{sid}, \texttt{pid}, \texttt{lock-Dep-Ok}) \overset{\tau^i_1}{\hookleftarrow} \mathcal{F}^{i-1}_{\text{2pSetupDep}}, U_{i\in[2,n]}$:

4) **Forward the deposit lock.** Invoke:

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_i.\text{id}, \texttt{Dep}_i, tr^i_{\text{splicing}}, T) \overset{\tau^i_1}{\hookrightarrow} \mathcal{F}^i_{\text{2pSetupDep}}$$

5) **Setup the Premium-lock state of γ_i.** IF receive $(\texttt{lock-Dep-Ok})$ from $\mathcal{F}^i_{\text{2pSetupDep}}$, then:

$$(\texttt{sid}, \texttt{pid}, \texttt{setup}, tr^i_{\text{splicing}}, \texttt{Prm}_i, T, tr^i_{\text{redeem}}) \overset{\tau^{i'}_1}{\hookrightarrow} \mathcal{F}^i_{\text{2pSetupPrm}}$$

and enter the lock-premium phase.

$(\texttt{sid}, \texttt{pid}, \texttt{lock-Dep-Ok}) \overset{\tau_2}{\hookleftarrow} \mathcal{F}^n_{\text{2pSetupDep}}, U_{n+1}$:

6) **Enter the lock-premium phase.**

`Lock-premium phase`

U_{n+1}:

1) **Lock the premium.** Invoke:

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_n.id, \texttt{Prm}_n, tr^n_{redeem}, T, tx^n_{\texttt{revoke-Dep}}) \overset{\tau_2}{\hookrightarrow} \mathcal{F}^n_{\text{2pSetupPrm}}$$

2) **Enable redeem handler and prepare for principal-lock.** If $(\texttt{lock-Prm-Ok})$ returned at τ'_2, then:

$$(\texttt{sid}, \texttt{pid}, \texttt{setup}, \texttt{Prcpl}, tr^n_{\text{splicing}}, tr^n_{\text{redeem}}) \overset{\tau'_2}{\hookrightarrow} \mathcal{F}^n_{\text{2pSetupPrcpl}}$$

then enter the next phase.

$(\texttt{sid}, \texttt{pid}, \texttt{lock-Prm-Ok}) \overset{\tau^i_2}{\hookleftarrow} \mathcal{F}^i_{\text{2pSetupPrm}}, U_{i\in[2,n]}$:

3) **Lock premium over γ_{i-1}:**

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_{i-1}.id, \texttt{Prm}_{i-1}, tr^{i-1}_{redeem}, T, tr^{i-1}_{\text{redeem}}) \overset{\tau^i_2}{\hookrightarrow} \mathcal{F}^{i-1}_{\text{2pSetupPrm}}$$

4) **Setup the Principal-Lock state.** If $(\texttt{sid}, \texttt{pid}, \texttt{lock-Prm-Ok})$ received at $\tau^{i'}_2$:

$$(\texttt{sid}, \texttt{pid}, \texttt{setup}, \texttt{Prcpl}, tr^{i-1}_{\text{splicing}}, tr^{i-1}_{\text{redeem}}) \overset{\tau^{i'}_2}{\hookrightarrow} \mathcal{F}^{i-1}_{\text{2pSetupPrcpl}}$$

and enter the lock-principal phase. Otherwise, terminate.

$(\texttt{sid}, \texttt{pid}, \texttt{lock-Prm-Ok}) \overset{\tau_3}{\hookleftarrow} \mathcal{F}^1_{\text{2pSetupPrm}}, U_1$:

5) **Enter the lock-principal phase.**

`Lock-principal phase`

U_1:

1) **Lock principal within γ_1.** Invoke:

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_1.\text{id}, \texttt{Prm},) \overset{\tau_3}{\hookrightarrow} \mathcal{F}^1_{\text{2pSetupPrcpl}}$$

2) **Setup the next phase.** If receive $(\texttt{lock-Prcpl-Ok})$ return at τ'_3:

$$(\texttt{sid}, \texttt{pid}, \texttt{setup}, tr^1_{\text{splicing}}, tr^1_{\text{redeem}}, \texttt{Dep}_1) \overset{\tau'_3}{\hookrightarrow} \mathcal{F}^1_{\text{2pRfdDep}}$$

$(\texttt{sid}, \texttt{pid}, \texttt{lock-Prcpl-Ok}) \overset{\tau^i_3}{\hookleftarrow} \mathcal{F}^{i-1}_{\text{2pSetupPrcpl}}, U_{i\in[2,n]}$:

3) **Forward the Principal lock.** Invoke:

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_i.\text{id}, \texttt{Prm}, tr^i_{\text{splicing}}, tr^i_{\text{redeem}}) \overset{\tau^i_3}{\hookrightarrow} \mathcal{F}^i_{\text{2pSetupPrcpl}}$$

5) **Setup the deposit-refund phase.** If it returns $(\texttt{lock-Prcpl-Ok})$ at $\tau^{i'}_3$, Then:

$$(\texttt{sid}, \texttt{pid}, \texttt{setup}, tr^i_{\text{splicing}}, tr^i_{\text{redeem}}, \texttt{Dep}_i) \overset{\tau^{i'}_3}{\hookrightarrow} \mathcal{F}^i_{\text{2pRfdDep}}$$

and enter the next phase.

$(\texttt{sid}, \texttt{pid}, \texttt{lock-Prcpl-Ok}) \overset{\tau_4}{\hookleftarrow} \mathcal{F}^n_{\text{2pSetupPrcpl}}, U_{n+1}$:

6) **Release the secret.** $(\texttt{sid}, \texttt{pid}, \texttt{release-secret}, s) \overset{\tau_4}{\hookrightarrow} U_0$, and enter the deposit-refund phase.

`Deposit-Refund phase`

U_{n+1}:

1) **Refund the payer side deposit.** Invoke:

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_n.id, \texttt{Dep}_n, tr^n_{\text{splicing}}, tr^n_{\text{redeem}}) \overset{\tau_4}{\hookrightarrow} \mathcal{F}^n_{\text{2pRfdDep}}$$

2) **Setup the settle phase.** If $(\texttt{refund-Dep-Ok})$ received from $\mathcal{F}^n_{\text{2pRfdDep}}$:

$$(\texttt{setup}, lb - \texttt{Prcpl} + \texttt{Prm}_n, rb + \texttt{Prcpl} - \texttt{Prm}_n) \overset{\tau'_4}{\hookrightarrow} \mathcal{F}^n_{\text{2pSettle}}$$

then enter the next phase. Otherwise, terminate.

$(\texttt{sid}, \texttt{pid}, \texttt{refund-Dep-Ok}) \overset{\tau^i_4}{\hookleftarrow} \mathcal{F}^i_{\text{2pRfdDep}}, U_{i\in[2,n]}$:

3) **Forward the revoke over γ_{i-1}:**

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_{i-1}.id, \texttt{Dep}_{i-1}, tr^{i-1}_{\text{splicing}}, tr^{i-1}_{\text{redeem}}) \overset{\tau^{i'}_4}{\hookrightarrow} \mathcal{F}^{i-1}_{\text{2pRfdDep}}$$

4) **Setup the settle phase.** If $(\texttt{refund-Dep-Ok})$ received from $\mathcal{F}^{i-1}_{\text{2pRfdDep}}$:

$$(\texttt{sid}, \texttt{pid}, \texttt{setup}, lb - \texttt{Prcpl} + \texttt{Prm}_{i-1}, rb + \texttt{Prcpl} - \texttt{Prm}_{i-1}) \overset{\tau^{i''}_4}{\hookrightarrow} \mathcal{F}^{i-1}_{\text{2pSettle}}$$

then enter the next phase. Otherwise, terminate.

$U_1 \overset{\tau_5}{\hookleftarrow} (\texttt{refund-Dep-Ok})$ from $\mathcal{F}^1_{\text{2pRfdDep}}$:

5) **Setup the settle phase.**

$$(\texttt{sid}, \texttt{pid}, \texttt{setup}, lb - \texttt{Prcpl} + \texttt{Prm}_1 - f_1, rb + \texttt{Prcpl} - \texttt{Prm}_1 + f_1) \overset{\tau'_5}{\hookrightarrow} \mathcal{F}^0_{\text{2pSettle}}$$

6) **Notify payer.** $(\texttt{sid}, \texttt{pid}, \texttt{revoke-Dep-Ok}) \overset{\tau''_5}{\hookrightarrow} U_0$.

$U_0 \overset{\tau_6}{\hookleftarrow} (\texttt{sid}, \texttt{pid}, \texttt{revoke-Dep-Ok})$ from U_1:

7) **Check the secret.** Check if received $(\mathtt{sid}, \mathtt{pid}, \mathtt{release\text{-}secret}, s)$ from U_{n+1}, and $\mathrm{Open}(s, h) = 1$, then enter the settle phase. Otherwise, abort.

`Settle phase`

U_0:

1) **Settle the payment.**

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{init}, \gamma_0.id, lb - x - \sum_{j=1}^{n} f_j, rb + x + \sum_{j=1}^{n} f_j) \overset{\tau_6}{\hookrightarrow} \mathcal{F}^0_{\text{2pSettle}}$$

And once $\mathtt{settle\text{-}Ok} \overset{\tau_6'}{\hookrightarrow} \mathcal{F}^0_{\text{2pSettle}}$, then return $(\mathtt{sid}, \mathtt{pid}, \mathtt{settled}, s) \overset{\tau_6'}{\hookrightarrow} \mathcal{E}$, close the splicing launcher

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{close}) \overset{\tau_6'}{\hookrightarrow} \mathcal{F}^0_{\text{splicing-launcher}}$$

ends the main protocol.

$(\mathtt{sid}, \mathtt{pid}, \mathtt{settle\text{-}Ok}) \overset{\tau_6^i}{\hookleftarrow} \mathcal{F}^{i-1}_{\text{2pSettle}}, U_{i\in[1,n]}$:

2) **Continue settlement chain:**

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{init}, \gamma_i.id, lb - \mathtt{Prcpl} + \mathtt{Prm}_i, rb + \mathtt{Prcpl} - \mathtt{Prm}_i) \overset{\tau_6^{i'}}{\hookrightarrow} \mathcal{F}^i_{\text{2pSettle}}$$

3) **Close the splicing launcher.** If $i = 1$,

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{close}) \overset{\tau_6^{i'}}{\hookrightarrow} \mathcal{F}^1_{\text{splicing-launcher}}$$

and ends the protocol.

4) $(\mathtt{sid}, \mathtt{pid}, \mathtt{settled}) \overset{\tau_7}{\hookrightarrow} \mathcal{E}$ and end.

F.1 Formal Descriptions of sub-procedures

2pSetupSplicing Protocol: $\Pi^i_{\text{2pSetupSplicing}}$

Parameters owned by both U_0 and U_1:

- `pidSet`: the set of all payment ids. Initially empty.

$(\mathtt{sid}, \mathtt{pid}, \mathtt{setup}, tr^1_{\text{splicing}}) \overset{\leq\tau}{\hookleftarrow} \mathcal{E}, U_1$:

0) If $\mathtt{pid} \in \mathtt{pidSet}$, ignore. Otherwise, add $\mathtt{pid}$ to $\mathtt{pidSet}$, save tr^1_{splicing}.

$(\mathtt{sid}, \mathtt{pid}, \mathtt{init}, \gamma_0.id, tx_{\text{splicing}}) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_0$:

1) **Request splicing.** If $\mathtt{pid} \in \mathtt{pidSet}$, then abort. Otherwise, add $\mathtt{pid}$ to $\mathtt{pidSet}$ then sign the new state $\sigma^l_{\text{splicing}} := \mathrm{approveTx}(tx_{\text{splicing}}, 0)$

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{request\text{-}splicing}, \gamma_0.id, tx_{\text{splicing}}, \sigma^l_{\text{splicing}}) \overset{\tau}{\hookrightarrow} U_1$$

$(\mathtt{sid}, \mathtt{pid}, \mathtt{request\text{-}splicing}, \gamma_0.id, tx_{\text{splicing}}, \sigma^l_{\text{splicing}}) \overset{\tau+1}{\hookleftarrow} U_0, U_1$:

2) **Check splicing transaction.** If

$$\top = \mathrm{checkSplicing}(\gamma_0.id, tx_{\text{splicing}}, tr^1_{\text{splicing}}, \sigma^l_{\text{splicing}})$$

Otherwise, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{splicing\text{-}fail}) \overset{\tau+1}{\hookrightarrow} \mathcal{E}$.

3) **Approve splicing.**
   - $\sigma^r_{\text{splicing}} := \mathrm{approveTx}(tx_{\text{splicing}}, 0)$
   - $tx_{\text{splicing}} := \mathrm{genUpdateTx}(tx_{\text{splicing}}, \sigma^l_{\text{splicing}}, \sigma^r_{\text{splicing}})$

4) **Splicing γ_0.** Send

$$(\mathtt{sid}_C, \mathtt{UPDATE\text{-}SPLICING}, \gamma_0.id, tx_{\text{splicing}}) \overset{\tau+1}{\hookrightarrow} \mathcal{F}_{Channels}$$

5) If $(\mathtt{sid}_C, \mathtt{UPDATED\text{-}SPLICING}) \overset{\tau'}{\hookleftarrow} \mathcal{F}_{Channels}$, then send $(\mathtt{sid}, \mathtt{pid}, \mathtt{splicing\text{-}Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$.

By round $\tau + t_{update} + 1$, U_0:

6) If $(\mathtt{sid}_C, \mathtt{UPDATED\text{-}SPLICING}) \hookleftarrow \mathcal{F}_{Channels}$ was received, then send $(\mathtt{sid}, \mathtt{pid}, \mathtt{splicing\text{-}Ok}) \hookrightarrow \mathcal{E}$. Otherwise, close the channel γ_0 by sending $(\mathtt{sid}_C, \mathtt{CLOSE}, \gamma_0.\mathrm{id}) \hookrightarrow \mathcal{F}_{Channels}$ and abort.

2pSetupDeposit Protocol: $\Pi^i_{\text{2pSetupDep}}$

Parameters owned by both U_i and U_{i+1}:

- `pidSet`: the set of all payment ids. Initially empty.

$(\mathtt{sid}, \mathtt{pid}, \mathtt{setup}, \mathtt{Dep}_i, T, tr^i_{\text{splicing}}) \overset{\leq\tau}{\hookleftarrow} \mathcal{E}, U_{i+1}$:

0) If $\mathtt{pid} \in \mathtt{pidSet}$, then abort. Otherwise, add $\mathtt{pid}$ to $\mathtt{pidSet}$, save $\mathtt{Dep}_i, T, tr^i_{\text{splicing}}$.

$(\mathtt{sid}, \mathtt{pid}, \mathtt{init}, \gamma_i.\mathrm{id}, \mathtt{Dep}_i, tr^i_{\text{splicing}}, T) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_i$:

1) **Generate the lock-deposit transactions.** If $\mathtt{pid} \in \mathtt{pidSet}$, then abort. Otherwise, add $\mathtt{pid}$ to $\mathtt{pidSet}$ and:

$$(tx^i_{\text{lock-Dep}}, tx^i_{\text{revoke-Dep}}) := \mathrm{genDepTx}(\gamma_i.\mathrm{id}, \mathtt{Dep}_i, tr^i_{\text{splicing}}, T)$$

2) **Approve the deposit revoke.**
   - $\sigma^{i,l}_{\text{revoke-Dep}} := \mathrm{approveTx}(tx^i_{\text{revoke-Dep}}, 0)$
   - $tx^i_{\text{revoke-Dep}}.\mathrm{ScriptSigs}[0] := (\sigma^{i,l}_{\text{revoke-Dep}})$

3) **Request for lock deposit.**

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{request\text{-}lock\text{-}Dep}, \gamma_i.\mathrm{id}, tx^i_{\text{lock-Dep}}, tx^i_{\text{revoke-Dep}}) \overset{\tau}{\hookrightarrow} U_{i+1}$$

$(\mathtt{sid}, \mathtt{pid}, \mathtt{request\text{-}lock\text{-}Dep}, \cdots) \overset{\tau+1}{\hookleftarrow} U_i, U_{i+1}$:

4) **Check the proposed state.** Check if $status = \mathtt{enabled}$ and

$$\top = \mathrm{checkDep}(\gamma_i.\mathrm{id}, tx^i_{\text{lock-Dep}}, tx^i_{\text{revoke-Dep}}, \mathtt{Dep}_i, T, tr^i_{\text{splicing}})$$

Otherwise, abort.

5) **Approve deposit and its revoke logic.**
   - $\sigma^{i,r}_{\text{revoke-Dep}} := \mathrm{approveTx}(tx^i_{\text{revoke-Dep}}, 1)$
   - $\sigma^{i,r}_{\text{lock-Dep}} := \mathrm{approveTx}(tx^i_{\text{lock-Dep}}, 0)$
   - Add $(\sigma^{i,r}_{\text{revoke-Dep}})$ in $tx^i_{\text{revoke-Dep}}.\mathrm{ScriptSigs}[0]$

6) **Send approval to U_i:**

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{approved\text{-}lock\text{-}Dep}, tx^i_{\text{revoke-Dep}}, \sigma^{i,r}_{\text{lock-Dep}}) \overset{\tau+1}{\hookrightarrow} U_i$$

$(\mathtt{sid}, \mathtt{pid}, \mathtt{approved\text{-}lock\text{-}Dep}, tx^i_{\text{revoke-Dep}}, \sigma^{i,r}_{\text{lock-Dep}}) \overset{\tau+2}{\hookleftarrow} U_{i+1}, U_i$:

7) **Check the signatures.** Check if $tx^i_{\text{revoke-Dep}}$ contains U_i's and U_{i+1}'s signatures and $\sigma^{i,r}_{\text{lock-Dep}}$ is valid.

8) **Approve the new state.**
   - $\sigma^{i,l}_{\text{lock-Dep}} := \mathrm{approveTx}(tx^i_{\text{lock-Dep}}, 0)$
   - $tx^i_{\text{lock-Dep}} := \mathrm{genUpdateTx}(tx^i_{\text{lock-Dep}}, \sigma^{i,l}_{\text{lock-Dep}}, \sigma^{i,r}_{\text{lock-Dep}})$

9) **Update the channel state.**

$$(\mathtt{sid}_C, \mathtt{UPDATE}, \gamma_i.id, tx^i_{\text{lock-Dep}}) \overset{\tau+2}{\hookrightarrow} \mathcal{F}_{Channels}$$

10) **Initialize splicing handler.** If $(\mathtt{sid}_C, \mathtt{UPDATED}, \gamma_i.\mathrm{id}) \overset{\tau'}{\hookleftarrow} \mathcal{F}_{Channels}$, then:

$$(\mathtt{sid}^i_{SH}, \mathtt{init}, \gamma_i.id, T, tr^i_{\text{splicing}}, tx^i_{\text{revoke-Dep}}) \overset{\tau'}{\hookrightarrow} \mathcal{F}^i_{\text{splicing-handler}}$$

Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{lock\text{-}Dep\text{-}Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$.

$(\mathtt{sid}_C, \mathtt{UPDATED}, \gamma_i.\mathrm{id}) \overset{\tau'}{\hookleftarrow} \mathcal{F}_{Channels}, U_{i+1}$:

11) Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{lock\text{-}Dep\text{-}Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$.

2pSetupPrm Protocol: $\Pi^i_{\text{2pSetupPrm}}$

Parameters owned by both U_i and U_{i+1}

- pidSet: the set of all the payment ids.

$(\mathtt{sid}, \mathtt{pid}, \mathtt{setup}, tr^i_{\text{splicing}}, \mathtt{Prm}_i, T, tr^i_{\text{redeem}}) \overset{\leq\tau}{\hookleftarrow} \mathcal{E}, U_i$:

0) If pid ∈ pidSet, then abort. Otherwise, add pid to pidSet. Then save $T, \mathtt{Prm}_i, tr^i_{\text{redeem}}, tr^i_{\text{splicing}}$

$(\mathtt{sid}, \mathtt{pid}, \mathtt{init}, \gamma_i.\text{id}, \mathtt{Prm}_i, tr^i_{\text{redeem}}, T, tr^i_{\text{splicing}}) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_{i+1}$:

1) **Construct Premium-Lock txs.** If pid ∈ pidSet, then abort. Otherwise, add pid to pidSet and generate the lock-premium txs:

$$(tx^i_{\text{lock-Prm}},\ tx^i_{\text{revoke-Prm}},\ tx^{i'}_{\text{revoke-Dep}}) := \text{genPrmTx}(\gamma.id, \mathtt{Prm}_i, T, tr^i_{\text{redeem}}, tr^i_{\text{splicing}})$$

2) **approve the revoke transactions.**
   - $\sigma^{i,r}_{\text{revoke-Prm}} := \text{approveTx}(tx_{\text{revoke-Prm}}, 0)$
   - $\sigma^{i',r}_{\text{revoke-Dep}} := \text{approveTx}(tx^{i'}_{\text{revoke-Dep}}, 0)$
   - $tx^{i'}_{\text{revoke-Prm}}.\text{ScriptSigs}[0] := (\sigma^{i,r}_{\text{revoke-Prm}})$
   - $tx^{i'}_{\text{revoke-Dep}}.\text{ScriptSigs}[0] := (\sigma^{i',r}_{\text{revoke-Dep}})$

$(\mathtt{sid}, \mathtt{pid}, \mathtt{request\text{-}Prm}, \gamma_i.id, tx^i_{\text{lock-Prm}},\ tx^i_{\text{revoke-Prm}},\ tx^{i'}_{\text{revoke-Dep}}) \overset{\tau}{\hookrightarrow} U_i$

$(\mathtt{sid}, \mathtt{pid}, \mathtt{request\text{-}Prm}, \ldots) \overset{\tau+1}{\hookleftarrow} U_{i+1}, U_i$:

3) **Check the proposed state.** Check if pid ∈ pidSet and

$$\top = \text{checkPrm}(\gamma_i.\text{id}, T, \mathtt{Prm}_i, tx^i_{\text{lock-Prm}}, tx^i_{\text{revoke-Prm}}, tx^{i'}_{\text{revoke-Dep}}, tx_{\text{redeem}}, tr^i_{\text{splicing}}, tr^i_{\text{redeem}}, T)$$

4) **Approve the revoke transactions.**
   - $\sigma^{i,l}_{\text{revoke-Prm}} := \text{approveTx}(tx^i_{\text{revoke-Prm}}, 0)$
   - $\sigma^{i',l}_{\text{revoke-Dep}} := \text{approveTx}(tx^{i'}_{\text{revoke-Dep}}, 0)$
   - Add $(\sigma^{i,l}_{\text{revoke-Prm}})$ in $tx^i_{\text{revoke-Prm}}.\text{ScriptSigs}[0]$
   - Add $(\sigma^{i',l}_{\text{revoke-Dep}})$ in $tx^{i'}_{\text{revoke-Dep}}.\text{ScriptSigs}[0]$
5) **Approve the new state.**

$$\sigma^{i,l}_{\text{lock-Prm}} := \text{approveTx}(tx^i_{\text{lock-Prm}}, 0)$$

6) **Update the splicing handler.**

$$(\text{sid}^i_{SH}, \mathtt{pid}, \mathtt{UPDATE}, tx^{i'}_{\text{revoke-Dep}}) \overset{\tau+1}{\hookrightarrow} \mathcal{F}^i_{\text{splicing-handler}}$$

7) **Request to finalize the premium lock.**

$(\mathtt{sid}, \mathtt{pid}, \mathtt{approved\text{-}Prm}, \sigma^{i,l}_{\text{lock-Prm}}, tx^i_{\text{revoke-Prm}}, tx^{i'}_{\text{revoke-Dep}}) \overset{\tau+1}{\hookrightarrow} U_{i+1}$

$(\mathtt{sid}, \mathtt{pid}, \mathtt{approved\text{-}Prm}, \cdots) \overset{\tau+2}{\hookleftarrow} U_i, U_{i+1}$:

8) **Check the signatures.** Check if U_0 signs the revoke transactions, and provide valid signature over the premium lock transaction.
9) **Update the channel state.**
   - $\sigma^{i,r}_{\text{lock-Prm}} := \text{approveTx}(tx^i_{\text{lock-Prm}}, 0)$
   - $tx^i_{\text{lock-Prm}} := \text{genUpdateTx}(tx^i_{\text{lock-Prm}}, \sigma^{i,l}_{\text{lock-Prm}}, \sigma^{i,r}_{\text{lock-Prm}})$
   - $(\text{sid}_C, \mathtt{UPDATE}, \gamma_i.id, tx^i_{\text{lock-Prm}}) \overset{\tau+2}{\hookrightarrow} \mathcal{F}_{Channels}$
10) **Initialize redeem handler.**

    If $(\text{sid}_C, \mathtt{UPDATED}, \gamma_i.id, tx^i_{\text{lock-Prm}}.\text{Outs}) \overset{\tau'}{\hookleftarrow} \mathcal{F}_{Channels}$, then:
   - $\sigma := \text{approveTx}(tx^i_{\text{revoke-Prm}}, 1), tx^i_{\text{revoke-Prm}}.\text{ScriptSigs}[1] := (\sigma)$
   - $(\mathtt{init}, \gamma_i.id, tr^i_{redeem}, tx^i_{\text{revoke-Prm}}) \overset{\tau'+2'}{\hookrightarrow} \mathcal{F}^i_{\text{redeem-handler}}$
   - $(\mathtt{sid}, \mathtt{pid}, \mathtt{lock\text{-}Prm\text{-}Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$

$(\text{sid}_C, \mathtt{UPDATED}, \gamma_i.id, tx^i_{\text{lock-Prm}}.\text{Outs}) \overset{\tau'}{\hookleftarrow} \mathcal{F}_{Channels}, U_i$:

11) **Return the final state.**

    $(\mathtt{sid}, \mathtt{pid}, \mathtt{lock\text{-}Prm\text{-}Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$
    By round $\tau + t_{update} + 2$, if channel still not updated, terminate this instance.

2pSetupPrcpl Protocol: $\Pi^i_{\text{2pSetupPrcpl}}$

Parameters owned by both U_i and U_{i+1}:

- pidSet: the set of all payment ids. Initially empty.

$(\mathtt{sid}, \mathtt{pid}, \mathtt{setup}, \mathtt{Prcpl}, tr^i_{\text{splicing}}, tr^i_{\text{redeem}}) \overset{\leq\tau}{\hookleftarrow} \mathcal{E}, U_{i+1}$:

0) If pid ∈ pidSet, then abort. Otherwise, add pid to pidSet, save $tr^i_{\text{splicing}}, tr^i_{\text{redeem}}, \mathtt{Prcpl}$.

$(\mathtt{sid}, \mathtt{pid}, \mathtt{init}, \gamma_i.\text{id}, \mathtt{Prcpl}, tr^i_{\text{splicing}}, tr^i_{\text{redeem}}) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_i$:

1) **Generate the lock-principal transactions.** If pid ∈ pidSet, then abort. Otherwise, add pid to pidSet and:

$$(tx^i_{\text{lock-Prcpl}},\ tx^{i'}_{\text{revoke-Prm}},\ tx^i_{\text{revoke-Prcpl}}) := \text{genPrcplTx}(\gamma_i.id, \mathtt{Prcpl}, tr^i_{\text{splicing}}, tr^i_{\text{redeem}})$$

2) **Approve the revoke transactions.**
   - $\sigma^{i,l}_{\text{revoke-Prcpl}} := \text{approveTx}(tx^i_{\text{revoke-Prcpl}}, 0)$
   - $tx^i_{\text{revoke-Prcpl}}.\text{ScriptSigs}[0] := (\sigma^{i,l}_{\text{revoke-Prcpl}})$
   - $\sigma^{i',l}_{\text{revoke-Prm}} := \text{approveTx}(tx^{i'}_{\text{revoke-Prm}}, 0)$
   - $tx^{i'}_{\text{revoke-Prm}}.\text{ScriptSigs}[0] := (\sigma^{i',l}_{\text{revoke-Prm}})$
3) **Request for locking principal.**

$(\mathtt{sid}, \mathtt{pid}, \mathtt{request\text{-}lock\text{-}Prcpl}, \gamma_i.id, tx^i_{\text{lock-Prcpl}},\ tx^{i'}_{\text{revoke-Prm}}, tx^i_{\text{revoke-Prcpl}}) \overset{\tau}{\hookrightarrow} U_{i+1}$

$(\mathtt{sid}, \mathtt{pid}, \mathtt{request\text{-}lock\text{-}Prcpl}, \cdots) \overset{\tau+1}{\hookleftarrow} U_i, U_{i+1}$:

4) **Check proposed state.** Check if $status$ = enabled and

$$\top = \text{checkPrcpl}(\gamma_i.\text{id}, \mathtt{Prcpl}, tx^i_{\text{lock-Prcpl}}, tx^i_{\text{revoke-Prcpl}}, tx^{i'}_{\text{revoke-Prm}}, tr^i_{\text{splicing}}, tr^i_{\text{redeem}})$$

   If any check fails, abort.
5) **Approve revoke transactions.**
   - $\sigma^{i,r}_{\text{revoke-Prcpl}} := \text{approveTx}(tx^i_{\text{revoke-Prcpl}}, 0)$
   - $\sigma^{i',r}_{\text{revoke-Prm}} := \text{approveTx}(tx^{i'}_{\text{revoke-Prm}}, 0)$
   - Add $(\sigma^{i,r}_{\text{revoke-Prcpl}})$ in $tx^i_{\text{revoke-Prcpl}}.\text{ScriptSigs}[0]$
   - Add $(\sigma^{i',r}_{\text{revoke-Prm}})$ in $tx^{i'}_{\text{revoke-Prm}}.\text{ScriptSigs}[0]$
6) **Update the redeem handler**

$$(\text{sid}^i_{RH}, \mathtt{UPDATE}, tx^{i'}_{\text{revoke-Prm}}) \overset{\tau+1}{\hookrightarrow} \mathcal{F}^i_{\text{redeem-handler}}$$

7) **Approve the new state.**
   - $\sigma^{i,r}_{\text{lock-Prcpl}} := \text{approveTx}(tx^i_{\text{lock-Prcpl}}, 0)$

$(\mathtt{sid}, \mathtt{pid}, \mathtt{approved\text{-}lock\text{-}Prcpl}, tx^i_{\text{revoke-Prcpl}}, tx^{i'}_{\text{revoke-Prm}}, \sigma^{i,r}_{\text{lock-Prcpl}}) \overset{\tau+1}{\hookrightarrow} U_i$

$(\mathtt{sid}, \mathtt{pid}, \mathtt{approved\text{-}lock\text{-}Prcpl}, \cdots) \overset{\tau+2}{\hookleftarrow} U_{i+1}, U_i$:

8) **Check the signature.** Check if $\sigma^{i,r}_{\text{lock-Prcpl}}$ is valid, and $tx^i_{\text{revoke-Prcpl}}$ and $tx^{i'}_{\text{revoke-Prm}}$ contain all required signatures.
9) **Approve the new state.**
   - $\sigma^{i,l}_{\text{lock-Prcpl}} := \text{approveTx}(tx^i_{\text{lock-Prcpl}}, 0)$
   - $tx^i_{\text{lock-Prcpl}} := \text{genUpdateTx}(tx^i_{\text{lock-Prcpl}}, \sigma^{i,l}_{\text{lock-Prcpl}}, \sigma^{i,r}_{\text{lock-Prcpl}})$

10) **Update the channel state.**

$$(\mathsf{sid}_C, \mathtt{UPDATE}, \gamma_i.id, tx^i_{\mathtt{lock\text{-}Prcpl}}) \overset{\tau+2}{\hookrightarrow} \mathcal{F}_{Channels}$$

11) **Update the splicing trigger.** If $(\mathsf{sid}_C, \mathtt{UPDATED}, \gamma_i.\mathrm{id}) \overset{\tau'}{\hookrightarrow} \mathcal{F}_{Channels}$, then:
- $\sigma := \mathrm{approveTx}(tx^i_{\mathtt{revoke\text{-}Prcpl}}, 1)$
- $tx^i_{\mathtt{revoke\text{-}Prcpl}}.\mathrm{ScriptSigs}[1] := (\sigma)$
- $(\mathsf{sid}^i_{SH}, \mathtt{UPDATE}, tx^i_{\mathtt{revoke\text{-}Prcpl}}) \overset{\tau'}{\hookrightarrow} \mathcal{F}^i_{\text{splicing-handler}}$
- Send $(\mathsf{sid}, \mathsf{pid}, \mathtt{lock\text{-}Prcpl\text{-}Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$

$(\mathsf{sid}_C, \mathtt{UPDATED}, \gamma_i.\mathrm{id}) \overset{\tau'}{\hookrightarrow} \mathcal{F}_{Channels}, U_{i+1}$:

12) Send $(\mathsf{sid}, \mathsf{pid}, \mathtt{lock\text{-}Prcpl\text{-}Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$.

2pRefundDeposit Protocol: Π^i_{2pRfdDep}

Parameters owned by both U_i and U_{i+1}:
- pidSet: the set of all payment ids. Initially empty.

$(\mathsf{sid}, \mathsf{pid}, \mathtt{setup}, tr^i_{\text{splicing}}, tr^i_{\text{redeem}}, \mathsf{Dep}_i) \overset{\leq\tau}{\hookrightarrow} \mathcal{E}, U_i$:

0) If pid ∈ pidSet, then abort. Otherwise, add pid to pidSet, save $tr^i_{\text{splicing}}, tr^i_{\text{redeem}}, \mathsf{Dep}_i$.

$(\mathsf{sid}, \mathsf{pid}, \mathtt{init}, \gamma_i.\mathrm{id}, tr^i_{\text{splicing}}, tr^i_{\text{redeem}}, \mathsf{Dep}_i) \overset{\tau}{\hookrightarrow} \mathcal{E}, U_{i+1}$:

1) **Construct deposit refund txs.** If pid ∈ pidSet, then abort. Otherwise, add pid to pidSet and:

$$(tx^i_{\mathtt{rfd\text{-}Dep}}, tx^{i'}_{\mathtt{revoke\text{-}Prcpl}}, tx^{i''}_{\mathtt{revoke\text{-}Prm}}) := \mathrm{genDepRfdTx}(\gamma_i.\mathrm{id}, \mathsf{Dep}_i, tr^i_{\text{splicing}}, tr^i_{\text{redeem}})$$

2) **Approve the revoke transactions.**
- $\sigma^{i',r}_{\mathtt{revoke\text{-}Prcpl}} := \mathrm{approveTx}(tx^{i'}_{\mathtt{revoke\text{-}Prcpl}}, 0)$
- $\sigma^{i'',r}_{\mathtt{revoke\text{-}Prm}} := \mathrm{approveTx}(tx^{i''}_{\mathtt{revoke\text{-}Prm}}, 0)$
- $tx^{i'}_{\mathtt{revoke\text{-}Prcpl}}.\mathrm{ScriptSigs}[0] := (\sigma^{i',r}_{\mathtt{revoke\text{-}Prcpl}})$
- $tx^{i''}_{\mathtt{revoke\text{-}Prm}}.\mathrm{ScriptSigs}[0] := (\sigma^{i'',r}_{\mathtt{revoke\text{-}Prm}})$

3) **Request for deposit refund.**

$$(\mathsf{sid}, \mathsf{pid}, \mathtt{request\text{-}Rfd\text{-}Dep}, \gamma_i.id, tx^i_{\mathtt{rfd\text{-}Dep}}, tx^{i'}_{\mathtt{revoke\text{-}Prcpl}}, tx^{i''}_{\mathtt{revoke\text{-}Prm}}) \overset{\tau}{\hookrightarrow} U_i$$

$(\mathsf{sid}, \mathsf{pid}, \mathtt{request\text{-}Rfd\text{-}Dep}, \ldots) \overset{\tau+1}{\hookrightarrow} U_{i+1}, U_i$:

4) **Check the tx validity.** Check if pid ∈ pidSet and

$$\top = \mathrm{checkRfdDep}(\gamma_i.\mathrm{id}, \mathsf{Dep}_i, tx^i_{\mathtt{rfd\text{-}Dep}}, tx^{i'}_{\mathtt{revoke\text{-}Prcpl}}, tx^{i''}_{\mathtt{revoke\text{-}Prm}}, tr^i_{\text{splicing}}, tr^i_{\text{redeem}})$$

5) **Approve revoke transactions.**
- $\sigma^{i',l}_{\mathtt{revoke\text{-}Prcpl}} := \mathrm{approveTx}(tx^{i'}_{\mathtt{revoke\text{-}Prcpl}}, 0)$
- $\sigma^{i'',l}_{\mathtt{revoke\text{-}Prm}} := \mathrm{approveTx}(tx^{i''}_{\mathtt{revoke\text{-}Prm}}, 0)$
- Add $(\sigma^{i',l}_{\mathtt{revoke\text{-}Prcpl}})$ in $tx^{i'}_{\mathtt{revoke\text{-}Prcpl}}.\mathrm{ScriptSigs}[0]$
- Add $(\sigma^{i'',l}_{\mathtt{revoke\text{-}Prm}})$ in $tx^{i''}_{\mathtt{revoke\text{-}Prm}}.\mathrm{ScriptSigs}[0]$

6) **Update the splicing handler.**

$$(\mathsf{sid}^i_{SH}, \mathsf{pid}, \mathtt{UPDATE}, tx^{i'}_{\mathtt{revoke\text{-}Prcpl}}) \overset{\tau+1}{\hookrightarrow} \mathcal{F}^i_{\text{splicing-handler}}$$

7) **Approve the new state.**

$$\sigma^{i,l}_{\mathtt{rfd\text{-}Dep}} := \mathrm{approveTx}(tx^i_{\mathtt{rfd\text{-}Dep}}, 0)$$

8) **Send approval to U_{i+1}:**

$$(\mathsf{sid}, \mathsf{pid}, \mathtt{approved\text{-}rfd\text{-}Dep}, \sigma^{i,l}_{\mathtt{rfd\text{-}Dep}}, tx^{i'}_{\mathtt{revoke\text{-}Prcpl}}, tx^{i''}_{\mathtt{revoke\text{-}Prm}}) \overset{\tau+1}{\hookrightarrow} U_{i+1}$$

$(\mathsf{sid}, \mathsf{pid}, \mathtt{approved\text{-}rfd\text{-}Dep}, \ldots) \overset{\tau+2}{\hookrightarrow} U_i, U_{i+1}$:

9) **Check the signatures.** Check if $\sigma^{i,l}_{\mathtt{rfd\text{-}Dep}}$ is valid, and $tx^{i'}_{\mathtt{revoke\text{-}Prcpl}}$ and $tx^{i''}_{\mathtt{revoke\text{-}Prm}}$ contain valid signatures.

10) **Approve the new state.**
- $\sigma^{i,r}_{\mathtt{rfd\text{-}Dep}} := \mathrm{approveTx}(tx^i_{\mathtt{rfd\text{-}Dep}}, 0)$
- $tx^i_{\mathtt{rfd\text{-}Dep}} := \mathrm{genUpdateTx}(tx^i_{\mathtt{rfd\text{-}Dep}}, \sigma^{i,l}_{\mathtt{rfd\text{-}Dep}}, \sigma^{i,r}_{\mathtt{rfd\text{-}Dep}})$

11) **Update the channel state.**

$$(\mathsf{sid}_C, \mathtt{UPDATE}, \gamma_i.id, tx^i_{\mathtt{rfd\text{-}Dep}}) \overset{\tau+2}{\hookrightarrow} \mathcal{F}_{Channels}$$

12) **Update the redeem handler.** If $(\mathsf{sid}_C, \mathtt{UPDATED}, \gamma_i.\mathrm{id}, tx^i_{\mathtt{rfd\text{-}Dep}}.\mathrm{Outs}) \overset{\tau'}{\hookrightarrow} \mathcal{F}_{Channels}$, then:
- $\sigma := \mathrm{approveTx}(tx^{i''}_{\mathtt{revoke\text{-}Prm}}, 1)$
- $tx^{i''}_{\mathtt{revoke\text{-}Prm}}.\mathrm{ScriptSigs}[1] := (\sigma)$
- $(\mathsf{sid}^i_{RH}, \mathsf{pid}, \mathtt{UPDATE}, tx^{i''}_{\mathtt{revoke\text{-}Prm}}) \overset{\tau'}{\hookrightarrow} \mathcal{F}^i_{\mathtt{redeem\text{-}handler}}$
- $(\mathsf{sid}, \mathsf{pid}, \mathtt{rfd\text{-}Dep\text{-}Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$

$(\mathsf{sid}_C, \mathtt{UPDATED}, \gamma_i.\mathrm{id}, tx^i_{\mathtt{rfd\text{-}Dep}}.\mathrm{Outs}) \overset{\tau'}{\hookrightarrow} \mathcal{F}_{Channels}, U_i$:

13) **Return the final state.** $(\mathsf{sid}, \mathsf{pid}, \mathtt{rfd\text{-}Dep\text{-}Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$

2pSettle Protocol: Π^i_{2pSettle}

Parameters owned by both U_i and U_{i+1}:
- pidSet: the set of all payment ids. Initially empty.

$(\mathsf{sid}, \mathsf{pid}, \mathtt{setup}, lb', rb') \overset{\leq\tau}{\hookrightarrow} \mathcal{E}, U_{i+1}$:

0) If pid ∈ pidSet, then abort. Otherwise, add pid to pidSet, save lb' and rb'.

$(\mathsf{sid}, \mathsf{pid}, \mathtt{init}, \gamma_i.id, lb', rb') \overset{\tau}{\hookrightarrow} \mathcal{E}, U_i$:

1) **Generate paid state over γ_i.** If pid ∈ pidSet, then abort. Otherwise, add pid to pidSet and:

$$tx^i_{\mathtt{settle}} := \mathrm{genSettleTx}(\gamma_i.id, lb', rb')$$

2) **Request for settle.**

$$(\mathsf{sid}, \mathsf{pid}, \mathtt{request\text{-}settle}, \gamma_i.id, tx^i_{\mathtt{settle}}) \overset{\tau}{\hookrightarrow} U_{i+1}$$

$(\mathsf{sid}, \mathsf{pid}, \mathtt{request\text{-}settle}, \gamma_i.id, tx^i_{\mathtt{settle}}) \overset{\tau+1}{\hookrightarrow} U_i, U_{i+1}$:

3) **Check the balance**. Check if pid ∈ pidSet and verify that $tx^i_{\mathtt{settle}}$ reflects the agreed balance of lb' and rb'. If any check fails, abort.

4) **Approve the settlement.**

$$\sigma^{i,r}_{\mathtt{settle}} := \mathrm{approveTx}(tx^i_{\mathtt{settle}}, 0)$$

5) **Send approval to U_i:**

$$(\mathsf{sid}, \mathsf{pid}, \mathtt{approved\text{-}settle}, \sigma^{i,r}_{\mathtt{settle}}) \overset{\tau+1}{\hookrightarrow} U_i$$

$(\mathsf{sid}, \mathsf{pid}, \mathtt{approved\text{-}settle}, \sigma^{i,r}_{\mathtt{settle}}) \overset{\tau+2}{\hookrightarrow} U_{i+1}, U_i$:

6) **Check the signature validity.** Verify that $\sigma^{i,r}_{\mathtt{settle}}$ is valid. Otherwise, abort.

7) **Generate the settle state of γ_i.**
- $\sigma^{i,l}_{\mathtt{settle}} := \mathrm{approveTx}(tx^i_{\mathtt{settle}}, 0)$
- $tx^i_{\mathtt{settle}} := \mathrm{genUpdateTx}(tx^i_{\mathtt{settle}}, \sigma^{i,l}_{\mathtt{settle}}, \sigma^{i,r}_{\mathtt{settle}})$

8) **Update the channel state.**

$$(\mathsf{sid}_C, \mathtt{UPDATE}, \gamma_i.id, tx^i_{\mathtt{settle}}) \overset{\tau+2}{\hookrightarrow} \mathcal{F}_{Channels}$$

9) If $(\texttt{sid}, \texttt{UPDATED}, \gamma_i.\text{id}, tx^i_{\texttt{settle}}.\text{Outs}) \overset{\tau'}{\hookleftarrow} \mathcal{F}_{Channels}$, then send $(\texttt{sid}, \texttt{pid}, \texttt{settle-Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$.

$(\texttt{sid}, \texttt{UPDATED}, \gamma_i.\text{id}, tx^i_{\texttt{settle}}.\text{Outs}) \overset{\tau'}{\hookleftarrow} \mathcal{F}_{Channels}, U_{i+1}$:

10) **Return the final state.** $(\texttt{sid}, \texttt{pid}, \texttt{settle-Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$
By round $\tau + t_{update} + 2$, if channel still not updated, abort.

Splicing Launcher Protocol: $\Pi^i_{\text{splicing-launcher}}$, $i \in \{0, 1\}$

Parameters owned by U_i:

- pidSet: the set of all payment ids. Initially empty.

$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_i.id, T) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_i$:

1) If $\texttt{pid} \in \texttt{pidSet}$, then abort. Otherwise, add pid to pidSet.
2) Check if $U_i = \gamma_i.lu$ or $U_i = \gamma_i.ru$. If not, abort.
3) Save T.
4) At round $T - t_{close}$, close channel γ_i:

$$(\texttt{ssid}, \texttt{CLOSE}, \gamma_i.id) \xrightarrow{T - t_{close}} \mathcal{F}_{Channels}$$

5) If $(\texttt{ssid}, \texttt{CLOSED}, \gamma_i.\text{id}) \overset{\tau'}{\hookleftarrow} \mathcal{F}_{Channels}$, then send $(\texttt{sid}, \texttt{pid}, \texttt{splicing-launched}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$.

$(\texttt{sid}, \texttt{pid}, \texttt{CLOSE}) \overset{\tau^* \geq \tau}{\longleftrightarrow} \mathcal{E}, U_i$:

6) Terminate the protocol.

Redeem Launcher Protocol: $\Pi^i_{\text{redeem-launcher}}$

Parameters owned by U_i:

- pidSet: the set of all payment ids. Initially empty.

$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_i.id, T, tr, tx) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_i$:

1) If $\texttt{pid} \in \texttt{pidSet}$, then abort. Otherwise, add pid to pidSet.
2) Save T, tr, tx.
3) At every round $\tau' \in [\tau, T + t_{close} + \Delta]$, if tr is confirmed over $\mathcal{G}_{\text{L}}$, then:

$$(\texttt{sid}, \texttt{POST}, tx) \overset{\tau'}{\hookrightarrow} \mathcal{G}_{\text{L}}$$

4) If tx is confirmed on $\mathcal{G}_{\text{L}}$, send $(\texttt{sid}, \texttt{pid}, \texttt{redeem-complete}) \hookrightarrow \mathcal{E}$, and terminate.
5) Terminate at round $T + t_{close} + \Delta$.

Slash Launcher Protocol: $\Pi^i_{\text{slash-launcher}}$

Parameters owned by U_i:

- pidSet: the set of all payment ids. Initially empty.

$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_i.id, T, tr, tx) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_i$:

1) If $\texttt{pid} \in \texttt{pidSet}$, then abort. Otherwise, add pid to pidSet.
2) Save T, tr, tx.
3) At round $T + t_{close} + \Delta$, if tr is an unspent transaction output in $\mathcal{G}_{\text{L}}$, then:
   - $\sigma := \text{approveTx}(tx, 0)$
   - $tx.\text{ScriptSigs}[0] := (\sigma)$
   - $(\texttt{ssid}, \texttt{POST}, tx) \xrightarrow{T + t_{close} + \Delta} \mathcal{G}_{\text{L}}$
   - If tx fails to be confirmed within Δ, generate other tx' to spend the UTXO and submit:

$$(\texttt{ssid}, \texttt{POST}, tx') \xrightarrow{T + t_{close} + 2\Delta} \mathcal{G}_{\text{L}}$$

   - When either tx or tx' is confirmed, send $(\texttt{sid}, \texttt{pid}, \texttt{slash-complete}) \hookrightarrow \mathcal{E}$.
4) IF tr is not confirmed within $T + t_{close} + \Delta$, then terminate.

Splicing Handler Protocol: $\Pi^i_{\texttt{splicing-handler}}$

Parameters owned by U_i:

- pidSet: the set of all payment ids. Initially empty.
- owner: the owner of the splicing handler.
- T: the expire time to end this handler.
- tr: the trigger UTXO we need to observe.
- txs: transactions to be broadcasted.

$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_i.id, T, tr, tx) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_i$:

1) If $\texttt{pid} \in \texttt{pidSet}$, then abort. Otherwise, add pid to pidSet.
2) Check if $U_i = \gamma_i.lu$. If not, abort.
3) Set owner $:= U_i$, $T := T$, $tr := tr$, $txs := \{tx\}$.

$(\texttt{sid}, \texttt{pid}, \texttt{UPDATE}, tx') \overset{\tau'}{\longleftrightarrow}$ owner:

4) Add tx' to txs.

At each round $\tau' \in [\tau, T]$, if tr is confirmed over $\mathcal{G}_{\text{L}}$, then:

5) Close channel γ_i by sending:

$$(\texttt{ssid}_C, \texttt{CLOSE}, \gamma_i.id) \overset{\tau'}{\hookrightarrow} \mathcal{F}_{Channels}$$

6) After t_{close} rounds, broadcast the valid transaction from txs to $\mathcal{G}_{\text{L}}$:

$$(\texttt{ssid}_L, \texttt{POST}, tx_{\text{valid}}) \xrightarrow{\tau' + t_{close}} \mathcal{G}_{\text{L}}$$

7) Send $(\texttt{sid}, \texttt{pid}, \texttt{splicing-handled}) \xrightarrow{\tau' + t_{close}} \mathcal{E}$, and terminate.
8) Terminate at round T.

Redeem Handler Protocol: $\Pi^i_{\texttt{redeem-handler}}$

Parameters owned by U_{i+1}:

- pidSet: the set of all payment ids. Initially empty.
- owner: the owner of the redeem handler.
- T: the expire time to end this handler.
- tr: the trigger UTXO we need to observe.
- txs: transactions to be broadcasted.

$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_i.id, T, tr, tx) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_{i+1}$:

1) If $\texttt{pid} \in \texttt{pidSet}$, then abort. Otherwise, add pid to pidSet.
2) Check if $U_{i+1} = \gamma_i.ru$. If not, abort.
3) Set owner $:= U_{i+1}$, $T := T$, $tr := tr$, $txs := \{tx\}$.
4) At round $T - t_{close}$, close channel γ_i by sending:

$$(\texttt{ssid}_C, \texttt{CLOSE}, \gamma_i.id) \xrightarrow{T - t_{close}} \mathcal{F}_{Channels}$$

$(\texttt{sid}, \texttt{pid}, \texttt{UPDATE}, tx') \overset{\tau'}{\longleftrightarrow}$ owner:

5) Check if the sender is the owner. If not, abort.
6) Add tx' to txs.

At each round $\tau' \in [\tau, T]$, if tr is confirmed over $\mathcal{G}_{\text{L}}$, then:

7) Close channel γ_i by sending:

$$(\texttt{ssid}_C, \texttt{CLOSE}, \gamma_i.id) \overset{\tau'}{\hookrightarrow} \mathcal{F}_{Channels}$$

8) After t_{close} rounds, find and broadcast the valid transaction from txs to $\mathcal{G}_{\text{L}}$:

$$(\texttt{ssid}_L, \texttt{POST}, tx_{\text{valid}}) \xrightarrow{\tau' + t_{close}} \mathcal{G}_{\text{L}}$$

9) Send $(\texttt{sid}, \texttt{pid}, \texttt{redeem-handled}) \xrightarrow{\tau' + t_{close}} \mathcal{E}$, and terminate.
10) Terminate the protocol at T.

G Auxiliary Algorithms

G.1 Helper functions.

This section includes several useful functions that help to construct, approve transactions, and update the state of the payment channel.

genOnion($\mathcal{M}, tx_{\text{splicing}}, tx_{\text{redeem}}$)

(1) Parse $\mathcal{M}$ to obtain the number of relays (n), base payment amount x, payment lifetime T, and the relay fee f_i for each $i \in [1, n]$.
(2) Calculate the padding fee, $f_0 := \max\{F - \sum_{j=1}^{n} f_j, 0\}$.
(3) Initialize message, $m := []$.
(4) **for** $i \in [1, n]$:
- $\texttt{Dep}_i := c_{\text{payee}} + (n - i)c_{\text{relay}}$.
- $\texttt{Prm}_i := \sum_{j=0}^{i} f_j$
- $\texttt{Prcpl} := x + \sum_{j=0}^{n} f_j$.
- $tr^i_{\text{splicing}} := (tx_{\text{splicing}}, i - 1)$
- $tr^i_{\text{redeem}} := (tx_{\text{redeem}}, i)$

(5) $m[1] := (\texttt{Dep}_1, \texttt{Prcpl}, tr^1_{\text{splicing}}, T)$
(6) $m[n + 1] := (\texttt{Prm}_n, \texttt{Prcpl}, tr^n_{\text{redeem}}, T)$
(7) $\forall i \in [2, n]$: $m[i] := (\texttt{Dep}_i, \texttt{Prm}_{i-1}, \texttt{Prcpl}, tr^i_{\text{splicing}}, tr^i_{\text{redeem}}, tr^{i-1}_{\text{splicing}}, tr^{i-1}_{\text{redeem}}, T)$
(8) **Return** m.

checkPayeeStake(U, T):

(1) **If** $\exists\, tx \in \mathcal{G}_{\text{L}}$.TXs, where tx contains an unspent output with index i, such that:
- tx.Outs$[i]$.amt $\geq c_{\text{burn}} + F$, and
- tx.Outs$[i].\phi$ requires the spending transaction tx_{spent} satisfy:
  - If tx.Outs$[i]$ is spent before T_{expire}, then

$$tx_{\text{spent}}.\text{Outs}[0] = \{\text{amt} : c_{\text{burn}}, \quad \phi = \text{Sig}(\perp))$$

  - After T_{expire}, U can spend the output with a valid signature.
- $T_{\text{expire}} \gg T$

(2) **If** all checks pass, **return** (tx, i).
(3) Otherwise, **return** $\perp$.

genSplicingTxs$(\gamma.\text{id}, \text{Out}_{\texttt{stake}}, \mathcal{M})$:

(1) Lookup the corresponding channel γ based on $\gamma.id$.
(2) Parse $\mathcal{M}$ to obtain the parameters within it: $\mathcal{M} = (U, \Gamma, x, \text{Fees}, \mathcal{S})$.
(3) Construct the splicing transaction tx_{splicing}:
- $\hat{\theta}_0 := \{\text{amt} : x + n\epsilon, \phi : ((s \leq \text{rel}(t_{\text{close}} + \Delta)) \wedge \text{Sig}(U_{n+1})) \vee (\text{Sig}(U_0) \geq \text{rel}(t_{\text{close}} + \Delta))\}$
- For $i \in [1, n]$, $\hat{\theta}_i := \{\text{amt} : \epsilon, \phi : \text{Sig}(U_i) \leq \text{rel}(2t_{\text{close}} + \Delta)\}$.
- $\theta_0 := \{\text{amt} : lb - x - (2n)\epsilon, \phi : \text{Sig}(U_0)\}$.
- $\theta_1 := \{\text{amt} : rb, \phi : \text{Sig}(U_1)\}$.
- $tx_{\text{splicing}} := \text{genState}(\gamma_0, \{\hat{\theta}_0, \ldots, \hat{\theta}_n, \theta_0, \theta_1\})$.

(4) Generate the redeem transaction tx_{redeem}:
- $\theta_0 := \{\text{amt} : x, \phi : \text{Sig}(U_{n+1})\}$.
- For each $i \in [1, n]$, $\theta_{i+1} := \{\text{amt} : \epsilon, \phi : \text{Sig}(U_i) \leq \text{rel}(t_{\text{close}} + \Delta)\}$.
- $tx_{\text{redeem}} := \{\text{Ins}[0] : \{tx : tx_{\text{splicing}}, \text{idx} : 0\}, \text{Outs} : (\theta_0, \ldots, \theta_{n+1})\}$.

(5) Generate the slash transaction tx_{slash}:
- $\theta_0 := \{\text{amt} : x + n\epsilon, \phi : \text{Sig}(U_0)\}$.
- $\theta_1 := \{\text{amt} : c_{\text{burn}}, \phi : \text{Sig}(0)\}$.
- $tx_{\text{slash}} := \{\text{Ins}[0] : \{tx : tx_{\text{splicing}}, \text{idx} : 0\}, \text{Ins}[1] : \text{Out}_{\texttt{stake}}, \text{Outs} : (\theta_{\text{slash}}, \theta_{\text{return}})\}$.

(6) Return $(tx_{\text{splicing}}, tx_{\text{redeem}}, tx_{\text{slash}})$.

checkSlashTx$(tx_{\text{splicing}}, tx_{\text{redeem}}, tx_{\text{slash}}, h, x)$:

(1) For tx_{splicing}, check if tx_{splicing}.Outs$[0]$.amt $\geq x$ and that tx_{splicing}.Outs$[0].\phi$ enables U_{n+1} to redeem the payment within rel(Δ).
(2) For tx_{redeem} and tx_{slash}, check if they are constructed using the function genSplicingTxs.
(3) **If** all the above conditions are satisfied, **return** $\top$. Otherwise, **return** $\perp$.

approveTx(tx, idx) *called by* U :

(1) Parse the "idx"-th input of transaction tx to obtain the spending UTXO, $Out_{\text{spending}} := tx$.Ins[idx].
(2) **If** the spending UTXO is locked with Sig(U), then **return** $\sigma := \text{Sign}(U.\text{sk}, tx)$.
(3) Otherwise, **return** $\perp$.

checkSlashSig$(\sigma_{\text{slash}}, tx_{\text{slash}})$:

(1) Check if σ_{slash} is a valid signature to spend tx_{slash}.Ins$[0]$.
(2) **If** yes, **return** $\top$, otherwise, **return** $\perp$.

genUpdateTx$(\gamma_{id}, tx_{\text{state}}, \sigma^l, \sigma^r)$:

(1) Construct a channel udpate transaction tx_{update} copying the state of tx_{state}, if σ^l and σ^r are valid signatures to spend the left and right channel states, respectively.
(2) Return tx_{update}.

genDepTx$(\gamma.id, \texttt{Dep}, tr_{splicing}, T)$:

(1) Lookup the corresponding channel γ based on $\gamma.id$.
(2) Define the deposit lock output:

$$\theta_0 := \left\{ \text{amt} : \texttt{Dep}, \quad \phi : \begin{cases} Sig(\gamma.lu) \wedge Sig(\gamma.ru) \vee \\ Sig(\gamma.ru) \geq T \end{cases} \right.$$

(3) Define the remained outputs:
- $\theta_1 := \{\text{amt} : lb - \texttt{Dep}, \phi : \text{Sig}(\gamma.lu)\}$
- $\theta_2 := \{\text{amt} : rb, \phi : \text{Sig}(\gamma.ru)\}$

(4) Construct the lock-deposit transaction:

$$tx_{\texttt{lock-Dep}}.\text{Outs} := \{\theta_0, \theta_1, \theta_2\}$$

(5) Define the deposit revoke output:

$$tx_{\texttt{revoke-Dep}}.\text{Outs}[0] := \left\{ \text{amt} : \texttt{Dep} + \epsilon, \quad \phi : \text{Sig}(\gamma.lu) \right\}$$

(6) In the revoke transaction, set the first input to refer to the lock transaction:

$$tx_{\texttt{revoke-Dep}}.\text{Ins}[0] := \{tx : tx_{\texttt{lock-Dep}}, \text{idx} : 0\}$$

(7) Link the splicing transaction to the revoke deposit transaction:

$$tx_{\texttt{revoke-Dep}}.\text{Ins}[1] := tr_{splicing}$$

(8) **Return** $(tx_{\texttt{lock-Dep}}, tx_{\texttt{revoke-Dep}})$.

genPrmTx$(\gamma.id, \texttt{Prm}, T, tr_{redeem}, tr_{\text{splicing}})$:

(1) Lookup the corresponding channel γ based on $\gamma.id$.
(2) Fetch the previously locked deposit:

$$\theta_0 := \gamma.\text{state}[0]$$

(3) Define the new premium lock output:

$$\theta_1 := \left\{ \text{amt} : \texttt{Prm}, \quad \phi : \begin{cases} \text{Sig}(\gamma.lu) \wedge \text{Sig}(\gamma.ru) \vee \\ \text{Sig}(\gamma.lu) \geq T + t_{\text{close}} + \Delta) \end{cases} \right\}$$

(4) Construct the remained outputs:

$$\theta_2 := \gamma.\text{state}[2]; \quad \theta_3 := \{\text{amt} : rb - \texttt{Prm}, \phi : \text{Sig}(\gamma.ru)\}$$

(5) Construct the lock-premium transaction:

$$tx_{\texttt{lock-Prm}}.\text{Outs} := \{\theta_0 \ldots, \theta_3\}$$

(6) Construct new deposit revoke (left revoke) tx' inputs:

$$tx'_{\texttt{revoke-Dep}}.\text{Outs}[0] := \left\{ \text{amt} : \gamma.\text{state}[0]{+}\epsilon, \quad \phi : \text{Sig}(\gamma.lu) \right\}$$

(7) Set the first input to refer to latest state:

$$tx'_{\texttt{revoke-Dep}}.\text{Ins}[0] := \{tx : tx_{\texttt{lock-Prm}}, \text{ idx} : 0\}$$

(8) Link the splicing trigger to the revoke deposit transaction:

$$tx'_{\texttt{revoke-Dep}}.\text{Ins}[1] := tr_{\text{splicing}}$$

(9) Define the premium revoke output:

$$\theta_{\texttt{revoke-Prm}} := \left\{ \text{amt} : \texttt{Prm} + \epsilon, \quad \phi : \text{Sig}(\gamma.ru) \right\}$$

(10) Construct the revoke-premium transaction:

$$tx_{\texttt{revoke-Prm}}.\text{Ins}[0] := \{tx : tx_{\texttt{lock-Prm}}, \text{ idx} : 1\}$$

$$tx_{\texttt{revoke-Prm}}.\text{Ins}[1] := tr_{redeem}$$

(11) **Return** $(tx_{\texttt{lock-Prm}}, tx_{\texttt{revoke-Prm}}, tx'_{\texttt{revoke-Dep}})$.

genPrcplTx($\gamma.id, \texttt{Prcpl}, T, tr_{\text{splicing}}, tr_{redeem}$):

(1) Fetch the previously locked deposit:

$$\theta_0 := \gamma.\text{state}[0] \quad \theta_1 := \gamma.\text{state}[1]$$

(2) Lock the principal $theta_0$.amt := θ_0.amt + Prcpl.

(3) Construct the remained outputs:

$$\theta_2 := \{\text{amt} : \gamma.\text{state}[3].\text{amt} - \texttt{Prcpl}, \phi : \text{Sig}(\gamma.lu)\}$$

$$\theta_3 := \gamma.\text{state}[3];$$

(4) Construct the lock-principal transaction:

$$tx_{\texttt{lock-Prcpl}}.\text{Outs} := \{\theta_0 \ldots, \theta_3\}$$

(5) Construct new principal revoke (left revoke) tx' inputs:

$$tx_{\texttt{revoke-Prcpl}}.\text{Outs}[0] := \left\{ \text{amt} : \gamma.\text{state}[0]{+}\epsilon, \quad \phi : \text{Sig}(\gamma.lu) \right\}$$

(6) Set the first input to refer to latest state:

$$tx_{\texttt{revoke-Prcpl}}.\text{Ins}[0] := \{tx : tx_{\texttt{lock-Prcpl}}, \text{ idx} : 0\}$$

(7) Link the splicing trigger to the revoke principal transaction:

$$tx_{\texttt{revoke-Prcpl}}.\text{Ins}[1] := tr_{\text{splicing}}$$

(8) Define the premium revoke output:

$$\theta_{\texttt{revoke-Prm}} := \left\{ \text{amt} : \gamma.\text{state}[1] + \epsilon, \quad \phi : \text{Sig}(\gamma.ru) \right\}$$

(9) Construct the revoke-premium transaction:

$$tx_{\texttt{revoke-Prm}}.\text{Ins}[0] := \{tx : tx_{\texttt{lock-Prcpl}}, \text{ idx} : 1\}$$

$$tx_{\texttt{revoke-Prm}}.\text{Ins}[1] := tr_{redeem}$$

(10) **Return** $(tx_{\texttt{lock-Prcpl}}, tx_{\texttt{revoke-Prm}}, tx_{\texttt{revoke-Prcpl}})$.

genDepRfdTx($\gamma.id, \texttt{Dep}, tr_{\text{splicing}}, tr_{\text{redeem}}$):

(1) Preserve the previously locked deposit:

$$\theta_0 := \gamma.\text{state}[0] \quad \theta_1 := \gamma.\text{state}[1]$$

(2) refund the deposit $theta_0$.amt := θ_0.amt − Dep.

(3) Construct the remained outputs:

$$\theta_2 := \{\text{amt} : \gamma.\text{state}[3].\text{amt} + \texttt{Dep}, \phi : \text{Sig}(\gamma.lu)\}$$

$$\theta_3 := \gamma.\text{state}[3];$$

(4) Construct the refund-deposit transaction:

$$tx_{\texttt{refund-Dep}}.\text{Outs} := \{\theta_0 \ldots, \theta_3\}$$

(5) Construct new revoke tx:

$$tx_{\texttt{revoke-Prcpl}}.\text{Outs}[0] := \left\{ \text{amt} : \gamma.\text{state}[0]{+}\epsilon, \quad \phi : \text{Sig}(\gamma.lu) \right\}$$

(6) Set the first input to refer to latest state:

$$tx_{\texttt{revoke-Prcpl}}.\text{Ins}[0] := \{tx : tx_{\texttt{refund-Dep}}, \text{ idx} : 0\}$$

(7) Link the splicing trigger:

$$tx_{\texttt{revoke-Prcpl}}.\text{Ins}[1] := tr_{\text{splicing}}$$

(8) Define the premium revoke output:

$$\theta_{\texttt{revoke-Prm}} := \left\{ \text{amt} : \gamma.\text{state}[1] + \epsilon, \quad \phi : \text{Sig}(\gamma.ru) \right\}$$

(9) Construct the revoke-premium transaction:

$$tx_{\texttt{revoke-Prm}}.\text{Ins}[0] := \{tx : tx_{\texttt{refund-Dep}}, \text{ idx} : 1\}$$

$$tx_{\texttt{revoke-Prm}}.\text{Ins}[1] := tr_{redeem}$$

(10) **Return** $(tx_{\texttt{refund-Dep}}, tx_{\texttt{revoke-Prm}}, tx_{\texttt{revoke-Prcpl}})$.

genSettleTx($\gamma.i, lb', rb'$):

(1) Lookup the corresponding channel γ based on $\gamma.id$.

(2) Define the settlement output, which finalizes the payment.

$$\theta_0 := \left\{ \text{amt} : lb', \quad \phi : \text{Sig}(\gamma.lu) \right\}$$

(3) Define the paid output for the counterparty:

$$\theta_1 := \left\{ \text{amt} : rb', \quad \phi : \text{Sig}(\gamma.ru) \right\}$$

(4) Generate the settle transaction:

$$tx_{\texttt{settle}}.\text{Outs} := \text{genState}\left(\gamma, \{\theta_0, \theta_1\}\right)$$

(5) **Return** $tx_{\texttt{settle}}$.

checkSplicing($\gamma_0.id, tx_{\text{splicing}}, tr^1_{\text{splicing}}, \sigma^l_{\text{splicing}}$):

(1) Check if U_1's balance is not reduced in the new state represented by tx_{splicing}, compared with γ_0.state.

(2) Check if tx_{splicing} contains tr^1_{splicing}.

(3) Check if the signature $\sigma^l_{\text{splicing}}$ is valid to spend the first input of tx_{splicing}.

(4) **If** all conditions are satisfied, **return** $\top$, **else return** $\bot$.

checkDep(γ_i.id, $tx^i_{\texttt{lock-Dep}}, tx^i_{\texttt{revoke-Dep}}, \texttt{Dep}_i, T, tr^i_{splicing}$):

(1) Ensure construct txs correctly:

$$tx^i_{\texttt{lock-Dep}}, \; tx^i_{\texttt{revoke-Dep}} = \text{genDepTx}(\gamma.id, \texttt{Dep}_i, tr^i_{\text{splicing}}, T)$$

(2) U_i provides valid signatures to active the revoke tx.
(3) **If** all conditions are satisfied, **return** $\top$, **else return** $\bot$.

checkPrm(γ_i.id, T, $\texttt{Prm}_i$, $tx^i_{\texttt{lock-Prm}}$, $tx^i_{\texttt{revoke-Prm}}$, $tx^{i'}_{\texttt{revoke-Dep}}$, tx_{redeem}, tr^i_{splicing}, tr^i_{redeem}, T):

(1) Check if the latest state and corresponding revoke txs are correctly constructed.

$$(tx^i_{\texttt{lock-Prm}},\ tx^i_{\texttt{revoke-Prm}},\ tx^{i'}_{\texttt{revoke-Dep}}) = \text{genPrmTx}(\gamma.id, \texttt{Prm}_i, T, tr^i_{\text{redeem}}, tr^i_{\text{splicing}})$$

(2) U_{i+1} provide valid signatures to active the two revoke txs.
(3) **If** all true **return** $\top$, **else return** $\bot$.

checkPrcpl(γ_i.id, $\texttt{Prcpl}$, T, $tx_{\texttt{lock-Prcpl}}$, $tx_{\texttt{revoke-Prcpl}}$ $tx^{i'}_{\texttt{revoke-Prm}}$, $tr_{splicing}$, tr_{redeem}) :

(1) Check if $tx_{\texttt{lock-Prcpl}}$ and $tx_{\texttt{revoke-Prcpl}}$ are correctly constructed.

$$(tx_{\texttt{lock-Prcpl}},\ tx_{\texttt{revoke-Prcpl}},\ tx^{i'}_{\texttt{revoke-Prm}}) = \text{genPrcplTx}(\gamma.id, \texttt{Prcpl}, T, tr_{\text{splicing}}, tr_{\text{redeem}})$$

(2) Check if U_i provides the correct signature over the revoke transactions.
(3) **If** all conditions are satisfied, **return** $\top$, **else return** $\bot$.

checkRfdDep(γ_i.id, $\texttt{Dep}_i$, $tx^i_{\texttt{rfd-Dep}}$, $tx^{i'}_{\texttt{revoke-Prcpl}}$, $tx^{i''}_{\texttt{revoke-Prm}}$, $tr_{splicing}$, tr_{redeem}) :

(1) Check if the new state is correctly constructed.

$$(tx^i_{\texttt{rfd-Dep}},\ tx^{i'}_{\texttt{revoke-Prcpl}},\ tx^{i''}_{\texttt{revoke-Prm}}) = \text{genDepRfdTx}(\gamma.id, \texttt{Dep}_i, tr_{\text{splicing}}, tr_{\text{redeem}})$$

(2) Check if U_{i+1} provides the correct signature over the revoke transactions.
(3) **If** all conditions are satisfied, **return** $\top$, **else return** $\bot$.

H UC proof

We utilize the global UC framework (GUC) [14] in our formal security analysis. Compare with standard UC, GUC enables a global setup, allowing protocol to interact with pre-configured global functionalities.

H.1 Ideal functionalities

H.1.1 Channel states and revoke transactions. Consider a multi-hop payment $\mathcal{M}_{CE}$ following the definition within Section 2 with n relays, here we list all the possible state of each channel from γ_0 to γ_n. We first define the funds involved in *Zeus*.

- Payee's on-chain stake. $\texttt{Stake} \geq c_{\text{burn}} \approx 2.3$ USD.
- Payer's off-chain deposit in channel $\gamma_{i\in[1,n]}.\texttt{Dep}_i = c_{\text{payee}} + (n-i)\cdot c_{\text{relay}}$.
- Premium in channel $\gamma_{i\in[1,n]}.\texttt{Prm}^i = f_0 + \sum_{k=1}^{i} f_k$.
- Principal in channel $\gamma_{i\in[1,n]}.\texttt{Prcpl} = x + \sum_{k=1}^{n} f_k$.

For channel γ_0 we define the following states:

- $\text{S}^0_{\texttt{IDLE}}$: we use $\text{S}_{\texttt{IDLE}}(lb, rb)$ to denotes the initial state of γ_0 where lb and rb are the initial balances of U_0 and U_1 respectively.

$$\theta_0 = \begin{cases}\text{amt}: & lb\\ \phi: & \text{Sig}(U_0)\end{cases} \quad \theta_1 = \begin{cases}\text{amt}: & rb\\ \phi: & \text{Sig}(U_1)\end{cases}$$

- $\text{S}^0_{\texttt{SPLICED}}$: we use $\text{S}_{\texttt{SPLICED}}(x, n)$ to denotes the state of γ_0 after the splicing transaction, where U_0 splices out about $x + n\epsilon$ tokens and creates n trigger outputs. The state is represented by $n+3$ outputs $(\theta_0, \ldots, theta_{n+2})$:

$$\theta_0 = \begin{cases}\text{amt}: & x + n\epsilon\\ \phi: & \begin{cases}\text{reveal } s \wedge \text{Sig}(U_{n+1})\\ \text{Sig}(U_0) \geq \text{rel}(t_{\text{close}} + \Delta)\end{cases}\end{cases}$$

$$\text{for } i \in [1,n], \theta_i = \begin{cases}\text{amt}: & \epsilon\\ \phi: & \begin{cases}\text{Sig}(U_i)\\ \text{Sig}(U_0) \geq \text{rel}(2t_{\text{close}} + \Delta)\end{cases}\end{cases}$$

$$\theta_{n+1} = \begin{cases}\text{amt}: & lb - x - (2n)\epsilon\\ \phi: & \text{Sig}(U_0)\end{cases} \quad \theta_{n+2} = \begin{cases}\text{amt}: & rb\\ \phi: & \text{Sig}(U_1)\end{cases}$$

where θ_0 is the main output containing the payment amount plus triggers, $\theta_{i\in[1,n]}$ are the individual trigger outputs, and θ_{n+1}, θ_{n+2} represent the remaining balances of U_0 and U_1 respectively.

- $\text{S}^0_{\texttt{PAID}}$: we use $\text{S}_{\texttt{PAID}}(lb - x - \text{Fees}, rb + x + \text{Fees})$ to denotes the state of γ_0 after the payment is successfully completed through the PCN, where $\text{Fees} = \sum_{i=1}^{n} f_i$ is the total fees paid to all relays. The state is represented by 2 outputs (θ_0, θ_1):

$$\theta_0 = \begin{cases}\text{amt}: & lb - x - \text{Fees}\\ \phi: & \text{Sig}(U_0)\end{cases} \quad \theta_1 = \begin{cases}\text{amt}: & rb + x + \text{Fees}\\ \phi: & \text{Sig}(U_1)\end{cases}$$

For channel $\gamma_i, i \in [1,n]$, we define the following states:

- $\text{S}^i_{\texttt{IDLE}}$: we use $\text{S}^i_{\texttt{IDLE}}(lb, rb)$ to denotes the initial state of γ_i where lb and rb are the initial balances of U_i and U_{i+1} respectively.

$$\theta_0 = \begin{cases}\text{amt}: & lb\\ \phi: & \text{Sig}(U_i)\end{cases} \quad \theta_1 = \begin{cases}\text{amt}: & rb\\ \phi: & \text{Sig}(U_{i+1})\end{cases}$$

- $\text{S}^i_{\texttt{LOCKED-Dep}}$: we use $\text{S}^i_{\texttt{LOCKED-Dep}}(\texttt{Dep}_i, T)$ to denotes the state of γ_i after the deposit is locked by U_i.

$$\theta_0 = \begin{cases}\text{amt}: & \texttt{Dep}_i\\ \phi: & \begin{cases}\text{Sig}(U_i) \wedge \text{Sig}(U_{i+1})\\ \text{Sig}(U_{i+1}) \geq T\end{cases}\end{cases}$$

$$\theta_1 = \begin{cases}\text{amt}: & lb - \texttt{Dep}_i\\ \phi: & \text{Sig}(U_i)\end{cases} \quad \theta_2 = \begin{cases}\text{amt}: & rb\\ \phi: & \text{Sig}(U_{i+1})\end{cases}$$

- $\text{S}^i_{\texttt{LOCKED-Prm}}$: we use $\text{S}^i_{\texttt{LOCKED-Prm}}(\texttt{Dep}_i, \texttt{Prm}, T)$ to denotes the state of γ_i after the premium is locked by U_{i+1} over the previous $\text{S}^i_{\texttt{LOCKED-Dep}}$ state.

$$\theta_0 = \begin{cases}\text{amt}: & \texttt{Dep}_i\\ \phi: & \begin{cases}\text{Sig}(U_i) \wedge \text{Sig}(U_{i+1})\\ \text{Sig}(U_{i+1}) \geq T\end{cases}\end{cases}$$

$$\theta_1 = \begin{cases}\text{amt}: & \texttt{Prm}_i\\ \phi: & \begin{cases}\text{Sig}(U_i) \wedge \text{Sig}(U_{i+1})\\ \text{Sig}(U_i) \geq T + t_{\text{close}} + \Delta\end{cases}\end{cases}$$

$$\theta_2 = \begin{cases}\text{amt}: & lb - \texttt{Dep}_i\\ \phi: & \text{Sig}(U_i)\end{cases} \quad \theta_3 = \begin{cases}\text{amt}: & rb - \texttt{Prm}_i\\ \phi: & \text{Sig}(U_{i+1})\end{cases}$$

- $S^i_{\texttt{LOCKED-Prcpl}}$: we use $S^i_{\texttt{LOCKED-Prcpl}}(\texttt{Dep}_i + \texttt{Prcpl}, \texttt{Prm}_i, T)$ to denotes the state of γ_i after additional principal is locked by U_i over the previous $S^i_{\texttt{LOCKED-Prm}}$ state.

$$\theta_0 = \begin{cases} \text{amt}: & \texttt{Dep}_i + \texttt{Prcpl} \\ \phi: & \begin{cases} \text{Sig}(U_i) \wedge \text{Sig}(U_{i+1}) \\ \text{Sig}(U_{i+1}) \geq T \end{cases} \end{cases}$$

$$\theta_1 = \begin{cases} \text{amt}: & \texttt{Prm}_i \\ \phi: & \begin{cases} \text{Sig}(U_i) \wedge \text{Sig}(U_{i+1}) \\ \text{Sig}(U_i) \geq T + t_{\text{close}} + \Delta \end{cases} \end{cases}$$

$$\theta_2 = \begin{cases} \text{amt}: & lb - \texttt{Dep}_i - \texttt{Prcpl} \\ \phi: & \text{Sig}(U_i) \end{cases} \quad \theta_3 = \begin{cases} \text{amt}: & rb - \texttt{Prm}_i \\ \phi: & \text{Sig}(U_{i+1}) \end{cases}$$

- $S^i_{\texttt{REFUNDED-Dep}}$: we use $S^i_{\texttt{REFUNDED-Dep}}(\texttt{Prcpl}, \texttt{Prm}_i, T)$ to denotes the state of γ_i after U_i's locked amount reduced by Dep.

$$\theta_0 = \begin{cases} \text{amt}: & \texttt{Prcpl} \\ \phi: & \begin{cases} \text{Sig}(U_i) \wedge \text{Sig}(U_{i+1}) \\ \text{Sig}(U_{i+1}) \geq T \end{cases} \end{cases}$$

$$\theta_1 = \begin{cases} \text{amt}: & \texttt{Prm}_i \\ \phi: & \begin{cases} \text{Sig}(U_i) \wedge \text{Sig}(U_{i+1}) \\ \text{Sig}(U_i) \geq T + t_{\text{close}} + \Delta \end{cases} \end{cases}$$

$$\theta_2 = \begin{cases} \text{amt}: & lb - \texttt{Prcpl} \\ \phi: & \text{Sig}(U_i) \end{cases} \quad \theta_3 = \begin{cases} \text{amt}: & rb - \texttt{Prm}_i + \texttt{Dep}_i \\ \phi: & \text{Sig}(U_{i+1}) \end{cases}$$

- $S^i_{\texttt{PAID}}$: we use $S^i_{\texttt{PAID}}(\texttt{Prcpl}, \texttt{Prm}, T)$ to denotes the state of γ_i after U_i gives the principal to U_{i+1} while U_{i+1} gives the premium to U_i.

$$\theta_0 = \begin{cases} \text{amt}: & lb - \texttt{Prcpl} + \texttt{Prm}_i \\ \phi: & \text{Sig}(U_i) \end{cases} \quad \theta_1 = \begin{cases} \text{amt}: & rb + \texttt{Prcpl} - \texttt{Prm}_i \\ \phi: & \text{Sig}(U_{i+1}) \end{cases}$$

We define two mappings, which map a channel's state to a valid transaction:

Left Revoke Transaction.

$$\text{RvkTx}_L : (S^i, tr^i_{\text{splicing}}) \mapsto tx_{\texttt{rvk-L}}$$

Given the state of γ_i (S^i) as input, function $\text{RvkTx}_L(S^i)$ returns a valid transaction $tx_{\texttt{rvk\_L}}$ which can revoke all the token (payer's deposit or principal) locked by U_i once γ_0 is spliced out in state $S^0_{\texttt{SPLICED}}$, while the UTXO tr^i_{splicing} will finialized within $\mathcal{F}_{Channels}$. This transaction takes the 0-th output of S_i and tr^i_{splicing} as inputs, along with the valid spending signatures($\text{Sig}(U_i)$), and a single output

$$\begin{cases} \text{amt}: & \text{sum of inputs' amount} - F \\ \phi: & \text{Sig}(U_i) \end{cases}$$

Right Revoke Transaction.

$$\text{RvkTx}_R : (S^i, tr^i_{\text{redeem}}) \mapsto tx_{\texttt{rvk-R}}$$

We denote $\text{RvkTx}_R(S^i)$ as the transaction to revoke the token locked by U_{i+1} if the redeem trigger tr^i_{redeem} is finalized on $\mathcal{G}_{\text{L}}$. The revoke transaction $tx_{\texttt{rvk\_R}}$ takes the 1-th output of S_i and tr^i_{redeem} as inputs, along with the valid spending signatures($\text{Sig}(U_{i+1})$), and a single output

$$\begin{cases} \text{amt}: & \text{sum of inputs' amount} - F \\ \phi: & \text{Sig}(U_{i+1}) \end{cases}$$

H.1.2 *Ideal functionality of main protocol.*

$\mathcal{F}_{Zeus}$

Parameters:
- pidSet: a set of all payment identifiers. Initially empty.
- For each pid ∈ pidSet, store:
  a) x: payment amount configured by payee.
  b) $\mathcal{M}$: The multi-hop payment metadata configured by the payer. $\mathcal{M} = (U, \Gamma, x, \text{Fees}, \mathcal{S})$, where:
     - U is the payment path, $U = (U_0, U_1, \ldots, U_n, U_{n+1})$.
     - Γ is the sequence of payment channels, $\Gamma = (\gamma_0, \ldots, \gamma_n)$, where U_i establishes channel γ_i with U_{i+1} for $i = 0$ to n.
     - Fees is the sequence of relay fees, Fees $= (f_1, \ldots, f_n)$, where f_i is the fee charged by intermediary U_i.
     - $\mathcal{S}$ is the sequence of channel states.
  c) $T, \bar{T}$: T is the principal timelock set by payer, $\bar{T}$ is the principal timelock set by payee.
  d) s: credential that payee wants to sell.
  e) h: commitment of payer-wanted credential, configured by the payer.
  f) $\texttt{status}_i$ for $i \in [0, n+1]$: status flags for each participant

Functionality:

Payee setup call

$(\texttt{sid}, \texttt{pid}, \texttt{setup}, x, \bar{T}, s) \overset{0}{\hookleftarrow} \mathcal{E}, U_{n+1}$:

(1) If pid ∈ pidSet, ignore. Otherwise:
- Add pid to pidSet.
- Store $\bar{x}, \bar{T}, s$.
- Set $\bar{h} \coloneqq \text{Commit}(s)$.
- Set $\texttt{status}_{n+1} \coloneqq \texttt{setup-complete}$.

Payer setup call

$(\texttt{sid}, \texttt{pid}, \texttt{setup}, \mathcal{M}, h, T) \overset{0}{\hookleftarrow} \mathcal{E}, U_0$:

(1) If pid ∈ pidSet, ignore. Otherwise:
- Add pid to pidSet.
- Store $\mathcal{M}, T, h$.
- Check if payee has sufficient stake:

$$\text{Out}_{\texttt{stake}} \coloneqq \text{checkPayeeStake}(U_{n+1}, T)$$

  If not abort this payment instance.
- Set $\texttt{status}_0 \coloneqq \texttt{setup-complete}$
- If U_0 is honest, generate:

$$(tx_{\text{splicing}}, tx_{\text{redeem}}, tx_{\text{slash}}) \coloneqq \text{genSplicingTx}(\gamma_0.\text{id}, \text{Out}_{\texttt{stake}}, \mathcal{M}, h)$$

Accountability request

Upon $\texttt{status}_0 = \texttt{setup-complete}$ and $\texttt{status}_{n+1} = \texttt{setup-complete}$:

(1) If U_0 is honest:
- Send $(\texttt{sid}, \texttt{pid}, \texttt{request-acc}, tx_{\text{splicing}}, tx_{\text{redeem}}, tx_{\text{slash}}) \overset{1}{\hookrightarrow} \mathcal{S}$
- Set $\texttt{status}_0 \coloneqq \texttt{acc-requested}$

(2) If U_0 is corrupted, allow $\mathcal{S}$ to provide the request:

$$(\texttt{sid}, \texttt{pid}, \texttt{set-request}, tx_{\text{splicing}}, tx_{\text{redeem}}, tx_{\text{slash}}) \overset{1}{\hookleftarrow} \mathcal{S}$$

- Store the provided transactions.
- Set $\texttt{status}_0 \coloneqq \texttt{acc-requested}$.

Accountability approval

Upon $\texttt{status}_0 = \texttt{acc-requested}$ and $\texttt{status}_{n+1} = \texttt{setup-complete}$:

(1) If U_{n+1} is honest:
- Check if $\text{checkSlashTx}(tx_{\text{splicing}}, tx_{\text{redeem}}, tx_{\text{slash}}, \bar{h}, \bar{x}) \neq \perp$
- If valid:
  - Send $(\texttt{sid}, \texttt{pid}, \texttt{approved-acc}, \sigma_{\text{slash}}) \overset{2}{\hookrightarrow} \mathcal{S}$

– Set up redeem launcher:

$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_0.id, T, tx_{\text{splicing}}.\text{Outs}[0], tx_{\text{redeem}}) \overset{1}{\hookrightarrow} \mathcal{F}_{\text{redeem-launcher}}$

– Set $\texttt{status}_{n+1} := \texttt{acc-approved}$

(2) If U_{n+1} is corrupted, allow $\mathcal{S}$ to set the approval signature:

$$(\texttt{sid}, \texttt{pid}, \texttt{set-approval}, \texttt{response}, \sigma_{\text{slash}}) \overset{2}{\hookleftarrow} \mathcal{S}$$

If provided σ_{slash} is valid:

- Store σ_{slash}
- Set $\texttt{status}_{n+1} := \texttt{acc-approved}$

Generate and distribute setup messages

Upon $\texttt{status}_0 = \texttt{acc-requested}$ and $\texttt{status}_{n+1} = \texttt{acc-approved}$:

(1) If U_0 is honest:

- Generate $m_{\text{setup}} := \text{genOnion}(\mathcal{M}, tx_{\text{splicing}}, tx_{\text{redeem}})$
- Notify the simulator the setup message is set. For each $i \in [1, n]$, send $(\texttt{sid}, \texttt{pid}, \texttt{setup-msg-sent}) \overset{2}{\hookrightarrow} \mathcal{S}$
- Set $\texttt{status}_0 := \texttt{setup-generated}$

(2) If U_0 is corrupted, allow $\mathcal{S}$ to provide setup messages:

$$(\texttt{sid}, \texttt{pid}, \texttt{set-setup-msgs}, m_{\text{setup}}) \overset{2}{\hookleftarrow} \mathcal{S}$$

- Store the provided setup messages
- Set $\texttt{status}_0 := \texttt{setup-generated}$

Splice channel

Upon $\texttt{status}_0 = \texttt{setup-generated}$:

(1) If U_0 is honest:

- Send $(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_0.id, tx_{\text{splicing}}) \overset{3}{\hookrightarrow} \mathcal{F}^0_{\text{2pSetupSplicing}}$
- Set $\texttt{status}_0 := \texttt{splicing-initiated}$

(2) If U_0 is corrupted, allow $\mathcal{S}$ to indicate if splicing occurs:

$$(\texttt{sid}, \texttt{pid}, \texttt{set-splicing}, \texttt{initiated}) \overset{3}{\hookleftarrow} \mathcal{S}$$

- Set $\texttt{status}_0 := \texttt{splicing-initiated}$

Process setup messages by intermediaries

Upon $\texttt{status}_0 = \texttt{splicing-initiated}$:

(1) For each $i \in [1, n+1]$ with $m_{\text{setup}}[i]$ available:

- If U_i is honest:
  (a) If $i = 1$, U_1 set the next state of channel γ_0 to the splicing-out state($S^0_{\texttt{SPLICED}}$):
    – Extract Dep_1, $\texttt{Prcpl}$, T, *splicing trigger* (tr^1_{splicing}).
    – Setup the splicing state of γ_0:

$$(\texttt{sid}, \texttt{pid}, \texttt{setup}, tr^1_{\text{splicing}}) \overset{3}{\hookrightarrow} \mathcal{F}_{\text{2pSetupSplicing}}$$

  (b) If $i > 1$: U_i set the next state of channel γ_{i-1} to the lock-premium state($S^{i-1}_{\texttt{LOCKED-Prm}}$):
    – Extract Dep_i, $\texttt{Prm}_{i-1}$, $\texttt{Prcpl}$, T, tr^i_{splicing}, $tr^{i-1}_{\text{splicing}}$, tr^i_{redeem}, tr^{i-1}_{redeem}.R
    – Setup the lock-deposit state of γ_{i-1} by calling $\mathcal{F}_{\text{2pSetupDep}}$:

$$(\texttt{sid}, \texttt{pid}, tr^{i-1}_{redeem}\texttt{setup}, \text{Dep}_i + c_{relay}, T, tr^i_{\text{splicing}}.tx) \overset{3}{\hookrightarrow} \mathcal{F}^{i-1}_{\text{2pSetupDep}}$$

and set $\texttt{status}_i := \texttt{Lock-Dep}$

  (c) If $i = n+1$, U_{n+1} sets the next state of channel γ_n to the lock-premium state($S^0_{\texttt{LOCKED-Prm}}$):
    – Extract $\texttt{Prm}_n$, $\texttt{Prcpl}$, T, tr^n_{redeem}.
    – If $\texttt{Prcpl} - \texttt{Prm}_n \geq \tilde{x}$ and $T = \tilde{T}$, then setup the next state of channel γ_n to state $S^n_{\texttt{Lock-Prm}}$:

$$(\texttt{sid}, \texttt{pid}, \texttt{setup}, c_{\text{payee}}, T, tx_{\text{splicing}}) \overset{3}{\hookrightarrow} \mathcal{F}^n_{\text{2pSetupDep}}$$

and set $\texttt{status}_{n+1} := \texttt{Lock-Dep}$

Complete setup phase

Upon receiving $(\texttt{sid}, \texttt{pid}, \texttt{splicing-Ok})$ from $\mathcal{F}^0_{\text{2pSetupSplicing}}$:

(1) If U_0 is honest, enable splicing and slash launchers:

- $(\texttt{sid}, \texttt{pid}, \texttt{init}, T, \gamma_0.\text{id}) \hookrightarrow \mathcal{F}^0_{\text{splicing-launcher}}$
- $(\texttt{sid}, \texttt{pid}, \texttt{init}, T, tx_{\text{splicing}}.\text{Outs}[0], tx_{\text{slash}}) \hookrightarrow \mathcal{F}_{\text{slash-launcher}}$
- Set $\texttt{status}_0 := \texttt{Refund-Dep}$

`Lock-deposit phase`

$(\texttt{sid}, \texttt{pid}, \texttt{slicing-Ok}) \overset{\tau_1}{\hookleftarrow} \mathcal{F}_{\text{2pSetupSplicing}}$, if U_1 :

U_1 honest and $\texttt{status}_1 = \texttt{Lock-Dep}$:

1) **Lock payer's deposit within γ_1.** Invoke:

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_1.\text{id}, \text{Dep}_1, tr^1_{\text{splicing}}, T) \overset{\tau_1}{\hookrightarrow} \mathcal{F}^1_{\text{2pSetupDep}}$$

2) **Enable splicing launcher and setup the next state of γ_1** If $\mathcal{F}^1_{\text{2pSetupDep}}$ returns $(\texttt{lock-Dep-Ok})$, then:

- $(\texttt{sid}, \texttt{pid}, \texttt{init}, T, \gamma_0.\text{id}) \overset{\tau'_1}{\hookrightarrow} \mathcal{F}^1_{\text{splicing-launcher}}$
- $(\texttt{sid}, \texttt{pid}, tr^1_{\text{splicing}}, \texttt{Prm}_1, T, tr^1_{\text{redeem}}) \overset{\tau'_1}{\hookrightarrow} \mathcal{F}^1_{\text{2pSetupPrm}}$

and set $\texttt{status}_1 := \texttt{Lock-Prm}$

$(\texttt{sid}, \texttt{pid}, \texttt{lock-Dep-Ok}) \overset{\tau^i_1}{\hookleftarrow} \mathcal{F}^{i-1}_{\text{2pSetupDep}}, U_{i\in[2,n]}$:

U_i honest and $\texttt{status}_i = \texttt{Lock-Dep}$:

4) **Forward the deposit lock.** Invoke:

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_i.\text{id}, \text{Dep}_i, tr^i_{\text{splicing}}, T) \overset{\tau^i_1}{\hookrightarrow} \mathcal{F}^i_{\text{2pSetupDep}}$$

5) **Setup the Premium-lock state of γ_i.** IF receive $(\texttt{lock-Dep-Ok})$ from $\mathcal{F}^i_{\text{2pSetupDep}}$, then:

$$(\texttt{sid}, \texttt{pid}, \texttt{setup}, tr^i_{\text{splicing}}, \texttt{Prm}_i, T, tr^i_{\text{redeem}}) \overset{\tau^{i'}_1}{\hookrightarrow} \mathcal{F}^i_{\text{2pSetupPrm}}$$

and Set $\texttt{status}_i := \texttt{Lock-Prm}$

$(\texttt{sid}, \texttt{pid}, \texttt{lock-Dep-Ok}) \overset{\tau_2}{\hookleftarrow} \mathcal{F}^n_{\text{2pSetupDep}}, U_{n+1}$:

U_{n+1} honest and $\texttt{status}_{n+1} = \texttt{Lock-Dep}$:

6) Set $\texttt{status}_{n+1} := \texttt{Lock-Prm}$

`Lock-premium phase`

U_{n+1} honest and $\texttt{status}_{n+1} = \texttt{Lock-Prm}$:

1) **Lock the premium.** Invoke:

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_n.id, \texttt{Prm}_n, tr^n_{redeem}, T, tx^n_{\texttt{revoke-Dep}}) \overset{\tau_2}{\hookrightarrow} \mathcal{F}^n_{\text{2pSetupPrm}}$$

2) **Enable redeem handler and prepare for principal-lock.** If $(\texttt{lock-Prm-Ok})$ returned at τ'_2, then:

$$(\texttt{sid}, \texttt{pid}, \texttt{setup}, \texttt{Prcpl}, tr^n_{\text{splicing}}, tr^n_{\text{redeem}}) \overset{\tau'_2}{\hookrightarrow} \mathcal{F}^n_{\text{2pSetupPrcpl}}$$

then set $\texttt{status}_{n+1} := \texttt{Lock-Prcpl}$

$(\texttt{sid}, \texttt{pid}, \texttt{lock-Prm-Ok}) \overset{\tau^i_2}{\hookleftarrow} \mathcal{F}^i_{\text{2pSetupPrm}}, U_{i\in[2,n]}$:

U_i honest and $\texttt{status}_i = \texttt{Lock-Prm}$:

3) **Lock premium over γ_{i-1}:**

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_{i-1}.id, \texttt{Prm}_{i-1}, tr^{i-1}_{redeem}, T, tr^{i-1}_{\text{redeem}}) \overset{\tau^i_2}{\hookrightarrow} \mathcal{F}^{i-1}_{\text{2pSetupPrm}}$$

4) **Setup the Principal-Lock state.** If $(\texttt{sid}, \texttt{pid}, \texttt{lock-Prm-Ok})$ received at $\tau^{i'}_2$:

$$(\texttt{sid}, \texttt{pid}, \texttt{setup}, \texttt{Prcpl}, tr^{i-1}_{\text{splicing}}, tr^{i-1}_{\text{redeem}}) \overset{\tau^{i'}_2}{\hookrightarrow} \mathcal{F}^{i-1}_{\text{2pSetupPrcpl}}$$

and set $\texttt{status}_i := \texttt{Lock-Prcpl}$ Otherwise, terminate.

$(\texttt{sid}, \texttt{pid}, \texttt{lock-Prm-Ok}) \overset{\tau_3}{\hookleftarrow} \mathcal{F}^1_{\text{2pSetupPrm}}, U_1$:

U_1 honest and $\texttt{status}_1 = \texttt{Lock-Prm}$:

5) Set $\texttt{status}_1 := \texttt{Lock-Prcpl}$

`Lock-principal phase`

U_1 honest and $\texttt{status}_1 = \texttt{Lock-Prcpl}$:

1) **Lock principal within γ_1.** Invoke:

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_1.\texttt{id}, \texttt{Prm},) \overset{\tau_3}{\hookrightarrow} \mathcal{F}^1_{\text{2pSetupPrcpl}}$$

2) **Setup the next phase.** If receive $(\texttt{lock-Prcpl-Ok})$ return at τ_3':

$$(\texttt{sid}, \texttt{pid}, \texttt{setup}, tr^1_{\text{splicing}}, tr^1_{\text{redeem}}, \texttt{Dep}_1) \overset{\tau_3'}{\hookrightarrow} \mathcal{F}^1_{\text{2pRfdDep}}$$

and set $\texttt{status}_1 := \texttt{Rfd-Dep}$

$(\texttt{sid}, \texttt{pid}, \texttt{lock-Prcpl-Ok}) \overset{\tau_3^i}{\hookleftarrow} \mathcal{F}^{i-1}_{\text{2pSetupPrcpl}}, U_{i\in[2,n]}$:

U_i honest and $\texttt{status}_i = \texttt{Lock-Prcpl}$:

3) **Forward the Principal lock.** Invoke:

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_i.\texttt{id}, \texttt{Prm}, tr^i_{\text{splicing}}, tr^i_{\text{redeem}}) \overset{\tau_3^i}{\hookrightarrow} \mathcal{F}^i_{\text{2pSetupPrcpl}}$$

5) **Setup the deposit-refund phase.** If it returns $(\texttt{lock-Prcpl-Ok})$ at $\tau_3^{i'}$, Then:

$$(\texttt{sid}, \texttt{pid}, \texttt{setup}, tr^i_{\text{splicing}}, tr^i_{\text{redeem}}, \texttt{Dep}_i) \overset{\tau_3^{i'}}{\hookrightarrow} \mathcal{F}^i_{\text{2pRfdDep}}$$

and set $\texttt{status}_i := \texttt{Rfd-Dep}$

$(\texttt{sid}, \texttt{pid}, \texttt{lock-Prcpl-Ok}) \overset{\tau_4}{\hookleftarrow} \mathcal{F}^n_{\text{2pSetupPrcpl}}, U_{n+1}$:

U_{n+1} honest and $\texttt{status}_{n+1} = \texttt{Lock-Prcpl}$:

6) **Leak the secret.** If U_0 is corrupted, $(\texttt{sid}, \texttt{pid}, \texttt{release-secret}, s) \overset{\tau_4}{\hookrightarrow} \mathcal{S}$, Set $\texttt{status}_{n+1} := \texttt{Rfd-Dep}$

`deposit-Refund phase`

U_{n+1} honest and $\texttt{status}_{n+1} = \texttt{Rfd-Dep}$:

1) **Refund the payer side deposit.** Invoke:

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_n.id, \texttt{Dep}_n, tr^n_{\text{splicing}}, tr^n_{\text{redeem}}) \overset{\tau_4}{\hookrightarrow} \mathcal{F}^n_{\text{2pRfdDep}}$$

2) **Setup the settle phase.**
If $(\texttt{refund-Dep-Ok})$ received from $\mathcal{F}^n_{\text{2pRfdDep}}$:

$$(\texttt{setup}, lb - \texttt{Prcpl} + \texttt{Prm}_n, rb + \texttt{Prcpl} - \texttt{Prm}_n) \overset{\tau_4'}{\hookrightarrow} \mathcal{F}^n_{\text{2pSettle}}$$

then set $\texttt{status}_{n+1} := \texttt{Settle}$.

$(\texttt{sid}, \texttt{pid}, \texttt{refund-Dep-Ok}) \overset{\tau_4^i}{\hookleftarrow} \mathcal{F}^i_{\text{2pRfdDep}}, U_{i\in[2,n]}$:

U_i honest and $\texttt{status}_i = \texttt{Rfd-Dep}$:

3) **Forward the revoke over γ_{i-1}:**

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_{i-1}.id, \texttt{Dep}_{i-1}, tr^{i-1}_{\text{splicing}}, tr^{i-1}_{\text{redeem}}) \overset{\tau_4^{i'}}{\hookrightarrow} \mathcal{F}^{i-1}_{\text{2pRfdDep}}$$

4) **Setup the settle phase.**
If $(\texttt{refund-Dep-Ok})$ received from $\mathcal{F}^{i-1}_{\text{2pRfdDep}}$:

$$(\texttt{sid}, \texttt{pid}, \texttt{setup}, lb - \texttt{Prcpl} + \texttt{Prm}_{i-1}, rb + \texttt{Prcpl} - \texttt{Prm}_{i-1}) \overset{\tau_4^{i''}}{\hookrightarrow} \mathcal{F}^{i-1}_{\text{2pSettle}}$$

then set $\texttt{status}_i := \texttt{Settle}$.

$U_1 \overset{\tau_5}{\hookleftarrow} (\texttt{refund-Dep-Ok})$ from $\mathcal{F}^1_{\text{2pRfdDep}}$:

U_1 honest and $\texttt{status}_1 = \texttt{Rfd-Dep}$:

5) **Setup the settle phase.**

$$(\texttt{sid}, \texttt{pid}, \texttt{setup}, lb - \texttt{Prcpl} + \texttt{Prm}_1 - f_1, rb + \texttt{Prcpl} - \texttt{Prm}_1 + f_1) \overset{\tau_5'}{\hookrightarrow} \mathcal{F}^0_{\text{2pSettle}}$$

6) **Notify payer.** $(\texttt{sid}, \texttt{pid}, \texttt{revoke-Dep-Ok}) \overset{\tau_5''}{\hookrightarrow} U_0$.

$U_0 \overset{\tau_6}{\hookleftarrow} (\texttt{sid}, \texttt{pid}, \texttt{revoke-Dep-Ok})$ from U_1:

7) **Check the secret.** If U_{n+1} is corrupted, check if received

$$(\texttt{sid}, \texttt{pid}, \texttt{replace-secret}, \bar{s})$$

from $\mathcal{S}$, and $\text{Open}(\bar{s}, h) = 1$, then set $\texttt{status}_0 := \texttt{Settle}$. If U_{n+1} honest, directly set $\texttt{status}_0 := \texttt{Settle}$.

`Settle phase`

U_0 honest and $\texttt{status}_0 = \texttt{Settle}$:

1) **Settle the payment.**

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_0.id, lb - x - \sum_{j=1}^{n} f_j, rb + x + \sum_{j=1}^{n} f_j) \overset{\tau_6}{\hookrightarrow} \mathcal{F}^0_{\text{2pSettle}}$$

Once $\texttt{settle-Ok} \overset{\tau_6'}{\hookleftarrow} \mathcal{F}^0_{\text{2pSettle}}$, then return

$$(\texttt{sid}, \texttt{pid}, \texttt{settled}, s) \overset{\tau_6'}{\hookrightarrow} \mathcal{E}$$

and close the splicing launcher

$$(\texttt{sid}, \texttt{pid}, \texttt{close}) \overset{\tau_6'}{\hookrightarrow} \mathcal{F}^0_{\text{splicing-launcher}}$$

ends the main protocol.

$(\texttt{sid}, \texttt{pid}, \texttt{settle-Ok}) \overset{\tau_6^i}{\hookleftarrow} \mathcal{F}^{i-1}_{\text{2pSettle}}, U_{i\in[1,n]}$:

U_i honest and $\texttt{status}_i = \texttt{Settle}$:

2) **Continue settlement chain:**

$$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_i.id, lb - \texttt{Prcpl} + \texttt{Prm}_i, rb + \texttt{Prcpl} - \texttt{Prm}_i) \overset{\tau_6^{i'}}{\hookrightarrow} \mathcal{F}^i_{\text{2pSettle}}$$

3) **Close the splicing launcher.** If $i = 1$,

$$(\texttt{sid}, \texttt{pid}, \texttt{close}) \overset{\tau_6^{i'}}{\hookrightarrow} \mathcal{F}^1_{\text{splicing-launcher}}$$

and ends the protocol.

$(\texttt{sid}, \texttt{pid}, \texttt{settle-Ok}) \overset{\tau_7}{\hookleftarrow} \mathcal{F}^n_{\text{2pSettle}}, U_{n+1}$:

U_{n+1} honest and $\texttt{status}_{n+1} = \texttt{Settle}$:

4) $(\texttt{sid}, \texttt{pid}, \texttt{settled}) \overset{\tau_7}{\hookrightarrow} \mathcal{E}$ and end.

H.1.3 Sub protocol ideal functionalities.

$\mathcal{F}_{\text{2pSetupSplicing}}$

Parameters:

- pidSet: a set of all payment identifiers.
- For each $\texttt{pid} \in \texttt{pidSet}$, store:
  a) γ_0: the channel instance γ_0 to be updated.
  b) S^0_L, S^0_R: variables for the proposed next states of γ_0.
  c) tr^l_{splicing}: the trigger proposed by the left party U_0.
  d) tr^r_{splicing}: the trigger proposed by the right party U_1.

Functionality:

Setup call.

$(\texttt{sid}, \texttt{pid}, \texttt{setup}, tr^1_{\text{splicing}}) \overset{\leq\tau}{\hookleftarrow} \mathcal{E}, U_1$:

S1) If $\texttt{pid} \in \texttt{pidSet}$, then do nothing (ignore the message). Otherwise:
- Add pid to pidSet.
- Set the right trigger: $tr^r_{\text{splicing}} := tr^1_{\text{splicing}}$.

Initialization call.

$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_0.\texttt{id}, tx_{\text{splicing}}) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_0$:

I1) If $\texttt{pid} \in \texttt{pidSet}$, ignore. Otherwise:
- Add pid to pidSet.
- Set $S^0_L := tx_{\text{splicing}}.\text{Outs}$.

- Extract splicing triggers from tx_{splicing}, set tr^{l}_{splicing} accordingly.

I2) At round τ:
- If U_0 is dishonest, the simulator may modify the left-proposed state and trigger:

$$(\texttt{sid}, \texttt{pid}, \texttt{reset-left}, \bar{S}^0, \bar{tr}^{l}_{\text{splicing}}) \overset{\tau}{\hookleftarrow} \mathcal{S}$$

If received, set $\text{S}^0_{\text{L}} := \bar{S}^0$, $tr^{l}_{\text{splicing}} := \bar{tr}^{l}_{\text{splicing}}$.

I3) At round $\tau + 1$:
- If U_1 is corrupted, then send

$$(\texttt{sid}, \texttt{pid}, \texttt{left-state}, \text{S}^0_{\text{L}}) \overset{\tau+1}{\hookleftarrow} \mathcal{S}$$

- If U_1 is honest:
  - if $tr^{l}_{\text{splicing}} \neq tr^{r}_{\text{splicing}}$ or the proposed state is invalid, send:

$$(\texttt{sid}, \texttt{pid}, \texttt{splicing-fail}) \overset{\tau+1}{\hookleftarrow} \mathcal{E}$$

  - if states match ($\bar{S} = \text{S}^0_{\text{L}} = \text{S}^0_{\text{R}}$) and triggers match ($tr^{l}_{\text{splicing}} = tr^{r}_{\text{splicing}}$):

$$(\texttt{ssid}_C, \texttt{UPDATE-SPLICING}, \gamma_0.\text{id}, \text{stateTx}(\bar{S})) \overset{\tau+1}{\hookleftarrow} \mathcal{F}_{Channels}$$

Final outputs.

I4) If U_0 is honest and receives

$$(\texttt{ssid}_C, \texttt{UPDATED-SPLICING}, \gamma_0.\text{id}, \text{S}^0_{\text{L}}) \overset{\tau'}{\hookleftarrow} \mathcal{F}_{Channels}$$

then the functionality:
- Returns $(\texttt{sid}, \texttt{pid}, \texttt{splicing-Ok}) \overset{\tau'}{\hookleftarrow} \mathcal{E}$.

I5) If U_1 is honest and receives

$$(\texttt{ssid}_C, \texttt{UPDATED-SPLICING}, \gamma_0.\text{id}, \text{S}^0_{\text{R}}) \overset{\tau'}{\hookleftarrow} \mathcal{F}_{Channels}$$

then the functionality outputs $(\texttt{sid}, \texttt{pid}, \texttt{splicing-Ok}) \overset{\tau'}{\hookleftarrow} \mathcal{E}$.

I6) If by round $\tau + t_{update} + 1$, the honest U_0 has not received confirmation, the functionality:
- Sends channel close request:

$$(\texttt{ssid}_C, \texttt{CLOSE}, \gamma_0.\text{id}) \hookrightarrow \mathcal{F}_{Channels}$$

$\mathcal{F}^{i}_{\text{2pSetupDep}}$

Parameters:
- $\texttt{pidSet}$: a set of all payment identifiers.
- For each $\texttt{pid} \in \texttt{pidSet}$, store:
  a) γ_i: the channel instance γ_i to be updated.
  b) $\text{S}^i_{\text{L}}, \text{S}^i_{\text{R}}$: variables for the proposed next states of γ_i.
  c) tr^{l}_{splicing}: the trigger proposed by the left party U_i.
  d) tr^{r}_{splicing}: the trigger proposed by the right party U_{i+1}.
  e) T: the timelock parameter.
  f) $\texttt{Dep}_i$: the amount of deposit to be locked by U_i.

Functionality:
Setup call.

$(\texttt{sid}, \texttt{pid}, \texttt{setup}, \texttt{Dep}_i, T, tr^{i}_{\text{splicing}}) \overset{\leq\tau}{\hookleftarrow} \mathcal{E}, U_{i+1}$:

S1) If $\texttt{pid} \in \texttt{pidSet}$, then do nothing (ignore the message). Otherwise:
- Add $\texttt{pid}$ to $\texttt{pidSet}$.
- Set $\text{S}^i_{\text{R}} := \text{S}^i_{\texttt{LOCKED-Dep}}(\texttt{Dep}_i, T)$.
- Set the right triggers: $tr^{r}_{\text{splicing}} := tr^{i}_{\text{splicing}}$.
- Save T and $\texttt{Dep}_i$.

Initialization call.

$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_i.\text{id}, \texttt{Dep}_i, tr^{i}_{\text{splicing}}, T) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_i$:

I1) If $\texttt{pid} \notin \texttt{pidSet}$, ignore. Otherwise:
- Set $\text{S}^i_{\text{L}} := \text{S}^i_{\texttt{LOCKED-Dep}}(\texttt{Dep}_i, T)$.
- Set the left triggers: $tr^{l}_{\text{splicing}} := tr^{i}_{\text{splicing}}$.

I2) At round τ:
- If U_i is dishonest, the simulator may modify the left-proposed state and triggers:

$$(\texttt{sid}, \texttt{pid}, \texttt{reset-left}, \bar{S}^i, \bar{tr}^{l}_{\text{splicing}}) \overset{\tau}{\hookleftarrow} \mathcal{S}$$

If received, set $\text{S}^i_{\text{L}} := \bar{S}^i$, $tr^{l}_{\text{splicing}} := \bar{tr}^{l}_{\text{splicing}}$.

I3) At round $\tau + 1$:
- If U_{i+1} is corrupted, then send

$$(\texttt{sid}, \text{S}^i_{\text{L}}, \text{RvkTx}_L(\text{S}^i_{\text{L}}, tr^{l}_{\text{splicing}})) \overset{\tau+1}{\hookleftarrow} \mathcal{S}$$

and let the simulator modify the right-proposed state:

$$(\texttt{sid}, \texttt{pid}, \texttt{reset-right}, \bar{S}^i, \bar{tr}^{r}_{\text{splicing}}) \overset{\tau+1}{\hookleftarrow} \mathcal{S}$$

If received, set $\text{S}^i_{\text{R}} := \bar{S}^i$, $tr^{r}_{\text{splicing}} := \bar{tr}^{r}_{\text{splicing}}$.

I4) At round $\tau + 2$: If states match ($\bar{S} = \text{S}^i_{\text{L}} = \text{S}^i_{\text{R}}$), triggers match ($tr^{l}_{\text{splicing}} = tr^{r}_{\text{splicing}}$)
- If U_i honest:

$$(\texttt{ssid}_C, \texttt{UPDATE}, \gamma_i.\text{id}, \text{stateTx}(\bar{S})) \overset{\tau+2}{\hookleftarrow} \mathcal{F}_{Channels}$$

- If U_i dishonest,

$$(\texttt{sid}, \texttt{pid}, \text{stateTx}(\bar{S}), \text{RvkTx}_L(\bar{S}, tr^{l}_{\text{splicing}})) \overset{\tau+2}{\hookleftarrow} \mathcal{S}$$

Final outputs.

I5) If U_i is honest and receives

$$(\texttt{ssid}_C, \texttt{UPDATED}, \gamma_i.\text{id}, \text{S}^i_{\text{L}}) \overset{\tau'}{\hookleftarrow} \mathcal{F}_{Channels}$$

then the functionality:
- Initialize splicing handler:

$$(\texttt{ssid}^i_{SH}, \texttt{init}, \gamma_i.\text{id}, T, tr^{l}_{\text{splicing}}, \text{RvkTx}_L(\text{S}^i_{\text{L}}, tr^{l}_{\text{splicing}})) \overset{\tau'}{\hookrightarrow} \mathcal{F}^{i}_{\texttt{splicing-handler}}$$

- Returns $(\texttt{sid}, \texttt{pid}, \texttt{lock-Dep-Ok}) \overset{\tau'}{\hookleftarrow} \mathcal{E}$.

I6) If U_{i+1} is honest and receives

$$(\texttt{ssid}_C, \texttt{UPDATED}, \gamma_i.\text{id}, \text{S}^i_{\text{R}}) \overset{\tau'}{\hookleftarrow} \mathcal{F}_{Channels}$$

then the functionality outputs $(\texttt{sid}, \texttt{pid}, \texttt{lock-Dep-Ok}) \overset{\tau'}{\hookleftarrow} \mathcal{E}$.

$\mathcal{F}^{i}_{\text{2pSetupPrm}}$

Parameters:
- $\texttt{pidSet}$: a set of all payment identifiers. item γ_i: the channel instance γ_i to be updated.
- For each $\texttt{pid} \in \texttt{pidSet}$, store:
  a) γ_i: the channel instance γ_i to be updated.
  b) $\text{S}^i_{\text{L}}, \text{S}^i_{\text{R}}$: variables for the proposed next states of γ_i.
  c) $tr^{l}_{\text{splicing}}, tr^{l}_{\text{redeem}}$: the trigger proposed by the left party U_i.
  d) $tr^{r}_{\text{splicing}}, tr^{r}_{\text{redeem}}$: the trigger proposed by the right party U_{i+1}.

Functionality:

Setup call.

$(\texttt{sid}, \texttt{pid}, \texttt{setup}, \gamma_i.\text{id}, tr^i_{\text{splicing}}, \texttt{Prm}_i, T, tr^i_{\text{redeem}}) \overset{\leq\tau}{\hookleftarrow} \mathcal{E}, U_i$:

S1) If $\texttt{pid} \in \texttt{pidSet}$, then do nothing (ignore the message). Otherwise:
- Add pid to pidSet.
- Set $\text{S}^i_{\text{L}} := S^i_{\texttt{Lock-Prm}}(Prm_i, T)$.
- Set the left triggers: $tr^l_{\text{splicing}} := tr^i_{\text{splicing}}$ and $tr^l_{\text{redeem}} := tr^i_{\text{redeem}}$.

Initialization call.

$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_i.\text{id}, \texttt{Prm}_i, tr^i_{\text{redeem}}, T, tr^i_{\text{splicing}}) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_{i+1}$:

I1) If $\texttt{pid} \notin \texttt{pidSet}$, ignore. Otherwise:
- Set $\text{S}^i_{\text{R}} := \text{S}^i_{\texttt{LOCKED-Prm}}(\texttt{Prm}_i, T)$.
- Set the right triggers: $tr^r_{\text{splicing}} := tr^i_{\text{splicing}}$ and $tr^r_{\text{redeem}} := tr^i_{\text{redeem}}$.

I2) At round τ:
- If U_{i+1} is dishonest, the simulator may modify the right-proposed state and triggers:

  $(\texttt{sid}, \texttt{pid}, \texttt{reset-right}, \bar{S}^i, \bar{tr}^r_{\text{splicing}}, \bar{tr}^r_{\text{redeem}}) \overset{\tau}{\hookleftarrow} \mathcal{S}$

  If received, set $\text{S}^i_{\text{R}} := \bar{S}^i$, $tr^r_{\text{splicing}} := \bar{tr}^r_{\text{splicing}}$, and $tr^r_{\text{redeem}} := \bar{tr}^r_{\text{redeem}}$.

I3) At round $\tau + 1$:
- If U_i is corrupted, then send

  $(\texttt{sid}, \text{S}^i_{\text{R}}, \text{RvkTx}_L(\text{S}^i_{\text{R}}, tr^r_{\text{splicing}}), \text{RvkTx}_R(\text{S}^i_{\text{R}}, tr^r_{\text{redeem}})) \overset{\tau+1}{\hookrightarrow} \mathcal{S}$

  and let the simulator modify the left-proposed state:

  $(\texttt{sid}, \texttt{pid}, \texttt{reset-left}, \bar{S}^i, \bar{tr}^l_{\text{splicing}}, \bar{tr}^l_{\text{redeem}}) \overset{\tau+1}{\hookleftarrow} \mathcal{S}$

  If received, set $\text{S}^i_{\text{L}} := \bar{S}^i$, $tr^l_{\text{splicing}} := \bar{tr}^l_{\text{splicing}}$, and $tr^l_{\text{redeem}} := \bar{tr}^l_{\text{redeem}}$.
- If states match ($\bar{S} = \text{S}^i_{\text{L}} = \text{S}^i_{\text{R}}$), triggers match ($tr^l_{\text{splicing}} = tr^r_{\text{splicing}}$ and $tr^l_{\text{redeem}} = tr^r_{\text{redeem}}$), and U_i honest:

$(\texttt{ssid}^i_{SH}, \texttt{pid}, \texttt{UPDATE}, \text{RvkTx}_L(\bar{S}, tr^i_{\text{splicing}})) \overset{\tau+1}{\hookrightarrow} \mathcal{F}^i_{\text{splicing-handler}}$

I3) At round $\tau + 2$: If states match ($\bar{S} = \text{S}^i_{\text{L}} = \text{S}^i_{\text{R}}$), triggers match ($tr^l_{\text{splicing}} = tr^r_{\text{splicing}}$ and $tr^l_{\text{redeem}} = tr^r_{\text{redeem}}$)
- If U_{i+1} honest:

  $(\texttt{ssid}_C, \texttt{UPDATE}, \gamma_i.\text{id}, \text{stateTx}(\bar{S})) \overset{\tau+2}{\hookrightarrow} \mathcal{F}_{Channels}$
- If U_{i+1} dishonest,

  $(\texttt{sid}, \texttt{pid}, \text{stateTx}(\bar{S}), \text{RvkTx}_L(\bar{S}), \text{RvkTx}_R(\bar{S})) \overset{\tau+2}{\hookrightarrow} \mathcal{S}$

Final outputs.

I4) If U_{i+1} is honest and receives

$(\texttt{ssid}_C, \texttt{UPDATED}, \gamma_i.\text{id}, \text{S}^i_{\text{R}}) \overset{\tau'}{\hookleftarrow} \mathcal{F}_{Channels}$

then the functionality:
- update redeem handler:

  $(\texttt{ssid}^i_{RH}, \texttt{init}, \gamma_i.\text{id}, tr^i_{\text{redeem}}, \text{TxRvk}_R(\text{S}^i_{\text{R}}, tr^i_{\text{redeem}})) \overset{\tau}{\hookrightarrow} \mathcal{F}^i_{\text{redeem-handler}}$
- Returns $(\texttt{sid}, \texttt{pid}, \texttt{lock-Prm-Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$.

I4) If U_i is honest and receives

$(\texttt{ssid}_C, \texttt{UPDATED}, \gamma_i.\text{id}, \text{S}^i_{\text{L}}) \overset{\tau'}{\hookleftarrow} \mathcal{F}_{Channels}$

then the functionality outputs $(\texttt{sid}, \texttt{pid}, \texttt{lock-Prm-Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$.

$\mathcal{F}^i_{\text{2pSetupPrcpl}}$

Parameters:
- pidSet: a set of all payment identifiers.
- For each $\texttt{pid} \in \texttt{pidSet}$, store:
  a) γ_i: the channel instance γ_i to be updated.
  b) $\text{S}^i_{\text{L}}, \text{S}^i_{\text{R}}$: variables for the proposed next states of γ_i.
  c) $tr^l_{\text{splicing}}, tr^l_{\text{redeem}}$: the trigger proposed by the left party U_i.
  d) $tr^r_{\text{splicing}}, tr^r_{\text{redeem}}$: the trigger proposed by the right party U_{i+1}.
  e) Prcpl: the amount of principal to be locked by U_i.

Functionality:
Setup call.

$(\texttt{sid}, \texttt{pid}, \texttt{setup}, \texttt{Prcpl}, tr^i_{\text{splicing}}, tr^i_{\text{redeem}}) \overset{\leq\tau}{\hookleftarrow} \mathcal{E}, U_{i+1}$:

S1) If $\texttt{pid} \in \texttt{pidSet}$, then do nothing (ignore the message). Otherwise:
- Add pid to pidSet.
- Set $\text{S}^i_{\text{R}} := \text{S}^i_{\texttt{LOCKED-Prcpl}}(\texttt{Prcpl})$.
- Set the right triggers: $tr^r_{\text{splicing}} := tr^i_{\text{splicing}}$ and $tr^r_{\text{redeem}} := tr^i_{\text{redeem}}$.
- Save Prcpl.

Initialization call.

$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_i.\text{id}, \texttt{Prcpl}, tr^i_{\text{splicing}}, tr^i_{\text{redeem}}) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_i$:

I1) If $\texttt{pid} \notin \texttt{pidSet}$, ignore. Otherwise:
- Set $\text{S}^i_{\text{L}} := \text{S}^i_{\texttt{LOCKED-Prcpl}}(\texttt{Prcpl})$.
- Set the left triggers: $tr^l_{\text{splicing}} := tr^i_{\text{splicing}}$ and $tr^l_{\text{redeem}} := tr^i_{\text{redeem}}$.

I2) At round τ:
- If U_i is dishonest, the simulator may modify the left-proposed state and triggers:

  $(\texttt{sid}, \texttt{pid}, \texttt{reset-left}, \bar{S}^i, \bar{tr}^l_{\text{splicing}}, \bar{tr}^l_{\text{redeem}}) \overset{\tau}{\hookleftarrow} \mathcal{S}$

  If received, set $\text{S}^i_{\text{L}} := \bar{S}^i$, $tr^l_{\text{splicing}} := \bar{tr}^l_{\text{splicing}}$, and $tr^l_{\text{redeem}} := \bar{tr}^l_{\text{redeem}}$.

I3) At round $\tau + 1$:
- If U_{i+1} is corrupted, then send

  $(\texttt{sid}, \text{S}^i_{\text{L}}, \text{RvkTx}_L(\text{S}^i_{\text{L}}, tr^l_{\text{splicing}}), \text{RvkTx}_R(\text{S}^i_{\text{L}}, tr^l_{\text{redeem}})) \overset{\tau+1}{\hookrightarrow} \mathcal{S}$

  and let the simulator modify the right-proposed state:

  $(\texttt{sid}, \texttt{pid}, \texttt{reset-right}, \bar{S}^i, \bar{tr}^r_{\text{splicing}}, \bar{tr}^r_{\text{redeem}}) \overset{\tau+1}{\hookleftarrow} \mathcal{S}$

  If received, set $\text{S}^i_{\text{R}} := \bar{S}^i$, $tr^r_{\text{splicing}} := \bar{tr}^r_{\text{splicing}}$, and $tr^r_{\text{redeem}} := \bar{tr}^r_{\text{redeem}}$.

I4) At round $\tau + 2$: If states match ($\bar{S} = \text{S}^i_{\text{L}} = \text{S}^i_{\text{R}}$), triggers match ($tr^l_{\text{splicing}} = tr^r_{\text{splicing}}$ and $tr^l_{\text{redeem}} = tr^r_{\text{redeem}}$)
- If U_i honest:

  $(\texttt{ssid}_C, \texttt{UPDATE}, \gamma_i.\text{id}, \text{stateTx}(\bar{S})) \overset{\tau+2}{\hookrightarrow} \mathcal{F}_{Channels}$
- If U_i dishonest,

$(\texttt{sid}, \texttt{pid}, \text{stateTx}(\bar{S}), \text{RvkTx}_L(\bar{S}, tr^l_{\text{splicing}}), \text{RvkTx}_R(\bar{S}, tr^l_{\text{redeem}})) \overset{\tau+2}{\hookrightarrow} \mathcal{S}$

Final outputs.

I5) If U_i is honest and receives

$(\texttt{ssid}_C, \texttt{UPDATED}, \gamma_i.\text{id}, \text{S}^i_{\text{L}}) \overset{\tau'}{\hookleftarrow} \mathcal{F}_{Channels}$

then the functionality:
- Update splicing handler:

$(\texttt{ssid}^i_{SH}, \texttt{UPDATE}, \text{RvkTx}_L(\text{S}^i_{\text{L}}, tr^l_{\text{splicing}})) \overset{\tau'}{\hookrightarrow} \mathcal{F}^i_{\text{splicing-handler}}$

- Returns $(\texttt{sid}, \texttt{pid}, \texttt{lock-Prcpl-Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$.

I6) If U_{i+1} is honest and receives

$$(\texttt{ssid}_C, \texttt{UPDATED}, \gamma_i.\text{id}, \text{S}_\text{R}^i) \overset{\tau'}{\hookrightarrow} \mathcal{F}_{Channels}$$

then the functionality outputs $(\texttt{sid}, \texttt{pid}, \texttt{lock-Prcpl-Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$.

$\mathcal{F}^i_{\text{2pRfdDep}}$

Parameters:
- pidSet: a set of all payment identifiers.
- For each $\texttt{pid} \in \texttt{pidSet}$, store:
  a) γ_i: the channel instance γ_i to be updated.
  b) $\text{S}_\text{L}^i, \text{S}_\text{R}^i$: variables for the proposed next states of γ_i.
  c) $tr^l_{\text{splicing}}, tr^l_{\text{redeem}}$: the trigger proposed by the left party U_i.
  d) $tr^r_{\text{splicing}}, tr^r_{\text{redeem}}$: the trigger proposed by the right party U_{i+1}.
  e) $\texttt{Dep}_i$: the amount of deposit to be refunded to U_i.

Functionality:
Setup call.

$(\texttt{sid}, \texttt{pid}, \texttt{setup}, tr^i_{\text{splicing}}, tr^i_{\text{redeem}}, \texttt{Dep}_i) \overset{\leq\tau}{\hookrightarrow} \mathcal{E}, U_i$:

S1) If $\texttt{pid} \in \texttt{pidSet}$, then do nothing (ignore the message). Otherwise:
- Add pid to pidSet.
- Set $\text{S}_\text{L}^i := \text{S}^i_{\texttt{REFUNDED-Dep}}(\texttt{Dep}_i)$.
- Set the left triggers: $tr^l_{\text{splicing}} := tr^i_{\text{splicing}}$ and $tr^l_{\text{redeem}} := tr^i_{\text{redeem}}$.
- Save $\texttt{Dep}_i$.

Initialization call.

$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_i.\text{id}, tr^i_{\text{splicing}}, tr^i_{\text{redeem}}, \texttt{Dep}_i) \overset{\tau}{\hookrightarrow} \mathcal{E}, U_{i+1}$:

I1) If $\texttt{pid} \notin \texttt{pidSet}$, ignore. Otherwise:
- Set $\text{S}_\text{R}^i := \text{S}^i_{\texttt{REFUNDED-Dep}}(\texttt{Dep}_i)$.
- Set the right triggers: $tr^r_{\text{splicing}} := tr^i_{\text{splicing}}$ and $tr^r_{\text{redeem}} := tr^i_{\text{redeem}}$.

I2) At round τ:
- If U_{i+1} is dishonest, the simulator may modify the right-proposed state and triggers:

  $(\texttt{sid}, \texttt{pid}, \texttt{reset-right}, \bar{S}^i, \bar{tr}^r_{\text{splicing}}, \bar{tr}^r_{\text{redeem}}) \overset{\tau}{\hookleftarrow} \mathcal{S}$

  If received, set $\text{S}_\text{R}^i := \bar{S}^i$, $tr^r_{\text{splicing}} := \bar{tr}^r_{\text{splicing}}$, and $tr^r_{\text{redeem}} := \bar{tr}^r_{\text{redeem}}$.

I3) At round $\tau + 1$:
- If U_i is corrupted, then send

  $(\texttt{sid}, \text{S}_\text{R}^i, \text{RvkTx}_L(\text{S}_\text{R}^i, tr^r_{\text{splicing}}), \text{RvkTx}_R(\text{S}_\text{R}^i, tr^r_{\text{redeem}})) \overset{\tau+1}{\hookrightarrow} \mathcal{S}$

  and let the simulator modify the left-proposed state:

  $(\texttt{sid}, \texttt{pid}, \texttt{reset-left}, \bar{S}^i, \bar{tr}^l_{\text{splicing}}, \bar{tr}^l_{\text{redeem}}) \overset{\tau+1}{\hookleftarrow} \mathcal{S}$

  If received, set $\text{S}_\text{L}^i := \bar{S}^i$, $tr^l_{\text{splicing}} := \bar{tr}^l_{\text{splicing}}$, and $tr^l_{\text{redeem}} := \bar{tr}^l_{\text{redeem}}$.

I4) At round $\tau + 2$: If states match ($\bar{S} = \text{S}_\text{L}^i = \text{S}_\text{R}^i$), triggers match ($tr^l_{\text{splicing}} = tr^r_{\text{splicing}}$ and $tr^l_{\text{redeem}} = tr^r_{\text{redeem}}$)
- If U_{i+1} honest:

  $(\texttt{ssid}_C, \texttt{UPDATE}, \gamma_i.\text{id}, \text{stateTx}(\bar{S})) \overset{\tau+2}{\hookrightarrow} \mathcal{F}_{Channels}$
- If U_{i+1} dishonest,

$(\texttt{sid}, \texttt{pid}, \text{stateTx}(\bar{S}), \text{RvkTx}_L(\bar{S}, tr^r_{\text{splicing}}), \text{RvkTx}_R(\bar{S}, tr^r_{\text{redeem}})) \overset{\tau+2}{\hookrightarrow} \mathcal{S}$

Final outputs.

I5) If U_i is honest and receives

$$(\texttt{ssid}_C, \texttt{UPDATED}, \gamma_i.\text{id}, \text{S}_\text{L}^i) \overset{\tau'}{\hookrightarrow} \mathcal{F}_{Channels}$$

then the functionality outputs $(\texttt{sid}, \texttt{pid}, \texttt{rfd-Dep-Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$.

I6) If U_{i+1} is honest and receives

$$(\texttt{ssid}_C, \texttt{UPDATED}, \gamma_i.\text{id}, \text{S}_\text{R}^i) \overset{\tau'}{\hookrightarrow} \mathcal{F}_{Channels}$$

then the functionality:
- Update splicing and redeem handlers:

$(\texttt{ssid}^i_{SH}, \texttt{UPDATE}, \text{RvkTx}_L(\text{S}_\text{R}^i, tr^r_{\text{splicing}})) \overset{\tau'}{\hookrightarrow} \mathcal{F}^i_{\texttt{splicing-handler}}$

$(\texttt{ssid}^i_{RH}, \texttt{UPDATE}, \text{RvkTx}_R(\text{S}_\text{R}^i, tr^r_{\text{redeem}})) \overset{\tau'}{\hookrightarrow} \mathcal{F}^i_{\texttt{redeem-handler}}$

- Returns $(\texttt{sid}, \texttt{pid}, \texttt{rfd-Dep-Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$.

$\mathcal{F}^i_{\text{2pSettle}}$

Parameters:
- pidSet: a set of all payment identifiers.
- For each $\texttt{pid} \in \texttt{pidSet}$, store:
  a) γ_i: the channel instance γ_i to be updated.
  b) lb': the new left balance for U_i.
  c) rb': the new right balance for U_{i+1}.

Functionality:
Setup call.

$(\texttt{sid}, \texttt{pid}, \texttt{setup}, lb', rb') \overset{\leq\tau}{\hookrightarrow} \mathcal{E}, U_{i+1}$:

S1) If $\texttt{pid} \in \texttt{pidSet}$, then do nothing (ignore the message). Otherwise:
- Add pid to pidSet.
- Save lb' and rb'.

Initialization call.

$(\texttt{sid}, \texttt{pid}, \texttt{init}, \gamma_i.\text{id}, lb', rb') \overset{\tau}{\hookrightarrow} \mathcal{E}, U_i$:

I1) If $\texttt{pid} \notin \texttt{pidSet}$, ignore. Otherwise:
- Set $\text{S}_\text{L}^i := \text{S}^i_{\texttt{PAID}}(lb', rb')$.

I2) At round τ:
- If U_{i+1} is honest, leak $(\texttt{sid}, \text{S}_\text{L}^i) \overset{\tau}{\hookrightarrow} \mathcal{S}$.

I3) At round $\tau + 1$:
- If U_i is honest, leak $(\texttt{sid}, \text{S}_\text{R}^i) \overset{\tau+1}{\hookrightarrow} \mathcal{S}$.

I4) At round $\tau + 2$: If states match ($\bar{S} = \text{S}_\text{L}^i = \text{S}_\text{R}^i$):
- Let $tx^i_{\texttt{settle}} := \text{stateTx}(\bar{S})$.
- Send $(\texttt{ssid}_C, \texttt{UPDATE}, \gamma_i.id, tx^i_{\texttt{settle}}) \overset{\tau+2}{\hookrightarrow} \mathcal{F}_{Channels}$.

Final outputs.

I5) If U_i is honest and receives

$$(\texttt{ssid}_C, \texttt{UPDATED}, \gamma_i.\text{id}, \text{S}_\text{L}^i) \overset{\tau'}{\hookrightarrow} \mathcal{F}_{Channels}$$

then the functionality outputs $(\texttt{sid}, \texttt{pid}, \texttt{settle-Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$.

I6) If U_{i+1} is honest and receives

$$(\texttt{ssid}_C, \texttt{UPDATED}, \gamma_i.\text{id}, \text{S}_\text{R}^i) \overset{\tau'}{\hookrightarrow} \mathcal{F}_{Channels}$$

then the functionality outputs $(\texttt{sid}, \texttt{pid}, \texttt{settle-Ok}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$.

$\mathcal{F}^i_{\text{splicing-launcher}}$

Parameters:
- pidSet: the set of all payment ids. Initially empty.

$(\mathsf{sid}, \mathsf{pid}, \mathtt{init}, \gamma_i.id, T) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_i$:

1) If $\mathsf{pid} \in \mathtt{pidSet}$, ignore. Otherwise, add pid to $\mathtt{pidSet}$.
2) Check if $U_i = \gamma_i.lu$ or $U_i = \gamma_i.ru$. If not, ignore.
3) Save T. If U_i is honest, then:
   - At round $T - t_{close}$, send:
   $$(\mathsf{ssid}, \mathtt{CLOSE}, \gamma_i.id) \xhookrightarrow{T-t_{close}} \mathcal{F}_{Channels}$$
   - If $(\mathsf{sid}_C, \mathtt{CLOSED}, \gamma_i.\mathrm{id}) \overset{\tau'}{\hookleftarrow} \mathcal{F}_{Channels}$, output $(\mathsf{sid}, \mathsf{pid}, \mathtt{splicing\text{-}launched}) \overset{\tau'}{\hookrightarrow} \mathcal{E}$.

$(\mathsf{sid}, \mathsf{pid}, \mathtt{CLOSE}) \overset{\tau^* \geq \tau}{\hookleftarrow} \mathcal{E}, U_i$:

4) Terminate the protocol if U_i honest.

$\mathcal{F}^i_{\text{redeem-launcher}}$

Parameters:
- $\mathtt{pidSet}$: the set of all payment ids. Initially empty.

$(\mathsf{sid}, \mathsf{pid}, \mathtt{init}, \gamma_i.id, T, tr, tx) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_i$:

1) If $\mathsf{pid} \in \mathtt{pidSet}$, ignore. Otherwise, add pid to $\mathtt{pidSet}$.
2) Save T, tr, tx. If U_i is honest, then:
   - At each round $\tau' \in [\tau, T + t_{close} + \Delta]$, if tr is confirmed on $\mathcal{G}_{\mathrm{L}}$, send:
   $$(\mathsf{ssid}, \mathtt{POST}, tx) \overset{\tau'}{\hookrightarrow} \mathcal{G}_{\mathrm{L}}$$
   - If tx is confirmed on $\mathcal{G}_{\mathrm{L}}$, output $(\mathsf{sid}, \mathsf{pid}, \mathtt{redeem\text{-}complete}) \hookrightarrow \mathcal{E}$.
3) Terminate this instance if U_i is honest.

$\mathcal{F}^i_{\text{slash-launcher}}$

Parameters:
- $\mathtt{pidSet}$: the set of all payment ids. Initially empty.

$(\mathsf{sid}, \mathsf{pid}, \mathtt{init}, \gamma_i.id, T, tr, tx) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_i$:

1) If $\mathsf{pid} \in \mathtt{pidSet}$, ignore. Otherwise, add pid to $\mathtt{pidSet}$.
2) Save T, tr, tx. If U_i is honest, then:
   - At round $T + t_{close} + \Delta$, if tr is an unspent transaction output in $\mathcal{G}_{\mathrm{L}}$, send:
   $$(\mathsf{ssid}, \mathtt{POST}, tx) \xhookrightarrow{T+t_{close}+\Delta} \mathcal{G}_{\mathrm{L}}$$
   - If tx fails to confirm within Δ, send an alternative transaction:
   $$(\mathsf{ssid}, \mathtt{POST}, tx') \xhookrightarrow{T+t_{close}+2\Delta} \mathcal{G}_{\mathrm{L}}$$
   - When either transaction confirms, output $(\mathsf{sid}, \mathsf{pid}, \mathtt{slash\text{-}complete}) \hookrightarrow \mathcal{E}$.
3) Terminate this instance if U_i is honest and tr not finalized when $T + t_{close} + \Delta$.

$\mathcal{F}^i_{\texttt{splicing-handler}}$

Parameters:
- $\mathtt{pidSet}$: the set of all payment ids. Initially empty.
- For each $\mathsf{pid} \in \mathtt{pidSet}$, store:
  a) γ_i: the channel instance.
  b) T: expiration time.
  c) tr: trigger UTXO.
  d) txs: set of transactions to broadcast.

$(\mathsf{sid}, \mathsf{pid}, \mathtt{init}, \gamma_i.id, T, tr, tx) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_i$:

1) If $\mathsf{pid} \in \mathtt{pidSet}$ or $U_i \neq \gamma_i.lu$, ignore. Otherwise:
   - Add pid to $\mathtt{pidSet}$.
   - Set $txs := \{tx\}$, and store T, tr.

$(\mathsf{sid}, \mathsf{pid}, \mathtt{UPDATE}, tx') \overset{\tau'}{\hookleftarrow} U_i$:

2) If $\mathsf{pid} \notin \mathtt{pidSet}$ or $U_i \neq \gamma_i.lu$, ignore. Otherwise:
   - Add tx' to txs.

If U_i is honest:

3) At each round $\tau' \in [\tau, T]$, if tr is confirmed on $\mathcal{G}_{\mathrm{L}}$:
   - Send $(\mathsf{ssid}_C, \mathtt{CLOSE}, \gamma_i.id) \overset{\tau'}{\hookrightarrow} \mathcal{F}_{Channels}$.
   - After t_{close} rounds, send $(\mathsf{ssid}_L, \mathtt{POST}, tx_{\text{valid}}) \xhookrightarrow{\tau'+t_{close}} \mathcal{G}_{\mathrm{L}}$.
   - Output $(\mathsf{sid}, \mathsf{pid}, \mathtt{splicing\text{-}handled}) \xhookrightarrow{\tau'+t_{close}} \mathcal{E}$.
4) Terminate this instance if tr is not confirmed on $\mathcal{G}_{\mathrm{L}}$ by T.

$\mathcal{F}^i_{\texttt{redeem-handler}}$

Parameters:
- $\mathtt{pidSet}$: the set of all payment ids. Initially empty.
- For each $\mathsf{pid} \in \mathtt{pidSet}$, store:
  a) γ_i: the channel instance.
  b) T: expiration time.
  c) tr: trigger UTXO.
  d) txs: set of transactions to broadcast.

$(\mathsf{sid}, \mathsf{pid}, \mathtt{init}, \gamma_i.id, T, tr, tx) \overset{\tau}{\hookleftarrow} \mathcal{E}, U_{i+1}$:

1) If $\mathsf{pid} \in \mathtt{pidSet}$ or $U_{i+1} \neq \gamma_i.ru$, ignore. Otherwise:
   - Add pid to $\mathtt{pidSet}$.
   - Set $txs := \{tx\}$, and store T, tr.

$(\mathsf{sid}, \mathsf{pid}, \mathtt{UPDATE}, tx') \overset{\tau'}{\hookleftarrow} U_{i+1}$:

2) If $\mathsf{pid} \notin \mathtt{pidSet}$ or $U_{i+1} \neq \gamma_i.ru$, ignore. Otherwise:
   - Add tx' to txs.

If U_{i+1} is honest:

3) At round $T - t_{close}$, send:
   $$(\mathsf{ssid}_C, \mathtt{CLOSE}, \gamma_i.id) \xhookrightarrow{T-t_{close}} \mathcal{F}_{Channels}$$
4) At each round $\tau' \in [\tau, T]$, if tr is confirmed on $\mathcal{G}_{\mathrm{L}}$:
   - Send $(\mathsf{ssid}_C, \mathtt{CLOSE}, \gamma_i.id) \overset{\tau'}{\hookrightarrow} \mathcal{F}_{Channels}$.
   - After t_{close} rounds, send $(\mathsf{ssid}_L, \mathtt{POST}, tx_{\text{valid}}) \xhookrightarrow{\tau'+t_{close}} \mathcal{G}_{\mathrm{L}}$.
   - Output $(\mathsf{sid}, \mathsf{pid}, \mathtt{redeem\text{-}handled}) \xhookrightarrow{\tau'+t_{close}} \mathcal{E}$.
5) Terminate this instance if tr is not confirmed on $\mathcal{G}_{\mathrm{L}}$ by T.

Here, we denotes all the sub ideal functionalities as $\mathcal{F}_{\mathsf{sub}}$.

H.2 UC proofs

In this section, we prove that our main/sub protocols GUC realizes the corresponding ideal functionalities.

H.2.1 UC proofs for the main protocol.

Lemma 2. Given EUF-CMA secure signature scheme Σ and perfectly hiding and binding commitment scheme C, the setup phase of Π_{Zeus} GUC-realizes $\mathcal{F}_{Zeus}$ in the $\{\mathcal{F}_{\mathtt{prelim}}, \mathcal{F}^0_{\text{splicing-launcher}}, \mathcal{F}_{\text{redeem-launcher}}, \mathcal{F}^0_{\text{2pSetupSplicing}}, \{\mathcal{F}^i_{\text{2pSetupPrm}}\}_{i\in[1,n]}\}$-hybrid world.

Proof. We prove that for any environment $\mathcal{E}$ and adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that $\mathcal{E}$ cannot distinguish between the real world execution with Π_{Zeus} and $\mathcal{A}$ from the ideal world execution with $\mathcal{F}_{Zeus}$ and $\mathcal{S}$. As the commitment scheme is perfectly hiding, if U_{n+1} is honest, no one can compute a $s' \neq s$, such that $\mathrm{Open}(s', h) = 1$. And by secure Σ, adversary can not forge signatures of honest parties.

Our simulator $\mathcal{S}$ observe the attacks in the real world and simulates the behavior of corrupted parties in the ideal world. We analyze all possible corruption scenarios:

Simulator for the setup phase of $\mathcal{F}_{Zeus}$

Case 1: All parties honest. $\mathcal{S}$ does nothing since the ideal functionality handles all operations correctly.

Case 2: Only payer U_0 corrupted.

- At round τ, $\mathcal{S}$ intercepts when corrupted U_0 sends $(\mathtt{sid}, \mathtt{pid}, \mathtt{setup}, \mathcal{M}, h, T)$ to the environment. $\mathcal{S}$ records these values.
- At round 1, $\mathcal{S}$ intercepts when corrupted U_0 sends $(\mathtt{sid}, \mathtt{pid}, \mathtt{request\text{-}acc}, tx_{\text{splicing}}, tx_{\text{redeem}}, tx_{\text{slash}})$ to U_{n+1} in the real world. Then the simulator $\mathcal{S}$ replay the attack to $\mathcal{F}_{Zeus}$ through the `set-request` interface:

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{set\text{-}request}, tx_{\text{splicing}}, tx_{\text{redeem}}, tx_{\text{slash}}) \overset{1}{\hookrightarrow} \mathcal{F}_{Zeus}$$

- At round 2, $\mathcal{S}$ receives the valid slash signature leaked from the ideal functionality:

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{approved\text{-}acc}, \sigma_{\text{slash}}) \overset{2}{\hookleftarrow} \mathcal{F}_{Zeus}$$

  $\mathcal{S}$ forwards relay this message in the ideal world.
- At round 2, if corrupted U_0 sends setup messages m_{setup} to intermediaries in the real world, $\mathcal{S}$ captures them and forwards to the ideal functionality:

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{set\text{-}setup\text{-}msgs}, m_{\text{setup}}) \overset{2}{\hookrightarrow} \mathcal{F}_{Zeus}$$

- At round 3, if corrupted U_0 initiates splicing by sending $(\mathtt{sid}, \mathtt{pid}, \mathtt{init}, \gamma_0.id, tx_{\text{splicing}})$ to $\mathcal{F}^0_{\text{2pSetupSplicing}}$, $\mathcal{S}$ relays this to the ideal functionality:

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{set\text{-}splicing}, \mathtt{initiated}) \overset{\tau+4}{\hookleftarrow} \mathcal{F}_{Zeus}$$

- If U_0 deviates from the protocol or aborts at any point, $\mathcal{S}$ replicates this behavior in the ideal world by not forwarding the corresponding message(s) to the ideal functionality or the U_0 in the ideal world.

Case 3: Only payee U_{n+1} corrupted.

- At round τ, $\mathcal{S}$ intercepts when corrupted U_{n+1} sends $(\mathtt{sid}, \mathtt{pid}, \mathtt{setup}, x, \bar{T}, s)$ to the environment. $\mathcal{S}$ records these values.
- At round $\tau + 1$, $\mathcal{S}$ receives from the ideal functionality:

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{request\text{-}acc}, tx_{\text{splicing}}, tx_{\text{redeem}}, tx_{\text{slash}}) \overset{\tau+1}{\hookleftarrow} \mathcal{F}_{Zeus}$$

  $\mathcal{S}$ forwards this message to corrupted U_{n+1}.
- At round $\tau + 2$, $\mathcal{S}$ observes how corrupted U_{n+1} responds:
  - If U_{n+1} verifies the transactions and sends $(\mathtt{sid}, \mathtt{pid}, \mathtt{approved\text{-}acc}, \sigma_{\text{slash}})$ with a valid signature σ_{slash}, $\mathcal{S}$ forwards to the ideal functionality:

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{set\text{-}approval}, \mathtt{approve}, \sigma_{\text{slash}}) \overset{\tau+2}{\hookrightarrow} \mathcal{F}_{Zeus}$$

  - If U_{n+1} rejects or provides an invalid signature, $\mathcal{S}$ indicates rejection:

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{set\text{-}approval}, \mathtt{reject}, \perp) \overset{\tau+2}{\hookrightarrow} \mathcal{F}_{Zeus}$$

- If corrupted U_{n+1} sets up the redeem launcher, $\mathcal{S}$ simulates this action internally since it's handled automatically by the ideal functionality for honest U_{n+1}.

Case 4: Both payer U_0 and payee U_{n+1} corrupted.

- $\mathcal{S}$ observes all messages exchanged between corrupted U_0 and U_{n+1} in the real world.
- $\mathcal{S}$ replicates the behavior by forwarding the necessary messages to the ideal functionality:
  - When U_0 sends $(\mathtt{sid}, \mathtt{pid}, \mathtt{request\text{-}acc}, tx_{\text{splicing}}, tx_{\text{redeem}}, tx_{\text{slash}})$ to U_{n+1}, $\mathcal{S}$ forwards:

    $(\mathtt{sid}, \mathtt{pid}, \mathtt{set\text{-}request}, tx_{\text{splicing}}, tx_{\text{redeem}}, tx_{\text{slash}}) \hookrightarrow \mathcal{F}_{Zeus}$
  - When U_{n+1} responds with $(\mathtt{sid}, \mathtt{pid}, \mathtt{approved\text{-}acc}, \sigma_{\text{slash}})$, $\mathcal{S}$ forwards:

    $(\mathtt{sid}, \mathtt{pid}, \mathtt{set\text{-}approval}, \mathtt{approve}, \sigma_{\text{slash}}) \hookrightarrow \mathcal{F}_{Zeus}$
  - When U_0 generates m_{setup}, $\mathcal{S}$ forwards:

    $(\mathtt{sid}, \mathtt{pid}, \mathtt{set\text{-}setup\text{-}msgs}, m_{\text{setup}}) \hookrightarrow \mathcal{F}_{Zeus}$
  - When U_0 initiates splicing, $\mathcal{S}$ indicates:

    $(\mathtt{sid}, \mathtt{pid}, \mathtt{set\text{-}splicing}, \mathtt{initiated}) \hookrightarrow \mathcal{F}_{Zeus}$
- If either party deviates from the protocol or aborts, $\mathcal{S}$ replicates this in the ideal world by not forwarding the corresponding message(s).

Case 5: Some intermediaries U_i corrupted.

- Since intermediaries don't participate in the setup phase beyond receiving setup messages, $\mathcal{S}$ only needs to handle the forwarding of these messages.
- At round 3, when setup messages are distributed, $\mathcal{S}$ receives:

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{setup\text{-}msg}, i, m_{\text{setup}}[i]) \overset{3}{\hookleftarrow} \mathcal{F}_{Zeus}$$

  for each corrupted intermediary U_i and forwards these messages to the corresponding corrupted party. The simulator $\mathcal{S}$ will also mimic the real-world interaction with other ideal functionalities in the ideal world.

Indistinguishability Analysis:

We now analyze each case to show that the environment $\mathcal{E}$ cannot distinguish between the real and ideal executions:

Case 1: All parties honest. The simulator does nothing as the ideal functionality handles all operations correctly according to the protocol specification. The execution follows exactly the same steps as the real-world protocol, so indistinguishability holds.

Case 2: Only payer U_0 corrupted. When U_0 is corrupted, the simulator $\mathcal{S}$ ensures that:

- All messages sent by corrupted U_0 in the real world are properly forwarded to the ideal functionality.
- All responses issued from honest U_{n+1} in the real world, are simulated by the leaked message from the ideal functionality in the real world.
- The transaction validation logic in the ideal world matches the real world.

From $\mathcal{E}$'s perspective, the interaction between corrupted U_0 and honest U_{n+1} proceeds identically in both worlds. The message formats, timing, and validation checks are identical. If U_0 deviates or aborts, this behavior is accurately reflected in both worlds.

Case 3: Only payee U_{n+1} corrupted. When U_{n+1} is corrupted, the simulator $\mathcal{S}$ ensures that:

- All messages sent by honest U_0 (in the real world) are properly simulated to corrupted U_{n+1} in the ideal world, using the leaked messages from the ideal functionality.
- All responses from corrupted U_{n+1} are properly translated into instructions for the ideal functionality.
- The signature validation and transaction verification logic in the ideal world matches the real world.

From $\mathcal{E}$'s perspective, the interaction between honest U_0 and corrupted U_{n+1} proceeds identically in both worlds. If U_{n+1} provides an invalid signature or rejects the accountability request, this behavior is accurately reflected in both worlds.

Case 4: Both payer U_0 and payee U_{n+1} corrupted. When both main parties are corrupted, the simulator simply replicates their behavior in the ideal world by forwarding appropriate messages to the ideal functionality. Since all messages and protocol steps are replicated exactly, the real and ideal worlds are indistinguishable to $\mathcal{E}$.
Case 5: Some intermediaries U_i corrupted. Intermediaries only receive setup messages during the setup phase. The simulator ensures that corrupted intermediaries receive exactly the same messages in both worlds. Since their role in the setup phase is limited to receiving these messages, indistinguishability holds.

Therefore, Π_{Zeus} GUC-realizes $\mathcal{F}_{Zeus}$ in the $\{\mathcal{F}_{\texttt{prelim}}, \mathcal{F}_{\texttt{sub}}\}$-hybrid world. □

Theorem 5. Given EUF-CMA secure signature scheme Σ and perfectly hiding and binding commitment scheme C, the Π_{Zeus} UC-realizes $\mathcal{F}_{Zeus}$ in the $\{\mathcal{F}_{\texttt{prelim}}, \mathcal{F}_{\texttt{sub}}\}$-hybrid world.

Proof. This is trivial to prove considering Lemma. 2 and the fact that in the rest phases of Π_{Zeus}, the parties are only interacting with outside ideal functionalities in $\mathcal{F}_{\texttt{sub}}$ and $\mathcal{F}_{\texttt{prelim}}$. The only interaction among the users is that U_{n+1} sends the secret s to U_0 at the end of the `Lock-Principal` phase, and U_0 verifies this at the end of the `Deposit-refund` phase. $\mathcal{F}_{Zeus}$ allows $\mathcal{S}$ to modify the secret if U_{n+1} is corrupted. As U_{n+1} is corrupted, $\mathcal{S}$ can directly replay the actual secret $\bar{s}$ sent from U_{n+1} to $\mathcal{F}_{Zeus}$. By the perfectly binding of C, the execute trace will be the same no matter if $\bar{s} = s$ or not. □

H.2.2 UC proofs for the sub-protocols.

Lemma 3. $\Pi^i_{\text{2pSetupSplicing}}$ GUC-realizes $\mathcal{F}^i_{\text{2pSetupSplicing}}$ in the $\mathcal{F}_{\texttt{prelim}}$ - hybrid world.

Proof. We prove that for any environment $\mathcal{E}$ and adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that $\mathcal{E}$ cannot distinguish between the real world execution with $\Pi^i_{\text{2pSetupSplicing}}$ and $\mathcal{A}$ from the ideal world execution with $\mathcal{F}^i_{\text{2pSetupSplicing}}$ and $\mathcal{S}$.

Our simulator $\mathcal{S}$ handles the communication with the ideal functionality $\mathcal{F}^i_{\text{2pSetupSplicing}}$ and simulates the behavior of the corrupted parties in the ideal world. We construct the simulator $\mathcal{S}$ for all possible corruption scenarios:

Simulator for $\mathcal{F}^i_{\text{2pSetupSplicing}}$

Case 1: Both U_0 and U_1 honest. $\mathcal{S}$ does nothing.

Case 2: Only U_1 honest, U_0 corrupted.

- At round τ, $\mathcal{S}$ intercepts the `request-splicing` messages from corrupted U_0 to U_1 in the real-world protocol:
  - The proposed splicing transaction tx_{splicing}
  - U_0's signature $\sigma^l_{\text{splicing}}$
- $\mathcal{S}$ verifies if $\sigma^l_{\text{splicing}}$ is valid. Then $\mathcal{S}$ extracts the proposed state $\tilde{S}_{left} := (tx_{\text{splicing}}.\text{Outs})$ and corresponding trigger from tx_{splicing}, and resets the left-proposed state in the ideal functionality:

$$(\texttt{sid}, \texttt{pid}, \texttt{reset-left}, \tilde{S}_{left}, \tilde{tr}^l_{\text{splicing}}) \overset{\tau}{\hookrightarrow} \mathcal{F}^i_{\text{2pSetupSplicing}}$$

- At τ', when channel updated to the new state and corrupted U_0 returns `UPDATED-SPLICING`, $\mathcal{S}$ replays this the `UPDATED-SPLICING` message in the ideal world.

Case 3: Only U_0 honest, U_1 corrupted.

- At round $\tau + 1$, $\mathcal{S}$ receives from the ideal functionality:

$$(\texttt{sid}, \texttt{pid}, \texttt{left-state}, \text{S}^0_{\text{L}}) \overset{\tau+1}{\longleftrightarrow} \mathcal{F}^i_{\text{2pSetupSplicing}}$$

  The simulator $\mathcal{S}$ extract the corresponding transaction, U_0's signature σ^l_{S} over the new state, and the trigger tr^l_{splicing} from S^0_{L}.
- In the ideal world, $\mathcal{S}$ simulate U_0's ' message to corrupted U_1:

$$(\texttt{sid}, \texttt{pid}, \texttt{request-splicing}, \gamma_0.id, tx^0_{\text{S}}, \sigma^l_{\text{S}}) \hookrightarrow U_1$$

  where $tx^0_{\text{S}}.\text{Outs} = \text{S}^0_{\text{L}}$.
- At round $\tau + 1$, $\mathcal{S}$ observes how corrupted U_1 responds in the real world:
  - If U_1 verifies the transaction and update the channel with signature $\sigma^r_{\text{splicing}}$, as U_1 is corrupted, $\mathcal{S}$ can construct this valid update message and forward to the channel ideal functionality $\mathcal{F}_{Channels}$ in the ideal world.

Case 4: Both U_0, U_1 corrupted. $\mathcal{S}$ just blocks any inputs to the ideal functionality, and directly replays all the messages observed from the real world within the ideal world.

Indistinguishability Analysis:

We now analyze each case to show that the environment $\mathcal{E}$ cannot distinguish between the real and ideal executions:
Case 1: Both U_0 and U_1 honest. The simulator does nothing, and the ideal functionality handles all the logic. The execution follows the same steps as the real-world protocol, so indistinguishability holds.
Case 2: Only U_1 honest, U_0 corrupted. When U_0 is corrupted, the simulator $\mathcal{S}$ ensures that:

- In the ideal world, the left-proposed state configured in the ideal functionality exactly matches what the corrupted U_0 proposes in the real world.
- The channel update in the ideal world happens if and only if the state is valid and would have been accepted in the real world.
- If U_0 aborts in the real world, the behavior is replicated in the ideal world.

For the honest U_1, from $\mathcal{E}$'s perspective, the ideal world's trace is indistinguishable from the real world's trace. U_1 receives the same request message, verifies it in the same way, and proceeds with the channel update if valid.
Case 3: Only U_0 honest, U_1 corrupted. When U_1 is corrupted, the simulator $\mathcal{S}$ ensures that:

- The right-proposed state in the ideal functionality reflects the corrupted U_1's response to the proposal.
- The channel update occurs in the ideal world if and only if U_1 trigger such update in the real world.
- If U_1 reports failure in the real world, the same happens in the ideal world.

For the honest U_0, its view in both worlds is identical: it sends a proposal of new state, and if U_1 approved, the channel is updated to the new proposed state.
Case 4: Both U_0 and U_1 corrupted. The simulator directly replays all messages in the ideal world, ensuring the behavior is identical to the real world.

Therefore, $\Pi^i_{\text{2pSetupSplicing}}$ GUC-realizes $\mathcal{F}^i_{\text{2pSetupSplicing}}$ in the $\mathcal{F}_{\texttt{prelim}}$-hybrid world. □

Lemma 4. $\Pi^{i}_{\text{2pSetupDep}}$ GUC-realizes $\mathcal{F}^{i}_{\text{2pSetupDep}}$ in the $\{\mathcal{F}_{\texttt{prelim}}, \mathcal{F}^{i}_{\text{splicing-handler}}\}$-hybrid world.

Proof. We prove that for any environment $\mathcal{E}$ and adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that $\mathcal{E}$ cannot distinguish between the real world execution with $\Pi^{i}_{\text{2pSetupDep}}$ and $\mathcal{A}$ from the ideal world execution with $\mathcal{F}^{i}_{\text{2pSetupDep}}$ and $\mathcal{S}$.

Our simulator $\mathcal{S}$ handles the communication with the ideal functionality $\mathcal{F}^{i}_{\text{2pSetupDep}}$ and simulates the behavior of the corrupted parties. We analyze the possible corruption scenarios:

Simulator for $\mathcal{F}^{i}_{\text{2pSetupDep}}$

Case 1: Both U_i and U_{i+1} honest. $\mathcal{S}$ does nothing.

Case 2: Only U_{i+1} honest, U_i corrupted.

- At round τ, $\mathcal{S}$ intercepts the messages from corrupted U_i to U_{i+1} in the real-world protocol:
  - The proposed deposit-lock transaction $tx^{i}_{\texttt{lock-Dep}}$
  - The associated revoke transaction $tx^{i}_{\texttt{revoke-Dep}}$
- $\mathcal{S}$ verifies if U_i provided valid signatures for the revoke transaction. Then $\mathcal{S}$ extracts the proposed state $\tilde{S}_{left} := (tx^{i}_{\texttt{lock-Dep}}.\text{Outs})$, and corresponding trigger ($\bar{tr}^{l}_{\text{splicing}} := tx^{i}_{\texttt{revoke-Dep}}.\text{Ins}[1]$), and resets the left-proposed state in the ideal functionality:
$$(\text{sid}, \text{pid}, \texttt{reset-left}, \tilde{S}_{left}, \bar{tr}^{l}_{\text{splicing}}) \overset{\tau}{\hookrightarrow} \mathcal{F}^{i}_{\text{2pSetupDep}}$$
- At round $\tau + 2$, if $\mathcal{S}$ receives:
$$(\text{sid}, \text{pid}, \text{stateTx}(\bar{S}), \text{RvkTx}_L(\bar{S}, tr^{l}_{\text{splicing}})) \overset{\tau+2}{\hookleftarrow} \mathcal{F}^{i}_{\text{2pSetupDep}}$$
then $\mathcal{S}$ saves these values for future use.
- $\mathcal{S}$ observes whether the corrupted U_i updates the channel and initializes the splicing handler in the real world:
  - If U_i sends $(\text{ssid}_C, \texttt{UPDATE}, \gamma_i.id, tx^{i}_{\texttt{lock-Dep}})$ to $\mathcal{F}_{Channels}$, then $\mathcal{S}$ simply replays the same message in the ideal world:
$$(\text{ssid}_C, \texttt{UPDATE}, \gamma_i.id, \text{stateTx}(\bar{S})) \hookrightarrow \mathcal{F}_{Channels}$$
  - If U_i initializes the splicing handler after successful channel update, $\mathcal{S}$ replays this action in the ideal world.
  - If U_i aborts or deviates, $\mathcal{S}$ replicates the same behavior in the ideal world.

Case 3: Only U_i honest, U_{i+1} corrupted.

- At round $\tau + 1$, $\mathcal{S}$ receives from the ideal functionality:
$$(\text{sid}, \text{S}^{i}_{\text{L}}, tx^{i}_{\texttt{revoke-Dep}}) \overset{\tau+1}{\hookleftarrow} \mathcal{F}^{i}_{\text{2pSetupDep}}$$
- In the ideal world, $\mathcal{S}$ simulates U_i sending the request message to corrupted U_{i+1}:
$$(\text{sid}, \text{pid}, \texttt{request-lock-Dep}, \gamma_i.id, tx^{i}_{\text{S}}, \bar{tx}^{i}_{\texttt{revoke-Dep}}) \hookrightarrow U_{i+1}$$
where $tx^{i}_{\text{S}}.\text{Outs} = \text{S}^{i}_{\text{L}}$, $\bar{tx}^{i}_{\texttt{revoke-Dep}}$ just contains U_i's signature.
- $\mathcal{S}$ observes how corrupted U_{i+1} responds at $\tau + 1$:
  - If U_{i+1} verifies the transactions and sends a valid approval message containing signatures, $\mathcal{S}$ extracts the right-proposed state and sets:
$$(\text{sid}, \text{pid}, \texttt{reset-right}, \text{S}^{i}_{\text{L}}, tr^{l}_{\text{splicing}}) \overset{\tau+1}{\hookleftarrow} \mathcal{F}^{i}_{\text{2pSetupDep}}$$
  - If U_{i+1} aborts or sends invalid signatures, $\mathcal{S}$ sets:
$$(\text{sid}, \text{pid}, \texttt{reset-right}, \bot, \bot) \overset{\tau+1}{\hookleftarrow} \mathcal{F}^{i}_{\text{2pSetupDep}}$$

Case 4: Both U_i, U_{i+1} corrupted. $\mathcal{S}$ just blocks any inputs to the ideal functionality, and directly replays all the messages observed from the real world within the ideal world.

Indistinguishability Analysis:

We now analyze each case to show that the environment $\mathcal{E}$ cannot distinguish between the real and ideal executions:

Case 1: Both U_i and U_{i+1} honest. The simulator does nothing, and the ideal functionality handles all the logic. The execution follows exactly the same steps as the real-world protocol, so indistinguishability holds.

Case 2: Only U_{i+1} honest, U_i corrupted. When U_i is corrupted, the simulator $\mathcal{S}$ ensures that:

- At round τ, the left-proposed state in the ideal functionality exactly matches what the corrupted U_i proposes in the real world.
- At round $\tau + 2$, if U_i updates the channel in the real world, $\mathcal{S}$ replicates this behavior in the ideal world.
- If U_i initializes the splicing handler in the real world, $\mathcal{S}$ ensures the same happens in the ideal world.

For the honest U_{i+1}, from $\mathcal{E}$'s perspective, the ideal world's trace is indistinguishable from the real world's trace. U_{i+1} receives the same request message, verifies it in the same way, and provides approval if the state is valid.

Case 3: Only U_i honest, U_{i+1} corrupted. When U_{i+1} is corrupted, the simulator $\mathcal{S}$ ensures that:

- At round $\tau+1$, the right-proposed state in the ideal functionality reflects the corrupted U_{i+1}'s response to the proposal.
- The channel update occurs in the ideal world if and only if U_{i+1} approves the proposal in the real world.
- The splicing handler is initialized in the ideal world if and only if the channel update succeeds in the real world.

For the honest U_i, its view in both worlds will be identical: U_i sends a request, and if approved, updates the channel and initializes the splicing handler.

Case 4: Both U_i and U_{i+1} corrupted. The simulator directly replays all messages in the ideal world, ensuring the behavior is identical to the real world.

Therefore, $\Pi^{i}_{\text{2pSetupDep}}$ GUC-realizes $\mathcal{F}^{i}_{\text{2pSetupDep}}$ in the $\{\mathcal{F}_{\texttt{prelim}}, \mathcal{F}^{i}_{\text{splicing-handler}}\}$-hybrid world. □

Lemma 5. $\Pi^{i}_{\text{2pSetupPrm}}$ GUC-realizes $\mathcal{F}^{i}_{\text{2pSetupPrm}}$ in the $\{\mathcal{F}_{\texttt{prelim}}, \mathcal{F}^{i}_{\text{splicing-handler}}, \mathcal{F}^{i}_{\text{redeem-handler}}\}$-hybrid world.

Proof. We prove that for any environment $\mathcal{E}$ and adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that $\mathcal{E}$ cannot distinguish between the real world execution with $\mathcal{F}^{i}_{\text{2pSetupPrm}}$ and $\mathcal{A}$ from the ideal world execution with $\mathcal{F}^{i}_{\text{2pSetupPrm}}$ and $\mathcal{S}$.

Our simulator $\mathcal{S}$ handles the communication with the ideal functionality $\mathcal{F}^{i}_{\text{2pSetupPrm}}$ and simulates the behavior of the corrupted parties. We analyze the three possible corruption scenarios:

Simulator for $\mathcal{F}^{i}_{\text{2pSetupPrm}}$

Case 1: Both U_i and U_{i+1} honest. $\mathcal{S}$ does nothing.

Case 2: Only U_i honest, U_{i+1} corrupted.

- At round τ, $\mathcal{S}$ intercepts the messages from corrupted U_{i+1} to U_i in the real-world protocol:

  - The proposed premium-lock transaction $tx^{i}_{\texttt{lock-Prm}}$
  - The associated revoke transactions $tx^{i}_{\texttt{revoke-Prm}}$ and $tx^{i'}_{\texttt{revoke-Dep}}$
- $\mathcal{S}$ verifies if U_{i+1} provided valid signatures for the revoke transactions. Then $\mathcal{S}$ extracts the proposed state $\tilde{S}_{right} := (tx^{i}_{\texttt{lock-Prm}}.\text{Outs})$, and corresponding triggers ($\bar{tr}^{r}_{\text{redeem}} := tx^{i}_{\texttt{revoke-Prm}}.\text{Ins}[1]$ and $\bar{tr}^{r}_{\text{splicing}} := tx^{i'}_{\texttt{revoke-Dep}}.\text{Ins}[1]$), replay in the ideal world, and resets the right-proposed state in the ideal functionality:

$$(\texttt{sid}, \texttt{pid}, \texttt{reset-right}, \tilde{S}_{right}, \bar{tr}^{r}_{\text{splicing}}, \bar{tr}^{r}_{\text{redeem}}) \overset{\tau}{\hookrightarrow} \mathcal{F}^{i}_{\text{2pSetupPrm}}$$

- At round $\tau + 2$, if $\mathcal{S}$ receives:

$$(\texttt{sid}, \texttt{pid}, \text{stateTx}(\bar{S}), \text{RvkTx}_L(\bar{S}), \text{RvkTx}_R(\bar{S})) \overset{\tau+2}{\hookleftarrow} \mathcal{F}^{i}_{\text{2pSetupPrm}}$$

then $\mathcal{S}$ saves these values for future use.
- $\mathcal{S}$ observes whether the corrupted U_{i+1} updates the channel in the real world:
  - If U_{i+1} sends $(\texttt{ssid}_C, \texttt{UPDATE}, \gamma_i.id, tx^{i}_{\texttt{lock-Prm}})$ to $\mathcal{F}_{Channels}$, then $\mathcal{S}$ simply replays the same message in the ideal world:

$$(\texttt{ssid}_C, \texttt{UPDATE}, \gamma_i.id, \text{stateTx}(\bar{S})) \hookrightarrow \mathcal{F}_{Channels}$$

  - If U_{i+1} aborts or deviates, $\mathcal{S}$ replicates the same behavior in the ideal world.
- Similarly, $\mathcal{S}$ also observes if U_{i+1} updates the redeem handler. If U_{i+1} updates the redeem handler, then $\mathcal{S}$ can replay it in the ideal world.

Case 3: Only U_{i+1} honest, U_i corrupted.

- At round $\tau + 1$, $\mathcal{S}$ receives from the ideal functionality:

$$(\texttt{sid}, \text{S}^{i}_{\text{R}}, tx^{R}_{rvk-L}, tx^{L}_{rvk-R}) \overset{\tau+1}{\hookleftarrow} \mathcal{F}^{i}_{\text{2pSetupPrm}}$$

- In the ideal world, $\mathcal{S}$ simulates U_{i+1} sending the approval message to corrupted U_i:

$$(\texttt{sid}, \texttt{pid}, \texttt{approve-Prm}, \gamma_i.id, tx^{i}_{S}, \bar{tx}^{R}_{rvk-L}, \bar{tx}^{R}_{rvk-R}) \hookrightarrow U_i$$

where $tx^{i}_{S}.\text{Outs} = \text{S}^{i}_{\text{R}}$, $\bar{tx}^{R}_{rvk-L}, \bar{tx}^{R}_{rvk-R}$ just remove U_i's signature.
- $\mathcal{S}$ observes how corrupted U_i responds at $\tau + 1$:
  - If U_i verifies the transactions and sends a valid signature $\sigma^{i,l}_{\texttt{lock-Prm}}$ for $tx^{i}_{\texttt{lock-Prm}}$, then $\mathcal{S}$ does not need to reset the left-proposed state in the ideal functionality.
  - If U_i aborts or sends invalid signatures, $\mathcal{S}$ sets:

$$(\texttt{sid}, \texttt{pid}, \texttt{reset-left}, \bot, \bot, \bot) \overset{\tau+1}{\hookleftarrow} \mathcal{F}^{i}_{\text{2pSetupPrm}}$$

- $\mathcal{S}$ also observes if U_i updates the splicing handler. If U_i , then $\mathcal{S}$ can replay this action is the ideal world, as the corresponding revoke txs are leaked from the ideal functionality.

Case 4: Both U_i, U_{i+1} corrupted. $\mathcal{S}$ just block any inputs to the ideal functionality, and directly replays all the messages observed from the real world within the ideal world.

Indistinguishability Analysis:

We now analyze each case to show that the environment $\mathcal{E}$ cannot distinguish between the real and ideal executions. It is obvious that in case 1, and case 4, such indistinguishability holds. We focus on the remaining cases.

Case 2: Only U_i honest, U_{i+1} corrupted. When U_{i+1} is corrupted, the simulator $\mathcal{S}$ ensures that:

- At $\tau + 1$, the right-proposed state and the corresponding revoke txs in the ideal functionality exactly matches what the corrupted U_{i+1} proposes sent in the `approve-lock-Prm` message in the real world.
- At $\tau+2$, the $\mathcal{S}$ will replay the `approved-lock-Prm` message sent from U_i, when the state and triggers match.
- The channel update occurs in the ideal world if and only if it occurs in the real world.

For the honest U_i, from $\mathcal{E}$'s perspective, the ideal world's trace is indistinguishable from the real world's trace. In the real world, U_i updates the splicing handler only if the proposed state and triggers match. So as the idea world's, guranteed by ideal functionality's logic.

Case 3: Only U_{i+1} honest, U_i corrupted. When U_i is corrupted, the simulator $\mathcal{S}$ ensures that:

- At round $\tau + 1$, he left-proposed state in the ideal functionality reflects the corrupted U_i's response (the `approved-lock-Prm`) to the proposal, dynamically updated by $\mathcal{S}$.
- The channel update occurs in the ideal world if and only if U_i ackowledge the U_{i+1} proposal in the real world. This is guaranteed by the ideal functionality's logic in the ideal world.
- The splicing handler is updated in the ideal world if and only if U_i updates it in the real world. And $\mathcal{S}$ has the access to the revoke txs leaked by the ideal functionality, so it can replay the same action in the ideal world.

For the honest U_{i+1}, its view in both worlds will be identical: U_{i+1} updates the channel if approval is received, and the redeem handler is initialized if the channel update succeeds.

Therefore, $\Pi^{i}_{\text{2pSetupPrm}}$ GUC-realizes $\mathcal{F}^{i}_{\text{2pSetupPrm}}$ the $\{\mathcal{F}_{\texttt{prelim}}, \mathcal{F}^{i}_{\text{splicing-handler}}, \mathcal{F}^{i}_{\text{redeem-handler}}\}$-hybrid world. □

Lemma 6. $\Pi^{i}_{\text{2pSetupPrcpl}}$ GUC-realizes $\mathcal{F}^{i}_{\text{2pSetupPrcpl}}$ in the $\{\mathcal{F}_{\texttt{prelim}}, \mathcal{F}^{i}_{\text{splicing-handler}}, \mathcal{F}^{i}_{\text{redeem-handler}}\}$-hybrid world.

Proof. We prove that for any environment $\mathcal{E}$ and adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that $\mathcal{E}$ cannot distinguish between the real world execution with $\Pi^{i}_{\text{2pSetupPrcpl}}$ and $\mathcal{A}$ from the ideal world execution with $\mathcal{F}^{i}_{\text{2pSetupPrcpl}}$ and $\mathcal{S}$.

Our simulator $\mathcal{S}$ handles the communication with the ideal functionality $\mathcal{F}^{i}_{\text{2pSetupPrcpl}}$ and simulates the behavior of the corrupted parties. We analyze the possible corruption scenarios:

Simulator for $\mathcal{F}^{i}_{\text{2pSetupPrcpl}}$

Case 1: Both U_i and U_{i+1} honest. $\mathcal{S}$ does nothing.

Case 2: Only U_{i+1} honest, U_i corrupted.

- At round τ, $\mathcal{S}$ intercepts the messages from corrupted U_i to U_{i+1} in the real-world protocol:
  - The proposed principal-lock transaction $tx^{i}_{\texttt{lock-Prcpl}}$
  - The associated revoke transactions $tx^{i}_{\texttt{revoke-Prcpl}}$ and $tx^{i'}_{\texttt{revoke-Prm}}$
- $\mathcal{S}$ verifies if U_i provided valid signatures for the revoke transactions. Then $\mathcal{S}$ extracts the proposed state $\tilde{S}_{left} := (tx^{i}_{\texttt{lock-Prcpl}}.\text{Outs})$, and corresponding triggers ($\bar{tr}^{l}_{\text{splicing}} := tx^{i}_{\texttt{revoke-Prcpl}}.\text{Ins}[1]$ and $\bar{tr}^{l}_{\text{redeem}} := tx^{i'}_{\texttt{revoke-Prm}}.\text{Ins}[1]$), and resets the left-proposed state in the ideal functionality:

$$(\texttt{sid}, \texttt{pid}, \texttt{reset-left}, \tilde{S}_{left}, \bar{tr}^{l}_{\text{splicing}}, \bar{tr}^{l}_{\text{redeem}}) \overset{\tau}{\hookrightarrow} \mathcal{F}^{i}_{\text{2pSetupPrcpl}}$$

- At round $\tau + 2$, if $\mathcal{S}$ receives:

$$(\texttt{sid}, \texttt{pid}, \text{stateTx}(\bar{S}), \text{RvkTx}_L(\bar{S}, tr^{l}_{\text{splicing}}), \text{RvkTx}_R(\bar{S}, tr^{l}_{\text{redeem}})) \overset{\tau+2}{\hookleftarrow} \mathcal{F}^{i}_{\text{2pSetupPrcpl}}$$

then $\mathcal{S}$ saves these values for future use.
- $\mathcal{S}$ observes whether the corrupted U_i updates the channel and splicing handler in the real world:
  - If U_i sends $(\mathtt{ssid}_C, \mathtt{UPDATE}, \gamma_i.id, tx^i_{\mathtt{lock\text{-}Prcpl}})$ to $\mathcal{F}_{Channels}$, then $\mathcal{S}$ simply replays the same message in the ideal world:
    $$(\mathtt{ssid}_C, \mathtt{UPDATE}, \gamma_i.id, \mathsf{stateTx}(\bar{S})) \hookrightarrow \mathcal{F}_{Channels}$$
  - If U_i updates the splicing handler after successful channel update, $\mathcal{S}$ also replays this action in the ideal world.
  - If U_i aborts or deviates, $\mathcal{S}$ replicates the same behavior in the ideal world.

Case 3: Only U_i honest, U_{i+1} corrupted.
- At round $\tau+1$, $\mathcal{S}$ receives from the ideal functionality:
  $$(\mathsf{sid}, \mathrm{S}^i_{\mathrm{L}}, tx^i_{\mathtt{rvk\text{-}L}}, tx^i_{\mathtt{rvk\text{-}R}}) \overset{\tau+1}{\hookleftarrow} \mathcal{F}^i_{\text{2pSetupPrcpl}}$$
- In the ideal world, $\mathcal{S}$ simulates U_i sending the request message to corrupted U_{i+1}:
  $$(\mathsf{sid}, \mathsf{pid}, \mathtt{request\text{-}lock\text{-}Prcpl}, \gamma_i.id, tx^i_{\mathrm{S}}, tx^i_{\mathtt{rvk\text{-}Prcpl}}, tx^{i'}_{\mathtt{rvk\text{-}Prm}}) \hookrightarrow U_{i+1}$$
  where $tx^i_{\mathrm{S}}.\mathrm{Outs} = \mathrm{S}^i_{\mathrm{L}}$, $tx^i_{\mathtt{rvk\text{-}Prcpl}}$ and $tx^{i'}_{\mathtt{rvk\text{-}Prm}}$ contain U_i's signatures.
- $\mathcal{S}$ observes how corrupted U_{i+1} responds at $\tau+1$:
  - If U_{i+1} verifies the transactions and sends a valid approval message with signatures for the revoke transactions and the principal lock transaction, $\mathcal{S}$ extracts the right-proposed state and sets:
    $$(\mathsf{sid}, \mathsf{pid}, \mathtt{reset\text{-}right}, \mathrm{S}^i_{\mathrm{L}}, tr^l_{\text{splicing}}, tr^l_{\text{redeem}}) \overset{\tau+1}{\hookleftarrow} \mathcal{F}^i_{\text{2pSetupPrcpl}}$$
  - If U_{i+1} updates the redeem handler in the real world, $\mathcal{S}$ ensures this happens in the ideal world as well.
  - If U_{i+1} aborts or sends invalid signatures, $\mathcal{S}$ sets:
    $$(\mathsf{sid}, \mathsf{pid}, \mathtt{reset\text{-}right}, \bot, \bot, \bot) \overset{\tau+1}{\hookleftarrow} \mathcal{F}^i_{\text{2pSetupPrcpl}}$$

Case 4: Both U_i, U_{i+1} corrupted. $\mathcal{S}$ just blocks any inputs to the ideal functionality and directly replays all the messages observed from the real world within the ideal world.

Indistinguishability Analysis:

We now analyze each case to show that the environment $\mathcal{E}$ cannot distinguish between the real and ideal executions:

Case 1: Both U_i and U_{i+1} honest. Similar to previous proofs, the simulator does nothing, and the ideal functionality handles all the logic. The execution follows exactly the same steps as the real-world protocol, so indistinguishability holds.

Case 2: Only U_{i+1} honest, U_i corrupted. When U_i is corrupted, the simulator $\mathcal{S}$ ensures that:

- At round τ, the left-proposed state in the ideal functionality exactly matches what the corrupted U_i proposes in the real world.
- At round $\tau+2$, if U_i updates the channel in the real world, $\mathcal{S}$ replicates this behavior in the ideal world.
- If U_i updates the splicing handler in the real world after channel update, $\mathcal{S}$ ensures the same happens in the ideal world.

For the honest U_{i+1}, from $\mathcal{E}$'s perspective, the ideal world's trace is indistinguishable from the real world's trace. U_{i+1} receives the same request message, verifies it in the same way, and provides approval if the state is valid. The revoke transactions and update of the redeem handler proceed identically in both worlds.

Case 3: Only U_i honest, U_{i+1} corrupted. Similar to the proof for $\mathcal{F}^i_{\text{2pSetupPrm}}$, when U_{i+1} is corrupted, the simulator $\mathcal{S}$ ensures that:

- At round $\tau+1$, the right-proposed state in the ideal functionality reflects the corrupted U_{i+1}'s response to the proposal.
- The channel update occurs in the ideal world if and only if U_{i+1} approves the proposal in the real world.
- The redeem handler is updated in the ideal world if and only if U_{i+1} updates it in the real world.

For the honest U_i, its view in both worlds will be identical: U_i sends a request, and if approved, updates the channel and the splicing handler.

Case 4: Both U_i and U_{i+1} corrupted. The simulator directly replays all messages in the ideal world, ensuring the behavior is identical to the real world.

Therefore, $\Pi^i_{\text{2pSetupPrcpl}}$ GUC-realizes $\mathcal{F}^i_{\text{2pSetupPrcpl}}$ in the $\{\mathcal{F}_{\mathtt{prelim}}, \mathcal{F}^i_{\text{splicing-handler}}, \mathcal{F}^i_{\text{redeem-handler}}\}$-hybrid world. □

Lemma 7. Π^i_{2pRfdDep} GUC-realizes $\mathcal{F}^i_{\text{2pRfdDep}}$ in the $\{\mathcal{F}_{\mathtt{prelim}}, \mathcal{F}^i_{\text{splicing-handler}}, \mathcal{F}^i_{\text{redeem-handler}}\}$-hybrid world.

Proof. We prove that for any environment $\mathcal{E}$ and adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that $\mathcal{E}$ cannot distinguish between the real world execution with Π^i_{2pRfdDep} and $\mathcal{A}$ from the ideal world execution with $\mathcal{F}^i_{\text{2pRfdDep}}$ and $\mathcal{S}$.

Our simulator $\mathcal{S}$ handles the communication with the ideal functionality $\mathcal{F}^i_{\text{2pRfdDep}}$ and simulates the behavior of the corrupted parties. We analyze the possible corruption scenarios:

Simulator for $\mathcal{F}^i_{\text{2pRfdDep}}$

Case 1: Both U_i and U_{i+1} honest. $\mathcal{S}$ does nothing.

Case 2: Only U_i honest, U_{i+1} corrupted.
- At round τ, $\mathcal{S}$ intercepts the messages from corrupted U_{i+1} to U_i in the real-world protocol:
  - The proposed deposit refund transaction $tx^i_{\mathtt{rfd\text{-}Dep}}$
  - The associated revoke transactions $tx^{i'}_{\mathtt{revoke\text{-}Prcpl}}$ and $tx^{i''}_{\mathtt{revoke\text{-}Prm}}$
- $\mathcal{S}$ verifies if U_{i+1} provided valid signatures for the transactions. Then $\mathcal{S}$ extracts the proposed state $\tilde{S}_{right} := (tx^i_{\mathtt{rfd\text{-}Dep}}.\mathrm{Outs})$, and corresponding triggers ($\tilde{tr}^r_{\text{splicing}} := tx^{i'}_{\mathtt{revoke\text{-}Prcpl}}.\mathrm{Ins}[1]$ and $\tilde{tr}^r_{\text{redeem}} := tx^{i''}_{\mathtt{revoke\text{-}Prm}}.\mathrm{Ins}[1]$), and resets the right-proposed state in the ideal functionality:
  $$(\mathsf{sid}, \mathsf{pid}, \mathtt{reset\text{-}right}, \tilde{S}_{right}, \tilde{tr}^r_{\text{splicing}}, \tilde{tr}^r_{\text{redeem}}) \overset{\tau}{\hookrightarrow} \mathcal{F}^i_{\text{2pRfdDep}}$$
- At round $\tau+2$, if $\mathcal{S}$ receives following from $\mathcal{F}^i_{\text{2pRfdDep}}$:
  $$(\mathsf{sid}, \mathsf{pid}, \mathsf{stateTx}(\bar{S}), \mathrm{RvkTx}_L(\bar{S}, tr^r_{\text{splicing}}), \mathrm{RvkTx}_R(\bar{S}, tr^r_{\text{redeem}}))$$
  then $\mathcal{S}$ saves these values for future use.
- $\mathcal{S}$ observes whether the corrupted U_{i+1} updates the channel and the handlers in the real world:
  - If U_{i+1} sends $(\mathtt{ssid}_C, \mathtt{UPDATE}, \gamma_i.id, tx^i_{\mathtt{rfd\text{-}Dep}})$ to $\mathcal{F}_{Channels}$, then $\mathcal{S}$ simply replays the same message in the ideal world:
    $$(\mathtt{ssid}_C, \mathtt{UPDATE}, \gamma_i.id, \mathsf{stateTx}(\bar{S})) \hookrightarrow \mathcal{F}_{Channels}$$
  - If U_{i+1} updates the splicing and redeem handlers after successful channel update, $\mathcal{S}$ also replays these actions in the ideal world.
  - If U_{i+1} aborts or deviates, $\mathcal{S}$ replicates the same behavior in the ideal world.

Case 3: Only U_{i+1} honest, U_i corrupted.

- At round $\tau+1$, $\mathcal{S}$ receives from the ideal functionality:

$$(\mathtt{sid}, \mathrm{S}_{\mathrm{R}}^{i}, tx_{\mathtt{rvk\text{-}L}}^{i}, tx_{\mathtt{rvk\text{-}R}}^{i}) \xleftarrow{\tau+1} \mathcal{F}_{\mathrm{2pRfdDep}}^{i}$$

- In the ideal world, $\mathcal{S}$ simulates U_{i+1} sending the request message to corrupted U_i:

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{request\text{-}Rfd\text{-}Dep}, \gamma_i.id, tx_{\mathtt{rfd\text{-}Dep}}^{i}, tx_{\mathtt{revoke\text{-}Prcpl}}^{i'}, tx_{\mathtt{revoke\text{-}Prm}}^{i''}) \hookrightarrow U_i$$

  where $tx_{\mathtt{rfd\text{-}Dep}}^{i}.\mathrm{Outs} = \mathrm{S}_{\mathrm{R}}^{i}$, and the revoke transactions contain U_{i+1}'s signatures.
- $\mathcal{S}$ observes how corrupted U_i responds at $\tau+1$:
  - If U_i verifies the transactions and sends a valid approval message with signature $\sigma_{\mathtt{rfd\text{-}Dep}}^{i,l}$ for the refund transaction, $\mathcal{S}$ extracts the left-proposed state and sets:

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{reset\text{-}left}, \mathrm{S}_{\mathrm{R}}^{i}, tr_{\mathrm{splicing}}^{r}, tr_{\mathrm{redeem}}^{r}) \xleftarrow{\tau+1} \mathcal{F}_{\mathrm{2pRfdDep}}^{i}$$

  - If U_i aborts or sends invalid signatures, $\mathcal{S}$ sets:

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{reset\text{-}left}, \bot, \bot, \bot) \xleftarrow{\tau+1} \mathcal{F}_{\mathrm{2pRfdDep}}^{i}$$

Case 4: Both U_i, U_{i+1} corrupted. $\mathcal{S}$ just blocks any inputs to the ideal functionality and directly replays all the messages observed from the real world within the ideal world.

Indistinguishability Analysis:

We now analyze each case to show that the environment $\mathcal{E}$ cannot distinguish between the real and ideal executions:

Case 1: Both U_i and U_{i+1} honest. Similar to previous proofs, the simulator does nothing, and the ideal functionality handles all the logic. The execution follows exactly the same steps as the real-world protocol, so indistinguishability holds.

Case 2: Only U_i honest, U_{i+1} corrupted. When U_{i+1} is corrupted, the simulator $\mathcal{S}$ ensures that:

- At round τ, the right-proposed state in the ideal functionality exactly matches what the corrupted U_{i+1} proposes in the real world.
- At round $\tau+2$, if U_{i+1} updates the channel in the real world, $\mathcal{S}$ replicates this behavior in the ideal world.
- If U_{i+1} updates the splicing and redeem handlers in the real world after channel update, $\mathcal{S}$ ensures these same actions occur in the ideal world.

For the honest U_i, from $\mathcal{E}$'s perspective, the ideal world's trace is indistinguishable from the real world's trace. U_i receives the same request message, verifies it in the same way, and provides approval if the state is valid. The transaction verification process and signature checks proceed identically in both worlds.

Case 3: Only U_{i+1} honest, U_i corrupted. Similar to previous proofs, when U_i is corrupted, the simulator $\mathcal{S}$ ensures that:

- At round $\tau+1$, the left-proposed state in the ideal functionality reflects the corrupted U_i's response to the proposal.
- The channel update occurs in the ideal world if and only if U_i approves the proposal in the real world.

For the honest U_{i+1}, its view in both worlds will be identical: U_{i+1} sends a request, and if approved, updates the channel and the splicing/redeem handlers.

Case 4: Both U_i and U_{i+1} corrupted. The simulator directly replays all messages in the ideal world, ensuring the behavior is identical to the real world.

Therefore, $\Pi_{\mathrm{2pRfdDep}}^{i}$ GUC-realizes $\mathcal{F}_{\mathrm{2pRfdDep}}^{i}$ in the $\{\mathcal{F}_{\mathtt{prelim}}, \mathcal{F}_{\mathrm{splicing\text{-}handler}}^{i}, \mathcal{F}_{\mathrm{redeem\text{-}handler}}^{i}\}$-hybrid world. □

Lemma 8. $\Pi_{\mathrm{2pSettle}}^{i}$ GUC-realizes $\mathcal{F}_{\mathrm{2pSettle}}^{i}$ in the $\mathcal{F}_{\mathtt{prelim}}$-hybrid world.

Proof. We prove that for any environment $\mathcal{E}$ and adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that $\mathcal{E}$ cannot distinguish between the real world execution with $\Pi_{\mathrm{2pSettle}}^{i}$ and $\mathcal{A}$ from the ideal world execution with $\mathcal{F}_{\mathrm{2pSettle}}^{i}$ and $\mathcal{S}$.

Our simulator $\mathcal{S}$ handles the communication with the ideal functionality $\mathcal{F}_{\mathrm{2pSettle}}^{i}$ and simulates the behavior of the corrupted parties. We analyze the possible corruption scenarios:

Simulator for $\mathcal{F}_{\mathrm{2pSettle}}^{i}$

Case 1: Both U_i and U_{i+1} honest. $\mathcal{S}$ does nothing.

Case 2: Only U_{i+1} honest, U_i corrupted.

- At round τ, $\mathcal{S}$ intercepts the messages from corrupted U_i to U_{i+1} in the real-world protocol:
  - The proposed settlement transaction $tx_{\mathtt{settle}}^{i}$
- $\mathcal{S}$ verifies that $tx_{\mathtt{settle}}^{i}$ reflects the balances specified in the setup call. Then $\mathcal{S}$ extracts the proposed state $\tilde{S}_{left} := (tx_{\mathtt{settle}}^{i}.\mathrm{Outs})$ and resets the left-proposed state in the ideal functionality:

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{reset\text{-}left}, \tilde{S}_{left}) \xhookrightarrow{\tau} \mathcal{F}_{\mathrm{2pSettle}}^{i}$$

- At round $\tau+2$, if $\mathcal{S}$ receives:

$$(\mathtt{sid}, \mathtt{pid}, \mathrm{stateTx}(\bar{S})) \xleftarrow{\tau+2} \mathcal{F}_{\mathrm{2pSettle}}^{i}$$

  then $\mathcal{S}$ checks if the corrupted U_i would have updated the channel state in the real world.
- If the state would have been updated in the real world, $\mathcal{S}$ replicates this behavior in the ideal world:

$$(\mathtt{ssid}_{\mathcal{C}}, \mathtt{UPDATE}, \gamma_i.id, \mathrm{stateTx}(\bar{S})) \hookrightarrow \mathcal{F}_{Channels}$$

- If U_i aborts or deviates in the real world, $\mathcal{S}$ replicates the same behavior in the ideal world.

Case 3: Only U_i honest, U_{i+1} corrupted.

- At round $\tau+1$, $\mathcal{S}$ receives from the ideal functionality:

$$(\mathtt{sid}, \mathrm{S}_{\mathrm{L}}^{i}) \xleftarrow{\tau+1} \mathcal{F}_{\mathrm{2pSettle}}^{i}$$

- In the ideal world, $\mathcal{S}$ simulates U_i sending the request message to corrupted U_{i+1}:

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{request\text{-}settle}, \gamma_i.id, tx_{\mathtt{settle}}^{i}) \hookrightarrow U_{i+1}$$

  where $tx_{\mathtt{settle}}^{i}.\mathrm{Outs} = \mathrm{S}_{\mathrm{L}}^{i}$.
- $\mathcal{S}$ observes how corrupted U_{i+1} responds:
  - If U_{i+1} verifies the transaction and sends a valid approval with signature $\sigma_{\mathtt{settle}}^{i,r}$, $\mathcal{S}$ sets:

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{reset\text{-}right}, \mathrm{S}_{\mathrm{L}}^{i}) \xleftarrow{\tau+1} \mathcal{F}_{\mathrm{2pSettle}}^{i}$$

  - If U_{i+1} aborts or sends invalid signatures, $\mathcal{S}$ sets:

$$(\mathtt{sid}, \mathtt{pid}, \mathtt{reset\text{-}right}, \bot) \xleftarrow{\tau+1} \mathcal{F}_{\mathrm{2pSettle}}^{i}$$

Case 4: Both U_i, U_{i+1} corrupted. $\mathcal{S}$ just blocks any inputs to the ideal functionality and directly replays all the messages observed from the real world within the ideal world.

Indistinguishability Analysis:

We now analyze each case to show that the environment $\mathcal{E}$ cannot distinguish between the real and ideal executions:
Case 1: Both U_i and U_{i+1} honest. Similar to previous proofs, the simulator does nothing, and the ideal functionality handles all the logic. The execution follows exactly the same steps as the real-world protocol, so indistinguishability holds.
Case 2: Only U_{i+1} honest, U_i corrupted. When U_i is corrupted, the simulator $\mathcal{S}$ ensures that:

- At round τ, the left-proposed state in the ideal functionality exactly matches what the corrupted U_i proposes in the real world.
- At round $\tau + 2$, if U_i updates the channel in the real world, $\mathcal{S}$ replicates this behavior in the ideal world.

For the honest U_{i+1}, from $\mathcal{E}$'s perspective, the ideal world's trace is indistinguishable from the real world's trace. U_{i+1} receives the same settlement request, verifies it in the same way, and provides approval if the state correctly reflects the agreed balances.
Case 3: Only U_i honest, U_{i+1} corrupted. Similar to previous proofs, when U_{i+1} is corrupted, the simulator $\mathcal{S}$ ensures that:

- At round $\tau+1$, the right-proposed state in the ideal functionality reflects the corrupted U_{i+1}'s response to the settlement proposal.
- The channel update occurs in the ideal world if and only if U_{i+1} approves the proposal in the real world.

For the honest U_i, its view in both worlds will be identical: U_i sends a settlement request, and if approved, updates the channel to the new balances.
Case 4: Both U_i and U_{i+1} corrupted. The simulator directly replays all messages in the ideal world, ensuring the behavior is identical to the real world.

Therefore, Π^i_{2pSettle} GUC-realizes $\mathcal{F}^i_{\text{2pSettle}}$ in the $\mathcal{F}_{\texttt{prelim}}$-hybrid world. □

Lemma 9. $\Pi^i_{\text{splicing-launcher}}$ GUC-realizes $\mathcal{F}^i_{\text{splicing-launcher}}$ in the $\mathcal{F}_{\texttt{prelim}}$-hybrid world.

Lemma 10. $\Pi^i_{\text{redeem-launcher}}$ GUC-realizes $\mathcal{F}^i_{\text{redeem-launcher}}$ in the $\mathcal{F}_{\texttt{prelim}}$-hybrid world.

Lemma 11. $\Pi^i_{\text{slash-launcher}}$ GUC-realizes $\mathcal{F}^i_{\text{slash-launcher}}$ in the $\mathcal{F}_{\texttt{prelim}}$-hybrid world.

Lemma 12. $\Pi^i_{\text{splicing-handler}}$ GUC-realizes $\mathcal{F}^i_{\text{splicing-handler}}$ in the $\mathcal{F}_{\texttt{prelim}}$-hybrid world.

Lemma 13. $\Pi^i_{\text{redeem-handler}}$ GUC-realizes $\mathcal{F}^i_{\text{redeem-handler}}$ in the $\mathcal{F}_{\texttt{prelim}}$-hybrid world.

Lemma 9, Lemma 10, Lemma 11, Lemma 12, and Lemma 13 are trivial to prove, as the protocol and corresponding ideal functionalities only consists of a single party. By the honest assumption, the real-world execution is identical to the ideal world.

For abbreviation, we denote the set of all sub protocols as $\mathcal{F}_{\texttt{sub}}$, including all the 2psetup protocols and the sub-protocols in the splicing and redeem launcher/handlers.

H.3 Proof of security and efficiency goals

In this section, we formally prove the main ideal functionality $\mathcal{F}_{Zeus}$ in the $\{\mathcal{F}_{\texttt{prelim}}, \mathcal{F}_{\texttt{sub}}\}$-hybrid model achieves the design goals. Here, we provide the formal definition of the design goals, and then prove the security of *Zeus*.

Setup. Given a multi-hop payment denoted by $\mathcal{M}_{CE}$, where $\mathcal{M}_{\text{CE}} := (\mathcal{M}, s, h)$. $\mathcal{M}$ contains the metadata for the multi-hop payment, and s is the credential required by the payer, h is the corresponding commitment.

For the metadata $\mathcal{M}$, $\mathcal{M} := (U, \Gamma, x, \text{Fees}, \mathcal{S})$ where $U = (U_0, U_1, ..., U_n, U_{n+1})$ is an ordered sequence of $n+2$ parties with payer U_0, payee U_{n+1}, and intermediate relays U_1 through U_n. $\Gamma = (\gamma_0, ..., \gamma_n)$ denotes the sequence of payment channels, where U_i establishes channel γ_i with U_{i+1}, for $i = 0$ to n. The payment amount is x, and Fees $= (f_1, ..., f_n)$ represents the sequence of relay fees where f_i is the fee charged by intermediary U_i. We denote the on-chain transaction cost as F. $\mathcal{S}$ contains all the states of channel, as defined in Appendix H.1.1. In *Zeus*, T denotes the timelock for the principal, while Δ is the time required for the on-chain transaction to be confirmed.

Assumptions. Our model follows a synchronous and authenticated communication model, with a PPT adversary $\mathcal{A}$ controlling all but explicitly-specified honest parties.

H.3.1 Balance security. Here, we give the formal definition of balance security in *Zeus*.

Definition H.1. (Balance Security) Given a multi-hop payment $\mathcal{M}_{\text{CE}}$, we say an ideal functionality $\mathcal{F}_{MHP}$ achieves *balance security*: 1) if for any PPT adversary $\mathcal{A}$, the balance of any honest party $U_{i\in[1,n+1]}$ will not decrease compared with their initial balance; and 2) if payer U_0 and payee U_{n+1} are honest, U_0 pays U_1 in the first channel γ_0 only if U_{n+1} gets paid in γ_n.

Theorem 6 (*Zeus* achieves balance security)**.** Given a multi-hop payment $\mathcal{M}_{\text{CE}}$, the main ideal functionality $\mathcal{F}_{Zeus}$ is executed in the $\{\mathcal{F}_{\texttt{prelim}}, \mathcal{F}_{\texttt{sub}}\}$-hybrid world achieves balance security.

Proof. We prove balance security by analyzing all possible channel states that an honest party might encounter during protocol execution. For each honest party, we demonstrate that their total balance across all channels will not decrease below their initial balance, regardless of adversarial behavior.

For simplicity, we assume all channels have the same initial balance (lb, rb), where lb is the left balance and rb is the right balance for any channel $\gamma_{i\in[0,n]}$. We use the notation $\gamma_i \rightarrow S^i_{\texttt{STATE}}$ to denote that channel γ_i is in state $S^i_{\texttt{STATE}}$. When the on-chain splicing-redemption happens, the relays can optimistically revokes its locked funds off-chain. So we omit the cost of revocation in our analysis.

We first shows that if payer and payee are honest, payer pays relay U_1 in channel γ_0 iff payee receives the payment x. In $\mathcal{F}_{Zeus}$, if U_0 and U_{n+1} are honest, U_0 settles γ_0 to `Paid` iff U_{n+1} ensures γ_n is in the `Refunded-Dep` state. Once the `Refunded-Dep` state times out, it is equivalent to the `Paid` state (in both states, the payee has a new balance of $rb + x$). If γ_n never enters the `Refunded-Dep` state, the honest payer U_0 will never settle γ_0 to `Paid`, and the splicing-redemption process will revoke all off-chain payments.

We then analyze balance security for the other three parties: honest first relay U_1, honest intermediate relays $U_{i\in[2,n]}$, and honest payee U_{n+1} under different cases.

Case 1: Honest first relay U_1. We analyze all possible state combinations of γ_0 and γ_1:

- **Idle states**: $\gamma_0 \to S^0_{\texttt{IDLE}}$ and $\gamma_1 \to S^1_{\texttt{IDLE}}$.
  In this case, the balance of U_1 remains unchanged at $lb + rb$.
- **Spliced first channel**: $\gamma_0 \to S^0_{\texttt{SPLICED}}$ and $\gamma_1 \to S^1_{\texttt{IDLE}}$.
  When γ_0 is in state $S^0_{\texttt{SPLICED}}$, as defined in Appendix H.1.1, U_1 still has its original balance rb. Combined with balance lb from γ_1, U_1's total balance remains $lb + rb$.
- **Spliced first channel with payer's deposit locked in second channel**: $\gamma_0 \to S^0_{\texttt{SPLICED}}$ and $\gamma_1 \to S^1_{\texttt{LOCKED-Dep}}$.
  If γ_1 settles in state $S^1_{\texttt{LOCKED-Dep}}$ after γ_0 is spliced, the ideal functionality $\mathcal{F}_{Zeus}$ ensures U_1 can revoke its locked deposit $\texttt{Dep}_1$ through transaction $\text{RvkTx}_L(S^1_{\texttt{LOCKED-Dep}}, tr^1_{splicing})$. This is guaranteed by the temporal constraints enforced by $\mathcal{F}^1_{\text{splicing-launcher}}$ and $\mathcal{F}^1_{\text{splicing-handler}}$, which ensures U_1 has sufficient time ($t_{close} + \Delta$) to perform this revocation after splicing. Thus, U_1's balance remains $lb + rb$.
- **First channel spliced while premium locked in second**: $\gamma_0 \to S^0_{\texttt{SPLICED}}$ and $\gamma_1 \to S^1_{\texttt{LOCKED-Prm}}$.
  Similar to the previous case, U_1 can revoke its locked deposit through $\mathcal{F}^1_{\text{splicing-handler}}$ while maintaining its balance of $lb + rb$.
- **First channel spliced while principal locked in secondk**: $\gamma_0 \to S^0_{\texttt{SPLICED}}$ and $\gamma_1 \to S^1_{\texttt{LOCKED-Prcpl}}$.
  Similar to the previous case, U_1 can revoke its locked deposit through $\mathcal{F}^1_{\text{splicing-handler}}$ while maintaining its balance of $lb + rb$.
- **First channel spliced while deposit refunded in second**: $\gamma_0 \to S^0_{\texttt{SPLICED}}$ and $\gamma_1 \to S^1_{\texttt{REFUNDED-Dep}}$.
  Similar to the previous case, U_1 maintains its balance of at lease $lb + rb$. If payee fails to revoke its premium in-time, U_1 will get an extra payoff of $\texttt{Prm}_1 = f_0 + f_1$ in channel γ_1.
- **Paid first channel while deposit refunded in second**: $\gamma_0 \to S^0_{\texttt{PAID}}$ and $\gamma_1 \to S^1_{\texttt{REFUNDED-Dep}}$.
  When γ_0 reaches state $S^0_{\texttt{PAID}}$, U_1's balance in γ_0 increases to $rb + x + \sum_{i=1}^{n} f_i$ by definition. In γ_1 with state $S^1_{\texttt{REFUNDED-Dep}}$, U_1's balance becomes $lb - \texttt{Prcpl} + \texttt{Prm}_1$. Substituting $\texttt{Prcpl} = x + \sum_{i=1}^{n} f_i$ and $\texttt{Prm}_1 = f_0 + f_1$, we get:

  $$lb - (x + \sum_{i=1}^{n} f_i) + (f_0 + f_1) = lb - x - \sum_{i=2}^{n} f_i$$

  Therefore, U_1's total balance becomes:

  $$(rb + x + \sum_{i=1}^{n} f_i) + (lb - x - \sum_{i=2}^{n} f_i) = lb + rb + f_1$$

  This exceeds the initial balance by f_1, achieving balance security.
- **Both channels in paid state**: $\gamma_0 \to S^0_{\texttt{PAID}}$ and $\gamma_1 \to S^1_{\texttt{PAID}}$.
  In this optimal case, U_1 securely earns its relay fee f_1, and its total balance increases to $lb + rb + f_1$.

Therefore, *Zeus* achieves balance security for honest relay U_1 in all possible channel state combinations.

Case 2: Honest intermediate relays $U_{i\in[2,n]}$. We analyze all possible state combinations of γ_{i-1} and γ_i, considering both on-chain and off-chain scenarios:

- **Idle states**: $\gamma_{i-1} \to S^{i-1}_{\texttt{IDLE}}$ and $\gamma_i \to S^i_{\texttt{IDLE}}$.
  In this baseline case, U_i maintains its initial balance of $lb + rb$.
- **Left channel with deposit locked, right channel idle**: $\gamma_{i-1} \to S^{i-1}_{\texttt{LOCKED-Dep}}$ and $\gamma_i \to S^i_{\texttt{IDLE}}$.
  This analysis has two subcases:
  - If γ_0 is not spliced before time T (optimistic case): U_i's balance in γ_{i-1} increases by $\texttt{Dep}_{i-1}$ when the timelock expires, while balance in γ_i remains lb. Total balance increases to $lb + rb + \texttt{Dep}_{i-1}$.
  - If γ_0 is spliced before T (dispute case): The splicing trigger activates $\mathcal{F}^{i-1}_{\text{splicing-handler}}$, allowing U_i to revoke any locked funds in γ_{i-1} through RvkTx_L. Balance remains $lb + rb$.
- **Both channels with deposit locked**: $\gamma_{i-1} \to S^{i-1}_{\texttt{LOCKED-Dep}}$ and $\gamma_i \to S^i_{\texttt{LOCKED-Dep}}$.
  This analysis has two subcases:
  - Without splicing: U_i's balance in γ_{i-1} increases by $\texttt{Dep}_{i-1}$ while decreasing in γ_i by $\texttt{Dep}_i$. By protocol design (as specified in $\mathcal{F}^i_{\text{2pSetupDep}}$), $\texttt{Dep}_{i-1} - \texttt{Dep}_i = c_{\text{relay}} > 0$, ensuring total balance doesn't decrease.
  - With splicing: The ideal functionality $\mathcal{F}^i_{\text{splicing-handler}}$ allows U_i to recover its locked deposit in γ_i, so balance is at least $lb + rb$. If its left relay U_{i-1} fails to revoke its locked funds in γ_{i-1}, U_i get a bonus of $\texttt{Dep}_{i-1}$ in γ_{i-1}.
- **Left channel with deposit locked, right channel with premium locked**: $\gamma_{i-1} \to S^{i-1}_{\texttt{LOCKED-Dep}}$ and $\gamma_i \to S^i_{\texttt{LOCKED-Prm}}$.
  This analysis has two subcases:
  - Without splicing: U_i gains $\texttt{Dep}_{i-1}$ in γ_{i-1} and $\texttt{Prm}_i$ in γ_i. So total balance increases to $lb + rb + c_{\text{relay}} + \texttt{Prm}_i$.
  - With splicing: U_i can revoke its deposit in γ_i through $\mathcal{F}^i_{\text{splicing-handler}}$, while maintaining balance in γ_{i-1}. If redemption doesn't occur, U_i also retains $\texttt{Prm}_i$. So the balance of U_i is at least $lb + rb$.
- **Both channels with premium locks**: $\gamma_{i-1} \to S^{i-1}_{\texttt{LOCKED-Prm}}$ and $\gamma_i \to S^i_{\texttt{LOCKED-Prm}}$.
  This analysis has three subcases:
  - Without splicing: U_i's balance changes by $(\texttt{Prm}_i - \texttt{Prm}_{i-1}) + (\texttt{Dep}_{i-1} - \texttt{Dep}_i)$. By protocol design in $\mathcal{F}^i_{\text{2pSetupPrm}}$, we have $\texttt{Prm}_i - \texttt{Prm}_{i-1} = f_i$ and $\texttt{Dep}_{i-1} - \texttt{Dep}_i = c_{\text{relay}}$, ensuring non-negative change.
  - With splicing but no redemption: U_i can recover deposit in γ_i and retain premium $\texttt{Prm}_i$, increasing balance by f_i.
  - With splicing and redemption: Both deposit and premium are revoked, maintaining original balance ($lb + rb$).

- **Left channel with principal, right with premium**: $\gamma_{i-1} \to S^{i-1}_{\texttt{LOCKED-Prcpl}}$ and $\gamma_i \to S^{i}_{\texttt{LOCKED-Prm}}$.
  This analysis has three subcases:
  - Without splicing: U_i gains additional principal `Prcpl` compared to the previous "Both channels with premium locks, no splicing"case, increasing balance further.
  - With splicing but no redemption: U_i can revoke principal in γ_i, with a balance of at least $(lb + rb + f_i)$.
  - With splicing and redemption: All locked funds are revoked, maintaining initial balance.
- **Both channels with principal locks**: $\gamma_{i-1} \to S^{i-1}_{\texttt{LOCKED-Prcpl}}$ and $\gamma_i \to S^{i}_{\texttt{LOCKED-Prcpl}}$.
  This analysis has three subcases:
  - Without splicing: `Prcpl` is identical in both channels by design, so total balance remains $(lb+rb+c_{\text{relay}}+f_i)$.
  - With splicing: U_i can revoke principal in both channels through RvkTx_L transactions facilitated by $\mathcal{F}^{i-1}_{\text{splicing-handler}}$ and $\mathcal{F}^{i}_{\text{splicing-launcher}}$, maintaining balance of $(lb + rb + f_i)$
  - With splicing and redemption: All locked funds are revoked, maintaining initial balance.
- **Left channel with principal, right with deposit refund**: $\gamma_{i-1} \to S^{i-1}_{\texttt{LOCKED-Prcpl}}$ and $\gamma_i \to S^{i}_{\texttt{REFUNDED-Dep}}$.
  This analysis has three subcases:
  - Without splicing: Compare with "Both channels with principal locks, no splicing" case, U_i has a better payoff, as its pay less in channel γ_i.
  - With splicing and no redemption: U_i can revoke principal in γ_{i-1} through RvkTx_L, while retaining the deposit refund in γ_i.
  - With splicing and redemption: All locked funds are revoked, maintaining initial balance.
- **Both channels with deposit refunds**: $\gamma_{i-1} \to S^{i-1}_{\texttt{REFUNDED-Dep}}$ and $\gamma_i \to S^{i}_{\texttt{REFUNDED-Dep}}$.
  In this state, U_i's balance in γ_{i-1} is $lb - \texttt{Prcpl} + \texttt{Prm}_{i-1}$, and in γ_i it's $rb - \texttt{Prm}_i + \texttt{Dep}_i$. By protocol design, $\texttt{Prm}_{i-1} \geq \texttt{Prcpl}$ and $\texttt{Dep}_i \geq \texttt{Prm}_i$, ensuring total balance doesn't decrease. This holds regardless of whether splicing or redemption occurs.
- **Left channel paid, right with deposit refund**: $\gamma_{i-1} \to S^{i-1}_{\texttt{PAID}}$ and $\gamma_i \to S^{i}_{\texttt{REFUNDED-Dep}}$
  When $\gamma_{i-1} \to S^{i-1}_{\texttt{PAID}}$, U_i's balance becomes $rb + \texttt{Prcpl} - \texttt{Prm}_{i-1} = rb + x + \sum_{j=i}^{n} f_j$. In γ_i, the balance is $lb - \texttt{Prcpl} + \texttt{Prm}_i = lb - x - \sum_{j=i+1}^{n} f_j$. Therefore, total balance is $lb + rb + f_i$, which exceeds the initial amount. This remains true regardless of splicing activity since γ_{i-1} has already settled.
- **Both channels paid**: $\gamma_{i-1} \to S^{i-1}_{\texttt{PAID}}$ and $\gamma_i \to S^{i}_{\texttt{PAID}}$
  In this optimal case, U_i securely earns relay fee f_i, with total balance increasing to $lb + rb + f_i$.

Therefore, *Zeus* achieves balance security for all honest relays $U_{i\in[2,n]}$ across all possible channel state combinations.

Case 3: Honest payee U_{n+1}. We analyze the payee's balance security:

- **Without splicing**: For all possible states of channel γ_n without splicing, the balance of U_{n+1} never decreases below rb. In optimal case ($\gamma_n \to S^{n}_{\texttt{PAID}}$), the balance increases to $rb + x$.
- **With splicing**: If γ_0 is spliced, the payee U_{n+1} can redeem the payment on-chain through $\mathcal{F}^{n}_{\text{redeem-launcher}}$ and obtain the payment amount x, while successfully revoking all off-chain locked funds. The timing guarantees provided by $\mathcal{F}_{Zeus}$ ensure U_{n+1} has sufficient time to complete this redemption within $t_{close} + \Delta$ after splicing.

Through this comprehensive analysis of all possible channel states and transitions, we have demonstrated that *Zeus* achieves balance security for all honest parties under any adversarial behavior. □

H.3.2 Correctness.

Definition H.2. Given a multi-hop payment $\mathcal{M}_{\text{CE}}$, we say a MHP ideal functionality $\mathcal{F}_{MHP}$ achieves *correctness* if the final state of all channels will be in the `Paid` state when all parties are honest.

Theorem 7 (*Zeus* achieves correctness)**.** Given a multi-hop payment $\mathcal{M}_{\text{CE}}$, if the ideal functionality $\mathcal{F}_{Zeus}$ is executed in the $\{\mathcal{F}_{\texttt{prelim}}, \mathcal{F}_{\texttt{sub}}\}$-hybrid model, then $\mathcal{F}_{Zeus}$ achieves correctness.

The proof of this theorem is straightforward, when every party honest, all channel will be in the `Paid` state.

H.3.3 Coin availability.

Definition H.3. Given a multi-hop payment $\mathcal{M}_{\text{CE}}$, we say a ideal functionality $\mathcal{F}_{MHP}$ achieves *coin availability* if for any corrupted PPT $\mathcal{A}$, the locked funds issued from honest U_i will not be locked forever.

Theorem 8 (*Zeus* achieves coin availability)**.** Given a multi-hop payment $\mathcal{M}_{\text{CE}}$, if the ideal functionality $\mathcal{F}_{Zeus}$ is executed in the $\{\mathcal{F}_{\texttt{prelim}}, \mathcal{F}_{\texttt{sub}}\}$-hybrid model, then $\mathcal{F}_{Zeus}$ achieves coin availability.

This theorem is straightforward, by examining all possible states listed in Appendix H.1.1 as the locked funds in any channel will be released after a limited timelock. The timelock is irrelevant with the length of payment path, so no funds will be locked forever.

H.3.4 Fee security. Here, we provide the formal definition of fee security for a MHP ideal functionality. We demonstrate that our main ideal functionality $\mathcal{F}_{Zeus}$ achieves fee security.

Definition H.4 (Fee Security)**.** Given a multi-hop payment $\mathcal{M}_{\text{CE}}$, we say an ideal functionality $\mathcal{F}_{MHP}$ satisfies fee security if for any honest relay U_i ($i \in [1, n]$), in the presence of a PPT adversary $\mathcal{A}$ controlling all other parties, the following guarantee holds: if the payee U_{n+1} receives the payment with the amount x off-chain, then U_i must receive its corresponding relay fee f_i if it has provided liquidity (locked funds) during the protocol execution.

Theorem 9 (*Zeus* achieves fee security)**.** Given a multi-hop payment $\mathcal{M}_{\text{CE}}$, if the ideal functionality $\mathcal{F}_{Zeus}$ is executed in the $\{\mathcal{F}_{\texttt{prelim}}, \mathcal{F}_{\texttt{sub}}\}$-hybrid model, then $\mathcal{F}_{Zeus}$ achieves fee security.

Proof. We analyze three distinct cases to demonstrate that *Zeus* achieves fee security as defined in Definition H.4. For each case, we show that when the payee receives the payment x off-chain,

any honest relay U_i that has provided liquidity must receive its corresponding fee f_i.

Case 1: No on-chain process occurs. For an honest relay U_i, we show that if the payee receives the payment off-chain, relay U_i will receive its corresponding relay fee f_i. Recall the proof of balance security (Appendix H.3.1), where we analyzed the balance of any honest relay U_i under all possible conditions. We observe that once U_i locks the payer's deposit off-chain, it can achieve one of the following payoff states:

- $lb + rb + c_{\text{relay}}$: This occurs when the protocol aborts during the premium-lock phase. Here, U_i has only locked the payer's deposit, with amount $\texttt{Dep}_i = c_{\text{payee}} + (n-i)c_{\text{relay}}$. As demonstrated in Appendix C.2, the value of c_{relay} is sufficient to compensate for the relay cost associated with the locked $\texttt{Dep}_i$.
- $lb+rb+c_{\text{relay}}+f_i$: This occurs when the protocol aborts during the deposit-refund phase. In this scenario, U_i receives a payoff of $c_{\text{relay}} + f_i$, which adequately covers the liquidity cost of both the locked principal and the payer's deposit.
- $lb + rb + f_i$: This represents either successful payment completion or protocol abortion during the settlement phase. In either case, relay U_i receives its designated relay fee f_i, satisfying the fee security requirement.

Case 2: Splicing transaction finalizes on-chain before timeout T. Based on our analysis in Appendix H.3.1, an honest relay U_i can achieve one of the following payoffs:

- $lb + rb$: This occurs when the off-chain protocol aborts before U_{i+1} locks the premium, and the splicing transaction is finalized. Although U_i receives no additional compensation in this scenario, the payer has chosen to settle the payment directly on-chain rather than through the relay network. Therefore, the fee security property is not violated since U_i did not contribute to the successful payment routing.
- $lb + rb + f_i$: This occurs when the premium is successfully locked in γ_i. In this case, relay U_i receives its fee f_i as required by the fee security definition.

In both subcases, the fee security guarantee is maintained: either U_i did not contribute to payment routing (no premium locked) or it receives its fee f_i.

Case 3: Splicing transaction finalizes on-chain before T, with payee redemption on-chain before $T + \Delta$. In this final scenario, if all neighbors of the honest relay U_i behave rationally, U_i may achieve a zero additional payoff (receiving only its initial balance $lb + rb$). However, this does not violate fee security because the payment is settled directly on-chain rather than through the off-chain relay path. Since Definition H.4 specifically addresses fee security when "the payee U_{n+1} receives the payment with the amount x off-chain," this on-chain settlement case falls outside the scope of the guarantee.

Across all three cases, we have demonstrated that whenever the payee receives the payment off-chain, any honest relay that has provided liquidity during protocol execution receives its corresponding fee, thereby satisfying the fee security requirement as defined in Definition H.4. □

H.3.5 Griefing resistance against cost-sensitive adversaries. In this part, we prove that our ideal functionality $\mathcal{F}_{Zeus}$ achieves griefing resistance against cost-sensitive adversaries defined in Section 2, which requires that the cost incurred by an adversary to conduct a griefing attack exceeds the financial damage inflicted on honest parties.

Theorem 10 (*Zeus* achieves griefing resistance against cost-sensitive adversaries)**.** Given a multi-hop payment $\mathcal{M}_{\text{CE}}$, if the ideal functionality $\mathcal{F}_{Zeus}$ is executed in the $\{\mathcal{F}_{\texttt{prelim}}, \mathcal{F}_{\texttt{sub}}\}$-hybrid model, then $\mathcal{F}_{Zeus}$ achieves griefing resistance against 1-cost-sensitive adversaries.

Proof. We demonstrate that $\mathcal{F}_{Zeus}$ satisfies Theorem 10 by showing that for any possible PPT adversary $\mathcal{A}$ controlling S_A, the griefing damage (liquidity cost imposed on honest parties due to locked funds) is strictly less than the griefing cost (minimum transaction costs incurred by the adversary to execute the attack).

Let us denote by F the cost of executing a standard (1 input, 1 output) Bitcoin transaction, f the relay fee charged by each party, r the time-value interest rate for locked Bitcoin, T the principal timelock duration, and Δ the confirmation time for on-chain transactions. We analyze all possible adversarial collusion patterns:

Case 1: Corrupted payer U_0. We consider two attack strategies:

- *Attack 1: Abort after premium-lock phase.* If U_0 aborts the protocol after the premium-lock phase. If the splicing revocation does not occur before time T, each honest relay $U_{i\in[1,n]}$ receive a compensation of $c_{\text{relay}} + f_i$. If splicing occurs, the adversary U_0 incurs a transaction cost of at least F. The total liquidity cost (griefing damage) for honest parties (GD_1) is:

$$r \cdot \left[\left(nc_{\text{payee}} + \frac{n(n+1)}{2}c_{\text{relay}}\right)T + \left(nf_0 + \frac{n(n+1)}{2}f\right)(T+\Delta)\right]$$

  Using the configuration parameters specified in Appendix C.2, even in the worst-case scenario with $T = 2$ hours and $\Delta = 1$ hour, $\text{GD}_1 < 0.01$ USD, which is significantly less than the adversary's griefing cost of $F \approx 0.3$ USD.
- *Attack 2: Revoke payment after reaching refunded-deposit state.* If the payer attempts to revoke an off-chain payment after all channels have reached the refunded-deposit state, the liquidity cost (GD_2) is:

$$r \cdot \left[(nx + nf_0 + n^2 f)(T) + \left(nf_0 + \frac{n(n+1)}{2}f\right)(T+\Delta)\right]$$

  For a rational payment system, the total relay fees must satisfy $nf \le F$, as otherwise direct on-chain payments would be more economical. With a standard fee rate of $fr = 0.01\%$, we can rewrite the payment amount as $x = \frac{f}{fr}$, yielding:

$$\begin{aligned}\text{GD}_2 &= r \cdot \left[\left(\frac{nf}{fr} + nf_0 + n^2 f\right)(T) + \left(nf_0 + \frac{n(n+1)}{2}f\right)(T+\Delta)\right] \\ &\le r \cdot \left[\left(\frac{F}{fr} + 2nF\right)T + (nF + (n+1)F/2)(T+\Delta)\right] \\ &\approx 0.23F < F\end{aligned}$$

  Thus, the griefing cost exceeds the griefing damage.

Case 2: Corrupted payee U_{n+1}. U_{n+1} has two griefing strategy: 1) grief multiple payer's deposit while burning its on-chain stake by aborting *Zeus* once after payer's deposit fully locked; 2) griefing the whole principal by aborting after the lock-principal phase when locked funds reach their maximum, forcing the honest payer to splice γ_0, and then redeeming it. For the first strategy, Appendix C.2 in the manuscript have shown that given a small payer's deposit, we can reduce the griefing factor lower than one. Now we focus on the second strategy, where adversary incurs a griefing cost of at least F, and the griefing damage (GD_3) is:

$$r \cdot \left[(n(x + c_{\text{payee}}) + n^2(f + c_{\text{relay}}))(T) + \left(nf_0 + \frac{n(n+1)}{2} f \right) (T + \Delta) \right]$$

Under the same assumption that $nf \leq 2F$, this griefing damage is approximately 0.25 USD, which remains less than the griefing cost of F.

Case 3: Corrupted payer U_0 and payee U_{n+1}. With both endpoints corrupted, the griefing damage remains the same as in Case 2:

$$\text{GD}_4 = \text{GD}_3$$

However, the adversary now incurs a significantly higher griefing cost of at least $2F$, as both corrupted endpoints must place on-chain transactions to execute the attack. Therefore, the griefing cost exceeds the griefing damage.

Case 4: Corrupted payer U_0 with some relays. This case is similar to Case 1, where the griefing cost remains F, but the griefing damage decreases since liquidity provided by corrupted relays is excluded from the damage calculation. Thus, the griefing cost exceeds the griefing damage.

Case 5: Corrupted payee U_{n+1} with some relays. Similar to Case 2, the griefing cost is F, but the griefing damage decreases due to the exclusion of corrupted relays' liquidity from the damage calculation. Therefore, the griefing cost exceeds the griefing damage.

Case 6: Corrupted payer U_0, payee U_{n+1}, and some relays. This case combines the characteristics of Case 3 and Cases 4-5. The griefing cost remains at least $2F$ due to the necessary on-chain transactions, while the griefing damage is reduced compared to Case 3. Therefore, the griefing cost strictly exceeds the griefing damage.

For all possible adversarial coalitions defined in Section 2, we have demonstrated that the griefing cost consistently exceeds the griefing damage. Therefore, the ideal functionality $\mathcal{F}_{Zeus}$ achieves the griefing resistance property. □

H.3.6 Constant collateral.

Definition H.5. Given a multi-hop payment $\mathcal{M}_{\text{CE}}$, we say an ideal functionality $\mathcal{F}_{MHP}$ achieves *constant collateral* if the locked funds in any channel $\gamma_{i \in [0,n]}$ have constant timelocks, independent of the payment path length n.

Theorem 11 (*Zeus* achieves constant collateral)**.** Given a multi-hop payment $\mathcal{M}_{\text{CE}}$, if the ideal functionality $\mathcal{F}_{Zeus}$ is executed in the $\{\mathcal{F}_{\texttt{prelim}}, \mathcal{F}_{\texttt{sub}}\}$-hybrid model, then $\mathcal{F}_{Zeus}$ achieves constant collateral.

Similar to the coin availability, the theorem is straightforward, ranging all funds locked in any possible state, the timelock is bounded and irrelevant with the length of payment path, so no funds will be locked forever.

H.3.7 Credential exchange support.

Definition H.6 (Credential exchange support)**.** Given a multi-hop payment $\mathcal{M}_{\text{CE}} := \mathcal{M}, s, h$, we say an ideal functionality $\mathcal{F}_{Zeus}$ achieves *credential exchange* if:

- Honest payer U_0 pays x iff it receives credential s from payee U_{n+1}.
- Honest payee U_{n+1} gives credential s iff it receives payment x from payer U_0.

Theorem 12 (*Zeus* achieves credential exchange)**.** Given a multi-hop payment $\mathcal{M}_{\text{CE}}$, if the ideal functionality $\mathcal{F}_{Zeus}$ is executed in the $\{\mathcal{F}_{\texttt{prelim}}, \mathcal{F}_{\texttt{sub}}\}$-hybrid model, then $\mathcal{F}_{Zeus}$ achieves credential exchange.

Proof. We prove credential exchange by analyzing two cases: honest payer and honest payee.

Case 1: Honest payer U_0. We demonstrate that U_0 pays x if and only if it receives credential s from payee U_{n+1}. According to $\mathcal{F}_{Zeus}$ and the sub-functionality $\mathcal{F}_{\text{splicing-launcher}}$:

- **If U_0 receives credential s:** This occurs only when the protocol successfully reaches the settlement phase, allowing U_0 to update channel γ_0 to the PAID state. In this state, the payer transfers amount x to the first relay.
- **If U_0 does not receive credential s by round $T - 2t_{close} - \Delta$:** The payer will splice channel γ_0, revoking all off-chain payments. In this case, U_0 can offer the payment directly on-chain through the HTLC mechanism, which guarantees U_0 pays x if and only if it receives s.

Case 2: Honest payee U_{n+1}. We prove that U_{n+1} gives credential s if and only if it receives payment x. According to $\mathcal{F}_{Zeus}$, the payee U_{n+1} reveals the secret s only in two scenarios:

- **Off-chain settlement:** When channel γ_n reaches the REFUNDED-Dep state. We can verify that in all possible resolution paths of this state:
  - If channel γ_n times out in REFUNDED-Dep state, U_{n+1} receives amount x.
  - If channel γ_n further settles to PAID state, U_{n+1} receives amount x.
  - If splicing occurs before the timeout, U_{n+1} can submit a redemption transaction with credential s on-chain, receiving amount x.
- **On-chain settlement:** When the splicing transaction finalizes in $\mathcal{F}_{Channels}$ before time T, the payee reveals s only during the redemption transaction, which guarantees receipt of amount x on-chain.

In both cases, the honest payee reveals credential s if and only if it receives payment x, either off-chain or on-chain.

Therefore, $\mathcal{F}_{Zeus}$ achieves credential exchange as defined in Definition H.6. □