Baker: A Privacy-Preserving, NIZK-free and Efficient Payment Channel Hub Supporting Bidirectional Channels

Wenjing Li, Zi Li, Yuan Zhang, Sheng Zhong
Nanjing University, China
Email: liwenjing@smail.nju.edu.cn, zili@smail.nju.edu.cn, zhangyuan@nju.edu.cn, zhongsheng@nju.edu.cn

***Abstract*—Payment Channel Hub (PCH) improves blockchain scalability by enabling off-chain transactions via an untrusted intermediary known as the tumbler. However, existing PCHs either fail to guarantee the unlinkability privacy or rely on inefficient non-interactive zero-knowledge (NIZK) proofs. Recently, Ge et al. proposed Accio, a privacy-preserving PCH that eliminates the need for NIZK proofs. Nevertheless, Accio only supports unidirectional channels which results in high on-chain costs and routing inefficiencies. In this paper, we present Baker, the first bidirectional payment channel hub that operates without NIZK proofs and guarantees unlinkability. Unlike prior PCH solutions that maintain channel balance using a single state, Baker introduces a novel design in which each non-tumbler user maintains two separate pockets to record the channel balance. To ensure payment atomicity, Baker further designs a novel cryptographic primitive named Aggregatable Adaptor Signature (AAS) to enable atomic signature exchanges and signature aggregation. We implement Baker and empirically demonstrate its advantages over state-of-the-art protocols. Compared to BlindHub, which relies on NIZK proofs for privacy, Baker reduces off-chain communication overhead to 0.0036%. Moreover, the off-chain computation overhead of Baker is 7% of that of BlindHub and 40% of TBPChannel. Relative to Accio, Baker incurs only 80% of its on-chain cost and enjoys a 25% higher average transaction success rate.**

***Index Terms*—Payment Channel Hub, Bidirectional Channel, NIZK-free, Privacy**

I. Introduction

Scalability remains one of the fundamental challenges hindering the widespread adoption of blockchain systems. Among the various proposed solutions [1]–[3], the payment channel (e.g., [4], [5]) is one of the most widely adopted approaches. The core idea is to allow two parties to deposit their respective funds (i.e., initial channel balances) into a jointly controlled on-chain address, thereby enabling multiple off-chain transactions by updating the channel state [6]. Since only opening and closing the channel require interaction with the blockchain, this design reduces on-chain congestion and greatly improves overall system throughput [7].

Establishing a direct payment channel between every pair of users is often impractical, as it requires each user to lock a substantial amount of funds on-chain. Payment Channel Hubs (PCHs) avoid this by introducing an untrusted centralized intermediary known as a tumbler. In a PCH, each user maintains only a single channel with the hub, enabling payments between arbitrary users to be routed in one hop via the tumbler.

Security is the fundamental requirement of a PCH. Each transaction involves two subpayments: one on the payer-tumbler channel and another on the tumbler-payee channel. To ensure *atomicity*, both subpayments must either succeed or fail together [8]. The presence of the centralized intermediary also raises privacy concerns, as the tumbler may link the payer to the payee. This motivates the need for *unlinkability*—a privacy property that prevents transactions from being associated with the specific payer and payee [9]. As summarized in Table I, most PCHs achieve these two essential goals. However, they remain susceptible to advanced threats like the abort attack, which can degrade performance or disrupt fairness.

Moreover, existing PCH protocols that achieve both security and privacy often sacrifice efficiency and thus practicality. TumbleBit [9] introduces the puzzle-promising and puzzle-solving paradigm to ensure atomicity but enforces a fixed payment amount across all transactions to preserve unlinkability. This constraint requires a large transaction to be split into multiple fixed-size payments. It is inefficient because each fixed-size payment incurs the full protocol overhead. Several subsequent protocols [10]–[13] adopt the same paradigm and inherit this limitation. BlindHub [14] improves the paradigm to achieve *flexible amount*, but still relies on computationally expensive NIZK proofs. According to [14], each transaction takes approximately 17 seconds to complete.

Accio [17] marks an important step forward for PCHs by addressing the security vulnerabilities and efficiency limitations of prior protocols. It resists the abort attack, removes reliance on NIZK proofs, and supports flexible payment amounts. Compared to BlindHub, Accio incurs significantly lower off-chain communication overhead, defined as the amount of data exchanged among participants during payment execution, as well as lower computational overhead, measured by the time required to complete a payment [17]. These advantages arise from its use of two opposing unidirectional channels with the tumbler to support bidirectional functionality. However, a unidirectional channel permits balance transfers in only one fixed direction, leading to faster balance exhaustion which increases on-chain costs and reduces channel network stability.

The above analyses of state-of-the-art PCH protocols raise the following question:

Is it possible to design a secure, privacy-preserving and efficient PCH protocol that does not rely on NIZK proofs,

TABLE I: Comparison with the state-of-the-art PCH protocols

	Unlinkability	Atomicity	Abort Attack Resistance	Bidirectional Channel	Flexible Amount	NIZK-free*
TumbleBit [9]	√	√	×	×	×	√
Bolt [15]	√	√	×	√	√	×
Perun [16]	×	√	√	√	√	√
A^2L [10]	√	√	×	√	×	×
A^2L^+,A^2L^{UC} [11]	√	√	×	√	×	×
BlindHub [14]	√	√	×	√	√	×
Accio [17]	√	√	√	×	√	√
AUPC [12]	√	√	×	√	×	×
AuditPCH [13]	√	√	×	√	×	×
Baker (our protocol)	√	√	√	√	√	√

* TumbleBit requires NIZK proofs in the setup phase.

supports flexible payment amounts and bidirectional channels?

In this paper, we present Baker, a novel PCH protocol that provides an affirmative answer. Our main contributions are summarized as follows:

- We analyze the limitations of unidirectional channels in PCHs and further classify the goals of PCH protocols.
- We propose a novel cryptographic primitive for Baker named Aggregatable Adaptor Signature (AAS), which implements adaptor signatures and signature aggregation involving three digital signatures. We formally define its security properties, provide a concrete instantiation, and present the formal security proof.
- We design Baker, which maintains the channel balance state through the non-tumbler user's payer pocket and anonymized payee pocket. During payment execution, transaction completion requires interaction only between the payer and the tumbler. We further formalize Baker in the GUC framework and prove its security.
- We implement Baker and several state-of-the-art PCH protocols. We evaluate their performance in terms of on-chain costs, off-chain overhead, and routing performance. Baker incurs only 80% of the on-chain cost of Accio. It incurs only 40% of the off-chain computation overhead of TBPChannel. Compared to BlindHub, Baker reduces communication overhead to 0.0036% and computation overhead to 7%. Moreover, Baker consistently maintains a high transaction success rate as the number of transactions increases, averaging 25% higher than Accio.

II. Related Work

We review existing PCH protocols and their techniques.

The puzzle promising and solving paradigm and its limitations. Since TumbleBit [9] first informally introduced the puzzle promising and solving paradigm for PCH, a number of subsequent works have adopted and extended this paradigm. The core idea is to bind a payment to a cryptographic puzzle. The tumbler generates a conditional payment as well as a puzzle (e.g. a one-way function's output on a random input) and sends them to the payee, who can claim the payment by submitting a valid solution (e.g. the input). To enable an (indirect) transaction from the payer, the payee forwards the puzzle to the payer who then pays the tumbler instead to "buy" the solution and then sends it to complete its transaction with the payee. A^2L [10] formally defines the above paradigm, while A^2L^+ and A^2L^{UC} [11] identify security limitations in A^2L and propose stronger variants. AUPC [12] proposes a new design for payment channels, and AuditPCH [13] augments the traditional PCH model by enabling auditability while preserving privacy. However, since the tumbler sees the payment amount of every puzzle and its corresponding payee, and also knows the amount of every solution purchase and its payer, it can use the amount to link a transaction's payer and payee when transactions have different amounts. To achieve unlinkability, the above protocols enforce fixed-amount payments, which require inefficiently splitting transactions into multiple fixed-value transfers. To address this limitation, Bolt [15] abandons the puzzle-based paradigm. However, it is limited to anonymous blockchains such as Monero [18] and ZCash [19]. BlindHub [14] is compatible with multiple blockchains and supports flexible amounts within the puzzle-based framework. Unfortunately, BlindHub relies heavily on NIZK proofs, with each transaction taking around 17 seconds [14].

Compromise for NIZK-free designs. Several PCH protocols have recognized that the use of NIZK proofs in highly interactive PCHs can hinder efficiency. TumbleBit [9] avoids NIZK proofs in its transaction phase by applying the cut-and-choose technique, although it still requires NIZK proofs in the setup phase. Furthermore, TumbleBit fails to address critical threats such as the abort attack, and it does not support variable payment amounts. Perun [16] introduces virtual channels atop payment channels to enable direct payer-payee transactions without NIZK proofs. However, establishing numerous virtual channels incurs high on-chain costs, and the tumbler's involvement compromises unlinkability. Accio [17] proposes a novel NIZK-free PCH that uses unidirectional channels to achieve unlinkable and variable-amount payments. However, implementing bidirectional functionality in Accio requires creating two unidirectional channels between each user and the tumbler, which doubles the on-chain cost. Moreover, due to the unidirectional fund flow, these channels cannot self-rebalance, leading to frequent exhaustion [20]. In practice, such frequent exhaustion reduces network stability and lowers the overall transaction success rate [21].

The overuse of smart contracts. PCH protocols [12], [16], [17], [22], [23] built on Turing-complete blockchains handle on-chain operations via smart contracts which are self-executing programs that encode and enforce agreements without trusted intermediaries. Once deployed, the code of a smart

contract is immutable and publicly accessible. While channel opening and closing inherently require on-chain interaction, some protocols over-rely on extra smart contracts to resolve disputes, which increases on-chain costs. To resolve disputes where the receiver reveals the secret but does not receive a receipt, AUPC [12] invokes the `PromiseClaim` function, whose worst-case cost is 4x the combined cost of opening and closing a channel. Similarly, NOCUST [22] allows users to submit challenges if the operator misbehaves, and the evaluation results show that the gas cost of executing the `Challenge` is 3x that of the full channel lifecycle.

III. Preliminaries

A. Unidirectional and bidirectional channels

A PCH comprises multiple payment channels between users and the tumbler. Based on the supported direction of fund transfers, the channels can be classified as **unidirectional** or **bidirectional**. Once established, a unidirectional channel supports payments only in one fixed direction, whereas a bidirectional channel enables transfers in both directions between the two channel parties. Although two oppositely directed unidirectional channels can simulate bidirectional functionality, this method introduces several limitations, as outlined below:

Increased on-chain costs. Each unidirectional channel requires separate smart contract calls for channel opening and closing, thereby doubling the number of on-chain operations compared with a single bidirectional channel. Consequently, using two unidirectional channels incurs higher on-chain costs.

Balance exhaustion. Bidirectional channels naturally support balance rebalancing through reverse payments [24]. But balances cannot move between the two opposing unidirectional channels, so each channel's available balance declines with each transaction. Over time, this may lead to insufficient funds for new payments, lowering the transaction success rate [25], [26]. For example, consider a bidirectional channel between A and B, and two opposing unidirectional channels between C and D, where each party initially holds 6 units. Both pairs execute two transactions of 3 units in each direction. After these transactions, the bidirectional channel retains sufficient balance for further transactions. In contrast, the two opposing unidirectional channels are both exhausted and become unusable. Restoring functionality requires closing the exhausted channels and opening new ones [27], incurring additional on-chain costs. When balance exhaustion becomes widespread, it undermines the efficiency and stability of the channel network.

B. Cryptographic Primitives

Randomizable Signature on Commitments (RSoC). In Baker, each user's payee pocket is anonymized by hiding its value within a commitment. It becomes valid only after being signed by the tumbler. RSoC allows the user to randomize both the commitment and its signature into a new commitment-signature pair without access to the tumbler's secret key. After randomization, the hidden value remains unchanged and the new pair is publicly verifiable. Therefore, the tumbler cannot link the randomized pocket to the original one, which ensures unlinkability. Formally, RSoC contains the PPT algorithms $\Sigma_{RSoC} := (\mathsf{ComGen}, \mathsf{ComSign}, \mathsf{ComVf}, \mathsf{ComSignVf}, \mathsf{ComSignPartVf}, \mathsf{ComUpd}, \mathsf{UpdVf}, \mathsf{ComRdm})$. The algorithm ComGen generates a commitment that hides the value and ComSign signs the commitment. ComVf and ComSignVf verify the commitment and its signature, respectively. ComSignPartVf verifies a partial signature. ComUpd updates the commitment without revealing the hidden value and UpdVf checks the correctness of the updated commitment–signature pair. ComRdm randomizes the commitment.

Adaptor signature. An adaptor signature scheme [4] extends a digital signature scheme, allowing a witness to be embedded into an incomplete signature (often called a pre-signature) so that the witness can be extracted once the complete signature is revealed. This forms the basis for designing atomic signature exchanges between the payer and the tumbler in Baker. Formally, the adaptor signature scheme consists of PPT algorithms $\Sigma_{AS} := (\mathsf{PreSign}, \mathsf{Adapt}, \mathsf{AdaptVerify}, \mathsf{Extract})$ over a base digital signature scheme Σ_{DS} and a hard relation $\mathcal{R}$, which consists of instance-witness pairs (Y, y) such that $(Y, y) \in \mathcal{R}$. $\mathsf{PreSign}(\mathsf{sk}, m, Y)$ generates a pre-signature $\hat{\sigma}$ for message m and statement Y, and $\mathsf{Adapt}(\hat{\sigma}, y)$ computes a full signature σ using witness y. $\mathsf{AdaptVerify}(\mathsf{pk}, m, \hat{\sigma}, Y)$ checks the validity of $\hat{\sigma}$ under the message m and statement Y. Given the valid signature pair $(\sigma, \hat{\sigma})$ and Y, $\mathsf{Extract}(\sigma, \hat{\sigma}, Y)$ extracts the witness y.

IV. Design Goals

To design a secure, privacy-preserving, and efficient PCH protocol, we define two categories of core goals. The first is *security and privacy goals* (**G1**–**G5**), which must be achieved even against malicious users. The second is *functionality and efficiency goals* (**G6**–**G9**), which capture the practical requirements of real-world PCH systems.

(G1) Unlinkability. The tumbler should not learn which payer corresponds to which payee in any particular transaction.

(G2) Atomicity. A payment should be atomic across the two involved channels. That is, the state update on the payer-tumbler channel and that on the tumbler-payee channel should either both take effect or both abort.

(G3) Balance sufficiency. A payment should be executable only if the two involved channels hold sufficient balance for the corresponding state transitions.

(G4) Settlement correctness. Channel settlement should always reflect the latest valid channel state. No party should be able to gain funds by submitting stale, inconsistent, or otherwise invalid states at any stage of the protocol.

(G5) Abort-attack resistance. The protocol should resist selective abort attacks by a malicious tumbler. In particular, the tumbler should not be able to deliberately abort a payment in a manner that reveals or helps infer the payer-payee relationship.

(G6) Channel bidirectionality. Each user should be able to act as either a payer or a payee over the same payment channel, without requiring two opposing unidirectional channels.

(G7) Flexible amount. The protocol should support payments of arbitrary amounts within the available channel balance, rather than restricting payments to a fixed denomination.

(G8) NIZK-free design. The protocol should avoid reliance on NIZK proofs to reduce off-chain computation and communication overhead, thereby improving practical efficiency.

(G9) Minimal on-chain functionality. The protocol should require only the smart contract functionalities necessary for channel opening and closing. Additional on-chain mechanisms, such as dedicated dispute-resolution contracts, should be avoided whenever possible to reduce deployment complexity and on-chain cost.

V. Solution Overview

We present Baker, a privacy-preserving and efficient PCH protocol that supports bidirectional channels without relying on NIZK proofs. In Baker, each user maintains two distinct pockets for its bidirectional channel with the tumbler: a payee pocket and a payer pocket. The value in the payee pocket remains hidden and is visible only to its owner, while the payer pocket is publicly visible to both parties. These pockets track the channel balance updates under different situations. For each payment, only one pocket per user requires an update.

Specifically, the payee sends the valid payee pocket to the payer, who then forwards the pocket along with its own valid payer pocket to the tumbler. The tumbler updates and signs both pockets, while the payer signs the updated payer pocket. The payer and the tumbler then atomically exchange the updated pockets and their corresponding signatures to complete the transaction, which guarantees atomicity. This atomic exchange is implemented using the RSoC scheme and the Aggregatable Adaptor Signature scheme (refer to §VIII for details). Since the payee pocket remains anonymized and the payee can randomize the pocket and its signature, unlinkability is ensured. Furthermore, the tumbler can verify channel balance sufficiency by inspecting the payer's pockets, without requiring NIZK proofs.

Below, we provide an overview of Baker.

Open. The user and the tumbler open a channel by invoking a smart contract, with both parties locking a specified amount of funds as the initial channel balance.

Pocket initialization. This procedure is performed only once, immediately after the channel is opened, to support future transactions. The payee pocket is initialized with the user's initial balance. The tumbler generates and signs the initial payee pocket, then sends it to the user with the corresponding blinding factor. Upon verification, the user randomizes the signed payee pocket to anonymize it. The payer pocket's initial value is 0. The user and the tumbler then sign the initial payer pocket and exchange their signatures, which are aggregated into a single valid signature on the payer pocket.

Payment. To illustrate how Baker processes a transaction, we consider the example where Alice pays Bob through the tumbler (Figure 1). Bob first sends the current anonymized payee pocket with the tumbler's valid signature on the pocket and the transaction amount (e.g., 1 coin) to Alice. Alice then forwards this information along with its current signed payer pocket to the tumbler. The tumbler updates and signs both Alice's payer pocket and the anonymized payee pocket. Alice obtains the updated pockets with signatures if and only if Alice provides its signature on the updated payer pocket to the tumbler. The valid signature on Alice's updated payer pocket is the aggregation of signatures from both Alice and the tumbler. This signature aggregation is carried out independently by both the payer and the tumbler. Finally, Alice forwards the updated anonymized payee pocket with a valid signature to Bob, who randomizes it for future payments.

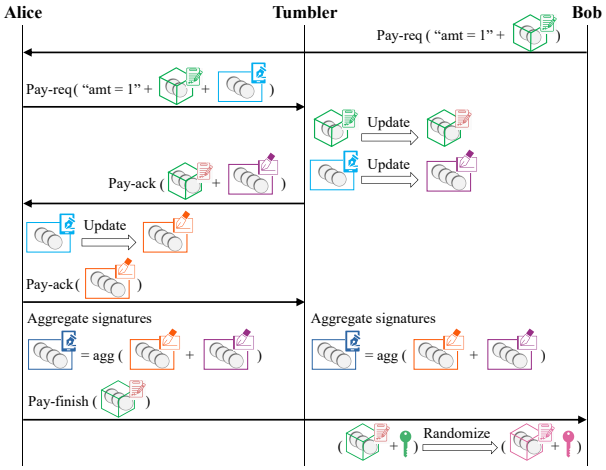


Fig. 1: Transaction execution. The pockets are represented solely by their recorded values for clarity. A colored cube surrounding the anonymized payee pocket indicates that it is hidden using a blinding factor, while a box around the payer pocket denotes that it remains in plaintext. A change in color reflects an update.

It is worth noting that (i) the payee pocket is anonymized, preventing the tumbler from identifying its owner, and (ii) all pocket updates and their associated signatures are publicly verifiable, allowing any party to discover forgery immediately.

Close. The channel is closed by invoking the smart contract. During the settlement process, the anonymized payee pocket is revealed. The user's final balance is computed as the payee pocket value minus the payer pocket value, while the tumbler's final balance equals Cap minus the user's final balance.

Challenge of bidirectional channels. In bidirectional channels, balance updates are non-monotonic, so a stale state may be more favorable to one party than the latest valid state. As a result, either party may settle the channel using a stale state for unfair gain. A secure protocol must therefore ensure that channel settlement always reflects the latest valid state.

Differences from Accio. Accio and Baker follow a similar transaction workflow: the payee first sends a transaction request to the payer, after which the payer interacts with the tumbler to complete the transaction. However, the two protocols differ fundamentally in design. Accio addresses the above challenge by replacing each bidirectional tumbler-user channel with two opposing unidirectional channels. This design simplifies state submission because, in each unidirec-

tional channel, the receiver's balance evolves monotonically, making the receiver naturally willing to submit the latest state. However, as discussed in §III-A, reliance on unidirectional channels introduces several practical and efficiency limitations. In contrast, Baker preserves the bidirectional channel structure and instead decomposes the channel balance into two monotonic components: a payee pocket recording the total amount received by the user and a payer pocket tracking the total amount paid by the user. These two pockets naturally align incentives for state submission: the user is incentivized to provide the latest payee pocket, while the tumbler is incentivized to provide the latest payer pocket. The channel balance can then be computed by combining the two pockets. In this way, Baker solves the challenge of bidirectional channels without inheriting the drawbacks of Accio's unidirectional design.

VI. Construction of RSoC

Similar to Accio and BlindHub, our construction follows the general framework proposed by Bauer et al. [28], with minor modifications to suit Baker. Although all three protocols refer to the study [28], their applications differ: BlindHub uses RSoC to bind the payment amount to a puzzle, Accio applies it to record unidirectional payee channel states, and Baker uses RSoC to initialize and update every user's payee pocket.

The RSoC scheme Σ_{RSoC} is built upon an asymmetric bilinear pairing $e : \mathbb{G} \times \hat{\mathbb{G}} \to \mathbb{G}_T$, where $\mathbb{G}$ and $\hat{\mathbb{G}}$ are additive cyclic groups generated by G and $\hat{G}$ respectively. All groups have the same prime order p.

- $\mathsf{Setup}(1^\lambda)$: Given a security parameter 1^λ, it outputs public parameters $params = (p, \mathbb{G}, G, \hat{\mathbb{G}}, \hat{G}, \mathbb{G}_T, e, P)$.
- $\mathsf{KeyGen}(params)$: It outputs a secret key $\mathsf{sk} = (x_0, x_1) \in (\mathbb{Z}_p^*)^2$ and the public key $\mathsf{vk} = (X_0 := x_0\hat{G}, X_1 := x_1\hat{G})$.
- $\mathsf{ComGen}(v, r)$: Given a message $v \in \mathbb{Z}_p$ and randomness $r \in \mathbb{Z}_p$, returns $cm = (C_0 := rG, C_1 := vG + rP)$.
- $\mathsf{ComSign}(cm = (C_0, C_1), \mathsf{sk})$: Samples $s \xleftarrow{\$} \mathbb{Z}_p^*$, returns the signature $\sigma = (Z, S, \hat{S}, T)$ with (1) $Z := \frac{1}{s}(G + x_0C_0 + x_1C_1)$, (2) $S := sG$, (3) $\hat{S} := s\hat{G}$, and (4) $T := \frac{1}{s}(x_0G + x_1P)$.
- $\mathsf{ComVf}(cm, v, r)$: It returns 1 if $C_0 = rG$ and $C_1 = vG + rP$, and returns 0 otherwise.
- $\mathsf{ComSignVf}(cm, \sigma, \mathsf{vk})$: It returns 0 if $\hat{S} = 0$. It returns 1 if the following equations hold and 0 otherwise: (1) $e(Z, \hat{S}) = e(G, \hat{G})e(C_0, X_0)\ e(C_1, X_1)$; (2) $e(G, \hat{S}) = e(S, \hat{G})$; and (3) $e(T, \hat{S}) = e(G, X_0)e(P, X_1)$.
- $\mathsf{ComSignPartVf}(cm = (C_0, C_1), cm_{\text{new}} = (C_0', C_1'), Z, \hat{S}, T, a, \mathsf{vk})$: It returns 0 if $\hat{S} = 0$. It returns 1 if the following equations hold and 0 otherwise: (1) $C_0' = C_0$; (2) $C_1' = C_1 + aG$; (3) $e(Z, \hat{S}) = e(G, \hat{G})e(C_0, X_0)\ e(C_1, X_1)$ and (4) $e(T, \hat{S}) = e(G, X_0)\ e(P, X_1)$.
- $\mathsf{ComUpd}(cm = (C_0, C_1), a)$: It returns an updated commitment $cm_{\text{new}} = (C_0' := C_0, C_1' := C_1 + aG)$.
- $\mathsf{UpdVf}(cm = (C_0, C_1), cm_{\text{new}} = (C_0', C_1'), \sigma_{\text{new}} = (Z, S, \hat{S}, T), a, \mathsf{vk})$: It returns 0 if $\hat{S} = 0$. It returns 1 if the following equations hold and 0 otherwise: (1) $C_0' = C_0$; (2) $C_1' = C_1 + aG$; (3) $e(Z, \hat{S}) = e(G, \hat{G})e(C_0', X_0)e(C_1', X_1)$; (4) $e(G, \hat{S}) = e(S, \hat{G})$; and (5) $e(T, \hat{S}) = e(G, X_0)e(P, X_1)$.
- $\mathsf{ComRdm}(cm, \sigma, r')$: $s' \xleftarrow{\$} \mathbb{Z}_p^*$. It outputs the randomized commitment $cm' = (C_0 + r'G, C_1 + r'P)$ and signature $\sigma' = (Z', S', \hat{S}', T')$ with (1) $Z' := \frac{1}{s'}(Z + r'T)$, (2) $S' := s'S$, (3) $\hat{S}' := s'\hat{S}$, and (4) $T' := \frac{1}{s'}T$.

More detailed formal security definitions and proofs for RSoC can be found in reference [28].

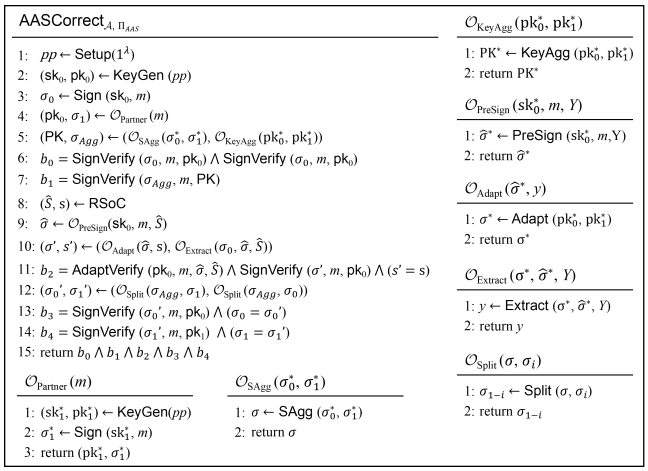


Fig. 2: Experiment $\mathsf{AASCorrect}_{\mathcal{A},\Pi_{\mathsf{AAS}}}$.

VII. Aggregatable Adaptor Signature

To ensure atomicity in Baker, the tumbler signs both the updated anonymized payee pocket and the updated payer pocket, and atomically exchanges them with the payer's signature on the updated payer pocket. This process involves three signatures, where the payer's and tumbler's signatures on the payer pocket must be aggregated. However, standard adaptor signatures neither support atomic exchanges involving three signatures nor enable signature aggregation. To address this, we design a novel cryptographic primitive called the Aggregatable Adaptor Signature (AAS).

Definition 1 (Aggregatable Adaptor Signature Scheme)**.** *An Aggregatable Adaptor Signature (*AAS*) scheme* Π_{AAS} *is constructed upon a digital signature scheme* Σ_{DS} *with the linear homomorphic addition property and a hard relation* $\mathcal{R}$*. The scheme* Π_{AAS} *consists of the following PPT algorithms:*

- $\mathsf{AAS.Setup}(1^\lambda)$*: Given a security parameter* 1^λ*, it outputs the public parameters* pp*.*
- $\mathsf{AAS.KeyGen}(\mathsf{pp})$*: Outputs a key pair* $(\mathsf{sk}, \mathsf{pk})$*.*
- $\mathsf{AAS.KeyAgg}(\mathsf{pk}_0, \mathsf{pk}_1)$*: Given two public keys* pk_0 *and* pk_1*, it outputs the aggregated public key* PK*.*
- $\mathsf{AAS.Sign}(\mathsf{sk}, m)$*: Given a secret key* sk *and a message* $m \in \{0,1\}^*$*, it outputs a valid digital signature* σ*.*
- $\mathsf{AAS.SAgg}(\sigma_0, \sigma_1)$*: Given two signatures* σ_0 *and* σ_1*, it outputs the aggregated signature* σ^**.*
- $\mathsf{AAS.Split}(\sigma^*, \sigma_i)$*: Given an aggregated signature* σ^* *and one of its components* σ_i*, it returns the other component* σ_{1-i}*, for* $i \in \{0,1\}$*.*
- $\mathsf{AAS.SignVerify}(\sigma, m, \mathsf{pk})$*: Returns 1 if* σ *is valid on* m *under* pk*, and 0 otherwise. The algorithm supports the verification of both individual and aggregated signatures.*

- The algorithms AAS.PreSign, AAS.Adapt, AAS.Extract and AAS.AdaptVerify *are the same as* PreSign, Adapt, Extract *and* AdaptVerify *of the adaptor signature.*

To rigorously analyze the security of AAS, we first define the formal security properties. Furthermore, we instantiate the AAS framework and provide a formal security proof, which are presented in Appendix A.

Definition 2 (Correctness)**.** *An* AAS *scheme* Π_{AAS} *is correct if for every PPT adversary* $\mathcal{A}$ *running the experiment* $\mathsf{AASCorrect}_{\mathcal{A},\Pi_{\mathsf{AAS}}}$ *defined in Figure 2,* it holds that $\Pr[\mathsf{AASCorrect}_{\mathcal{A},\Pi_{\mathsf{AAS}}}(\lambda) = 1] = 1$.

Definition 3 (Unforgeability)**.** *An* AAS *scheme* Π_{AAS} *achieves unforgeability if for every PPT adversary* $\mathcal{A}$ *running the experiment* $\mathsf{AASSignForge}_{\mathcal{A},\Pi_{\mathsf{AAS}}}$ *defined in Figure 3, the probability that* $\mathcal{A}$ *succeeds is negligible in the security parameter* λ*, i.e.,* $\Pr[\mathsf{AASSignForge}_{\mathcal{A},\Pi_{\mathsf{AAS}}}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$.

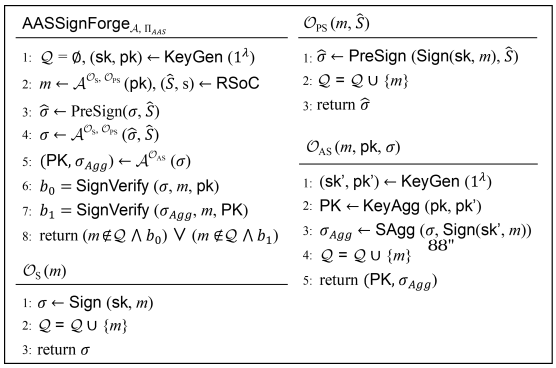


Fig. 3: Experiment $\mathsf{AASSignForge}_{\mathcal{A},\Pi_{\mathsf{AAS}}}$.

Definition 4 (Witness Extractability)**.** *An* AAS *scheme* Π_{AAS} *is witness extractable if for every PPT adversary* $\mathcal{A}$ *running the experiment* $\mathsf{AASWitExt}_{\mathcal{A},\Pi_{\mathsf{AAS}}}$ *defined in Figure 4, the probability that* $\mathcal{A}$ *succeeds is negligible in the security parameter* λ*, i.e., Pr*$[\mathsf{AASWitExt}_{\mathcal{A},\Pi_{\mathsf{AAS}}}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$.

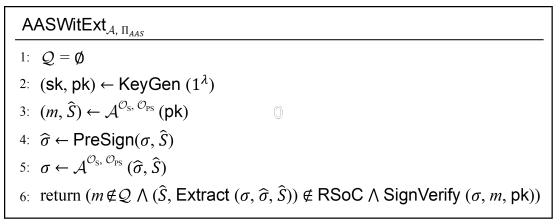


Fig. 4: Experiment $\mathsf{AASWitExt}_{\mathcal{A},\Pi_{\mathsf{AAS}}}$.

VIII. Baker's Design

This section provides a detailed description of the Baker protocol across three stages: channel opening, payment execution, and channel closing. In the open stage, the users open a channel on-chain and initialize two pockets for the user off-chain. During the payment, the users and the tumbler apply AAS and RSoC to update the pockets, thereby securely and efficiently recording channel balance updates. In the close stage, the users compute the latest channel balance from the pockets and close the channel on-chain. Finally, we analyze how Baker achieves all the goals in §IV.

A. Channel Opening

A user and the tumbler open a channel via a smart contract by depositing their respective funds into the contract address. The contract records their balances and channel details, then returns this information to both parties (Figure 7 (A)).

After the channel is established, the payee pocket P_{payee} and the payer pocket P_{payer} of the user must be initialized before the channel serves for payments. Let H be a secure hash function. Formally, these pockets are defined as follows:

$$P_{\text{payee}} = (C_0 = rG, C_1 = \alpha G + rP) \quad (1)$$

$$P_{\text{payer}} = SN \,||\, \beta \quad (2)$$

The payee pocket P_{payee} is a commitment (C_0, C_1) generated by the ComGen algorithm in RSoC. The component α is defined as $\alpha = H(id) \,||\, v_1$, where id denotes the unique channel identifier and v_1 is the total amount of coins the user has received as the payee.

The payer pocket P_{payer} includes a monotonically increasing serial number SN, where larger values indicate more recent pocket states. The component β is defined as $\beta = H(id) \,||\, v_2$, where id denotes the unique channel identifier and v_2 represents the total amount of coins the user has paid as the payer.

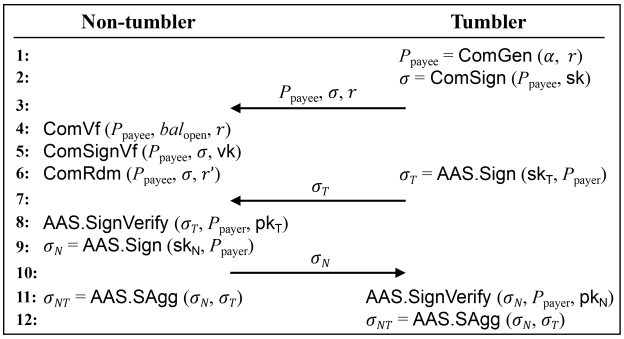


Fig. 5: Pocket initialization procedure. In this figure, $(\mathsf{sk}, \mathsf{vk})$ is the key pair of the tumbler in RSoC; $(\mathsf{sk}_\mathsf{T}, \mathsf{pk}_\mathsf{T})$ is the tumbler's key pair and $(\mathsf{sk}_\mathsf{N}, \mathsf{pk}_\mathsf{N})$ is the user's key pair in AAS.

During initialization, the tumbler generates the user's P_{payee}, where v_1 is set to the initial channel balance bal_{open} of the user, and signs P_{payee} (Lines 1-2, Figure 5). For P_{payer}, both the serial number SN and v_2 are initialized to zero. The user and the tumbler independently sign P_{payer} and exchange their signatures, which are then aggregated into the valid signature on P_{payer} (Lines 7-12, Figure 5). Upon verifying P_{payee}, the user randomizes the signed payee pocket $(P_{\text{payee}}, \sigma)$ to ensure its anonymity (Lines 3-6, Figure 5). This anonymized pocket is indistinguishable from its original, preventing the tumbler from learning either its content or its owner. The tumbler cannot infer the current channel balance from P_{payer} alone.

B. Payment Execution

The core idea of the payment is to simultaneously update the payer's P_{payer} and the payee's anonymized P_{payee} during the interaction with the tumbler, while ensuring the tumbler learns neither the payer–tumbler nor tumbler–payee channel balances and no party steals funds. Figure 6 illustrates the payment process via the example of Alice paying amt to Bob.

To initiate a payment, the payee sends the amount amt and the signed anonymized payee pocket $(P_{\text{payee}}, \sigma)$ to the payer. After verifying $(P_{\text{payee}}, \sigma)$, the payer sends a payment request to the tumbler (Lines 1-5, Figure 6), which includes $(P_{\text{payee}}, \sigma)$, amt, and Alice's payer pocket with the aggregate signature $(P_{\text{payer}}, \sigma_{AT})$. Upon receiving the request, the tumbler checks (Lines 6-10, Figure 6): **(C1)** the validity of the request, **(C2)** the validity of the anonymized payee pocket, **(C3)** the validity of the payer pocket, **(C4)** whether the payer pocket is the latest, and **(C5)** whether the payer has sufficient balance.

C1–C3 are checked by verifying the attached signatures. After each payment, the tumbler stores the updated payer pocket. To verify **C4**, it compares the received P_{payer} with the stored version. If they match, P_{payer} is the latest. At channel setup, the tumbler stores the payer's initial balance as bal_s. Since v_2 in P_{payer} is always visible, the tumbler verifies **C5** by computing $bal_s - v_2$. If $bal_s - v_2 \geq amt$, the payment proceeds. Otherwise, the tumbler requests the payer to reveal its current anonymized payee pocket and updates $bal_s = v_1$. The payer then re-anonymizes the revealed payee pocket with ComRdm. The payment proceeds only if updated bal_s still satisfies $bal_s - v_2 \geq amt$; otherwise, it fails due to insufficient balance. This procedure constitutes the whole check for **C5**, denoted as $\mathsf{Check}(bal_s, amt)$ in line 10 of Figure 6. Both the tumbler and the payer know bal_s in plaintext. If the tumbler disregards bal_s and requests the payer to reveal its payee pocket in each transaction, the payer can detect this deviation immediately and refuse the request to abort the payment.

Although verifying **C5** may reveal the payer's anonymized payee pocket, it does not compromise unlinkability. The pocket is not revealed in every transaction, and each payee pocket is randomized after a payment. So the tumbler cannot link the revealed pocket to any previously hidden ones to identify past payments in which the payer acted as a payee. Since the user can randomize a revealed payee pocket, the tumbler cannot infer the user's participation as a payee in future transactions.

Moreover, verifying **C5** does not trigger any rebalancing between the payer pocket and the payee pocket. Although it may update bal_s, it never changes v_1 or v_2. Therefore, throughout the channel lifetime, both v_1 and v_2 remain monotonically non-decreasing. As a result, verifying **C5** never creates an incentive for the user to submit a stale payee pocket or for the tumbler to prefer a stale payer pocket. This preserves the incentive structure required for correct settlement.

If all checks succeed, the tumbler invokes ComUpd to increment the value in the anonymized payee pocket by amt without learning its content, and signs the updated pocket $P_{\text{payee-new}}$ as $\sigma_{\text{new}} = (Z_{\text{new}}, S_{\text{new}}, \hat{S}_{\text{new}}, T_{\text{new}})$. Meanwhile, the tumbler generates the updated payer pocket $P_{\text{payer-new}}$ by incrementing the serial number SN by 1 and increasing v_2 by amt, and signs it as $\sigma_{\text{part-T}}$. (Lines 11-13, Figure 6).

Only the pockets with valid signatures are valid. Specifically, the updated payee pocket $P_{\text{payee-new}}$ must be signed by the tumbler, while the updated payer pocket $P_{\text{payer-new}}$ requires an aggregate signature jointly produced by both the payer and the tumbler. To ensure security and atomicity, the tumbler performs an atomic exchange: it provides σ_{new} and $\sigma_{\text{part-T}}$ in return for the payer's signature $\sigma_{\text{part-A}}$ on $P_{\text{payer-new}}$. The partial signature $\hat{S}_{\text{new}} = s\hat{G}$ in σ_{new} is a hard relation, where s is a secret scalar. The tumbler generates a pre-signature $\hat{\sigma}_{\text{part-T}}$ using $\hat{S}_{\text{new}}$ via AAS.PreSign, and sends it to the payer along with $P_{\text{payee-new}}$ and the corresponding partial signature $(Z_{\text{new}}, \hat{S}_{\text{new}}, T_{\text{new}})$. At this point, the payer has not yet obtained the full signatures $\sigma_{\text{part-T}}$ and σ_{new}. After verifying the $\hat{\sigma}_{\text{part-T}}$ and $(Z_{\text{new}}, \hat{S}_{\text{new}}, T_{\text{new}})$, the payer sends its own signature $\sigma_{\text{part-A}}$ on $P_{\text{payer-new}}$ to the tumbler. After verifying $\sigma_{\text{part-A}}$, the tumbler returns $\sigma_{\text{part-T}}$, which allows the payer to reconstruct σ_{new}. Finally, both parties obtain the aggregate signature $\sigma_{\text{AT-new}}$ (Lines 14–27, Figure 6). The payer forwards $(P_{\text{payee-new}}, \sigma_{\text{new}})$ to the payee. After verification, the payee randomizes the signed pocket for future payment (Lines 28-31, Figure 6).

Bob to Alice:
1: P_{payee}, σ, amt

Alice:
2: Abort if ComSignVf (P_{payee}, σ, vk) $\neq 1$
3: $msg = (P_{\text{payee}}, \sigma, amt, P_{\text{payer}}, \sigma_{\text{AT}})$
4: $\sigma_{\text{req}} \leftarrow$ Sign (msg, sk_{req})

Alice to Tumbler:
5: msg, σ_{req}

Tumbler:
6: Abort if: (a) Vf (msg, vk_{req}, σ_{req}) $\neq 1$
7: (b) ComSignVf (P_{payee}, σ, vk) $\neq 1$
8: (c) AAS.SignVerify (σ_{AT}, P_{payer}, PK_{AT}) $\neq 1$
9: (d) $P_{\text{payer}} \neq P_{\text{payer-store}}$
10: (e) Check (bal_s, amt) $\neq 1$
11: $P_{\text{payee-new}} \leftarrow$ ComUpd (P_{payee}, amt)
12: $\hat{\sigma}_{\text{new}} = (Z_{\text{new}}, S_{\text{new}}, \hat{S}_{\text{new}}, T_{\text{new}}) \leftarrow$ ComSign ($P_{\text{payee-new}}$, sk)
13: $(P_{\text{payer-new}}, \sigma_{\text{part-T}}) \leftarrow (P_{\text{payer}} + (\mathcal{I}||amt)$, AAS.Sign ($\text{sk}_{\text{T}}$, $P_{\text{payer-new}}$))
14: $\hat{\sigma}_{\text{part-T}} =$ AAS.PreSign (sk_{T}, $P_{\text{payer-new}}$, $\hat{S}_{\text{new}}$)

Tumbler to Alice:
15: $\hat{\sigma}_{\text{part-T}}$, $P_{\text{payee-new}}$, Z_{new}, $\hat{S}_{\text{new}}$, T_{new}

Alice:
16: $(P_{\text{payer-new}}, \sigma_{\text{part-A}}) \leftarrow (P_{\text{payer}} + (\mathcal{I}||amt)$, AAS.Sign ($\text{sk}_{\text{A}}$, $P_{\text{payer-new}}$));
17: Abort if: (a) AAS.AdaptVerify (pk_{T}, $P_{\text{payer-new}}$, $\hat{\sigma}_{\text{part-T}}$, $\hat{S}_{\text{new}}$) $\neq 1$
18: (b) ComSignPartVf (P_{payee}, $P_{\text{payee-new}}$, Z_{new}, $\hat{S}_{\text{new}}$, T_{new}, amt, vk) $\neq 1$

Alice to Tumbler:
19: $\sigma_{\text{part-A}}$

Tumbler:
20: Abort if AAS.SignVerify ($\sigma_{\text{part-A}}$, $P_{\text{payer-new}}$, pk_{A}) $\neq 1$
21: $\sigma_{\text{AT-new}} =$ AAS.SAgg ($\sigma_{\text{part-A}}$, $\sigma_{\text{part-T}}$)
22: Store ($P_{\text{payer-new}}$, $\sigma_{\text{AT-new}}$)

Tumbler to Alice:
23: $\sigma_{\text{part-T}}$

Alice:
24: Abort if AAS.SignVerify ($\sigma_{\text{part-T}}$, $P_{\text{payer-new}}$, pk_{T}) $\neq 1$
25: $\sigma_{\text{AT-new}} =$ AAS.SAgg ($\sigma_{\text{part-A}}$, $\sigma_{\text{part-T}}$)
26: $\sigma_{\text{new}} = (Z_{\text{new}}$, AAS.Adapt($\hat{\sigma}_{\text{part-T}}$, AAS.Extract ($\sigma_{\text{part-T}}$, $\hat{\sigma}_{\text{part-T}}$, $\hat{S}_{\text{new}}$)), $\hat{S}_{\text{new}}$, $T_{\text{new}})$
27: Store ($P_{\text{payer-new}}$, $\sigma_{\text{AT-new}}$)

Alice to Bob:
28: $P_{\text{payee-new}}$, σ_{new}

Bob:
29: Abort if UpdVf (P_{payee}, $P_{\text{payee-new}}$, σ_{new}, amt, vk) $\neq 1$
30: $(P'_{\text{payee}}, \sigma'_{\text{new}}) \leftarrow$ ComRdm ($P_{\text{payee-new}}$, σ_{new}, r')
31: Store (P'_{payee}, σ'_{new}, r+r')

Fig. 6: Payment procedure in Baker.

C. Channel Closing

To close a channel, the latest channel state must be submitted to the blockchain for settlement, enforced by a smart contract (Figure 7 (B)). Specifically, the user submits the latest signed versions of both P_{payer} and P_{payee}, while the tumbler

retrieves and submits the latest signed P_{payer} from its local storage. Upon successfully verifying the attached signatures, the smart contract checks whether the two submitted versions of P_{payer} match. If they do not, the one with the larger SN is selected. The final balance of the non-tumbler user is computed as $bal_N = v_1 - v_2$ with v_1 from P_{payee} and v_2 from P_{payer}. The tumbler's balance is calculated as $bal_T = Cap - bal_N$, where Cap is the channel capacity. Since the payer pocket with the highest SN is used for settlement, the user cannot steal funds by submitting a stale P_{payer}. If the user submits a stale P_{payee}, this dishonest behavior only harms the user's own interest.

(A) Opening a channel

Upon receiving (OPEN_REQ, θ_1) from $\theta_1.sender$, abort if θ_1 is malformed or $\theta_1.id$ exists. Remove $\theta_1.balance$ from $\theta_1.sender$'s account on ledger $\mathcal{L}$. Then:

1. **(Timeout)** If not receiving (OPEN_REQ, θ_2) from $\theta_1.party$ within a time Δ, $\theta_1.sender$ calls (OPEN_TIMEOUT, θ_1) to refund θ_1.balance. Channel closed.

2. If (OPEN_REQ, θ_2) is received within Δ: Abort if θ_2 is illegal. Otherwise, remove $\theta_2.balance$ from $\theta_2.sender$'s account on ledger $\mathcal{L}$. If either party has insufficient funds, abort. Otherwise, emit OPEN_ACK to both parties.

(B) Closing a channel

Upon receiving (CLOSE_REQ, θ_1) from $\theta_1.sender$, abort if $\theta_1.id$ is unknown or $\theta_1.sender$ is not the user. Send (CLOSING) to $\theta_1.party$ in Δ rounds. Then:

1. **(Timeout)** If not receiving (CLOSE_REQ, θ_2) from $\theta_1.party$ within a time $6{+}2\Delta$, $\theta_1.sender$ calls (CLOSE_TIMEOUT, θ_1) to obtain all funds in channel.

2. If (CLOSE_REQ, θ_2) is received within Δ: Abort if θ_2 is illegal. Otherwise:
   (1) If θ_1 is the non-tumbler user, execute as follows:
   a. Any verification fails: If verification of $\theta_1.P_{\text{payee}}$ or $\theta_1.P_{\text{payer}}$ fails, set $bal_{\theta_1} = 0$. If verification of $\theta_2.P_{\text{payer}}$ fails, set $bal_{\theta_2} = 0$. Abort if both failed. Then the honest party receives the entire channel balance.
   b. All verifications succeed. Select P_{payer} with the larger SN. Credit $v_1 - v_2$ to $\theta_1.sender$ and $Cap - (v_1 - v_2)$ to $\theta_2.sender$ on ledger $\mathcal{L}$. Emit (CLOSE_ACK) to both parties.
   (2) If θ_2 is the non-tumbler user, apply the same a/b logic symmetrically.

Fig. 7: The smart contract in Baker.

D. Security Analysis

We model the security of Baker in the generalized universal composability (GUC) framework [29], which extends the universal composability (UC) framework to capture global setup assumptions [30]. Unlike traditional game-based definitions, the UC framework ensures security under arbitrary concurrent executions, allowing a protocol to retain its security guarantees even when composed with other protocols or instances.

Adversary Model. We adopt the static corruption model, in which the adversary fixes the set of corrupted participants before the protocol execution begins. This set remains unchanged throughout the execution. Once a participant is corrupted, the adversary $\mathcal{A}$ obtains full control over that participant, including access to its entire internal state and secret information.

In the security model, we define both a real world and an ideal world. In the real world, parties execute according to the prescribed protocol π while interacting with a smart contract functionality, denoted by $\mathcal{C}$. The adversary can corrupt a subset of parties and fully control their behavior. In the ideal world, the real protocol π and the smart contract $\mathcal{C}$ are replaced by an ideal functionality $\mathcal{F}$, which models a trusted third party. $\mathcal{F}$ receives inputs from the parties and executes the prescribed computation accordingly. Adversarial behavior in the ideal world is captured by a PPT simulator $\mathcal{S}$, which interacts with $\mathcal{F}$. In both the real and ideal executions, an environment $\mathcal{Z}$ provides inputs to the parties and observes their outputs.

Security is defined as follows: for every PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that no PPT environment $\mathcal{Z}$ can distinguish whether it is interacting with the real-world execution $(\pi, \mathcal{A}, \mathcal{Z})$ or the ideal-world execution $(\mathcal{F}, \mathcal{S}, \mathcal{Z})$. The formal definition is given below.

Definition 5 *A protocol π in the $\mathcal{C}$-hybrid world securely realizes the ideal functionality $\mathcal{F}$ with respect to the global ledger $\mathcal{L}(\Delta)$ if for every PPT adversary $\mathcal{A}$ there exists a PPT simulator $\mathcal{S}$ such that for all PPT environments $\mathcal{Z}$, we have:*

$$\mathsf{EXEC}^{\mathcal{L}(\Delta),\mathcal{C}}_{\pi,\mathcal{A},\mathcal{Z}}(1^\lambda) \approx \mathsf{IDEAL}^{\mathcal{L}(\Delta)}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(1^\lambda)$$

where λ is the security parameter.

The formal security proof of Baker is presented in Appendix B. Below, we analyze how Baker achieves the security and privacy goals (**G1**–**G5**) under this adversary model.

(G1) Unlinkability. During the payment, the tumbler never interacts with the payee. It receives the payee's anonymized payee pocket from the payer, preventing it from identifying the payee pocket's actual owner. Furthermore, since the payee randomizes the updated signed payee pocket after each transaction, the tumbler cannot link the payee's pocket to those in previous transactions. Although the payer may temporarily reveal its own anonymized payee pocket to prove balance sufficiency, this does not compromise unlinkability as discussed in §VIII-B.

(G2) Atomicity. Baker ensures atomicity through the atomic exchange between the payer and tumbler. In Figure 6, if the tumbler fails to send the pre-signature $\hat{\sigma}_{\text{part-T}}$ or the partial signature $(Z_{\text{new}}, \hat{S}_{\text{new}}, T_{\text{new}})$, Alice will not send $\sigma_{\text{part-A}}$. If the tumbler sends incorrect information, Alice detects it during verification, and no exchange occurs. Conversely, if the tumbler is honest but Alice withholds or sends an invalid $\sigma_{\text{part-A}}$, the tumbler refuses to send $\sigma_{\text{part-T}}$. In this case, the payer cannot extract $\sigma_{\text{part-T}}$ or reconstruct σ_{new}. If the tumbler receives a valid $\sigma_{\text{part-A}}$ but refuses to respond or sends an invalid $\sigma_{\text{part-T}}$, Alice requests to close the channel. Closure requires the tumbler to submit the payer pocket with the attached aggregate signature. If the tumbler fails to submit these within the specified time, the smart contract enforces the forfeiture of the tumbler's entire channel balance. If the tumbler submits the updated payer pocket $(P_{\text{payer-new}}, \sigma_{\text{AT-new}})$, Alice can retrieve $\sigma_{\text{AT-new}}$ from the chain and compute $\sigma_{\text{part-T}}$ with $\mathsf{AAS.Split}(\sigma_{\text{AT-new}}, \sigma_{\text{part-A}})$. Furthermore, Alice can reconstruct the full signature σ_{new} with AAS.Extract and AAS.Adapt, allowing both sub-payments to succeed. If instead the original state $(P_{\text{payer}}, \sigma_{\text{AT}})$ is submitted, both sub-payments fail and no funds are lost. To prevent nonce reuse attacks, AAS employs a commit-and-reveal mechanism for all parties' nonces, similar to MuSig [31]. Consequently, in each transaction, the nonces used by the tumbler and payer to sign the updated payer pocket are unique, ensuring that the tumbler cannot produce a fresh $\sigma_{\text{part-T}}$ different from the one in line 13 of Figure 6.

(G3) Balance sufficiency. The payee can determine from the current state of the tumbler-payee channel whether the tumbler has sufficient funds to complete the payment. If not, the payee does not send its payee pocket to the payer, and the payment is never initiated. Once the payment is initiated, the tumbler verifies whether the payer has sufficient balance by inspecting the plaintext payer pocket. If this information is not enough, the payer temporarily reveals its anonymized payee pocket to the tumbler. With both pockets of the payer, the tumbler can compute the current channel balance and determine whether the payment is feasible.

(G4) Settlement correctness. Any forged pocket is invalid because it fails signature verification. Moreover, the tumbler always holds the latest valid payer pocket of each user, so any stale P_{payer} can be detected during transaction execution. When closing the channel, any conflicting payer pockets are resolved by the smart contract in favor of the one with the largest SN. Therefore, a user cannot obtain extra funds by submitting a stale P_{payer}. At any stage of Baker, if a user uses a stale P_{payee}, this only harms the user's own interests.

(G5) Abort-attack resistance. In Baker, the payee's pocket is updated solely by the tumbler without the payee's involvement. As a result, even if the tumbler aborts a transaction, it cannot determine the payee's identity.

E. Analysis on the Achievement of Goals

While §VIII-D discusses how Baker achieves its security and privacy goals, this subsection focuses on its achievement of functionality and efficiency goals

(G6) Channel bidirectionality. Baker maintains bidirectional channel balances using two pockets rather than a shared state. This design supports bidirectional channels while avoiding the challenges of conventional bidirectional channels.

(G7) Flexible amount. During transactions, the tumbler in Baker neither interacts with the payee nor learns the hidden values in the payee pocket. Consequently, even with flexible payment amounts, the tumbler cannot infer a link between the payer and the payee based on the payment amount.

(G8) NIZK-free design. In [10], [11], [14], NIZK proofs are used to prove balance sufficiency and that a value y is a valid solution to the puzzle c_y. Baker determines balance sufficiency from the pockets and eliminates the puzzle promising and solving paradigm, avoiding NIZK proofs entirely.

(G9) Minimal on-chain functionality. In Baker, on-chain dispute resolution is only required if the tumbler fails to return a valid $\sigma_{\text{part-T}}$ after receiving a valid $\sigma_{\text{part-A}}$. This can be handled by the channel-closing contract without extra logic.

IX. Performance Evaluation

We develop a prototype implementation of Baker and evaluate its performance in comparison with other PCH protocols[1].

[1] https://anonymous.4open.science/r/Baker_All_Experiments_Code

A. On-chain Cost

On-chain costs occur only when users interact with the blockchain. Since each channel is opened and closed only once, these costs are one-time expenses during its lifetime.

Implementation Details. We implement the smart contract of Baker and Accio on Ethereum using Solidity. The AAS scheme is instantiated with Schnorr signatures and an RSoC component, where RSoC is constructed over the `ALT_BN128` curve. For comparison, we also adopt the smart contract of Perun as a baseline, which is publicly available at [32].

Cost Metrics. We evaluate the on-chain cost by measuring the gas consumption during smart contract execution on Ethereum. The total gas usage reflects both the volume of data processed and the computational complexity of the contract logic. In our evaluation, we set the gas price to 3.29 Gwei and the ETH price to $2,280 USD (as of April 20, 2026).

TABLE II: On-chain gas costs for channel opening and closing required to support bidirectional payment functionality. For Baker, we measure the gas cost for different initiators.

Protocol	Open	Open Timeout	Close	Close Timeout
Baker (Tumbler)	161,417	44,670	995,570	50,345
Baker (Non-tumbler)	159,618	44,701	999,508	50,573
Accio (Bidirectional)	262,434	N.A.†	1,170,547	95,090
Perun (Optimistic)*	70,485	N.A.	163,367	N.A.
Perun (Pessimistic)*	70,485	N.A.	641,176	100,691

† N.A.: not applicable, as the operations are not supported in these protocols.

Comparison. The deployment of the Baker's smart contract costs 2.01M gas ($15.08), while deploying the Accio's smart contract costs 2.49M gas ($18.68). Since a single deployment supports an arbitrary number of channel openings and closings, our evaluation focuses on the cost of opening and closing channels. Detailed results are shown in Table II. Accio achieves bidirectional functionality by establishing two unidirectional channels. Opening these channels incurs a total cost of 262K gas, approximately 160% of the gas required to open a channel in Baker. Thus, the total cost of opening and closing in Accio is 1.43M gas, whereas Baker requires only 80% of that. For Perun, the total on-chain cost of a virtual channel is 712K gas in the pessimistic case and 234K gas in the optimistic case. In practical scenarios with N participants, Perun requires $O(N^2)$ virtual channels, whereas Baker needs only $O(N)$ payment channels. Therefore, the actual total on-chain cost of Perun is $O(N^2)\times$712K/234K gas, while Baker's total on-chain cost is $O(N)\times$1.16M gas. Baker is more cost-effective than Perun as long as $N > 10$, even under Perun's optimistic case. Overall, compared with Accio and Perun, Baker reduces the on-chain costs borne by users.

B. Off-chain Overhead

All transactions in PCH are executed off-chain, and the overhead of executing a transaction impacts overall efficiency.

Implementation Details. We implement Baker, TBPChannel [33] and Accio in Python. Additionally, we evaluate A^2L and BlindHub using their public implementations [34] and [35]. To simulate real-world deployments of PCH protocols, we deploy three independent clients: payer, payee and tumbler.

Overhead Metrics. The off-chain overhead includes communication overhead (the volume of data exchanged among participants during protocol execution) and computation overhead (the time required to execute the protocol).

Communication Overhead. We evaluate the communication overhead in a single payment for each participant, with detailed results in Table III. In Baker, the total communication overhead per payment is 6.297 KB—approximately 0.0036% of that in BlindHub and 60% of the overhead required by A^2L. The higher overheads in BlindHub and A^2L stem from their NIZK proofs and multiple interaction rounds. A^2L achieves lower overhead than BlindHub by fixing the payment amount. In contrast, Baker is NIZK-free and reduces communication rounds by limiting interaction to only the payer and the tumbler. These reduce overhead while supporting flexible amounts. The communication overhead of Baker is comparable to that of Accio and TBPChannel, although slightly higher. It is because Accio employs unidirectional channels to reduce overhead. TBPChannel omits key verification steps and cryptographic operations essential for atomicity and unlinkability in PCHs, which leads to lower communication overhead.

TABLE III: Communication overhead

Protocol	Payer	Payee	Tumbler	Total
Baker	3.595 KB	1.481 KB	1.221 KB	6.297 KB
BlindHub	55,843 KB	87,855 KB	32,018 KB	175,716 KB
TBPChannel	0.807 KB	2.11 KB	1.215 KB	4.132 KB
A^2L	2.568 KB	5.177 KB	2.383 KB	10.128 KB
Accio	2.873 KB	1.371 KB	1.222 KB	5.466 KB

Computation Overhead. We evaluate the time taken by each participant to complete a single payment, as shown in Table IV. The computation overhead in Baker is about 1.2 seconds, which is about 7% of that in BlindHub, 40% of TBPChannel, and comparable to Accio. The higher overhead in BlindHub and TBPChannel results from their computationally expensive NIZK proof schemes. Accio lowers computational cost by employing unidirectional channels, which require fewer signature verifications per payment. However, this design weakens atomicity guarantees and reduces transaction efficiency. A^2L incurs the lowest computation overhead among all protocols, largely because it fixes the payment amount. But its security guarantees are flawed, as proven in [11].

TABLE IV: Computation overhead

Protocol	Payer	Payee	Tumbler	Total*
Baker	0.6644 s	0.2951 s	0.2572 s	1.2167 s
BlindHub	4.743 s	15.799 s	9.387 s	16.594 s†
TBPChannel	0.0835 s	1.4187 s	1.3301 s	2.8323 s
A^2L	0.0397 s	0.1894 s	0.2707 s	0.3101 s†
Accio	0.5057 s	0.2584 s	0.2578 s	1.0219 s

* The total time is the duration of a payment from start to finish.

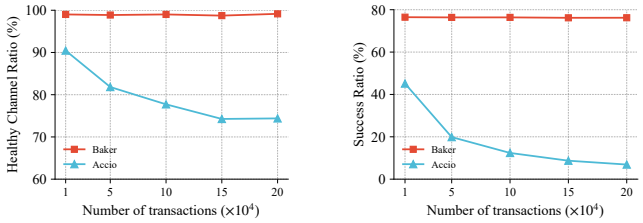
† In BlindHub and A^2L, the sum of time across roles exceeds the total time because different parties perform computations in parallel.

C. Routing Performance

To validate the limitations of unidirectional channels, we compare the routing performance of Baker with Accio.

Implementation Details. Since no PCH protocols have been deployed in practice, we evaluate routing performance using real-world data from the Lightning Network (LN) [36], a widely adopted off-chain payment system [37] for Bitcoin. LN consists of bidirectional channels and supports multi-hop payments [38]. Unlike PCH, there is no centralized hub in LN. We extract a subgraph from an LN snapshot dated April 14, 2026, which includes the highest-degree node (treated as the tumbler in our experiment) and its adjacent channels. The dataset contains 708 nodes and 707 channels with associated capacities. For bidirectional channels, capacity is evenly split between the two endpoints. For unidirectional channels, we split the capacity evenly between the two reverse channels. To simulate transactions, payer–payee pairs are randomly sampled from the subgraph, and transaction amounts are randomly generated within the channel capacity range.

Routing Metrics. We use *healthy channel ratio* and *success ratio* to evaluate the routing performance. The healthy channel ratio is the percentage of channels that remain non-exhausted (i.e., retain sufficient balance) after executing all transactions. The success ratio is the percentage of transactions that are successfully completed out of all generated transactions.



(a) Healthy channel ratio (b) Success ratio

Fig. 8: Routing Performance: Baker vs. Accio.

Comparison. Figure 8 illustrates how the healthy channel ratio and transaction success ratio evolve as the number of transactions increases. Baker consistently maintains a healthy channel ratio above 99%, indicating that channel exhaustion remains rare even under heavy transaction loads. In contrast, Accio's healthy channel ratio drops to 74% after 150,000 transactions, suggesting widespread channel exhaustion. In terms of success ratio, Baker maintains a stable transaction success ratio of 80% regardless of the number of transactions. By contrast, Accio's success ratio declines sharply as more channels become exhausted and can no longer support further transactions. On average, Baker outperforms Accio by 58% in success ratio. Moreover, an empirical study [39] shows that 70% of daily transactions in real-world off-chain payment systems occur frequently between a small number of payer–payee pairs. Under such realistic patterns, Baker's bidirectional-channel design mitigates channel exhaustion, thereby providing users with higher transaction success rates and less frequent channel rebalancing or reopening.

X. Conclusion

In this work, we present Baker, a privacy-preserving, NIZK-free, and efficient PCH that supports flexible amounts and bidirectional channels. Each non-tumbler user maintains two local

pockets to track channel balance updates. Crucially, the payee pocket of each user remains hidden from the tumbler and can be randomized along with its signature after each update, thereby ensuring both privacy and security. To achieve atomicity, we design an Aggregatable Adaptor Signature scheme for Baker. Experimental results show that Baker achieves low on-chain and off-chain costs while maintaining high transaction success rates and strong channel network stability.

REFERENCES

[1] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *2018 IEEE symposium on security and privacy (SP)*. IEEE, 2018, pp. 583–598.

[2] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten, "Arbitrum: Scalable, private smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1353–1370.

[3] R. Fernando and A. Roy, "Poster: Wip: Account zk-rollups from sum-check arguments," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 3594–3596.

[4] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostáková, M. Maffei, P. Moreno-Sanchez, and S. Riahi, "Generalized channels from limited blockchain scripts and adaptor signatures," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2021, pp. 635–664.

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

[6] V. Sivaraman, S. B. Venkatakrishnan, K. Ruan, P. Negi, L. Yang, R. Mittal, G. Fanti, and M. Alizadeh, "High throughput cryptocurrency routing in payment channel networks," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 777–796.

[7] G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi, "Concurrency and privacy with payment-channel networks," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 455–471.

[8] L. Aumayr, P. Moreno-Sanchez, A. Kate, and M. Maffei, "Blitz: Secure {Multi-Hop} payments without {Two-Phase} commits," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 4043–4060.

[9] E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg, "Tumblebit: An untrusted bitcoin-compatible anonymous payment hub," in *Network and distributed system security symposium*, 2017.

[10] E. Tairi, P. Moreno-Sanchez, and M. Maffei, "A2l: Anonymous atomic locks for scalability in payment channel hubs," in *2021 IEEE symposium on security and privacy (SP)*. IEEE, 2021, pp. 1834–1851.

[11] N. Glaeser, M. Maffei, G. Malavolta, P. Moreno-Sanchez, E. Tairi, and S. A. K. Thyagarajan, "Foundations of coin mixing services," in *Proceedings of the 2022 ACM SIGSAC conference on computer and communications security*, 2022, pp. 1259–1273.

[12] M. Minaei, P. Chatzigiannis, S. Jin, S. Raghuraman, R. Kumaresan, M. Zamani, and P. Moreno-Sanchez, "Unlinkability and interoperability in account-based universal payment channels," in *International Conference on Financial Cryptography and Data Security*. Springer, 2023, pp. 367–384.

[13] Y. Li, J. Weng, J. Lai, Y. Li, J. Sun, J. Wu, M. Li, P. Wu, and R. H. Deng, "Auditpch: Auditable payment channel hub with privacy protection," *IEEE Transactions on Information Forensics and Security*, 2024.

[14] X. Qin, S. Pan, A. Mirzaei, Z. Sui, O. Ersoy, A. Sakzad, M. F. Esgin, J. K. Liu, J. Yu, and T. H. Yuen, "Blindhub: Bitcoin-compatible privacy-preserving payment channel hubs supporting variable amounts," in *2023 IEEE symposium on security and privacy (SP)*. IEEE, 2023, pp. 2462–2480.

[15] M. Green and I. Miers, "Bolt: Anonymous payment channels for decentralized currencies," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 473–489.

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

[17] Z. Ge, J. Gu, C. Wang, Y. Long, X. Xu, and D. Gu, "Accio: Variable-amount, optimized-unlinkable and nizk-free off-chain payments via hubs," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1541–1555.

[18] "Monero," https://www.getmonero.org/, 2017.

[19] "Zcash," https://z.cash/, 2017.

[20] S. K. Mohanty and S. Tripathy, "Flexichain: Flexible payment channel network to defend against channel exhaustion attack," *ACM Transactions on Privacy and Security*, vol. 27, no. 4, pp. 1–26, 2024.

[21] W. Ni, P. Chen, L. Chen, P. Cheng, C. J. Zhang, and X. Lin, "Utility-aware payment channel network rebalance," *Proceedings of the VLDB Endowment*, vol. 17, no. 2, pp. 184–196, 2023.

[22] R. Khalil, A. Zamyatin, G. Felley, P. Moreno-Sanchez, and A. Gervais, "Commit-chains: Secure, scalable off-chain payments," *Cryptology ePrint Archive*, 2018.

[23] J. Zhang, Y. Ye, W. Wu, and X. Luo, "Boros: Secure and efficient off-blockchain transactions via payment channel hub," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 1, pp. 407–421, 2021.

[24] Z. Sui, J. K. Liu, J. Yu, M. H. Au, and J. Liu, "Auxchannel: Enabling efficient bi-directional channel for scriptless blockchains," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 138–152.

[25] P. Li, T. Miyazaki, and W. Zhou, "Secure balance planning of off-blockchain payment channel networks," in *IEEE INFOCOM 2020-IEEE conference on computer communications*. IEEE, 2020, pp. 1728–1737.

[26] M. Benedetti, F. De Sclavis, M. Favorito, G. Galano, S. Giammusso, A. Muci, and M. Nardelli, "Self-balancing semi-hierarchical payment channel networks for central bank digital currencies," in *2024 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. IEEE, 2024, pp. 530–536.

[27] Z. Ge, Y. Zhang, Y. Long, and D. Gu, "Shaduf: Non-cycle payment channel rebalancing." in *NDSS*, 2022.

[28] B. Bauer and G. Fuchsbauer, "Efficient signatures on randomizable ciphertexts," in *Security and Cryptography for Networks: 12th International Conference, SCN 2020, Amalfi, Italy, September 14–16, 2020, Proceedings 12*. Springer, 2020, pp. 359–381.

[29] R. Canetti, Y. Dodis, R. Pass, and S. Walfish, "Universally composable security with global setup," in *Theory of Cryptography: 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007. Proceedings 4*. Springer, 2007, pp. 61–85.

[30] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001, pp. 136–145.

[31] J. Nick, T. Ruffing, and Y. Seurin, "Musig2: Simple two-round schnorr multi-signatures," in *Annual International Cryptology Conference*. Springer, 2021, pp. 189–221.

[32] Perun, "Perunnetwork," https://github.com/PERUNnetwork/Perun, 2018.

[33] R. Chen, Y. Zhang, D. Li, Y. Liu, J. Liu, Q. Wu, J. Zhou, and W. Susilo, "Bitcoin-compatible privacy-preserving multi-party payment channels supporting variable amounts," *IEEE Transactions on Information Forensics and Security*, 2025.

[34] A2L, "a2l," https://github.com/etairi/a2l, 2022.

[35] BlindHub, "blind-hub," https://github.com/blind-channel/blind-hub, 2022.

[36] "Real-time lightning network," https://1ml.com/, 2017.

[37] A. Kurt, E. Erdin, K. Akkaya, S. Uluagac, and M. Cebe, "D-lnbot: A scalable, cost-free and covert hybrid botnet on bitcoin's lightning network," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 4, pp. 2162–2180, 2023.

[38] Q. Cai, J. Chen, D. Luo, G. Sun, H. Yu, and M. Guizani, "Deter-pay: A deterministic routing protocol in concurrent payment channel network," *IEEE Internet of Things Journal*, 2024.

[39] P. Wang, H. Xu, X. Jin, and T. Wang, "Flash: Efficient dynamic routing for offchain networks," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, 2019, pp. 370–381.

[40] C.-P. Schnorr, "Efficient signature generation by smart cards," *Journal of cryptology*, vol. 4, pp. 161–174, 1991.

APPENDIX A
SECURITY PROOF OF AAS

We instantiate the AAS framework using the Schnorr multi-signatures scheme [31], [40] and the RSoC scheme, and prove the security under this construction.

Aggregatable Schnorr-based Adaptor Signature. The construction is defined as follows:

- **Setup.** On input a security parameter 1^λ, it outputs the public parameters $\mathsf{pp} = (p, \hat{\mathbb{G}}, \hat{G}, H_{\mathsf{sig}}, H_{\mathsf{agg}})$, where p, $\hat{\mathbb{G}}$, and $\hat{G}$ are identical to those used in RSoC, and H_{sig} and H_{agg} are cryptographic hash functions.
- **KeyGen.** Each signer $i \in \{0, 1\}$ samples a secret key $\mathsf{sk}_i = x_i \xleftarrow{\$} \mathbb{Z}_p^*$ and sets the corresponding public key as $\mathsf{pk}_i = x_i\hat{G}$.
- **Aggregate key generation.** Let $L = (\mathsf{pk}_0, \mathsf{pk}_1)$ denote the ordered list of public keys for a fixed signer set. For each signer i, define the key-aggregation coefficient as $a_i = H_{\mathsf{agg}}(L \,\|\, \mathsf{pk}_i) \in \mathbb{Z}_p^*$. The aggregate public key is then computed as $\mathsf{PK} = a_0\mathsf{pk}_0 + a_1\mathsf{pk}_1$. This coefficient-based aggregation [31] binds each public key to the full signer list, thereby preventing rogue-key attacks.
- **Aggregate Signing and Verification.** AAS employs a commit-and-reveal nonce exchange [31] to prevent a malicious signer from adaptively choosing its nonce after observing the nonces of the honest signers. For $i \in \{0, 1\}$, each signer i samples a nonce $r_i \xleftarrow{\$} \mathbb{Z}_p^*$, computes the nonce commitment $R_i = r_i\hat{G}$, and first sends a commitment to R_i using Com. After all nonce commitments have been received, the signers reveal the corresponding R_i values. Let $R_{\mathsf{agg}} = \sum_{i=0}^{1} R_i$. The common challenge is computed as $c = H_{\mathsf{sig}}(R_{\mathsf{agg}} \,\|\, \mathsf{PK} \,\|\, m)$. Each signer outputs a signature share $\sigma_i = r_i + ca_ix_i$, which can be verified by checking whether $\sigma_i\hat{G} = R_i + ca_i\mathsf{pk}_i$. The aggregate signature is $\sigma^* = \sum_{i=0}^{1} \sigma_i$. It is verified by checking whether $\sigma^*\hat{G} = R_{\mathsf{agg}} + H_{\mathsf{sig}}(R_{\mathsf{agg}} \,\|\, \mathsf{PK} \,\|\, m)\mathsf{PK}$. The verification algorithm outputs 1 if the equation holds, and 0 otherwise.
- **Pre-signing and Verification.** In RSoC, the signature component $\hat{S} = s\hat{G}$ is a hard relation. Given an aggregate signature σ on a message m and the RSoC component $\hat{S} = s\hat{G}$, the signer generates a pre-signature $\hat{\sigma} = \sigma + s$. To verify the pre-signature, the verifier checks whether $\hat{\sigma}\hat{G} = R_{\mathsf{agg}} + H_{\mathsf{sig}}(R_{\mathsf{agg}} \,\|\, \mathsf{PK} \,\|\, m)\mathsf{PK} + \hat{S}$. If the equation holds, the pre-signature is accepted; otherwise, it is rejected.
- **Adaptation and Extraction.** The adaptor witness s is chosen to be the same scalar that defines the RSoC hard relation $(S, \hat{S}) = (sG, s\hat{G})$. Given a pre-signature $\hat{\sigma}$ and the witness s, the full signature is computed as $\sigma = \hat{\sigma} - s$. Conversely, given both the pre-signature and the full signature, one can extract the witness as $s = \hat{\sigma} - \sigma$. The extracted witness s allows the receiver to derive the missing RSoC component $S = sG$ while verifying its consistency with the public statement $\hat{S}$.

Theorem 1. *The construction is a secure AAS scheme.*

proof. The proof of this theorem consists of the following three lemmas.

Lemma 1. *The construction achieves the correctness property.*

proof. The correctness of the Schnorr digital signature scheme guarantees that signatures generated by AAS are accepted by the verification algorithm. Moreover, the correctness of the Schnorr adaptor signature scheme ensures both the adaptation correctness and the extraction correctness of AAS. For both the aggregated and split cases, correctness follows directly from the linearity of the aggregation and split operations, together with the fact that these operations preserve the Schnorr signature verification.

Lemma 2. *The construction achieves the unforgeability property.*

proof. We prove by reduction. Assume that there exists a PPT adversary $\mathcal{A}$ that wins the experiment $\mathsf{AASSignForge}_{\mathcal{A},\Pi_{\mathsf{AAS}}}$ with non-negligible probability. We then construct a PPT adversary $\mathcal{B}$ that uses $\mathcal{A}$ to forge against the EUF-CMA security of the underlying Schnorr signature scheme. The adversary $\mathcal{B}$ runs $\mathcal{A}$ as a subroutine and simulates the AAS signing-forgery experiment for it. For a fixed ordered public-key list $L = (\mathsf{pk}_0, \mathsf{pk}_1)$, the aggregate public key is computed as $\mathsf{PK} = a_0\mathsf{pk}_0 + a_1\mathsf{pk}_1$, where $a_i = H_{\mathsf{agg}}(L\|\mathsf{pk}_i)$. Hence, PK is a valid Schnorr public key in $\hat{\mathbb{G}}$. Whenever $\mathcal{A}$ makes a signing query on a message m, $\mathcal{B}$ answers it by returning a valid Schnorr signature under the aggregate public key PK. This perfectly simulates an honestly generated AAS aggregate signature, because an AAS aggregate signature has exactly the same verification equation as a Schnorr signature under PK, namely $\sigma\hat{G} = R_{\mathsf{agg}} + H_{\mathsf{sig}}(R_{\mathsf{agg}}\|\mathsf{PK}\|m)\mathsf{PK}$. Eventually, $\mathcal{A}$ outputs a purported forgery (m^*, σ^*). Since $\mathcal{A}$ wins the AAS signing-forgery experiment, σ^* is a valid AAS aggregate signature on m^* under PK, and m^* was not previously queried to the signing oracle. Therefore, $\sigma^*\hat{G} = R^*_{\mathsf{agg}} + H_{\mathsf{sig}}(R^*_{\mathsf{agg}}\|\mathsf{PK}\|m^*)\mathsf{PK}$. This is exactly the Schnorr verification equation under public key PK. Thus, $\mathcal{B}$ outputs (m^*, σ^*) as a valid Schnorr forgery. Therefore, if $\mathcal{A}$ can forge an AAS aggregate signature with non-negligible probability, then $\mathcal{B}$ can forge a Schnorr signature with non-negligible probability. This contradicts the EUF-CMA security of the Schnorr signature scheme. Hence, no PPT adversary can win the experiment $\mathsf{AASSignForge}_{\mathcal{A},\Pi_{\mathsf{AAS}}}$ with non-negligible probability.

Lemma 3. *The construction achieves the witness extractability property.*

proof. Let $\mathcal{A}$ be any PPT adversary that wins the witness-extractability experiment by outputting a valid pre-signature $\hat{\sigma}$ and a corresponding valid full signature σ for the same tuple $(m, \mathsf{PK}, R_{\mathsf{agg}}, \hat{S})$. We construct an efficient extractor $\mathcal{E}$ that recovers the adaptor witness. The extractor $\mathcal{E}$ computes $s = \hat{\sigma} - \sigma$. It remains to show that this extracted value is a valid witness for the adaptor statement $\hat{S}$. Since $\hat{\sigma}$ is valid, it satisfies $\hat{\sigma}\hat{G} = R_{\mathsf{agg}} + H_{\mathsf{sig}}(R_{\mathsf{agg}}\|\mathsf{PK}\|m)\mathsf{PK} + \hat{S}$. Since σ is a valid full aggregate signature for the same tuple, it satisfies

$\sigma\hat{G} = R_{\mathsf{agg}} + H_{\mathsf{sig}}(R_{\mathsf{agg}}\|\mathsf{PK}\|m)\mathsf{PK}$. Subtracting the two equations yields $(\hat{\sigma}-\sigma)G = \hat{S}$. Therefore, $s = \hat{\sigma}-\sigma$ satisfies $s\hat{G} = \hat{S}$ and is a valid witness for the adaptor statement. Furthermore, because the adaptor witness is chosen as the same scalar that defines the RSoC relation $(S, \hat{S}) = (sG, s\hat{G})$, the extractor can compute the missing RSoC component $S = sG$ and check that $\hat{S} = s\hat{G}$. Hence, any successful adversary in the witness-extractability experiment yields an efficiently extractable witness. Therefore, the construction satisfies witness extractability.

APPENDIX B
SECURITY PROOF OF BAKER

A. Security Model

Communication Network. Following prior work [16], [17], we assume a synchronous communication network in which protocol execution proceeds in discrete, globally synchronized rounds. The network guarantees bounded message delivery: any message sent by a party in round r is delivered to its intended party by the end of round $r+1$. Subject to this delay bound, the adversary controls the order of message delivery. All other communications (e.g., between a party and the environment, or between a party and the ideal functionality), as well as local computations, are assumed to be instantaneous and thus take 0 rounds.

Ledger Functionality $\mathcal{L}$. The underlying blockchain is modeled as a global ledger functionality $\mathcal{L}$, which is accessible by all parties, protocol instances, and ideal functionalities.

We assume that our protocol is executed among a fixed set $P = \{P_1, P_2, ..., P_n\}$. $\mathcal{L}$ supports the following operations:

- **Create.** Upon receiving (P_i, x_i) from the environment $\mathcal{Z}$, it stores x_i ($x_i \geq 0$) as the wallet funds of P_i.
- **Submit.** Upon receiving (op, P_i, v), the functionality proceeds as follows. If $op = \mathsf{add}$ and $P_i \in \mathcal{P}$ and $v \geq 0$, it sets $x_i = x_i + v$. If $op = \mathsf{sub}$ and $P_i \in \mathcal{P}$ and $v \geq 0$ and $x_i \geq v$, it sets $x_i = x_i - v$. Otherwise, it outputs fail.

Transactions are confirmed and added to the ledger state within at most Δ rounds, where Δ denotes the maximum confirmation delay. This ledger functionality $\mathcal{L}$ is denoted as $\mathcal{L}(\Delta)$. The adversary $\mathcal{A}$ cannot prevent valid transactions from being confirmed within Δ rounds. Once confirmed, ledger state transitions are immutable and publicly visible.

Smart Contract Functionality $\mathcal{C}$. We model the smart contract as an ideal functionality $\mathcal{C}$ on $\mathcal{L}(\Delta)$, resulting in a $\mathcal{C}$-hybrid world. $\mathcal{C}$ supports the same operations as shown in Figure 7. All state transitions executed by $\mathcal{C}$ are deterministic, publicly verifiable, and recorded on $\mathcal{L}(\Delta)$ upon confirmation.

B. Ideal Functionality of Baker

We define the ideal functionality $\mathcal{F}_{\mathsf{Baker}}$ in the $\mathcal{L}(\Delta)$-hybrid model. The protocol involves a designated tumbler P_T and a set of non-tumbler users. The honest parties are modeled as dummy parties that simply forward inputs from the environment to the functionality and return outputs from the functionality to the environment. For every corrupted party P, the functionality accepts inputs only from the simulator $\mathcal{S}$ on behalf of P, and any output intended for P is delivered to $\mathcal{S}$. The functionality maintains the following state:

A payment channel γ is defined as the tuple $\gamma = (\gamma.\mathsf{id}, \gamma.\mathsf{user}, \gamma.\mathsf{initbal}, \gamma.\mathsf{Cap}, \gamma.\mathsf{vpayer}, \gamma.\mathsf{vpayee})$. Here, $\gamma.\mathsf{id} \in \{0,1\}^*$ is the channel identifier, and $\gamma.\mathsf{user}$ denotes the identity of the non-tumbler user in the channel. The value $\gamma.\mathsf{initbal} \in \mathbb{R}_{\geq 0}$ represents the initial balance of the non-tumbler user. The $\gamma.\mathsf{Cap} \in \mathbb{R}_{>0}$ denotes the total channel capacity, defined as the sum of the initial balances of both parties. The $\gamma.\mathsf{vpayer} \in \mathbb{R}_{\geq 0}$ records the total amount paid by the non-tumbler user to the tumbler when acting as a payer in this channel, and is initialized to 0. The $\gamma.\mathsf{vpayee} \in \mathbb{R}_{\geq 0}$ records the total amount received by the non-tumbler user from the tumbler when acting as a payee, and is initialized to $\gamma.\mathsf{initbal}$.

During the channel opening and closing phases, the message θ is defined as the tuple $(\theta.\mathsf{sender}, \theta.\mathsf{party}, \theta.\mathsf{id}, \theta.\mathsf{balance}, \theta.\mathsf{sn}, \theta.\mathsf{vpayer}, \theta.\mathsf{vpayee})$. Here, $\theta.\mathsf{sender}$ denotes the message sender, and $\theta.\mathsf{party}$ denotes the other party in the channel. The value $\theta.\mathsf{balance} \in \mathbb{R}_{\geq 0}$ represents the initial balance of the message sender. The $\gamma.\mathsf{sn} \in \mathbb{Z}_{\geq 0}$ is the serial number of the latest channel state. The $\theta.\mathsf{id}$, $\theta.\mathsf{vpayer}$ and $\theta.\mathsf{vpayee}$ correspond to $\gamma.\mathsf{id}$, $\gamma.\mathsf{vpayer}$ and $\gamma.\mathsf{vpayee}$.

Channel Opening. Upon receiving an opening request of the form $(\mathsf{OPEN\_REQ}, \theta_1)$ from party $P_1 = \theta_1.\mathsf{sender}$, the functionality proceeds as follows:

If θ_1 is malformed, if $\theta_1.\mathsf{id}$ has already been used by an open or pending channel, or if P_1 does not have sufficient funds $\theta_1.\mathsf{balance}$ on $\mathcal{L}(\Delta)$, then the functionality ignores the request. Otherwise, within Δ rounds, sends $(\mathsf{sub}, P_1, \theta_1.\mathsf{balance})$ to $\mathcal{L}(\Delta)$, and records the request as pending.

The functionality then waits for at most Δ rounds for a matching request $(\mathsf{OPEN\_REQ}, \theta_2)$ from $P_2 = \theta_1.\mathsf{party}$. The request θ_2 is matching if it is well-formed, $\theta_2.\mathsf{sender} = P_2$, $\theta_2.\mathsf{party} = P_1$, and $\theta_2.\mathsf{id} = \theta_1.\mathsf{id}$. If no matching valid request is received within Δ rounds, or if P_2 does not have sufficient available funds $\theta_2.\mathsf{balance}$ on $\mathcal{L}(\Delta)$, the functionality sends $(\mathsf{add}, P_1, \theta_1.\mathsf{balance})$ to $\mathcal{L}(\Delta)$, removes the pending opening request, and outputs $(\mathsf{OPEN\_ABORT}, \theta_1.\mathsf{id})$ to P_1.

Otherwise, within Δ rounds the functionality sends $(\mathsf{sub}, P_2, \theta_2.\mathsf{balance})$ to $\mathcal{L}(\Delta)$ and creates a channel γ. Finally, the functionality outputs $(\mathsf{OPEN\_ACK}, \gamma)$ to both P_1 and P_2, removes the pending opening request, and terminates the channel opening.

Payment Execution. The payment functionality involves three parties: the payer P_A, the tumbler P_T, and the payee P_B. The functionality maintains an internal record of pending payment requests.

Upon receiving $(\mathsf{payeereq}, P_A, P_{target}, amt)$ from P_B or from $\mathcal{S}$ on behalf of a corrupted P_B (where $P_{target} = P_{actual}$ for some corrupted party P_{actual} colluding with a corrupted P_B and $P_{target} = P_B$ otherwise), the functionality finds the corresponding channel γ_1 such that $\gamma_1.\mathsf{user}$ is P_{target} and proceeds as follows.

- If no such channel γ_1 exists, or if $\gamma_1.\mathsf{Cap} - (\gamma_1.\mathsf{vpayee} - \gamma_1.\mathsf{vpayer}) < amt$, the functionality ignores the request.
- If P_A is corrupted, sends $(\mathsf{payeereq}, P_A, P_{target}, amt)$ to $\mathcal{S}$, capturing the information available to the corrupted payer. Otherwise, sends $(\mathsf{payeereq}, \perp, \perp, \perp)$ to $\mathcal{S}$.
- The functionality records a pending payment request from P_B for target P_{target} and waits for matching requests from the other parties for at most $6 + 2\Delta$ rounds. If no matching requests are received within this period, the functionality aborts the payment.

Upon receiving $(\mathsf{payerreq}, P_B, amt)$ from P_A or from $\mathcal{S}$ on behalf of a corrupted P_A, the functionality finds the corresponding channel γ_2 such that $\gamma_2.\mathsf{user}$ is P_A and proceeds as follows.

- If no such channel γ_2 exists or $\gamma_2.\mathsf{vpayee} - \gamma_2.\mathsf{vpayer} < amt$, the functionality ignores the request.
- The functionality checks whether there exists a pending payee-side request. If no such request exists, it ignores the request.
- If P_B is corrupted, sends $(\mathsf{payerreq}, P_A, P_B, amt)$ to $\mathcal{S}$. Otherwise, sends $(\mathsf{payerreq}, P_A, \perp, amt)$ to $\mathcal{S}$.
- The functionality records a pending payment request (P_A, P_B, amt) from P_A intended for P_B and waits for matching requests from the other parties for at most $6 + 2\Delta$ rounds. If no matching requests are received within this period, the functionality aborts the payment.

Upon receiving $(\mathsf{payreq}, P_A, P'_B, amt)$ from P_T (where $P'_B \in \{P_B, \perp\}$) or from $\mathcal{S}$ on behalf of a corrupted P_T, the functionality proceeds as follows.

- If matching pending requests from both P_A and P_B exist (with P_A targeting P_B, and P_B specifying P_{target}) within the waiting period $6+2\Delta$, the functionality atomically updates the channel states by setting $\gamma_1.\mathsf{vpayee} = \gamma_1.\mathsf{vpayee} + amt$ (for γ_1 of P_{target}) and $\gamma_2.\mathsf{vpayer} = \gamma_2.\mathsf{vpayer} + amt$. If no such matching pending requests exist, aborts.
- If P_A is honest, outputs $(\mathsf{paysuccess}, P_A, P_B, amt)$ to P_A. If P_B is honest, outputs $(\mathsf{paysuccess}, P_A, P_B, amt)$ to P_B. If P_T is honest, outputs $(\mathsf{paysuccess}, P_A, \perp, amt)$ to P_T. If P_A is corrupted or P_B is corrupted, the functionality outputs $(\mathsf{paysuccess}, P_A, P_B, amt)$ to $\mathcal{S}$. Otherwise, if P_T is corrupted, the functionality outputs $(\mathsf{paysuccess}, P_A, \perp, amt)$ to $\mathcal{S}$.
- The functionality removes the corresponding pending payment records.

Channel Closing. Upon receiving a channel closing request $(\mathsf{CLOSE\_REQ}, \theta_1)$ from party $P_1 = \theta_1.\mathsf{sender}$ or from $\mathcal{S}$ on behalf of a corrupted P_1, the functionality proceeds as follows:

If no channel instance with identifier $\theta_1.\mathsf{id}$ exists, or if P_1 is not a party of the corresponding channel, then ignores the request. Otherwise, the functionality marks the channel γ as closing. Within Δ rounds outputs $(\mathsf{CLOSING}, \theta_1.\mathsf{id})$ to P_2 (where $P_2 = \theta_1.\mathsf{party}$) if P_2 is honest, or to $\mathcal{S}$ on behalf of P_2 otherwise.

If no valid closing request $(\mathsf{CLOSE\_REQ}, \theta_2)$ from P_2 or $\mathcal{S}$ corresponding to the same $\theta_1.\mathsf{id}$ is received within $6 + 2\Delta$ rounds, the functionality sends $(\mathsf{add}, P_1, \gamma.Cap)$ and $(\mathsf{add}, P_2, 0)$ to $\mathcal{L}(\Delta)$ and aborts the channel closing.

Otherwise, upon receiving a valid $(\mathsf{CLOSE\_REQ}, \theta_2)$ from P_2 or from $\mathcal{S}$, the functionality determines the final balances $(\mathsf{bal}_1, \mathsf{bal}_2)$ according to the latest channel state encoded in θ_1 and θ_2. The functionality then sends $(\mathsf{add}, P_1, \mathsf{bal}_1)$ and $(\mathsf{add}, P_2, \mathsf{bal}_2)$ to $\mathcal{L}(\Delta)$. If P_1 is honest, outputs $(\mathsf{CLOSE\_ACK}, \gamma)$ to P_1. If P_2 is honest, outputs $(\mathsf{CLOSE\_ACK}, \gamma)$ to P_2. Otherwise, outputs $(\mathsf{CLOSE\_ACK}, \gamma)$ to $\mathcal{S}$. The functionality terminates the channel closing.

Since Baker requires 6 communication rounds to complete a payment, the functionality waits for at most $6 + 2\Delta$ rounds for the corresponding messages.

C. Simulator Construction

We construct a PPT simulator $\mathcal{S}$ that interacts with the ideal functionality $\mathcal{F}$ and simulates the view of any real-world adversary $\mathcal{A}$ for the environment $\mathcal{Z}$. The involved parties are P_A, P_B and P_T, each of which may be corrupted. Requests issued by corrupted parties may be forged, malformed, or invalid. The simulator internally emulates the protocol execution for all corrupted parties. Specifically, the simulator emulates the Setup phase by choosing a random trapdoor $k \in \mathbb{Z}_p^*$ and setting $P = kG$ for RSoC.

Channel Opening. Upon receiving an opening request of the form $(\mathsf{OPEN\_REQ}, \theta)$ from a corrupted party P, the simulator forwards the request to $\mathcal{F}$. If $\mathcal{F}$ ignores it, $\mathcal{S}$ outputs nothing.

If $\mathcal{F}$ outputs $(\mathsf{OPEN\_ACK}, \gamma)$, the simulator records the channel γ and sends $(\mathsf{OPEN\_ACK}, \gamma)$ to the corrupted party. If the corrupted party is P_T, $\mathcal{S}$ outputs $\sigma_N = \mathsf{AAS.Sign}(\mathsf{sk}_N, 0||H(\gamma.id)||0)$ to the corrupted party, where $\mathsf{AAS.SignVerify}(\sigma_N, 0||H(\gamma.\mathsf{id})||0, \mathsf{pk}_N) = 1$. If the corrupted party is the non-tumbler user, the simulator outputs $(P_{payee}, r) = (\mathsf{ComGen}(H(\gamma.\mathsf{id})||\gamma.\mathsf{initbal}, r), r)$, $\sigma = \mathsf{ComSign}(P_{payee}, \mathsf{sk})$ and $\sigma_T = \mathsf{AAS.Sign}(\mathsf{sk}_T, P_{payer})$ to the corrupted party, which can be verified through the corresponding verification algorithms.

Payment Execution. We analyze the payment execution based on the corruption combinations of the involved parties. Note that the cases where all parties are honest or all parties are corrupted are trivial, as $\mathcal{S}$ either passively relays messages or acts as a transparent translator to ensure perfect indistinguishability. For the remaining non-trivial scenarios, we consider the following six cases:

Case 1: Only P_A is corrupted.

Upon receiving $(\mathsf{payeereq}, P_A, P_B, amt)$ from $\mathcal{F}$, $\mathcal{S}$ generates a valid (P_{payee}, σ, amt) to P_A on behalf of P_B. Upon P_A sending the payment request to P_T, $\mathcal{S}$ ignores it if signatures are invalid or P_A has insufficient balance. Otherwise, $\mathcal{S}$ runs ComUpd and $\mathsf{ComSign}$ to generate $P_{payee-new}$ and its signature $\hat{\sigma}_{new} = (Z_{new}, S_{new}, \hat{S}_{new}, T_{new})$. Then $\mathcal{S}$ computes the

adaptor signature $\hat{\sigma}_{part-T}$ on the updated payer pocket and sends $(\hat{\sigma}_{part-T}, P_{payee-new}, Z_{new}, \hat{S}_{new}, T_{new})$ to P_A.

Upon receiving σ_{part-A} from P_A, $\mathcal{S}$ aborts if σ_{part-A} is invalid. Otherwise, $\mathcal{S}$ submits $(\mathsf{payerreq}, P_B, amt)$ to $\mathcal{F}$ on behalf of P_A, and returns the full AAS signature σ_{part-T} to P_A. If P_A subsequently forwards the updated pocket to P_B, $\mathcal{S}$ runs UpdVf using P_T's RSoC public key.

Case 2: Only P_B is corrupted.

Upon receiving the payment request (P_{payee}, σ, amt) from P_B destined for P_A, $\mathcal{S}$ uses the trapdoor information to extract the underlying actual identity P_{actual} and validity from P_{payee}. If the signature verification fails, $\mathcal{S}$ aborts.

When $\mathcal{F}$ receives the corresponding payer-side request from honest P_A and leaks the allowed information to $\mathcal{S}$, $\mathcal{S}$ submits $(\mathsf{payeereq}, P_A, P_{target}, amt)$ to $\mathcal{F}$ on behalf of P_B, and then waits for P_T to submit $(\mathsf{payreq}, P_A, P_B, amt)$ to $\mathcal{F}$ to execute the transfer.

Upon receiving $\mathsf{paysuccess}$ from $\mathcal{F}$, $\mathcal{S}$ executes ComUpd and $\mathsf{ComSign}$ to generate the valid $(P_{payee-new}, \sigma_{new})$ and sends them to P_B on behalf of P_A.

Case 3: Only P_T is corrupted.

Upon receiving $(\mathsf{payerreq}, P_A, \perp, amt)$ from $\mathcal{F}$ (initiated by P_A), $\mathcal{S}$ runs ComRdm on an arbitrary honest payee's current pocket to generate a dummy pocket (P_{sim}, σ_{sim}) and sends it with (P_{payer}, σ_{AT}) to P_T on behalf of P_A (simulating the payment request).

Upon receiving the response $(\hat{\sigma}_{part-T}, P'_{sim}, (Z'_{sim}, \hat{S}'_{sim}, T'_{sim}))$ from P_T, $\mathcal{S}$ aborts if signatures are invalid. Otherwise, $\mathcal{S}$ computes the AAS partial signature σ_{part-A} using sk_A and sends it to P_T.

Upon receiving the full signature σ_{part-T} from P_T, $\mathcal{S}$ verifies it. If valid, $\mathcal{S}$ submits $(\mathsf{payreq}, P_A, \perp, amt)$ to $\mathcal{F}$ on behalf of P_T (committing the payment).

Case 4: P_A and P_B are corrupted, but P_T is honest.

Upon P_A sending the payment request to P_T, $\mathcal{S}$ ignores it if signatures are invalid or P_A has insufficient balance. Otherwise, $\mathcal{S}$ runs ComUpd and $\mathsf{ComSign}$ to generate $P_{payee-new}$ and its signature $\hat{\sigma}_{new} = (Z_{new}, S_{new}, \hat{S}_{new}, T_{new})$. Then $\mathcal{S}$ computes the adaptor signature $\hat{\sigma}_{part-T}$ on the updated payer pocket and sends $(\hat{\sigma}_{part-T}, P_{payee-new}, Z_{new}, \hat{S}_{new}, T_{new})$ to P_A on behalf of P_T.

Upon receiving σ_{part-A} from P_A, $\mathcal{S}$ aborts if σ_{part-A} is invalid. Otherwise, $\mathcal{S}$ extracts the underlying identity P_{target} from the invoice used by P_A. $\mathcal{S}$ submits $(\mathsf{payerreq}, P_B, amt)$ to $\mathcal{F}$ on behalf of P_A, and submits $(\mathsf{payeereq}, P_A, P_{target}, amt)$ to $\mathcal{F}$ on behalf of P_B.

Case 5: P_A and P_T are corrupted, but P_B is honest.

Upon receiving $(\mathsf{payeereq}, P_A, P_B, amt)$ from $\mathcal{F}$, $\mathcal{S}$ sends the invoice (P_{payee}, σ, amt) to P_A on behalf of P_B.

Upon receiving the updated pocket $(P_{payee-new}, \sigma_{new})$ destined for P_B from P_A, $\mathcal{S}$ runs $\mathsf{UpdVf}(P_{payee}, P_{payee-new}, \sigma_{new}, amt, vk)$, where vk is the tumbler's public verification key. If valid, $\mathcal{S}$ submits $(\mathsf{payerreq}, P_B, amt)$ and $(\mathsf{payreq}, P_A, \perp, amt)$ to $\mathcal{F}$ on behalf of P_A and P_T respectively.

Case 6: P_T and P_B are corrupted, but P_A is honest.

Upon the corrupted P_B sending (P_{payee}, σ, amt) to P_A, $\mathcal{S}$ runs $\mathsf{ComSignVf}(P_{payee}, \sigma, vk)$ to verify. $\mathcal{S}$ aborts if the output is 0.

Upon $\mathcal{F}$ notifying $\mathcal{S}$ of $(\mathsf{payerreq}, P_A, P_B, amt)$, $\mathcal{S}$ constructs $msg = (P_{payee}, \sigma, amt, P_{payer}, \sigma_{AT})$. $\mathcal{S}$ signs σ_{req} on msg and sends (msg, σ_{req}) to P_T on behalf of P_A.

Upon receiving $(\hat{\sigma}_{part-T}, P_{payee-new}, Z_{new}, \hat{S}_{new}, T_{new})$ from P_T, the simulator $\mathcal{S}$ computes $P_{payer-new} = P_{payer} + (1 \,\|\, amt)$. The simulator $\mathcal{S}$ verifies the response by checking if $\mathsf{AAS.AdaptVerify}(\mathsf{pk}_T,\ P_{payer-new}, \hat{\sigma}_{part-T}, \hat{S}_{new}) = 1$ and $\mathsf{ComSignPartVf}(P_{payee}, P_{payee-new}, Z_{new}, \hat{S}_{new}, T_{new}, amt, vk) = 1$. If both verifications succeed, the simulator $\mathcal{S}$ generates σ_{part-A} and sends it to P_T. It then submits $(\mathsf{payreq}, P_A, \perp, amt)$ and $(\mathsf{payeereq}, P_A, P_B, amt)$ to $\mathcal{F}$ on behalf of P_T and P_B respectively. Otherwise, $\mathcal{S}$ aborts and submits nothing.

Channel Closing. Upon receiving a closing request of the form $(\mathsf{CLOSE\_REQ}, \theta)$ from a corrupted party P, $\mathcal{S}$ checks whether θ is well formed and whether the submitted payer/payee records are valid authenticated records associated with the channel identifier $\theta.\mathsf{id}$. In particular, $\mathcal{S}$ verifies the corresponding signatures, serial numbers, and channel identifiers exactly as the real contract $\mathcal{C}$ would do. If any check fails, $\mathcal{S}$ ignores the request. Otherwise, $\mathcal{S}$ forwards $(\mathsf{CLOSE\_REQ}, \theta)$ to $\mathcal{F}$ on behalf of the corrupted party P.

Upon receiving $(\mathsf{CLOSING}, \theta_1.\mathsf{id})$ from $\mathcal{F}$, $\mathcal{S}$ forwards the message to the corrupted party if it is not the channel-closing initiator. Upon receiving $(\mathsf{CLOSE\_ACK}, \gamma)$ from $\mathcal{F}$, $\mathcal{S}$ simulates the pocket events emitted by $\mathcal{C}$ and outputs the closing message to the corrupted party.

D. Theorem

Theorem 2. *Our protocol Baker running in the $\mathcal{C}$-hybrid world securely realizes the ideal functionality $\mathcal{F}$ with respect to $\mathcal{L}(\Delta)$.*

proof (sketch). We provide a proof sketch below:

Channel Opening. In the real world, channel opening is realized through a smart contract deployed on the blockchain. In the ideal world, the simulator $\mathcal{S}$ forwards all opening requests issued by corrupted parties to $\mathcal{F}$ and simulates the corresponding opening acknowledgments, initial payer/payee pockets, commitments, and signatures. Since these objects are generated using the same algorithms as in the real protocol, their distributions are identical to those in the real execution.

Payment Execution. The simulator $\mathcal{S}$ simulates all honest-party messages sent to corrupted parties. Valid payer-side and payee-side requests are mapped to the corresponding ideal inputs, whereas malformed requests, invalid signatures, invalid pocket updates, and requests with insufficient balances are rejected exactly as in the real protocol. When the payee is honest and the tumbler is corrupted, $\mathcal{S}$ simulates the payee-side pocket using a randomized commitment-signature pair. By the hiding and randomizability of RSoC, this simulated pocket is computationally indistinguishable from a real randomized payee pocket. Thus, the corrupted tumbler learns only the leakage allowed by $\mathcal{F}$, namely the payer identity and the

amount, but not the honest payee's identity. The atomicity of the payment follows from the ideal functionality and from the AAS-based exchange in the real protocol. Any successful deviation would imply either a forgery against AAS or a violation of the RSoC update-verification guarantees.

Channel Closing. The simulator $\mathcal{S}$ verifies closing states submitted by corrupted parties exactly as $\mathcal{C}$ does and forwards only valid closing requests to $\mathcal{F}$. During the challenge period, $\mathcal{S}$ simulates the same on-chain events and accepts only valid authenticated state records. The final settlement produced by $\mathcal{F}$ follows the same state-selection rule as $\mathcal{C}$: the latest valid payer and payee records determine the final balances. Therefore, the simulated closing transcript and pocket events are distributed as in the real execution.

Overall, the simulated messages generated for corrupted parties are either identically distributed to the real messages or computationally indistinguishable by the hiding and randomizability of RSoC. Invalid adversarial messages are rejected in both worlds, and valid adversarial messages induce the same state transitions in $\mathcal{F}$ as in the real execution. Hence, any environment distinguishing the two executions with non-negligible probability would yield either a forgery against the underlying AAS/RSoC authentication mechanisms, a violation of RSoC hiding or randomizability, or a discrepancy between the contract $\mathcal{C}$ and the ideal functionality $\mathcal{F}$. All these events occur only with negligible probability.

Therefore, Baker running in the $\mathcal{C}$-hybrid world securely realizes $\mathcal{F}$ with respect to $\mathcal{L}(\Delta)$.