IvyAPC: Auditable Generalized Payment Channels

Ming Li[1], Yuxian Li(✉)[1,2], Jian Weng[1], Yingjiu Li[3], Jiasi Weng[1], Junzuo Lai[1], Robert H. Deng[2]

[1] Jinan University, Guangzhou, China
{limjnu,liyuxianjnu,cryptjweng,Wengjiasi,junzuolai}@gmail.com
[2] Singapore Management University, Singapore robertdeng@smu.edu.sg
[3] University of Oregon,Eugene,USA yingjiul@uoregon.edu.

Abstract. Payment channels (PCs) are instrumental in enhancing blockchain scalability. As PCs become more prevalent, the imperative for independent and robust auditing mechanisms grows. Despite the critical need, there has been no extensive research on auditing PCs to ensure provable security. Challenges include maintaining global consensus and chronological integrity of off-chain transactions. Moreover, collusive parties pose a threat by potentially launching attacks that could disrupt the auditor's ability to verify transaction integrity.
This paper introduces IvyApc, the novel protocol designed for the auditable PC framework. IvyApc addresses the aforementioned challenges through two innovative techniques: (i) Accountable Assertions with Flexible Public Keys: This mechanism imposes penalties on parties attempting collusion during the audit process; (ii) Chain-Linking of Off-Chain Transactions: It guarantees a verifiable sequence of transactions, safeguarding against tampering within PCs. We validate the IvyApc protocol within the Universal Composability framework, demonstrating its adherence to the security prerequisites of completeness and soundness for auditing purposes. A prototype of IvyApc has been developed and tested for compatibility with Bitcoin's PC infrastructure, showcasing its practical applicability.

Keywords: Payment channel · Auditability · Layer-two solution.

1 Introduction

Auditing, essential for the external review of financial records, now extends to blockchain-based cryptocurrency transactions to meet finance's evolving needs [18,17,36,37,29,14,15,35]. Regulatory demands, such as New Jersey-AB 320 in Blockchain 2021 Legislation[4] prompt top accounting firms like Pricewaterhouse-Coopers (PwC) [39] and Ernst & Young (EY) [21] to invest in tools like EY's Blockchain Analyzer for transaction analysis [10].

[4] https://www.ncsl.org/research/financial-services-and-commerce/blockchain-2021-legislation.aspx

Despite extensive research on blockchain auditing, the focus remains narrow on on-chain transactions, overlooking off-chain transactions. This is particularly evident in Payment Channels (PCs) that allow multiple, unrecorded cryptocurrency exchanges, highlighting the necessity for expanded auditing practices [2,6,5,38,43,7,40,24,25]. The popularity of PCs like the Lightning [2] and Raiden Networks [4], their adoption in both academia and industry, underscores the need for their auditability.

Guidelines from the audit principle mentioned in Public Company Accounting Oversight Board [3] mandate auditors to ensure the completeness of annual financial reports and internal controls. This includes off-chain transactions, integral to blockchain finance, marking a departure from traditional audits due to significant challenges in auditing PCs, which is critical yet complex [3].

1.1 Challenges and Contributions

Lack of Order Information in Off-Chain Transactions. Most current PC protocols do not provide order information such as accurate timestamps for off-chain transactions, a feature not necessarily required by the protocols themselves but crucial for auditing purposes.

Lack of Global Consensus for Off-Chain Transactions. Off-chain transactions do not undergo the public consensus verification of blockchain, which is crucial for ensuring transaction validity on the blockchain [23]. This difference makes auditing off-chain transactions difficult, as off-chain transactions are verified locally by involved parties and auditors find it hard to collect all transactions.

Potential Collusions Among Transacting Parties. In PCs, the danger of collusion exists among transaction parties, who may submit incomplete off-chain transaction records to auditors, facilitating tax evasion and other malpractices. In extreme instances, they might only reveal the channel's opening and closing transactions, concealing all intermediate off-chain transactions. Further, collusive parties may alter the orders or contents of off-chain transactions or inject false off-chain transactions (e.g., fictitious charitable donations) into a PC.

To address these challenges, we propose IvyAPC, a universal and auditable PC framework incorporating generalized channels, a concept widely adopted in existing literature [5]. Our primary innovation lies in devising a strategy that compels transaction parties to accurately report off-chain transactions to auditors. This involves structuring off-chain transactions within a hash chain and creating a non-equivocation protocol that maintains hash chain validity, even amidst colluding attacks from transaction parties. Moreover, this non-equivocation protocol offers significant adaptability while using various payment systems, as discussed in Section 3. In the following, the contributions of this work are summarized:

– **Auditable generalized PC constructions**. We present IvyAPC, a protocol for auditable generalized PCs that enables auditors to detect misconduct in off-chain transactions by colluding parties. We establish a formal audit framework, emphasizing two crucial security features: completeness and succinctness (see Section §2.1). The core component of IvyAPC is the Accountable Assertions with a Flexible Public Key (AAFPK) mechanism, which is designed

to create auditable off-chain transactions. Unlike traditional accountable assertions, AAFPK utilizes versatile public keys to support multiple assertions within the same context without compromising secret keys, thereby enhancing non-equivocation in decentralized systems.
- **Formal security analysis**. We model IvyAPC in the Universal Composability (UC) framework and prove its security properties in the UC framework. In particular, we solve the problem that was left in CCS'15 [42] regarding the composability of accountable assertions.
- **Implementation and Evaluation**. We implemented a prototype of IvyAPC [16]. Our experimental results indicate that, compared to generalized channels, IvyAPC requires an additional computation cost of less than 0.5 seconds and an additional communication cost of less than 9KB per payment.

2 Overview

In this section, we start by introducing an audit model for personal computers and formalize the IvyAPC protocol.

2.1 Audit Model for PCs

The main participants in IvyAPC are composed of two roles: (i) transacting parties $\mathcal{P} := \{A, B, ...\}$, i.e., payers or payees, and (ii) auditor $\mathcal{D}$, e.g., an accounting firm such as Deloitte or PWC. In particular, $\mathcal{D}$ performs an audit after a channel has been closed and requires accessing off-chain transactions from transacting parties according to regulation policies (e.g., for tax checks). Compared with traditional information systems auditing that concentrates on *who* participate in financial activities and *when* did they reach an agreement on *what* [33,31,15], the audit model in IvyAPC also considers the total *income* or *spend* which relies on the order of off-chain transactions. The contents of auditing in PCs can be illustrated as the following three objects: **(1) The parties** refers to the parties who participate in PCs to perform off-chain transactions. **(2) The timestamp** is a critical factor representing the parties performing the transaction activity over time. It refers to the moment that an off-chain transaction is generated between transacting parties[5]. **(3) The data** can illustrate the consensus results (e.g., income or expenditure) that transacting parties have achieved. It mainly refers to the outcomes reached by parties in PCs, e.g., token exchange.

IvyAPC extends the basic PC protocol by adding the Audit operation as follows. Specifically, an *audit trail* is generated by transacting parties and refers to the necessary auditing information for an off-chain transaction.

Definition 1 (Auditable Payment Channel). *An auditable payment channel is defined as a tuple:* $\gamma := (\gamma.id, \gamma.us, \gamma.cash, \gamma.state, \gamma.ctList)$*, where* $\gamma.ctList := \big((\mathcal{T}_{\mathsf{off}_0}.tid, id_0), ..., (\mathcal{T}_{\mathsf{off}_n}.tid,\ id_n)\big)$ *refers to a list of off-chain transactions with*

[5] Since an un-published transaction does not have the attribution of timestamp, we thus make use of block timestamp to mark the timestamp of un-broadcast commitment transactions (see discussion §D.1 for details).

the index that happened in a PC. IvyAPC *is defined with respect to a blockchain* $\mathcal{L}$*, transacting parties* $\gamma.us := \{A, B\}$*, and an auditor* $\mathcal{D}$*. It proceeds with the five operations (*Create, Update, Close, Punish, Audit*):*

- Create$(A, B, x_A, x_B) \rightarrow 1/0$*. Two parties A and B collaboratively generate a funding transaction* $\mathcal{T}_{\mathsf{fund}}$ *which has two deposits* x_A *and* x_B*, respectively. It initializes an empty list ctList to store revoked commitment transactions. If* $\mathcal{T}_{\mathsf{fund}}$ *is published on blockchain* $\mathcal{L}$*, then an auditable payment channel* $\gamma := (\gamma.id, \gamma.us,\ \gamma.cash, \gamma.state, \gamma.ctList)$ *is created successfully by outputting 1; otherwise, output 0.*
- Update$(\gamma.id, \widetilde{state}) \rightarrow 1/0$*. Upon input an update state* $\widetilde{state} := (\widetilde{\theta}_1, ..., \widetilde{\theta}_l)$*, two parties collaboratively generate a new commitment transaction* $\mathcal{T}_{\mathsf{com}}$ *with* $\widetilde{state}$*. If succeed, then add* $(\mathcal{T}_{\mathsf{off}}, id)$ *to ctList and output 1, where id refers to the index of* $\mathcal{T}_{\mathsf{off}}$*; otherwise, output 0.*
- Close$(\gamma.id) \rightarrow 1/0$*. Upon input the identifier* $\gamma.id$ *of an* APC*, this operation checks if it is authorized by both parties. If yes, output 1; else, output 0.*
- Punish$(\gamma.id)$*. This operation is executed at the end of each round. It checks if some output of* $\mathcal{T}_{\mathsf{fund}}$ *has been spent. It punishes some party if it publishes a revoked commitment transaction or does not generate an audit trail following the* IvyAPC *protocol.*
- Audit$(\gamma.id, \mathcal{D}) \rightarrow 1/0$*. This operation is performed after* γ *has been closed. Upon receiving the audit information from both transacting parties, the auditor* $\mathcal{D}$ *verifies the validation of these information and reconstructs a final audit trail.* $\mathcal{D}$ *compares it with the published transaction on* $\mathcal{L}$*, and outputs 1 if it is equal; otherwise, it outputs 0.*

Particularly, a closed off-chain payment channel is *auditable* means that an auditor outputs "success (1)" if the payment channel satisfies the auditability requirements, and outputs "failure (0)" otherwise. This definition implies that an auditor can detect any changes made by rational users to the history of off-chain transactions committed in the channel, where the changes can be made either before or after the channel is closed. Note that we do not consider irrational users who may collude regardless of losing all their coins from a channel; such users may change transaction history before a channel is closed without being detected by an auditor.

Assume that A and B commit a sequence of off-chain transactions $(\mathcal{T}_{\mathsf{com}_0}, \ldots, \mathcal{T}_{\mathsf{com}_n})$ in a GC γ funded by $\mathcal{T}_{\mathsf{fund}}$ and closed in $\mathcal{T}_{\mathsf{com}_n}$. The corruption and collusion of the payer/payee P_1 or P_2 may present a modified history of off-chain transactions to an auditor $\mathcal{D}$. Following the current GC protocol [5], auditor $\mathcal{D}$ *cannot* detect any changes made to the original history of off-chain transactions (including changing of their orders, changing or hiding any subset of them, and inserting any new off-chain transactions among them) as long as the modified history is consistent with $\mathcal{T}_{\mathsf{fund}}$ and $\mathcal{T}_{\mathsf{com}_n}$; that is, each transaction in the modified history spends from the output of $\mathcal{T}_{\mathsf{fund}}$ and the last one of them updates the channel to the same state as $\mathcal{T}_{\mathsf{com}_n}$ does. Note that a change of the history of committed off-chain transactions in γ may be made either before or after ζ is

closed. If it is made before γ is closed, colluding users may formulate $\mathcal{T}_{\mathsf{com_n}}$ for channel closure such that $\mathcal{T}_{\mathsf{com_n}}$ is consistent with the manipulated history.

Adversary Model. We consider a Probabilistic Polynomial Time (PPT) adversary who may corrupt any user, in which case it controls the internal state of the corrupted user and all the following messages that the user sends to others. For any off-chain payment channel, we assume that its users may collude but they remain *rational* in the sense that they get compensated by all coins in the channel if their transaction counterparties misbehave. Note that this assumption of rational users is not new, but has been commonly used and heavily relied on by the existing payment channel protocols [5]. In the presence of $\mathcal{A}$, we identify the *completeness* property for characterizing the security of any auditable payment channel protocol.

Definition 2 (Completeness). *An auditable payment channel protocol* IvyAPC $:= (\mathsf{Create}, \mathsf{Update}, \mathsf{Close}, \mathsf{Punish}, \mathsf{Audit})$ *is complete if the following security properties hold for auditing each closed payment channel:* ***(1) Authenticity****. Adherence to the* IvyAPC *protocol ensures the validity of each commitment transaction, a property verifiable by any honest blockchain node.* ***(2) Zero-transaction Loss****. The* IvyAPC *protocol mandates the inclusion of all commitment transactions generated before the final commitment in the audit scope.* ***(3) Strict-transaction Order****. The* IvyAPC *protocol guarantees that the sequence of commitment transactions submitted for auditing accurately reflects their chronological occurrence in reality.*

3 Accountable Assertions with Flexible Public Key

3.1 Extractable Chameleon Hash with Flexible Public Key

It is a randomized hash function that can easily compute collisions given a trapdoor and allows anyone to extract the trapdoor given two different messages and random number pairs [42]. Compared with conventional extractable chameleon hash, we extend it with the flexible public key (called ECHFPK), where a public key or secret key can be transformed into a new representative of the same equivalence class, i.e., the pair of old and new key are related through a hard relation R [8]. ECHFPK includes the following PPT algorithms: $(\mathsf{GenCh}, \mathsf{Ch}, \mathsf{ChgChCPK}, \mathsf{ChgChCSK}, \mathsf{Col}, \mathsf{ExtractCsk})$:

- $(cpk, csk) \leftarrow \mathsf{GenCh}(1^\lambda)$: The key generation algorithm inputs the security parameter λ and outputs a public key cpk and a secret key csk (i.e., a trapdoor).
- $h \leftarrow \mathsf{Ch}(cpk, x; r)$: The evaluation algorithm generates a hash value h with the public key cpk, a message x, and a random r.
- $cpk' \leftarrow \mathsf{ChgChCPK}(cpk, \omega)$: The public key transformation algorithm takes cpk of an equivalence class $[cpk]_R$ and a public parameter ω as inputs. It outputs a different representative public key cpk', where $cpk' \in [cpk]_R$.
- $csk' \leftarrow \mathsf{ChgChCSK}(csk, \omega)$: The secret key transformation algorithm takes a trapdoor csk and public parameter ω as inputs, and outputs a different representative secret key csk'. This algorithm is reversible that given csk', it allows anyone to recover the secret key csk with the public ω.

– $r_1 \leftarrow \mathsf{Col}(csk', x_0, r_0, x_1)$: The collision-finding algorithm inputs a trapdoor csk' and a triple x_0, r_0, x_1 and outputs a value r_1 such that $\mathsf{Ch}(cpk', x_0; r_0) = \mathsf{Ch}(cpk', x_1; r_1)$.
– $csk' \leftarrow \mathsf{ExtractCsk}(cpk', (x_0, r_0, x_1, r_1))$: The extraction algorithm takes a public key cpk' and a 4-tuple (x_0, r_0, x_1, r_1) as inputs, and outputs csk'.

Specifically, extractable chameleon hash function satisfies: *(i) collision-resistance, (ii) uniformity, and (iii) extractability* [32,42]. The collision-resistance property states that the probability for a PPT adversary $\mathcal{A}$ to find a collision without a trapdoor is negligible. The uniformity property states that given two messages x_0 and x_1, for a uniformly random value r_0, the output of Col is also a uniformly distributed random value. The extractability property states that given two pairs (x_0, r_0) and (x_1, r_1), where $\mathsf{Ch}((\mathsf{cpk}', \mathsf{x_0}; \mathsf{r_0}))) = \mathsf{Ch}((\mathsf{cpk}', \mathsf{x_1}; \mathsf{r_1})))$ and $x_0 \neq x_1$, the trapdoor csk' can be extracted. The public and secret key transformation algorithms are instantiated according to the instantiation of extractable chameleon hash [32,42].

3.2 Accountable Assertions with Flexible Public Key Definition

Accountable Assertions with Flexible Public Key (called AAFPK) allows parties to make multiple statements in the same context under different representative secret keys without exposing the secret key. Our core idea is based on ECHFPK supporting that a key pair (apk, ask) can be transformed into a different representative key pair (apk', ask'). Note that AAFPK satisfies extractability which exposes the secret key if an assertor makes two statements in the same context under the same secret key.

Definition 3 (AAFPK). *The* AAFPK *protocol is a tuple of PPT algorithms* $\widetilde{\sum}$*:=(*Gen, Assert, Verify, ChgAPK, ChgASK, Extrtact*) such that:*

– $(apk, ask, auxsk) \leftarrow \mathsf{Gen}(1^\lambda)$*: The key generation algorithm inputs a security parameter* λ*, and outputs a public key* apk*, a secret key* ask*, an auxiliary secret information* $auxsk$*. For each public key, there is exactly one secret key.*
– $ask' \leftarrow \mathsf{ChgASK}(ask, \omega)$*: The secret key transformation algorithm takes a representative secret key* ask*, and a public parameter* ω *as inputs, and outputs a different representative secret key* ask'*. The algorithm is reversible in that given* ask'*, it allows anyone to recover the secret key* ask *with the public* ω*.*
– $apk' \leftarrow \mathsf{ChgAPK}(apk, \omega)$*: The public key transformation algorithm inputs a representative public key* apk *of equivalence class* $[apk]_R$*, and a public parameter* ω*, and outputs a different representative public key* $apk' \in [apk]_R$*.*
– $\tau / \perp \leftarrow \mathsf{Assert}(ask', auxsk, ct, st)$*: The assertion algorithm takes a secret key* ask'*, an auxiliary secret information* $auxsk$*, a context* ct*, a statement* st *as inputs. It outputs an assertion* τ *(or* $\perp$ *if the algorithm fails to execute).*
– $1/0 \leftarrow \mathsf{Verify}(apk', ct, st, \tau)$*: The verification algorithm takes a public key* apk'*, a context* ct*, a statement* st *and an assertion* τ *as inputs, and outputs 1 if* τ *is a valid assertion.*

– $ask'/\perp \leftarrow \mathsf{Extract}(apk', ct, st_0, st_1, \tau_0, \tau_1)$*: The extraction algorithm inputs a public key* apk'*, a context* ct*, two statements* st_0, st_1*, two assertions* τ_0, τ_1*, and outputs either* ask' *or* $\perp$ *to indicate failure.*

3.3 AAFPK Construction

Next, we present the construction of the AAFPK protocol. For simplification, here we mainly highlight the differences and refer the reader to [42] to see the construction of ECHFPK.

Key Generation. The key generation algorithm chooses L be a hash function: $\{0,1\}^* \rightarrow \{1, ..., m^{\ell-1}\}$, where m and ℓ are two positive integers that represent the branching and height of a tree. We assume that $\mathsf{L}(\cdot)$ is modeled as a random oracle. It also chooses $\mathsf{H}_0, \mathsf{H}_2$ be hash functions which are modeled as random oracles, H_1 be a collision-resistant hash function, and PRF be a pseudo-random function. Then, this algorithm generates the secret key $ask := csk$, auxiliary secret information $auxsk := \kappa$, where $(cpk, csk) \leftarrow \mathsf{ECHFPK.GenCh}(1^\lambda)$, $\kappa \in \{0,1\}^\lambda$ is a key for the PRF. It sets the public key as $apk := (cpk, z, \mathsf{L}, \mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2)$, where $z := \mathsf{H}_0(y_1^1, ..., y_m^1)$, $y_i^1 := \mathsf{Ch}(\mathsf{PRF}_\kappa(id, i, 0); \mathsf{PRF}_\kappa(id, i, 1))$, $i \in [m]$, id is an identifier for the position of the root node.

Secret Key Transformation. In this algorithm, the secret key, denoted as ask, is initially set to csk. It then transforms to become an updated secret key, ask', ensuring that ask and ask' belong to the same equivalence classes [28]. The transformation is performed using the function $\mathsf{ChgASK}(ask, \omega)$, which effectively converts the secret key into another secret key within the same class, represented as $ask' := \mathsf{ECHFPK.ChgChCSK}(csk, \omega)$. Here, ω stands for a specific public parameter chosen for this operation. It's important to note that this transformation allows the original secret key, ask, to be retrieved when given csk and ω. This feature ensures that the integrity of the secret key and link to its original form is maintained despite the update.

Public Key Transformation. This algorithm changes the public key $apk := (cpk, z, \mathsf{L}, \mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2)$ into an updated key $apk' := (cpk', z)$, where $cpk' := \mathsf{ECHFPK.ChgChCPK}(cpk, \omega)$ and ω is a chosen public parameter. This transformation allows the original secret key, ask, to be retrieved when given csk and ω.

Assertion. The assertion algorithm takes $(ask', auxsk, ct, st)$ as inputs. It computes the assertion path $\{Y_\ell, a_\ell, ..., Y_1, a_1\}$ from the leaf node Y_ℓ to the root node Y_1, where Y_ℓ stores the number $\mathsf{L}(\mathsf{ct})$ and each node Y_j contains m entries $Y_j := \{y_1^j, ..., y_m^j\}$, $j \in [m]$ at positions $a_j \in \{1, ..., m\}$. To assert a statement st in the context ct which is stored in Y_ℓ, the assertion algorithm computes ${r'}_{a_\ell}^\ell := \mathsf{Col}(csk', x_{a_\ell}^\ell, r_{a_\ell}^\ell, \mathsf{H}_1(st))$, where $(x_{a_\ell}^\ell, r_{a_\ell}^\ell)$ refer to the values that were set when the tree was initially generated. The entry refers to the position $\mathsf{L}(\mathsf{ct})$ is calculated as: $y_{a_\ell}^\ell = \mathsf{H}_2(\mathsf{Ch}(cpk', \mathsf{H}_1(st); {r'}_{a_\ell}^\ell), {r'}_{a_\ell}^\ell) = \mathsf{H}_2(\mathsf{Ch}(cpk', x_{a_\ell}^\ell; r_{a_\ell}^\ell), {r'}_{a_\ell}^\ell)$. Then the algorithm calculates other entries of Y_ℓ as: $y_i^\ell := \mathsf{Ch}(\mathsf{cpk}', \mathsf{x}_\mathsf{i}^\ell; {\mathsf{r}'}_\mathsf{i}^\ell)$, where $i \in [m] \backslash a_\ell$. It stores the entries $\{y_1^\ell, ..., y_m^\ell\}$ to Y_ℓ, and lets $z_\ell := \mathsf{H}_0(y_1^\ell, ..., y_m^\ell)$ and $f_\ell := (y_1^\ell, ..., y_{a_{\ell-1}}^\ell, y_{a_{\ell+1}}^\ell, ..., y_m^\ell)$. Similarly, the algorithm calculates other nodes

$(Y_{\ell-1}, a_{\ell-1}, ..., Y_1, a_1)$ as in the computation of (Y_ℓ, a_ℓ). The calculated assertion is $\tau := \big((r'^{\ell}_{a_\ell}, f_\ell, a_\ell), ..., (r'^{1}_{1}, f_1, a_1)\big)$, where $f_\epsilon := (y^\epsilon_1, ..., y^\epsilon_{a_\ell-1}, y^\epsilon_{a_\ell+1}, ..., y^\epsilon_m)$, where $\epsilon \in [\ell]$.

Verification. This algorithm takes (apk', ct, st, τ) as inputs, and parses apk' as (cpk', z) and τ as $\big((r'^{\ell}_{a_\ell}, f_\ell, a_\ell), ..., (r'^{1}_{1}, f_1, a_1)\big)$. Then it reconstructs the path from the leaf node Y_l to the root node Y_1, where $Y_1 := (y^1_1, ..., y^1_m)$. If the constructed root $\mathsf{H}(y^1_1, ..., y^1_m)$ is equal to z, it outputs 1; otherwise, it outputs 0.

Extraction. This algorithm inputs $(apk', ct, st_0, st_1, \tau_0, \tau_1)$ and reconstructs the path from the leaf node to the root node for both (ct, st_0, τ_0) and (ct, st_1, τ_1). During the reconstruction, there will exist a node that forms a collusion in ECHFPK. That is, there exist values (x_0, r_0) and (x_1, r_1) that enables ECHFPK.Ch $(cpk', x_0; r_0)$ =ECHFPK.Ch$(cpk', x_1; r_1)$. According to the extraction algorithm of ECHFPK, the secret key csk' can be extracted as $csk' \leftarrow$ ECHFPK. ExtractCsk $(cpk', x_0, r_0, x_1, r_1)$. If no collusion is found, the extraction algorithm outputs $\perp$.
Remark. When the assertor chooses a different representative key pair, denoted as (apk', ask'), to generate the signature τ, the verifiers must use the ChgPK(apk, ω) algorithm. This algorithm allows them to compute and verify the corresponding public key apk' necessary for validating τ. However, if the assertor claims different statements within the same context using the same representative key pair, the verifiers are then able to deduce the secret key ask'. This is done by effectively 'removing' the public component ω from the known quantities, revealing the secret key ask.

Theorem 1. *If* $\mathsf{L}, \mathsf{H}_0, \mathsf{H}_2$ *are modeled random oracle and the extractable chameleon hash with flexible public key scheme* ECHFPK *is collision-resistant, the accountable assertions with flexible public key scheme described in §3.3 is unforgeable.(The security proof is shown in the Appendix §C.3.)*

Theorem 2. *Let* $\widetilde{\sum}$ = (Gen, ChgAPK, ChgASK, Assert, Verify, Extrtact) *be an accountable assertions with flexible public key protocol. Then* $\widetilde{\sum}$ UC-realizes *the ideal accountable assertions functionality* $\mathcal{F}_{\mathsf{AA}}$ *if and only if the* AAFPK *protocol is unforgeable. (The security proof is in the Appendix §C.4.)*

4 IvyAPC Protocol

4.1 Protocol Specification

This section details the specific framework of our tool, as referenced in Figure 1. Our methodology requires that each extended GC γ presented for audit include at least one off-chain transaction and that it be closed by an off-chain transaction published on a blockchain. For simplicity, we conceptualize a complete transaction as comprising two distinct segments, symbolized by $\mathcal{T} := \{[\mathcal{T}], dat\}$. Here, $[\mathcal{T}] := \{in, out, v, tl\}$ represents the core payment component of $\mathcal{T}$, which lacks a signature. The dat refers to a data space, housing audit-relevant information (called audit trail)s.

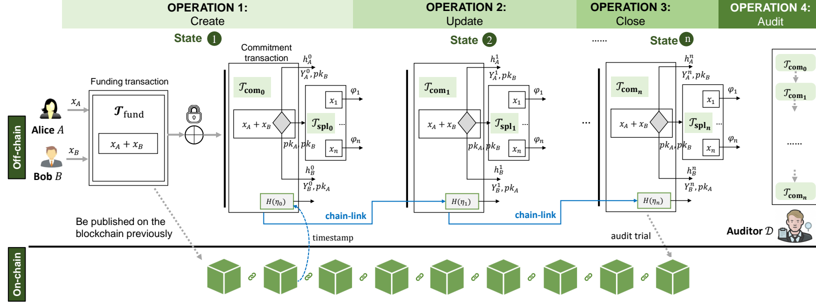


Fig. 1: An illustration of the IvyAPC protocol.

Create. Creating a payment channel involves three key steps for the transacting parties: generating the funding transaction $\mathcal{T}_{\mathsf{fund}}$, creating the initial commitment transaction $\mathcal{T}_{\mathsf{com}_0}$, and forming the initial split transaction $\mathcal{T}_{\mathsf{spl}_0}$. This procedure diverges from the GC protocol [5] in several crucial aspects: (i) *Assertion Exchange:* Transacting parties are required to exchange assertions using the AAFPK protocol during this operation. (ii) *Timestamping:* Each commitment transaction is marked with a timestamp using an on-chain transaction. (iii) *Digest Inclusion:* The digest of assertions and timestamps must be included in an unspent output.

The users A and B start by locally creating the funding transaction $[\mathcal{T}_{\mathsf{fund}}]$ and the commitment transaction $[\mathcal{T}_{\mathsf{com}_0}]$. To do this, each user generates their own set of keys: A generates $(apk_A, ask_A, auxsk_A)$ and B generates $(apk_B, ask_B, auxsk_B)$, using the $\widetilde{\sum}$.Gen function with a security parameter λ. Here, apk stands for the public key, ask for the secret key, and $auxsk$, an auxiliary secret key, is derived by hashing ask, specifically $auxsk := \mathsf{H}_0(ask)$. The secret keys ask_A and ask_B are termed as *colluding secrets*. They are crucial for ensuring that any attempt by A and B to alter the history of off-chain transactions in γ can be penalized. Following the key generation, A and B exchange their public keys (apk_A and apk_B), ensuring that $apk_A \neq apk_B$. They then store the pair $\xi_0 = (apk_A, apk_B)$ locally as the *audit trail* for the funding transaction $[\mathcal{T}_{\mathsf{fund}}]$. Before signing and broadcasting $[\mathcal{T}_{\mathsf{fund}}]$ to the blockchain, they incorporate $h_0 = \mathsf{H}_1(\xi_0)$ into it. Here, h_0 is included as an unspendable transaction output, such as using the OP_RETURN in Bitcoin.

Then, each party generate their accountable assertions τ_A^0, τ_B^0 of $[\mathcal{T}_{\mathsf{com}_0}]$ under the context of $[\mathcal{T}_{\mathsf{fund}}]$ (written as ct). These assertions are generated using the $\widetilde{\sum}$.Assert algorithm. Specifically, party A computes τ_A^0 as $\widetilde{\sum}.\mathsf{Assert}(\mathsf{ask_A}, \mathsf{auxsk_A}, \mathsf{H}_1(0, [\mathcal{T}_{\mathsf{fund}}]), [\mathcal{T}_{\mathsf{com}_0}])$, and similarly, party B computes τ_B^0. To ensure that the assertions are indeed exchanged and linked to the transacting parties, IvyAPC requires each party to generate proof. This proof associates each party's private revocation key (r) with the other party's assertion (τ) using a hash function. For instance, after party A sends its assertion τ_A^0 to party B, party B verifies it with $\widetilde{\sum}$.Verify. Upon successful verification, B generates a proof $\varsigma_B^0 := \mathsf{H}_0(\tau_A^0 || r_B^0)$. This mechanism also incorporates the timestamp of on-chain transactions to

mark the commitment transactions' timing accurately. The audit trail from the first commitment transaction is captured in the format:

$$\eta_0 := (\tau_A^0, \tau_B^0, \varsigma_A^0, \varsigma_B^0, [\mathcal{T}_{\mathsf{fund}}], [\mathcal{T}_{\mathsf{com}_0}], n), \\ \text{where } \varsigma_A^0 := \mathsf{H}_0(\tau_B^0 || r_A^0) \text{ and } \varsigma_B^0 := \mathsf{H}_0(\tau_A^0 || r_B^0), \tag{1}$$

where r_A^0 and r_B^0 respectively refer to the private revocation key of party A and B, $n := 0$ refers to the first index of $\mathcal{T}_{\mathsf{com}_0}$.

Once transacting parties generate the complete commitment transaction $\mathcal{T}_{\mathsf{com}_0} := \{[\mathcal{T}_{\mathsf{com}_0}], \mathsf{H}(\eta_0)\}$ locally, they can exchange the (pre-)signature $\mathcal{T}_{\mathsf{spl}_0}$, $\mathcal{T}_{\mathsf{com}_0}$, and $\mathcal{T}_{\mathsf{fund}}$ as in [5]. Upon the publication of $\mathcal{T}_{\mathsf{fund}}$ on the blockchain, an auditable payment channel, denoted as γ, is successfully established.

Update. In this operation, transacting parties pay for each other by updating the state of the channel. That is, they collaboratively update the state of γ from $\gamma.cash := (x_A, x_B)$ to $\gamma.\widetilde{cash} := (\widetilde{x_A}, \widetilde{x_B})$. This process involves several key steps: (1) *Invalidation of the Old Commitment Transaction:* The parties agree to invalidate the previous commitment transaction. This step is crucial for ensuring that only the most recent state of the channel can be finalized on the blockchain. (2) *Generation of New Transactions:* A new commitment transaction and a corresponding split transaction are generated. These transactions reflect the updated balances $(\widetilde{x_A}, \widetilde{x_B})$ of the parties. Each off-chain transaction, $[\mathcal{T}_{\mathsf{com}_n}]$, is constructed to distribute the total coins $(x_A + x_B)$ in γ based on the latest agreed-upon balances. If transaction $[\mathcal{T}_{\mathsf{com}_n}]$ is published on a blockchain by party A (without loss of generality), it activates specific spending conditions according to the updated distribution of funds: *(1) Colluding condition*: $(x_A + x_B)$ can be spent by a blockchain on-chain transaction that is verifiable w.r.t. X_A and X_B at any time after $[\mathcal{T}_{\mathsf{com}_n}]$ was published on blockchains. *(2) Publishing condition*: $(x_A + x_B)$ can be spent by B with a time lock Δ (n.e., after Δ time since $[\mathcal{T}_{\mathsf{com}_n}]$ was published on blockchains) if $[\mathcal{T}_{\mathsf{com}_n}]$ was revoked. *(3) Finalizing condition*: $(x_A + x_B)$ is spent by A and B for finalizing the channel state of γ with a time lock 2Δ if $[\mathcal{T}_{\mathsf{com}_n}]$ has not been revoked.

The *colluding condition* is used to punish a colluding user by their counterparty who can unilaterally spend all coins in γ if the history of committed off-chain transactions in γ is modified in their collusion attacks. Similar to the original GC protocol [5], the *publishing condition* is used to punish a user by their counterparty who can unilaterally spend all coins in γ if the user publishes a revoked off-chain transaction for channel closure. The only difference is that we add a time lock Δ to the publishing condition so that any collusion attack can be punished at a higher priority. The *finalizing condition* is also similar to that in the original GC protocol [5] except that its time lock is set to be twice as long to leave sufficient time for punishing any user publishing a revoked off-chain transaction before finalizing the channel state of γ. The parameter Δ serves as a system-wide constant, establishing a timeframe within which punitive actions can be executed on the blockchain to maintain channel integrity.

More concretely, a transacting party (e.g., A) first chooses a public parameter $\omega_A \in \mathbb{Z}_q^*$ and transforms the secret key $ask_A^n := \widetilde{\sum}.\mathsf{ChASK}(ask_A, n \cdot \omega_A) :=$

$ask_A \oplus n \cdot \omega_A$. It requires A to utilize a public parameter, e.g., $\omega_A := \mathsf{H}_0(\mathcal{T}_{\mathsf{fund}})$. When the n-th off-chain transaction $[\mathcal{T}_{\mathsf{com_n}}]$ ($n \geq 1$) is created in γ, users A and B exchange their accountable assertions τ_A^n, τ_B^n of $[\mathcal{T}_{\mathsf{com_n}}]$ under the context of $[\mathcal{T}_{\mathsf{fund}}]$, where $\tau_A^n \leftarrow \widetilde{\sum}.\mathsf{Assert}(ask_A, auxsk_A, ct, st_n)$ (i.e., $ct = \mathsf{H}_1(n, [\mathcal{T}_{\mathsf{fund}}])$ and $st_n := [\mathcal{T}_{\mathsf{com}_n}]$) and $\tau_B^n \leftarrow \widetilde{\sum}.\mathsf{Assert}(ask_B, auxsk_B, ct, st_n)$.

More concretely, a participant (e.g., A) selects a public parameter $\omega_A \in \mathbb{Z}_q^*$ and modifies their secret key to $ask_A^n := \widetilde{\sum}.\mathsf{ChASK}(\mathsf{ask_A}, \mathsf{n} \cdot \omega_\mathsf{A}) := \mathsf{ask_A} \oplus \mathsf{n} \cdot \omega_\mathsf{A}$, leveraging $\omega_A := \mathsf{H}_0(\mathcal{T}_{\mathsf{fund}})$. Upon generating the n-th off-chain transaction $[\mathcal{T}_{\mathsf{com_n}}]$ in γ, users A and B exchange their accountable assertions τ_A^n, τ_B^n of $[\mathcal{T}_{\mathsf{com_n}}]$ under the context of $[\mathcal{T}_{\mathsf{fund}}]$, where $\tau_A^n \leftarrow \widetilde{\sum}.\mathsf{Assert}(ask_A, auxsk_A, ct, st_n)$ (i.e., $ct = H_1(n, [\mathcal{T}_{\mathsf{fund}}])$ and $st_n := [\mathcal{T}_{\mathsf{com_n}}]$) and $\tau_B^n \leftarrow \widetilde{\sum}.\mathsf{Assert}(ask_B, auxsk_B, ct, st_n)$.

Then, they embed the *hash chain* $h_n = Hash_1(h_{n-1}||\xi_n)$ in $[\mathcal{T}_{\mathsf{com_n}}]$ before committing it[6] in γ. Specifically, upon receiving assertion τ_A^n from party A, party B first computes a new representative public key $apk_A^n = \widetilde{\sum}.\mathsf{ChAPK}(apk_A, n \odot \omega_A) := apk_A \odot n \cdot \omega_A$ and then verifies τ_A^n through using $\widetilde{\sum}.\mathsf{Verify}\ (apk_A^n, ct, st_n, \tau_A^n)$. If it outputs "1", party B generates a proof $\varsigma_B^n := \mathsf{H}_0(\tau_A^n||r_B^n)$, where r_B^n refers to party B' private revocation key to be used in $[\mathcal{T}_{\mathsf{com_n}}]$. They exchange the proof and generate the audit trail η_n as in Equation (1). The timestamp value ts_n in η_n is set as the earliest timestamp in the latest block. After that, they construct and sign the complete commitment transaction $\mathcal{T}_{\mathsf{com_n}} := \{[\mathcal{T}_{\mathsf{com_n}}], (\mathsf{H}_0(\eta_\mathsf{n}), \mathsf{H}_0(\eta_{\mathsf{n}-1}))\}$ and split transaction $\mathcal{T}_{\mathsf{spl_n}}$.

Close. Transacting parties close the channel by publishing a commitment transaction to the blockchain. Under the revocation mechanisms, transacting parties can close the channel peacefully without disrupting the fairness between them. If a transacting party closes the channel by publishing a revoked commitment transaction on the blockchain or regenerating a commitment transaction without including the digest of all previous commitment transactions, IvyAPC allows the other party to redeem the deposit of that party (see §D.4 for discussion).

Punish. This operation happens if (i) a transacting party publishes an old commitment transaction to the blockchain or (ii) makes two statements under the same representative secret key. For (i), suppose party A publishes an old commitment transaction, party B can utilize the pre-signature and complete signature of the transaction to calculate party A's secret witness y_A based on the extractability of adapter signature. With the witness y_A, party B can spend all the output of the channel. For (ii), party A's secret key ask_A would be extracted by B via $\widetilde{\sum}.\mathsf{Extract}$. In both cases, B can spend the output of transaction $\mathcal{T}_{\mathsf{com_n}}$ without interaction with A.

Audit. This operation is performed by auditor $\mathcal{D}$ who requires both transacting parties to provide information to be audited.

[6] In generalized channel [5], each off-chain transaction $[\mathcal{T}_{\mathsf{com_n}}]$ is constructed as a commitment transaction followed by a split transaction. The embedding should be performed on the commitment transaction as its unspendable transaction output.

Given a channel γ, an auditor O may audit a sequence of off-chain transactions $\gamma.ctList := \{\mathcal{T}_{\mathsf{com}_i}.tid, \mathcal{T}_{\mathsf{spl}_i}.tid, n\}_{i\in[n]}$, audit trails $\{\eta_i\}_{i\in[n]}$, revocation secrets $\{r^i_{\mathcal{P}}\}_{i\in[n],\mathcal{P}\in\{A,B\}}$, assertions and their exchange proofs $\{\varsigma^i_{\mathcal{P}}, \tau^i_{\mathcal{P}}\}_{i\in[n],\mathcal{P}\in\{A,B\}}$, and public parameters $\{\omega_{\mathcal{P}}\}_{\mathcal{P}\in\{A,B\}}$. The auditor outputs "success" *iff* all of the following *audit conditions* are meet:

1. *Checking all transactions*: The funding transaction $\mathcal{T}_{\mathsf{fund}}$ and the closing transaction $\mathcal{T}_{\mathsf{com}_n}$ were published on-chain; the sequence of off-chain transactions is non-empty and all of them were committed in γ by their users; the signature of each commitment transaction is valid..
2. *Checking all audit trails*: Check if $\{\tau^i_{\mathcal{P}}\}_{i\in[n],\mathcal{P}\in\{A,B\}}$ and $\{\varsigma^i_{\mathcal{P}}, r^i_{\mathcal{P}}\}_{i\in[n],\mathcal{P}\in\{A,B\}}$ are correct, i.e., if $\mathsf{H}_0(r^i_{\mathcal{P}}) = h^i_{\mathcal{P}}$ and $\varsigma^i_{\mathcal{P}} = \mathsf{H}_0(\tau^{\mathsf{i}}_{\mathcal{P}}||\mathsf{r}^{\mathsf{i}}_{\mathcal{P}})$ hold.
3. *Checking hash chains*: Check $\mathsf{H}_0(r^i_{\mathcal{P}}) = h^i_{\mathcal{P}}$ and $\varsigma^i_{\mathcal{P}} = \mathsf{H}_0(\tau^{\mathsf{i}}_{\mathcal{P}}||\mathsf{r}^{\mathsf{i}}_{\mathcal{P}})$ for $1 \leq i \leq n$.
4. *Checking consistency between $\widetilde{\eta_n}$ and η_n*: Compute the latest audit information $\widetilde{\eta_n}$ according to the provided information $\{\gamma.ctList, \{\eta_i, \omega_{\mathcal{P}}, r^i_{\mathcal{P}}\}_{i\in[n],\mathcal{P}\in\{A,B\}}\}$, and verify whether the hash value of $\widetilde{\eta_n}$ is equal to hash value of η_n that is included in the closing transaction $\mathcal{T}_{\mathsf{com}_n}$.
5. *Checking spending time of $\mathcal{T}_{\mathsf{com}_n}$*: If $\mathcal{T}_{\mathsf{com}_n}$ was spent on the blockchain, it was spent after 2Δ time since $\mathcal{T}_{\mathsf{com}_n}$ was published on the blockchain.

If the information submitted for audit is the original history of off-chain transactions committed in γ and associated audit trails, it is clear that the auditor outputs "success" following our methodology. Next, we clarify that our methodology enables the auditor to output "failure" if a modified history of off-chain transactions and associated audit trails (if any) are submitted for audit.

5 Security Analysis

Let sequence $(\bar{\mathcal{T}}_{\mathsf{com}_1}, \ldots, \bar{\mathcal{T}}_{\mathsf{com}_m})$ for $m \geq 1$ denote the original history of the off-chain transactions committed in channel γ funded by $\mathcal{T}_{\mathsf{fund}}$ and closed by $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$. For each $\bar{\mathcal{T}}_{\mathsf{com}_i}$ in the original history, we assume that users A and B already generated an audit trail and embedded a hash chain $\bar{h}_i$ (out of scope is that the users do not follow the extended GC in the first place). Let sequence $(\mathcal{T}_{\mathsf{com}_1}, \ldots, \mathcal{T}_{\mathsf{com}_1})$ denote a changed history of committed off-chain transactions in γ. In the following analysis we make a non-trivial assumption: (i) the changed history is non-empty (i.e., $n \geq 1$), (ii) $\mathcal{T}_{\mathsf{com}_i}$ is committed by both A and B, (iii) the i-th audit trailis generated for $\mathcal{T}_{\mathsf{com}_i}$ following our methodology, and (iv) a hash chain h_i is embedded in each $\mathcal{T}_{\mathsf{com}_i}$ following our methodology; otherwise, an auditor can trivially output "failure" by checking the first three *audit conditions* mentioned above. Any changes made to the original history in γ can be categorized into the exclusive cases:

- *Case I (i.e., arbitrary-ordering attack)*: $\{i : \bar{\mathcal{T}}_{\mathsf{com}_i} \neq \mathcal{T}_{\mathsf{com}_i}, 1 \leq i \leq m\} \neq \emptyset$.
- *Case II (i.e., discarding attack)*: $1 \leq n < m$ and $\bar{\mathcal{T}}_{\mathsf{com}_i} = \mathcal{T}_{\mathsf{com}_i}$ for $i \leq n < m$.
- *Case III*(i.e., injection attack): $1 \leq m < n$ and $\bar{\mathcal{T}}_{\mathsf{com}_i} = \mathcal{T}_{\mathsf{com}_i}$ for all $i \leq m < n$.

Case I means that at least one off-chain transaction in the original history was changed, which refers to the *arbitrary-ordering attack*. Let $m_0 = \min\{i : \bar{\mathcal{T}}_{\mathsf{com}_i} \neq \mathcal{T}_{\mathsf{com}_i}, 1 \leq i \leq m\}$. Since both A and B need to commit the changed off-chain transaction, $\mathcal{T}_{\mathsf{com}_{m_0}}$, it requires collusion between A and B. If A and B in collusion made the change *after* γ was closed, then the audit trail η_{m_0} for the changed transaction $\mathcal{T}_{\mathsf{com}_{m_0}}$ is different from the audit trail $\bar{\eta}_{m_0}$ for the original transaction $\bar{\mathcal{T}}_{\mathsf{com}_i}$; thus, the auditor can output "failure" by checking the consistency between $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ and $\mathcal{T}_{\mathsf{com}_n}$ (i.e., the 4-th audit condition): The auditor can detect that the hash chain $\bar{h}_m$ embedded in $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ (which is computed partly from $\bar{\eta}_{m_0}$) is different from the hash chain h_n embedded in $\mathcal{T}_{\mathsf{com}_n}$ (which is computed partly from η_{m_0}).

On the other hand, if A and B in collusion made the change from $\bar{\mathcal{T}}_{\mathsf{com}_{m_0}}$ to $\mathcal{T}_{\mathsf{com}_{m_0}}$ *before* γ was closed, they embed h_n into $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ such that the auditor cannot detect any inconsistency between $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ and $\mathcal{T}_{\mathsf{com}_n}$. In this case, the auditor can still output "failure" by checking the spending time of $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ (i.e., the 5-th audit condition). The auditor can detect that $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ was spent within Δ time since $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ was published on the blockchain. This is because A and B need to compute their accountable assertions $\tau_A^{m_0}, \tau_B^{m_0}$ and exchange them for constructing a valid audit trail η_{m_0}. The user, say A w.l.o.g., who receives $\tau_B^{m_0}$ before B receives $\tau_A^{m_0}$ (or B receives no $\tau_A^{m_0}$), can extract B's colluding secret $ask'_B := \widetilde{\sum}.Extract(apk'_B, ct, st_{m_0}, \bar{st}_{m_0}, \tau_B^{m_0}, \bar{\tau}_B^{m_0})$. Recall that in the original history, A had already received B's accountable assertion $\bar{\tau}_{B,m_0}$ in constructing $\bar{\eta}_{m_0}$. Now A can publish $\bar{\mathcal{T}}_{\mathsf{com}_{m_0}}$ on the blockchain as $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ and punish B by spending all coins in γ within Δ time under the colluding condition of $\bar{\mathcal{T}}_{\mathsf{com}_{m_0}}$. A rational A will do so because if A does not punish B but sends its accountable assertion $\tau_A^{m_0}$ to B, then B can punish A by publishing $\bar{\mathcal{T}}_{\mathsf{com}_{m_0}}$ and spending all coins in γ under the colluding condition of $\bar{\mathcal{T}}_{\mathsf{com}_{m_0}}$.

Case II means that no off-chain transaction in the original history was changed but a last segment of it was missing in the changed history, which refers to the *discarding attack*. In this case, each off-chain transaction in the changed history was ever revoked by A and B. If the change was made *after* γ was closed, then the auditor can output "failure" by checking the consistency between $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ and $\mathcal{T}_{\mathsf{com}_n}$ (i.e., the fourth audit condition): The auditor can detect that the hash chain $\bar{h}_m$ embedded in $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ is different from the hash chain h_n embedded in $\mathcal{T}_{\mathsf{com}_n}$. On the other hand, if the change was made *before* γ was closed, a user, A w.l.o.g., could publish $\mathcal{T}_{\mathsf{com}_n}$ for channel closure (i.e., $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}} = \mathcal{T}_{\mathsf{com}_n}$) such that the auditor cannot detect any inconsistency between $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ and $\mathcal{T}_{\mathsf{com}_n}$. In this case, the auditor can still output "failure" by checking the spending time of $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ (i.e., the fifth audit condition). The auditor can detect that $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ was spent in $[\Delta, 2\Delta)$ time since $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ was published on the blockchain. This is because a rational B punishes user A for publishing a revoked off-chain transaction $\mathcal{T}_{\mathsf{com}_n}$ on the blockchain under the publishing condition of $\mathcal{T}_{\mathsf{com}_n}$.

Case III means that no off-chain transaction in the original history was changed but additional off-chain transactions were appended in the changed history. If the change was made *after* γ was closed, then the auditor can output "failure" by

checking the consistency between $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ and $\mathcal{T}_{\mathsf{com}_n}$. The auditor can detect that the hash chain $\bar{h}_m$ embedded in $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ is different from the hash chain h_n embedded in $\mathcal{T}_{\mathsf{com}_n}$. Further, if the change was made *before* γ was closed, both A and B committed the additional off-chain transactions $\mathcal{T}_{\mathsf{com}_{m+1}}, \ldots, \mathcal{T}_{\mathsf{com}_n}$, computed their audit trails, and embedded their hash chains following our assumption. If γ was not closed on $\mathcal{T}_{\mathsf{com}_n}$ but on a revoked transaction, the auditor can output "failure" by detecting that $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ was spent in $[\Delta, 2\Delta)$ time since $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}}$ was published on the blockchain. If γ was closed on $\mathcal{T}_{\mathsf{com}_n}$ (i.e., $\mathcal{T}_{\mathsf{com}_{\mathsf{last}}} = \mathcal{T}_{\mathsf{com}_n}$), then the changed history $(\mathcal{T}_{\mathsf{com}_1}, \ldots, \mathcal{T}_{\mathsf{com}_n})$ can be treated as a new original history in γ since A and B continued committing additional off-chain transactions beyond the old original history before the channel was closed. Our above analysis on any changes made to the old original history can be applied recursively to any changes made to the new original history.

6 Instantiation and Implementation

6.1 Implementation

Our work is implemented using C++ and Javascript. It depends on the full source code of *libsecp256k1* [19] to perform elliptic curve computations, and accountable assertions [41] to realize the AAFPK protocol. Bitcoin environment was built locally on a personal computer ("Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz", 4GB RAM). We make use of *bitcoinjs-lib* [9] as a programming lib to interact with Bitcoin. The implementation is publicly available [16]. To evaluate the performance of IvyAPC, we modeled two parties as two clients in the computer and conducted the experiments by letting them generate 100 commitment transactions in a channel (i.e., $n = 99$). At the time of writing, the average price of Bitcoin is 10 satoshis per byte citebitcoinfees or 0.0039 USD per byte.

6.2 Evaluation Results

Compared with GCs [5], IvyAPC does not reduce the number of on-chain or off-chain transactions. Thus, we mainly present its extra computation and storage costs in generating audit trails in commitment transactions.

Computation Cost. Table 1a shown the computation cost of the AAFPK protocol. The results show that the public and secret key transform algorithms are efficient and take no more than 1 μs. The performance of the assertion and verification algorithm is as efficient as [42] which takes about 4.3 ms and 5.1 ms, respectively. Besides, the computation cost of an audit trail includes the assertion generation, verification, and exchange proof generation as in Equation (1), which takes 121.3 ms on average (considering local communication cost between two parties). In addition, we allow one party to extract the secret key of the other party in case of dishonest behavior. The Extract algorithm recovers a transformed secret key first and removes the public parameter to obtain the secret, which takes 5.4 ms on average.

Storage Cost. Regarding the storage cost, we mainly evaluate the size and transaction fees concerning the funding and commitment transactions. The results are shown in Table 1b. Specifically, each commitment transaction contains

a hash value for storing audit information, resulting in an extra cost of 32 bytes compared with GCs [5]. The size of a funding transaction and a commitment transaction are about 744 bytes and 1303 bytes, respectively, resulting in the cost of 2.90 USD and 5.08 USD on the Bitcoin blockchain. To generate an audit trail in a commitment transaction, a transacting party is required to send an assertion and its exchanging proof to the other party, which takes 4192 bytes. Besides, to provide the whole history of commitment transactions to the auditor, one party needs to store $100*(2|\tau|+|\mathcal{T}_{\mathsf{com}}|+2*|\varsigma|+|\mathcal{T}_{\mathsf{on}}.ts|)$, where the size of an assertion τ, ς, and $|\mathcal{T}_{\mathsf{on}}.ts|$ are 4160 bytes, 32 byte, 8 bytes, respectively. To audit 100 commitment transactions, one party is required to store about 0.92MB.

Operation	Average Time
ChgAPK/ ChgASK	1.0 μs
Assert	4.3 ms
Verify	5.1 ms
Extract	5.4 ms

(a) Computation Costs of the AAFPK Protocol

Operation	Size (bytes)	Cost (USD)
Funding Transaction	744	2.90
Commitment Transaction	1303	5.08
Audit Trail in a Commitment Transaction	4192	-

(b) Storage and On-Chain Costs of Transactions

Table 1: Computation and Storage Costs of the AAFPK Protocol

7 Conclusion

Payment Channels (PCs) have emerged as a pivotal solution for enhancing blockchain scalability. In this study, we tackled the auditing challenges associated with off-chain transactions and introduced IvyApc—a comprehensive, auditable protocol tailored for PCs. Our contributions are twofold: (i) Audit Model: We established a formal audit model that delineates the security criteria necessary for PC audits. (ii) Accountable Assertions Scheme: We devised the Accountable Assertions Scheme with Flexible Public Key (AAFPK) and integrated it into the IvyApc framework. Through rigorous proofs, we have affirmed that IvyApc fulfills the stringent security standards set by our audit model within the UC framework. Moreover, practical implementation tests confirm that IvyApc is not only theoretically sound but also viable for real-world applications.

Acknowledgement

The work was supported in part by the National Natural Science Foundation of China (Nos. 62102165, 62472198, U2001205,62332007, U22B2028, 62302192, U23A20303), the Science and Technology Major Project of Tibetan Autonomous Region of China (No. XZ202201ZD0006G), the Natural Science Foundation of Guangdong Province (No. 2024A1515010086), and the Science and Technology Program of Guangzhou (Nos. 2024A03J0464, 2024A04J3691), Guangdong Basic and Applied Basic Research Foundation (Grant No. 2023B1515040020) National Joint Engineering Research Center of Network Security Detection and Protection Technology, Guangdong Key Laboratory of Data Security and Privacy Preserving, Guangdong Hong Kong Joint Laboratory for Data Security and Privacy Protection, and the Ripple University Blockchain Research Initiative.

References

1. Kpmg blockchain services (2017)
2. Lightning network (2022)
3. Public company accounting oversight board (2022)
4. Raiden network (2022)
5. Aumayr, L., Ersoy, O., Erwig, A., Faust, S., Hostáková, K., Maffei, M., Moreno-Sanchez, P., Riahi, S.: Generalized bitcoin-compatible channels. Cryptology ePrint Archive **2020**, 476 (2020)
6. Aumayr, L., Maffei, M., Ersoy, O., Erwig, A., Faust, S., Riahi, S., Hostáková, K., Moreno-Sanchez, P.: Bitcoin-compatible virtual channels. In: S&P. pp. 901–918. IEEE (2021)
7. Aumayr, L., Thyagarajan, S.A., Malavolta, G., Monero-Sánchez, P., Maffei, M.: Sleepy channels: Bitcoin-compatible bi-directional payment channels without watchtowers. Cryptology ePrint Archive (2021)
8. Backes, M., Hanzlik, L., Kluczniak, K., Schneider, J.: Signatures with flexible public key: Introducing equivalence classes for public keys. In: Asiacrypt. pp. 405–434. Springer (2018)
9. BitcoinJS: bitcoinjs-lib. `https://github.com/bitcoinjs/bitcoinjs-lib` (2024), accessed: 2024-04-27
10. Bonyuet, D.: Overview and impact of blockchain on auditing. International Journal of Digital Accounting Research **20**, 31–43 (2020)
11. Camenisch, J., Drijvers, M., Gagliardoni, T., Lehmann, A., Neven, G.: The wonderful world of global random oracles. In: Eurocrypt. pp. 280–312. Springer (2018)
12. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS. pp. 136–145. IEEE (2001)
13. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: TCC. pp. 61–85. Springer (2007)
14. Chatzigiannis, P., Baldimtsi, F.: Miniledger: compact-sized anonymous and auditable distributed payments. In: ESORICS. pp. 407–429. Springer (2021)
15. Chatzigiannis, P., Baldimtsi, F., Chalkias, K.: Sok: auditability and accountability in distributed payment systems. In: ACNS. pp. 311–337. Springer (2021)
16. Consortium, A.C.: Auditable channel consortium implementation. `https://github.com/AuditableChannel-Consortium` (2024), accessed: 2024-04-27
17. Dagher, G.G., Bünz, B., Bonneau, J., Clark, J., Boneh, D.: Provisions: Privacy-preserving proofs of solvency for bitcoin exchanges. In: CCS. pp. 720–731 (2015)
18. Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., Breidenbach, L., Juels, A.: Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In: S&P. pp. 910–927. IEEE (2020)
19. Developers, B.C.: libsecp256k1. `https://github.com/bitcoin/secp256k1` (2024), accessed: 2024-04-27
20. Dziembowski, S., Eckey, L., Faust, S.: Fairswap: How to fairly exchange digital goods. In: CCS. pp. 967–984 (2018)
21. EY Global: Building a better working world. `https://www.ey.com/en_gl` (2024), accessed: 2024-02-21
22. Falzon, F., Elkhiyaoui, K., Manevich, Y., De Caro, A.: Short privacy-preserving proofs of liabilities. In: CCS. pp. 1805–1819 (2023)
23. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: Eurocrypt. pp. 281–310. Springer (2015)

24. Ge, Z., Gu, J., Wang, C., Long, Y., Xu, X., Gu, D.: Accio: Variable-amount, optimized-unlinkable and nizk-free off-chain payments via hubs. In: CCS. pp. 1541–1555 (2023)
25. Glaeser, N., Maffei, M., Malavolta, G., Moreno-Sanchez, P., Tairi, E., Thyagarajan, S.A.K.: Foundations of coin mixing services. In: CCS. pp. 1259–1273 (2022)
26. Gluchowski, A.: Zk rollup: scaling with zero-knowledge proofs. Matter Labs (2019)
27. Gudgeon, L., Moreno-Sanchez, P., Roos, S., McCorry, P., Gervais, A.: Sok: Layer-two blockchain protocols. In: International Conference on Financial Cryptography and Data Security. pp. 201–226. Springer (2020)
28. Hanser, C., Slamanig, D.: Structure-preserving signatures on equivalence classes and their application to anonymous credentials. In: Asiacrypt. pp. 491–511. Springer (2014)
29. Jiang, Y., Li, Y., Zhu, Y.: Auditable zerocoin scheme with user awareness. In: ICCSP. pp. 28–32 (2019)
30. Katz, J., Maurer, U., Tackmann, B., Zikas, V.: Universally composable synchronous computation. In: TCC. pp. 477–498. Springer (2013)
31. Kogan, B., Jajodia, S.: An audit model for object-oriented databases. In: Proceedings Seventh Annual Computer Security Applications Conference. pp. 90–91. IEEE Computer Society (1991)
32. Krawczyk, H., Rabin, T.: Chameleon hashing and signatures (1998)
33. Latham, D.C.: Department of defense trusted computer system evaluation criteria. Department of Defense (1986)
34. Li, Y., Susilo, W., Yang, G., Yu, Y., Phuong, T.V.X., Liu, D.: Non-equivocation in blockchain: Double-authentication-preventing signatures gone contractual. In: AsiaCCS. pp. 859–871 (2021)
35. Li, Y., Weng, J., Lai, J., Li, Y., Sun, J., Wu, J., Li, M., Wu, P., Deng, R.H.: Auditpch: Auditable payment channel hub with privacy protection. TIFS (2024)
36. Naganuma, K., Yoshino, M., Sato, H., Suzuki, T.: Auditable zerocoin. In: EuroS&PW. pp. 59–63. IEEE (2017)
37. Narula, N., Vasquez, W., Virza, M.: zkledger:privacy-preserving auditing for distributed ledgers. In: NSDI. pp. 65–80 (2018)
38. Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments (2016)
39. PricewaterhouseCoopers Global: Building trust for today and tomorrow. `https://www.pwc.com/gx/en.html` (2024), [Accessed: 21-February-2024]
40. Qin, X., Pan, S., Mirzaei, A., Sui, Z., Ersoy, O., Sakzad, A., Esgin, M.F., Liu, J.K., Yu, J., Yuen, T.H.: Blindhub: Bitcoin-compatible privacy-preserving payment channel hubs supporting variable amounts. In: S&P. pp. 2462–2480. IEEE (2023)
41. or Random, R.: Accountable assertions. `https://github.com/real-or-random/accas` (2024), accessed: 2024-04-27
42. Ruffing, T., Kate, A., Schröder, D.: Liar, liar, coins on fire! penalizing equivocation by loss of bitcoins. In: CCS. pp. 219–230 (2015)
43. Thyagarajan, S.A.K., Malavolta, G.: Lockable signatures for blockchains: Scriptless scripts for all signatures. In: S&P. pp. 937–954. IEEE (2021)
44. Tomescu, A., Devadas, S.: Catena: Efficient non-equivocation via bitcoin. In: S&P. pp. 393–409. IEEE (2017)
45. Yang, Y., Liang, X., Song, X., Dong, Y., Huang, L., Ren, H., Dong, C., Zhou, J.: Maliciously secure circuit private set intersection via SPDZ-compatible oblivious PRF. PETS (2025)

A Related Works

Blockchain auditability. There are several audit research efforts proposed in both industry and academia that leverage the decentralization and tamper-resistance of blockchains to improve traditional audit methods [1,36,37,29,10,14]. Traditional accounting firms like Deloitte and KPMG have incorporated blockchain technology to ensure transaction completeness and enhance audit and tax services, respectively [1]. Besides, some academic studies [36,37,29,14] have created new blockchain frameworks to audit on-chain transactions with minimal privacy compromise. For instance, Narula et al. introduced zkLedger, which offers privacy-preserving audit verification [37], while Chatzigiannis et al. developed Miniledger to maintain user anonymity during audits [14]. Moreover, some research has explored user/organization-level blockchain audits, allowing auditors to verify information on public ledgers. Proof of solvency methods demonstrate an organization's asset sufficiency without exposing sensitive data [17,22]. This contrasts with the less explored net income/spending proofs involving off-chain transactions. Ernst & Young developed the CAAT tool to manage accounting and tax reporting for blockchain-recorded cryptocurrency transactions [10]. However, a universal approach for auditing layer-two transactions in PCs is lacking.

Off-chain compression-chains. The technique is to compress a large number of transactions into a single transaction to reduce the cost of transaction verification. Particularly, ZK-rollup is to compress a large number of transactions into a single transaction while generating a zero-knowledge proof (zk-SNARK) to verify that all these transactions are valid [26]. This method commits a summary of transaction history to the blockchain, but it falls short as a comprehensive solution for auditing PCs due to its limitations: (1) its exclusive design for Ethereum, (2) dependence on a trusted data availability committee for complete transaction verification, and (3) inability to prevent discarding and arbitrary-ordering attacks in PC audits. Unlike traditional PCs, which operate without intermediaries, our proposed IvyAPC introduces no such entities, offering a more universal auditing approach.

Blockchain witnessing scheme. Blockchain witnessing serves to deter equivocation in both on-chain and off-chain transactions by using a confirmed on-chain transaction as a witness [42,44,34]. Although schemes like Catena [44] can prevent collusion in presenting false off-chain transactions to auditors by publishing a corresponding on-chain transaction for each off-chain one, this approach conflicts with the primary goal of PCs, which is to decrease the volume of on-chain transactions.

B Preliminaries

Accountable assertions protocol was initially proposed to prevent equivocation for Bitcoin-compatible PCs [42], e.g., double-spending in the off-chain. A party (called assertor $\mathcal{R}$) would be monetarily punished by exposing its secret key

if it asserts more than two statements in the same context. Formally, an accountable assertions protocol is a tuple of PPT algorithms $\sum :=$ (Gen, Assert, Verify, Extrtact) as follows:

- $(apk, ask, auxsk) \leftarrow \mathsf{Gen}(\lambda)$: is a PPT algorithm that on input of a security parameter λ outputs a public key *apk*, a secret key *ask*, and an auxiliary secret key *auxsk*.
- $\tau/\bot \leftarrow \mathsf{Assert}(ask, auxsk, ct, st)$: is a DPT algorithm that on input of a secret key *ask*, an auxiliary secret key *auxsk*, a context *ct*, and a statement *st*, outputs an assertion τ or $\bot$ to indicate failure.
- $1/0 \leftarrow \mathsf{Verify}(apk, ct, st, \tau)$: is a DPT algorithm that on input of a public key *apk*, a context *ct*, a statement *st*, and an assertion τ, outputs 1 iff τ is a valid assertion of *st* under *ct*, and outputs 0 otherwise.
- $ask/\bot \leftarrow \mathsf{Extract}(apk, ct, st_0, st_1, \tau_0, \tau_1)$: is a DPT algorithm that on the input of a public key *apk*, a context *ct*, two different statements st_0, st_1, and two assertions τ_0, τ_1, outputs *ask* if τ_0 and τ_1 are valid assertions of st_0 and st_1, respectively under *ct*, and outputs $\bot$ otherwise.

The accountable assertions protocol satisfies the security properties of extractability and secrecy (see §C.2 for formal definitions). In addition, we specify a property called *unforgeability* in the random oracle (see §1 for formal proof). It states that any party can not forge a valid assertion on behalf of an assertor.

Definition 4 (Unforgeability). *An accountable assertions protocol* $\sum :=$ (Gen, Assert, Verify, Extrtact) *is unforgeable if the following holds for any PPT forger F, negligible function $negl(\lambda)$ and the security parameter λ with large enough values:*

$$Prob \begin{bmatrix} (apk, ask, auxsk) \leftarrow \mathsf{Gen}(1^{\lambda}) \\ \wedge \tau \leftarrow F^{\mathsf{Assert}(\mathsf{ask},\mathsf{auxsk},\cdot,\cdot)} \\ \wedge F \text{ never asked } \mathsf{Assert}(\cdot) \text{ to assert pair } (\mathsf{ct}, \mathsf{st}); \\ 1 \leftarrow \mathsf{Verify}(\mathsf{apk}, \mathsf{ct}, \mathsf{st}, \tau) \end{bmatrix} < negl(\lambda) \tag{2}$$

C Security and Construction of AAFPK

C.1 Instantiation of AAFPK

The implementation of AAFPK is constructed based on Krawczyk and Rabin chameleon hash function [32]. We instantiate a stateless AAFPK in the elliptic curve setting and discrete logarithms assumption. Specifically, we choose the branch of a tree $n = 2$ and its height $\ell = 64$. Thus, the space of the context is $\{0, 1\}^{64}$. We choose the elliptic curve $secp256k1$ group of a base point g of order prime q. Let $\mathbb{G}$ be a group with a generator g and prime q. In addition, we let H_0, H_1, and H_2 be collision-resistant HMAC − SHA256 hash functions , where SHA256 is used to model as a global random oracle or a PRF function [45].

- **Key Generation**: The key generation algorithm takes a security parameter λ, and picks a secure elliptic curve with a base point g of prime order p and

q, where $p = 2q + 1$. It sets a global order number $n := 0$, then randomly chooses a value $\alpha \in \mathbb{Z}_q^*$, and sets a key pair $(csk, cpk) = (\alpha, X) \leftarrow \mathsf{Gen}(1^\lambda)$, where $X = g^\alpha \; mod \; p$. It finally outputs a public key $apk := (cpk, z)$, a secret key $ask := csk$, an auxiliary secret information $ausk := \mathsf{H_0(csk)}$, where $z := \mathsf{H_0(y_1^1, ..., y_m^1)},\ \mathsf{y_i^1} := \mathsf{CH(PRF(id, i, 0); PRF(id, i, 1))},\ \mathsf{i} \in [\mathsf{m}]$, m refers to the branching factor of a tree, id is an identifier for the position of the root node.
- **Secret Key Transformation**: The secret key transformation algorithm takes the secret key ask, and a value $\omega \in \mathbb{Z}_q^*$ as inputs, and outputs a different representative secret key $ask' := \mathsf{ChgASK(ask}, \omega) := \mathsf{ECHFPK.ChgChCSK(csk}, \omega) := \alpha + \omega$.
- **Public Key Transformation**: The public key transformation algorithm takes the public key apk, and a value ω as inputs, and outputs a different representative public key apk', where $apk' := \mathsf{ChgAPK(apk}, \omega) := (\mathsf{cpk', z}) := \big(\mathsf{ECHFPK.ChgChCPK(cpk}, \omega), \mathsf{z}\big) := (\mathsf{g}^{\alpha+\omega}, \mathsf{z})$.
- **Assertion**: The assertion algorithm takes $(ask, auxk,\ ct, st)$ as inputs, and computes the assertion path from the leaf node Y_ℓ to the root node Y_1, where ℓ refers to the height of the tree, $Y_c := (y_1^c, ..., y_m^c)$ for $c \in [\ell]$.
- **Verification**: The verification algorithm takes $(apk, ct, st, \tau, \omega)$ as inputs, and parses apk and τ. It reconstructs the path from the leaf node to the root node and verifies if the result equals the Merkle root.
- **Extraction**: The extraction algorithm takes $(cpk, ct, st_0, st_1, \tau_0, \tau_1)$ as inputs and reconstructs the path from the leaf node to the root node for both (st_0, τ_0) and (st_1, τ_1). It can extract ask using the $\mathsf{ECHFPK.ExtractCsk}$ algorithm.

C.2 Security of Accountable Assertions

An accountable assertions protocol fulfills the *extractability* and *secrecy* property [42]. Specifically, extractability states that if the assertor has made two different statements $st_0 \neq st_1$ in the same context ct, then the probability that the secret key can not be extracted is negligible.

Definition 5 (Extractability). *An accountable assertions protocol* $\sum := (\mathsf{Gen}, \mathsf{Assert}, \mathsf{Verify}, \mathsf{Extrtact})$ *is extractable if for every PPT adversary* $\mathcal{A}$*, the following holds:*

$$Prob \begin{bmatrix} \mathsf{Extrtact(apk, ct, st_0, st_1}, \tau_0, \tau_1) \neq \mathsf{ask} \\ \wedge \forall b \in \{0,1\}, \mathsf{Verify(apk, ct, st_b}, \tau_\mathsf{b}) = 1 \\ \wedge st_0 \neq st_1 : (apk, ct, st_0, st_1, \tau_0, \tau_1) \leftarrow \mathcal{A}(1^\lambda) \end{bmatrix} < negl(\lambda), \quad (3)$$

where $negl(\cdot)$ is a negligible function.

The secrecy property states that the adversary $\mathcal{A}$ is unable to extract the secret key by asking the challenger to assert the given statements for chosen contexts with a given order number.

Definition 6 (Secrecy). *An accountable assertions protocol* $\sum :=$ (Gen, Assert, Verify, Extrtact) *is secrecy if for every PPT adversary $\mathcal{A}$, there exists a negligible function $negl(\cdot)$ such that the probability of the following experiment* assertChal *returns 1 is negligible:*

Experiment $\mathsf{assertChal}_{\mathcal{A},\Pi}(\lambda)$:
1: $(apk, ask, auxsk) \leftarrow \mathsf{Gen}(1^\lambda)$
2: $Q := \emptyset$
3: $ask^* \leftarrow \mathcal{A}^{\mathsf{Assert}'(\mathsf{ask},\mathsf{auxsk},',')}(apk)$
4: return 1 iff $ask^* = ask$
5: $\quad \wedge(\nexists ct, st_0, st_1, st_0 \neq st_1$
6: $\quad \wedge\{(ct, st_0), (ct, st_1) \subseteq Q\})$
7: **Assertion Oracle** $\mathsf{Assert}'(\mathsf{ask}, \mathsf{auxsk}, \mathsf{ct}, \mathsf{st})$
8: $\quad Q := Q \cup \{ct, st\}$
9: $\quad$ return $\mathsf{Assert}(\mathsf{ask}, \mathsf{auxsk}, \mathsf{ct}, \mathsf{st})$

Definition 7 (Consistency). *An accountable assertions protocol* $\sum :=$ (Gen, Assert, Verify, Extrtact) *is consistent if for every PPT adversary $\mathcal{A}$, and any context and statement pair (ct, st), the following holds:*

$$Prob\begin{bmatrix}(apk, ask, ausk) \leftarrow \mathsf{Gen}(1^\lambda) \\ \tau \leftarrow \mathsf{Assert}(\mathsf{ask}, \mathsf{ausk}, \mathsf{ct}, \mathsf{st}) \\ \wedge b \leftarrow \mathsf{Verify}(\mathsf{apk}, \mathsf{ct}, \mathsf{st}, \tau) \\ \wedge b' \leftarrow \mathsf{Verify}(\mathsf{apk}, \mathsf{ct}, \mathsf{st}, \tau); \\ b \neq b'\end{bmatrix} < negl(\lambda) \tag{4}$$

The consistency property is implicit in conventional accountable assertions, which state that the probability of the verification algorithm outputting different results for the same (ct, st, τ) is negligible.

C.3 Security of AAFPK

In this subsection, we provide the security proof of Theorem 1.

Proof. Suppose that there exists an adversary $\mathcal{A}$ breaks the unforgeability of the accountable assertions with flexible public key scheme with non-negligible probability. It is easy to construct a PPT algorithm $\mathcal{B}$ that makes use of $\mathcal{A}$ to break the collision-resistance of ECHFPK with non-negligible probability. Algotithm $\mathcal{B}$ is given the public key cpk of ECHFPK, and runs $\mathcal{A}$ as follows.

$\mathcal{B}$ chooses a key κ for the pseudo-random function and computes $y_i^1 = \mathsf{CH}(\mathsf{cpk}, \mathsf{PRF}_\kappa(\mathsf{p}, \mathsf{i}, 0); \mathsf{PRF}_\kappa(\mathsf{p}, \mathsf{i}, 1))$ for $i \in [m]$, where p is an identifier for the position of the root node. Then, it sets $z = \mathsf{H}_0(y_1^1, \ldots, y_m^1)$ and sends $apk = (cpk, z)$ to the adversary. Let q be the maximum number of unique assertion queries of $\mathcal{A}$. $\mathcal{B}$ chooses q bitstrings $\{Q_i^{\mathsf{L}}\}_{i \leq i \leq q}$ in the output space of L and q bitstrings

$\{Q_i^{\mathsf{H}}\}_{i \leq i \leq q}$ in the output space of H_2 uniformly at random. Assuming that the leaf entry with the number Q_i^{L}, counted across all leaves from left to right, is $y_i = Q_i^{\mathsf{H}}$. $\mathcal{B}$ also maintains three lists L-list, H_0-list, H_2-list and a counter ix, where ix is set to be 0 initially. Let us now explain how $\mathcal{B}$ answers the query made by $\mathcal{A}$.

L **Query**: On input ct, $\mathcal{B}$ does the following. If there exists a record $\langle ct, I(ct), Q_{I(ct)}^{\mathsf{L}}\rangle$ in L-list, it return $Q_{I(ct)}^{\mathsf{L}}$; else it sets $I(ct) = ++ix$, adds $\langle ct, I(ct), Q_{I(ct)}^{\mathsf{L}}\rangle$ into L-list and returns $Q_{I(ct)}^{\mathsf{L}}$ to $\mathcal{A}$.

H_0 **Query**: On input $(y_1, \ldots, y_m)$, $\mathcal{B}$ does the following. If there exists a record $\langle (y_1, \ldots, y_m), z\rangle$ in H_0-list, it return z; else it chooses a random value z, adds $\langle (y_1, \ldots, y_m), z\rangle$ into H_0-list and returns z to $\mathcal{A}$.

H_2 **Query**: On input (s, r), $\mathcal{B}$ does the following. If there exists a record $\langle (s, r), y\rangle$ in H_2-list, it return y; else it chooses a random y, adds $\langle (s, r), y\rangle$ into H_2-list and returns y to $\mathcal{A}$.

$\mathsf{Assertion}$ **Query**: On input (ct, st), $\mathcal{B}$ chooses a random values r and sets $\mathsf{H_2(CH(H_1(st); r), r) = Q^{H}_{I(ct)}}$. Then $\mathcal{B}$ computes and returns the assertion path $((r, f_\ell^{I(ct)}, a_\ell^{I(ct)}), (r_{\ell-1}, f_{\ell-1}^{I(ct)}, a_{\ell-1}^{I(ct)}), \ldots, (r_1, f_1^{I(ct)}, a_1^{I(ct)}))$, where for $1 \leq i < \ell$, $r_i = \mathsf{PRF}_\kappa(\mathsf{p_i^{I(ct)}}, \mathsf{a_i^{I(ct)}}, 1)$ and sets $H_0(y_1^i, \ldots, y_m^i) = \mathsf{PRF}_\kappa(\mathsf{p_{i+1}^{I(ct)}}, \mathsf{a_{i+1}^{I(ct)}}, 0)$.

Observe that, with overwhelming probability, the simulation towards $\mathcal{A}$ is correct. If $\mathcal{A}$ outputs a valid assertion τ for (ct, st), which never queried by $\mathcal{A}$, one can find a collision somewhere on the path from the leaf to the root. This contradicts the collision-resistance of ECHFPK and concludes the proof for our construction.

C.4 Ideal Functionality $\mathcal{F}_{\mathsf{AA}}$ and Security Proof

The ideal functionality of $\mathcal{F}_{\mathsf{AA}}$ for the accountable assertions protocol is illustrated as in Fig. 2. Functionality $\mathcal{F}_{\mathsf{AA}}$ can also model the functionality of the AAFPK scheme. The assertor can send the key generation request KeyGen to transform the key pair.

In the following, we present the security proof of AAFPK for Theorem 2 in the UC framework. Specifically, an AAFPK scheme can be translated into a protocol $\widetilde{\sum}$. When assertor $\mathcal{R}$ runs $\widetilde{\sum}$ receives an input (KeyGen, sid), it checks the session identifier that $sid := (\mathcal{R}, sid')$ for some sid'. If yes, it runs Gen to generate the secret key pair $(ask, auxsk)$ and outputs the public key apk. When receiving an input (Assert, sid, apk', ct, st) from an sid that already generates an assertion secret key ask, it computes $\mathsf{Assert(ask, auxsk, ct, st)}$ and outputs (Asserted, sid, apk', ct, st, τ). When a participant receives an input $(\mathsf{Verify, sid, apk', ct, st}, \tau)$, it computes $b := \mathsf{Verify(ask, auxsk, ct, st}, \tau)$ and outputs (Verified, sid, b).

Proof. We prove $\widetilde{\sum}$ *UC-realizes* $\mathcal{F}_{\mathsf{AA}}$ in two steps. For the "only if" direction, we prove that $\widetilde{\sum}$ violates the unforgeability of Definition 4. That is, for any ideal adversary $\mathcal{S}$, the environment $\mathcal{E}$ is able to distinguish whether it interacts with

$\mathcal{A}$ in real-process $\widetilde{\sum}$ or $\mathcal{S}$ in idea-process $\mathcal{F}_{\mathsf{AA}}$. We assume that environment $\mathcal{E}$ does not corrupt any party or send messages to $\mathcal{A}$. Then, assume that $\widetilde{\sum}$ is not unforgeable, namely, there is a forger F that can successfully generate an assertion without running the Assert algorithm. Here, we let environment $\mathcal{E}$ run a copy of F internally. Whenever F asks its oracle to assert a statement st in a context ct, $\mathcal{E}$ sends $(\text{ASSERT}, sid, apk', ct, st)$ to $\mathcal{R}$ and sends the output to F. Suppose that F generates an assertion record (apk', st, ct, τ). Environment $\mathcal{E}$ proceeds as follows:

- If (apk', st, ct) was used to generate an assertion before, then $\mathcal{E}$ outputs 0 and aborts.
- Else, $\mathcal{E}$ sends $(\text{VERIFY}, sid, apk', ct, st, \tau)$ to some party $\mathcal{P}_i$, and outputs the result.

It can be seen that when $\mathcal{E}$ interacts with the adversary $\mathcal{S}$ in the ideal-process $\mathcal{F}_{\mathsf{AA}}$, $\mathcal{E}$ never outputs 1. However, when $\mathcal{E}$ interacts with the adversary $\mathcal{A}$ in the real-process $\widetilde{\sum}$, $\mathcal{E}$ outputs 0 with non-negligible probability due to the existence of the forger F.

On the other hand, for the "if" direction, we show that $\widetilde{\sum}$ does not realize $\mathcal{F}_{\mathsf{AA}}$. Namely, for any ideal adversary $\mathcal{S}$ and a real work adversary A, there is an environment $\mathcal{E}$ that can distinguish whether it interacts with $\mathcal{A}$ in real-process $\widetilde{\sum}$ or $\mathcal{S}$ in idea-process $\mathcal{F}_{\mathsf{AA}}$. The "generic" simulator $\mathcal{S}$ proceeds as follows:

1. **Key Generation**. Upon $(\mathsf{KeyGen}, \mathsf{sid}) \leftarrow \mathcal{F}_{\mathsf{AA}}$, where $sid = (\mathcal{R}, sid')$, then:
   - $\mathcal{R}$ is honest and $\mathcal{S}$ does not send KeyGen on behalf of corrupt parties.
   - Invoke the simulated assertor "$\mathcal{R}$", and send $(\mathsf{KeyGen}, \mathsf{sid}) \rightarrow$"$\mathcal{R}$", and wait for $(\mathsf{KeyGen}, \mathsf{sid}, \mathsf{apk}) \leftarrow$"$\mathcal{R}$".
   - Send $(\mathsf{KeyGen}, \mathsf{sid}, \mathsf{apk}) \rightarrow \mathcal{F}_{\mathsf{AA}}$.
2. **Assertion Generation**. Upon $(\mathsf{Assert}, \mathsf{sid}, \mathsf{apk}', \mathsf{ct}, \mathsf{st}) \leftarrow \mathcal{F}_{\mathsf{AA}}$, where $sid = (\mathcal{R}, sid')$, then:
   - If $\mathcal{R}$ is honest, $\mathcal{S}$ chooses a honest party "$\mathcal{P}_i$", and send $(\mathsf{Assert}, \mathsf{sid}, \mathsf{apk}', \mathsf{ct}, \mathsf{st}) \rightarrow$ "$\mathcal{P}_{\mathsf{i}}$", and wait for $(\mathsf{Asserted}, \mathsf{sid}, \tau) \leftarrow$ "$\mathcal{P}_{\mathsf{i}}$".
   - Send $(\mathsf{Asserted}, \mathsf{sid}, \tau) \rightarrow \mathcal{F}_{\mathsf{AA}}$.
3. **Assertion Verification**. Upon $(\mathsf{Verify}, \mathsf{sid}, \mathsf{apk}', \mathsf{ct}, \mathsf{st}, \tau) \leftarrow \mathcal{F}_{\mathsf{AA}}$, where $sid = (\mathcal{R}, sid')$:
   - Invoke a participant and run $f \leftarrow \widetilde{\Pi}.\mathsf{Verify}\ (\mathsf{apk}', \mathsf{ct}, \mathsf{st}, \tau)$, and send $(\mathsf{Verified}, \mathsf{sid}, \mathsf{f}) \rightarrow \mathcal{F}_{\mathsf{AA}}$.
4. **Extraction**. Upon $(\mathsf{Extract}, \mathsf{sid}, \mathsf{apk}', \mathsf{ct}, \mathsf{st}_0, \mathsf{st}_1, \tau_0, \tau_1) \leftarrow \mathcal{F}_{\mathsf{AA}}$, where $sid = (\mathcal{R}, sid')$:
   - $\mathcal{S}$ invokes a participant and runs $\mathsf{ask} \leftarrow \widetilde{\sum}.\ \mathsf{Extract}(\mathsf{apk}', \mathsf{ct}, \mathsf{st}_0, \mathsf{st}_1, \tau_0, \tau_1)$, and send $(\mathsf{Extracted},\ \mathsf{sid}, \mathsf{ask}) \rightarrow \mathcal{F}_{\mathsf{AA}}$.

We demonstrate the proof of this direction by constructing a forger F. Specifically, F runs a simulated copy of adversary $\mathcal{A}$ and environment $\mathcal{E}$. In addition, it simulates an interaction with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{AA}}$ for $\mathcal{E}$. In the key generation, F sends the public key apk from its input to $\mathcal{A}$, rather than generating $(apk, ask, auxk)$ by

running $\mathsf{Gen}(\cdot)$. In the assertion generation, F lets its oracle assert a statement st in a context ct and obtain the assertion τ, rather than running the assertion generation algorithm Assert. In particular, if $\mathcal{E}$ asks $\mathcal{A}$ to corrupt the assertor $\mathcal{R}$, then F aborts and outputs a failure message. Suppose that the simulated $\mathcal{E}$ activates an uncorrupted party $\mathcal{P}_i$ with an input $(\textsc{Verify}, sid, apk', ct, st, \tau)$, F verifies if (ct, st) pair was never asserted under the transformed public key ask' while $\mathsf{Verify}(\mathsf{apk}', \mathsf{ct}, \mathsf{st}, \tau) = 1$, i.e., constituting a forgery successfully. If yes, F outputs (ct, st, τ) and aborts. Otherwise, F continues the simulation.

$\mathcal{E}$'s view depends on the success probability of F. If this probability is negligible (i.e., the unforgeability property holds in the real world), then the success probability that environment $\mathcal{E}$ can distinguish an interaction with $\mathcal{A}$ in real-process $\widetilde{\sum}$ or $\mathcal{S}$ in ideal-process $\mathcal{F}_{\mathsf{AA}}$ is also negligible. However, this is contradictory to our security assumption. Thus, as long as the success probability of F is non-negligible, then $\mathcal{E}$'s view keeps the same in the real world and ideal world, i.e., $\widetilde{\sum}$ *UC-realizes* $\mathcal{F}_{\mathsf{AA}}$.

D Additional Material on IvyAPC

D.1 On-chain Timestamp Witnessing

Generally, the timestamp of an on-chain transaction is set as when it is sent into the transaction pool. In PCs, only the final commitment transaction will be sent to the transaction pool; hence, most commitment transactions do not have a timestamp. It is problematic if we let transacting parties set the timestamp of a commitment transaction in PCs, since they can set it arbitrarily for their benefit, breaking the basic audit requirement that demands transactions to attach a relatively accurate timestamp. Therefore, we introduce an on-chain timestamp witnessing approach to enable auditors to check the timestamp of commitment transactions.

As illustrated in Fig. 3, on-chain transactions that have been included in a block are ordered by their timestamp. Suppose that the latest block is BK_{121} when the later commitment transaction $\mathcal{T}_{\mathsf{com}_1}$ is generating. Party A chooses the earliest timestamp of on-chain transaction, i.e., $\mathcal{T}_{\mathsf{on}_0}.ts$, as a reference to set the timestamp of $\mathcal{T}_{\mathsf{com}_1}$. In particular, if transacting parties generate multiple commitment transactions within this block period, e.g., $\mathcal{T}_{\mathsf{com}_2}$ and $\mathcal{T}_{\mathsf{com}_3}$, they can choose a set of on-chain transactions with the earliest consecutive timestamps, i.e., $\mathcal{T}_{\mathsf{on}_1}.ts$ and $\mathcal{T}_{\mathsf{on}_2}.ts$, as references to set their timestamps.

Although dishonest parties might choose any block within the longest blockchain as a reference, they can only choose the block whose height is larger than or equal to the block height which includes the genesis commitment transaction, and less than the height of the latest block. Otherwise, the auditor could detect this dishonest behavior and regard a payment channel as an "illegal" channel. We have to point out that such an on-chain timestamp witnessing approach can not guarantee the precise timestamp of commitment transactions. Setting the

The ideal functionality $\mathcal{F}_{\mathsf{AA}}$ interacts with an assertor $\mathcal{R}$, a set of participants $\mathcal{V} := \{A, B, ...\}$ who act as the verifiers, and the ideal adversary $\mathcal{S}$ (i.e., the simulator).
Parameters: context and statement space $\mathcal{CS}$, $\mathcal{SS}$.
Variables: initially empty key, context, and assertion records KEY, CT, AT.

Key Generation

Upon $(\textsc{KeyGen}, sid) \leftarrow \mathcal{R}$, verify that $sid := (\mathcal{R}, sid')$ for some sid'. If not, then ignore the request. Otherwise, proceed as follows:

- If $sid \neq (\mathcal{R}, sid')$, then abort.
- Else, send $(\textsc{KeyGen}, sid) \rightarrow \mathcal{S}$, and wait for $(\textsc{KeyGened}, sid, apk) \leftarrow \mathcal{S}$. If not received, then abort.
- Create record $(\mathsf{KEY}, \mathsf{sid}, \mathsf{apk})$.
- Output $(\textsc{KeyGened}, sid, apk) \rightarrow \mathcal{R}$.

Assertion Generation

Upon $(\textsc{Assert}, sid, apk', ct, st) \leftarrow \mathcal{R}$ with $ct \in \mathcal{CS},\ st \in \mathcal{SS}$, verify that $sid := (\mathcal{R}, sid')$ for some sid'. If not, then ignore the request. Otherwise, proceed as follows:

- If no record $(\mathsf{KEY}, \mathsf{sid}, \mathsf{apk}')$ exists, then abort.
- If $\mathcal{R}$ is honest, then:
  - If a record $(\mathsf{CT}, \mathsf{sid}, \mathsf{apk}', \mathsf{ct})$ exists, then abort.
  - Else, send $(\textsc{Assert}, sid, ct, st) \rightarrow \mathcal{S}$, and wait for $(\textsc{Asserted}, sid, \tau) \leftarrow \mathcal{S}$. Verify that if $(\mathsf{AT}, \mathsf{sid}, \mathsf{apk}', \mathsf{ct}, \mathsf{st}, \tau, \mathsf{false})$ exists, then send an error message to $\mathcal{R}$ and abort.
  - Else, create record $(\mathsf{CT}, \mathsf{sid}, \mathsf{apk}', \mathsf{ct})$ and $(\mathsf{AT}, \mathsf{sid}, \mathsf{apk}', \mathsf{ct}, \mathsf{st}, \tau, \mathsf{true})$.
- Else, i.e., $\mathcal{R}$ is corrupt, then:
  - Send $(\textsc{Assert}, sid, ct, st) \rightarrow \mathcal{S}$, and wait for $(\textsc{Asserted}, sid, \tau) \leftarrow \mathcal{S}$. If not received, then abort.
  - Create record $(\mathsf{AT}, \mathsf{sid}, \mathsf{apk}', \mathsf{ct}, \mathsf{st}, \tau, \mathsf{true})$.
- Execute $\mathsf{ExtractAsk}(\mathsf{apk}', \mathsf{ct}, \mathsf{st}, \tau)$ and output $(\textsc{Asserted}, sid, ct, st, \tau) \rightarrow \mathcal{R}$.

Assertion Verification

Upon $(\textsc{Verify}, sid, apk', ct, st, \tau) \leftarrow V$ for $V \in \mathcal{V}$, then send $(\textsc{Verify}, sid, apk', ct, st, \tau) \rightarrow \mathcal{S}$. Upon receiving $(\textsc{Verified}, sid, ct, st, \tau, b) \leftarrow \mathcal{S}$, then proceed as follows:

- If $apk' = apk$ and a record $(\mathsf{AT}, \mathsf{sid}, \mathsf{apk}, \mathsf{ct}, \mathsf{st}, \tau, \mathsf{b})$ exists, then set $f \leftarrow b$.
- Else if $apk' = apk$, $\mathcal{R}$ is honest and no record $(\mathsf{AT}, \mathsf{sid}, \mathsf{apk}, \mathsf{ct}, \mathsf{st}, *, \mathsf{true})$ exists, set $f \leftarrow 0$ (*Unforgeability*).
- Else, if $(\mathsf{AT}, \mathsf{sid}, \mathsf{apk}, \mathsf{ct}, \mathsf{st}, \tau, \mathsf{b}')$ exists, then set $f \leftarrow b'$.
- Else, set $f \leftarrow b$, and create a record $(\mathsf{AT}, \mathsf{sid}, \mathsf{apk}', \mathsf{ct}, \mathsf{st}, \mathsf{f})$. Output $(\textsc{Verified}, sid, f) \rightarrow V$.

Subprocedure $\mathsf{ExtractAsk}(\mathsf{apk}', \mathsf{ct}, \mathsf{st}, \tau)$
(execute at the end of each assertion generation)

For each $(\mathsf{AT}, \mathsf{sid}, \mathsf{apk}', \mathsf{ct}, \mathsf{st}, \tau, \mathsf{true})$ generated in $\mathsf{AssertionGeneration}$, check if $(\mathsf{AT}, \mathsf{sid}, \mathsf{apk}, \mathsf{ct}, \mathsf{st}', \tau', \mathsf{true})$ exists for $apk = apk'$ and $st \neq st'$. If yes, send $(\textsc{Extract}, sid, apk', ct, st_0, st_1, \tau_0, \tau_1) \rightarrow \mathcal{S}$, and wait for $(\textsc{Extracted}, sid, ask') \leftarrow \mathcal{S}$. Then, output $(\textsc{Extracted}, sid, ask') \rightarrow \mathcal{P}$ (*Extractability*).

Fig. 2: The ideal functionality $\mathcal{F}_{\mathsf{AA}}$ modeling an accountable assertions protocol.

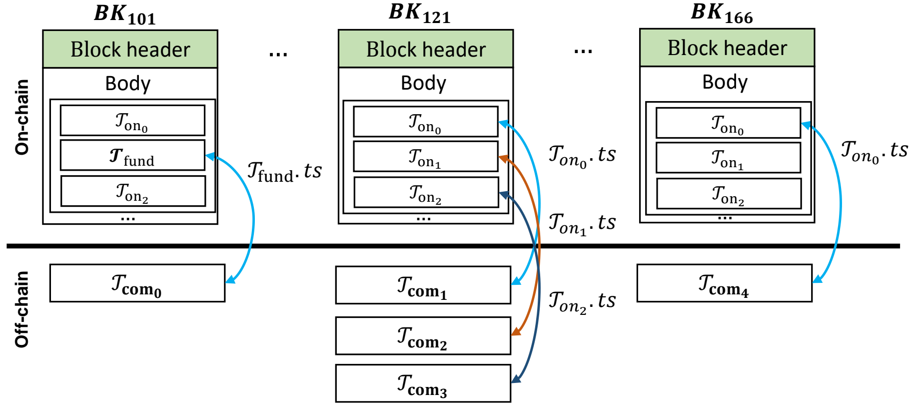


Fig. 3: An illustration of on-chain timestamp witnessing.

precise timestamp of commitment transactions without requiring a trusted third party is challenging. We leave this as our future work.

D.2 Ideal World Functionalities

In this work, the security analysis is established via the IvyAPC protocol [13]. The GUC framework permits the entity $\mathcal{E}$ unfettered interaction and engagement across diverse protocols sans any restrictions. This subsection embarks with an exposition of the foundational principles of the UC framework. Subsequently, it outlines the ideal functionalities encompassing the global random oracle and foundational blockchain-based protocols. Thereafter, it articulates the ideal functionality $\mathcal{F}_{AA}$, dedicated to the accountable assertions protocol, and $\mathcal{F}_{\mathsf{IvyAPC}}$, tailored for the IvyAPC protocol.

UC framework. This framework allows one to analyze a protocol in isolation using the *universal composition operation* [12]. This operation enables us to construct a protocol based on cryptographic building blocks while still proving its security. The UC framework is essentially a simulation-based approach that compares the execution of a protocol π in the *real world* with an idealized protocol $\mathcal{F}$ in the *ideal world*. In the real world, the execution of the protocol π is modeled as the tuples $\{\mathcal{E}, \mathcal{A}, \mathcal{P}\}$, where $\mathcal{E}$ refers to the environment who sends inputs to $\mathcal{A}$ and $\mathcal{P}$, and $\mathcal{P} = \{p_1, p_2, ...\}$ refer to the participants of π. Each participant may execute different modules of π. The adversary $\mathcal{A}$ that can corrupt a participant upon receiving the instruction from $\mathcal{E}$ and send any information on behalf of the corrupt participant. In the ideal world, the execution of the ideal functionality $\mathcal{F}$ is modeled as ideal functionality that interacts with $\mathcal{E}$ and the ideal adversary $\mathcal{S}$ who simulates the behaviors of the adversary $\mathcal{A}$.

Communication model.. The parties communicate with others through authenticated secure channels. We assume a synchronous communication network

The global functionality $\mathcal{G}_{\mathsf{Ledger}}$ is running with a set of participants $\{P_u\}_{u\in[\zeta]}$ who have a balance $\{c_u\}_{u\in[\zeta]} \in \mathbb{N}$, and a partial function F for frozen coin, and a global storage $\mathsf{Storage} := \emptyset$. It executes upon receiving the following queries:

Read Clock: Upon $(\text{CLOCK-READ}, sid) \leftarrow P_u$, then send $(\text{CLOCK-READ}, sid) \rightarrow \mathcal{G}_{\mathsf{clock}}$. Upon $(\mathsf{clock-Read}, \mathsf{sid}, \mathsf{n}) \leftarrow \mathcal{G}_{\mathsf{clock}}$, then send $(\text{CLOCK-READ}, sid, n) \rightarrow P_u$.
Read States: Upon $(\text{READ}, sid) \leftarrow P_u$, then send $(\text{RECEIPT}, \mathsf{Storage}[\mathsf{sid}]) \rightarrow *$, or $\perp \rightarrow *$ if not found.
Write States: Upon $(\text{WRITE}, sid, inps) \leftarrow P_u$, if $\mathsf{Storage}[\mathsf{sid}]$ not found, store $\mathsf{Storage}[\mathsf{sid}] = \mathsf{inps}$. If store success in blockchain, send $(\text{RECEIPT}, evids) \rightarrow *$. Otherwise, send $(\text{REJECT}, evids) \rightarrow *$.
Verify States: Upon $(\text{VERIFY}, sid, state) \leftarrow P_u$, if $state \in \mathsf{Storage}[\mathsf{sid}]$, output $(true) \rightarrow *$. Otherwise, send $(false) \rightarrow *$.
Update Funds: Upon $(\text{UPDATE}, P_u, \hat{c}) \leftarrow \mathcal{E}$ with $\hat{c} \geq 0$. Set $c_i = \hat{c}$ and send $(\text{UPDATED}, P_u, \hat{c}) \rightarrow *$.
Freeze Funds: Upon $(\text{FREEZE}, sid, P_u, \hat{c}) \leftarrow \mathcal{F}^a$, check if $c_i \geq \hat{c}$. If not, reply with $(nofunds, p_i, \hat{c})$. Otherwise, set $c_u = c_u - \hat{c}$, and store $(sid, \hat{c})$ in F, and $(\text{FROZEN}, sid, p_u, \hat{c}) \rightarrow *$.
Unfreeze Funds: Upon $(\text{UNFREEZE}, sid, P_u, \hat{c}) \leftarrow \mathcal{F}$, check if $(sid, \hat{c}') \in F$ with $\hat{c}' \geq \hat{c}$. If yes, update $(sid, \hat{c}')$ to $(sid, \hat{c}' - \hat{c})$ in F, set $c_u = c_u + \hat{c}$, and send $(\text{UNFROZEN}, sid, P_u, \hat{c}) \rightarrow *$.

[a] $\mathcal{L}.sid$ denotes a session sid of the functionality $\mathcal{L}$

Fig. 4: The global functionality $\mathcal{G}_{\mathsf{Ledger}}$ modeling a distributed ledger.

where each operation is performed in rounds. One round refers to the maximal time that a message is sent from one party to another. Further, the adversary $\mathcal{A}$ can control the arrival order of a message at a certain party in a given round.

Global Ledger Functionality. To support the security analysis, we utilize the existing model of [20] to model the basic properties of a global functionality $\mathcal{G}_{\mathsf{Ledger}}$ for a blockchain. Each party P have a balance $c_P \in \mathbb{N}$. Function F is used to map an identifier sid to a number of coins to be locked in the blockchain. The balance of party P can be updated via the instruction update, freeze and unfreeze sent from the environment $\mathcal{E}$. $\mathcal{G}_{\mathsf{Ledger}}$ has a public storage Storage to store a state and transactions list.

Ideal functionalities for global random oracles. In our setting, a global random oracle functionality $\mathcal{G}_{pRO}$ is utilized to construct the simulator [11]. Specifically, $\mathcal{G}_{pRO}$ is modeled as a programmable and observable hash function in the hybrid world that can be called both in the real and ideal world. It returns a random number $y \in \{0,1\}^{\mu}$ when receiving a hash query for a value r. If a query is performed before, then it returns the same value. Specifically, the programmability allows the adversary to fix the response for certain queries if they do not have queried before. The observability allows the adversary to observe all of the querying and response.

Ideal functionalities for blockchain-based protocols. Following [5,6,27], we use the GUC framework to model a global ledger $\mathcal{G}_{\mathsf{Ledger}}$ (see Appendix §D.1). It can support users to handle coins such as $\mathsf{Update}, \mathsf{Freeze}, \mathsf{Unfreeze}$ and $\mathsf{Read}, \mathsf{Write}$ operations. $\mathcal{G}_{\mathsf{Ledger}}$ relies on the global clock ideal functionality $\mathcal{G}_{\mathsf{Clock}}$ to set its execution round [30], allowing each party to be aware of the current round. We formalize the operation of $\mathsf{ReadClocks}$ in $\mathcal{G}_{\mathsf{Clock}}$ to obtain the current execution round. For simplicity, we utilize the blockchain length to represent the execution round. Specifically, environment $\mathcal{E}$ initializes $\mathcal{G}_{\mathsf{Ledger}}$ by (i) instructing $\mathcal{G}_{\mathsf{Ledger}}$ to generate public security parameter λ, and (ii) instructing each party to generate a key pair (pk, sk), and submitting all public keys to the $\mathcal{G}_{\mathsf{Ledger}}$. The states of $\mathcal{G}_{\mathsf{Ledger}}$ are public that all participants can access, while they can only be updated by environment $\mathcal{E}$ through a Update request. To update balance states, parties specify other ideal functionalities to send Freeze or $\mathsf{Unfreeze}$ message, rather than interacting with $\mathcal{G}_{\mathsf{Ledger}}$ directly.

Ideal functionality for accountable assertion. The ideal functionality $\mathcal{F}_{\mathsf{AA}}$ describes a setting where three parties exist, i.e., an assertor $\mathcal{R}$ who asserts a statement for a context, and a set of verifiers $\mathcal{P} := \{A, B, ...\}$ who are responsible for verifying the correctness of generated assertions, and the ideal adversary $\mathcal{S}$ (see §C.4 for the functionality $\mathcal{F}_{\mathsf{AA}}$). Let $\mathcal{F}_{\mathsf{AA}}$ provide an "assertion registry service" where $\mathcal{R}$ can register the statement, context, and assertion. As illustrated in Fig. 2, functionality $\mathcal{F}_{\mathsf{AA}}$ has three operations: $\mathsf{KeyGeneration}$, $\mathsf{AssertionGeneration}$, $\mathsf{AssertionVerification}$. The verifiers extract the secret key of $\mathcal{R}$ in case of equivocation using the subprocedure $\mathsf{ExtractAsk}$.

Key Generation. Upon receiving $(\textsc{KeyGen}, sid)$ from the registry $\mathcal{R}$, the ideal functionality $\mathcal{F}_{\mathsf{AA}}$ validates whether the assertor's identifier corresponds with the one encoded within the session identifier (sid). If a mismatch occurs, $\mathcal{F}_{\mathsf{AA}}$ disregards the KeyGen request, effectively ensuring that only the initiating party has the authority to generate or modify a secret key. Within the operational scope of $\mathcal{F}_{\mathsf{AA}}$, the simulator $\mathcal{S}$ possesses the capability to ascertain the value of the public key. This is predicated on the premise that the unforgeability and extractability underpinning accountable assertions are independent of the specific value of the public key.

Assertion Generation. In this phase, the ideal functionality denoted as $\mathcal{F}_{\mathsf{AA}}$, checks for any prior registration requests from any participant. This step is crucial to ensure that only the first party to make a request can use the assertion feature of the created functionality instance. Generating an assertion depends on the security status of $\mathcal{R}$. Specifically, if $\mathcal{R}$ has not been compromised, $\mathcal{F}_{\mathsf{AA}}$ applies a unique assertion rule for each public key, based on the CT (Context Table) record. This rule is stringent: if a duplicate context ct appears in CT, the process stops immediately, and an error is reported. However, if there are duplicate requests, $\mathcal{F}_{\mathsf{AA}}$ hands over the decision-making to the simulator $\mathcal{S}$ by issuing the Assert command. This step is designed to determine the outcome fairly. As the phase concludes, $\mathcal{F}_{\mathsf{AA}}$ initiates the $\mathsf{ExtractAsk}$ subroutine to ascertain whether two assertions, τ_0 and τ_1, bearing identical context ct, are present

in AT. Affirmative findings prompt $\mathcal{F}_{\mathsf{AA}}$ to transmit $(\mathsf{AT}, \mathsf{apk}', \mathsf{ct}, \mathsf{st}_0, \tau_0)$ and $(\mathsf{AT}, \mathsf{apk}', \mathsf{ct}, \mathsf{st}_1, \tau_1)$ to $\mathcal{S}$, thereby enabling $\mathcal{S}$ to simulate a party to compute a secret key ask'.

Assertion Verification. $\mathcal{F}_{\mathsf{AA}}$ ensures unforgeability given an honest $\mathcal{R}$. More concretely, without any assertions in AT, the chance of $\mathcal{F}_{\mathsf{AA}}$ returning "0" is negligible. Conversely, with a compromised $\mathcal{R}$, $\mathcal{S}$ controls the outcome, undermining unforgeability in real and ideal worlds. This illustrates that accountable assertions fail to block a corrupted $\mathcal{R}$ from validating any assertion for any statement-context pair, as $\mathcal{S}$ can act as $\mathcal{R}$. Thus, during a $\mathsf{Verify}(apk, ct, st, \tau)$ request, $\mathcal{S}$ force a "1" output, irrespective of the assertion history of (ct, st).

Ideal functionality for IvyAPC. To succinctly model the security framework of IvyAPC, we introduce an ideal functionality, $\mathcal{F}_{\mathsf{IvyAPC}}$, within the hybrid model. This functionality employs three pivotal components as subroutines: the programmable random oracle $\mathcal{G}_{pRO}$, a global ledger $\mathcal{G}_{\mathsf{Ledger}}$, and an ideal accountable assertions protocol $\mathcal{F}_{\mathsf{AA}}$ (refer to Section §D.2). $\mathcal{F}_{\mathsf{IvyAPC}}$, delineated in Figure 5, is designed in alignment with the generalized ideal Garbled Circuits (GC) functionality, as discussed in previous works [5,6,7]. It encapsulates five critical operations within its framework: (i) Create, to initiate channel openings; (ii) Update, for modifying channel states; (iii) Close, aimed at terminating a channel; (iv) Punish, which is invoked upon identifying misbehavior to impose penalties; and (v) Audit, for auditing the generated off-chain transactions.

Create. The Create operation is composed of two main phases: *audit trail generation phase* and *preparation phase*. (1) *Audit Trail Generation Phase*: This phase begins when the parties involved in the transaction exchange a *partial* audit trail and notify $\mathcal{E}$. If any party fails to respond, it's interpreted as a refusal to create the channel. After exchanging the partial audit trails, the parties compile a complete audit trail. This complete trail is then used to create the initial commitment transaction. (2) *Preparation Phase*: The preparation phase kicks off when $\mathcal{F}_{\mathsf{IvyAPC}}$ receives a set of transaction identifiers, denoted as $\overrightarrow{tid} := (tid_1, ..., tid_k)$ from $\mathcal{S}$, where k represents the total number of ways the final state $\gamma.state$ of the channel can be revealed. The creation of an auditable payment channel γ is considered successful following the publication of the funding transaction $\mathcal{T}_{\mathsf{fund}}$ on the blockchain.

Update. This procedure unfolds in two distinct phases: *preparation and revocation* . (1) *Preparation Phase*: This phase begins when $\mathcal{F}_{\mathsf{IvyAPC}}$ receives transaction identifiers from $\mathcal{S}$. To establish a new state, $\mathcal{F}_{\mathsf{IvyAPC}}$ mandates the parties to exchange audit trails. This audit data is encapsulated within a commitment transaction before the exchange of (pre-)signatures between the parties. (2) *Revocation Phase*: This phase ensures that an off-chain payment is only considered successful after both parties have nullified the previous state. Either party can halt the audit trail generation process. This can happen if a party chooses not to send an initial audit agreement message, labeled AUDIT-PREOK, or fails to dispatch a setup confirmation, SETUP-OK. It's crucial to understand that a

The ideal functionality $\mathcal{F}_{\mathsf{IvyAPC}}$ is parameterized with two algorithms $\mathsf{ForceClose}$and $\mathsf{Compare}$. It interacts with two parties A and B, and an auditor $\mathcal{D}$, the ideal adversary $\mathcal{S}$. Besides, it internally interacts with a global ledger $\mathcal{G}_{\mathsf{Ledger}}$. It executes upon receiving the following requests:

Create. Upon $(\textsc{create}, id, \gamma, \mathcal{T}^{A}_{on}.tid) \leftarrow A$, verify that it is well-formed (contain valid identifier and signature). If not, ignore the request. Otherwise, proceed as follows:

1) If already received $(\textsc{create}, id, \gamma, \mathcal{T}^{B}_{on}.tid, B)$ (i.e., both parties agree to open the channel) within a particular round[a], then send $(\textsc{audit-PreReq}, \gamma.id) \rightarrow A$. If $(\textsc{audit-PreOK}, \gamma.id) \leftarrow A$, then send $(\textsc{audit-Req}, \gamma.id) \rightarrow A$. If $(\textsc{audit-OK}, \gamma.id) \leftarrow A$, then wait if a transaction $\mathcal{T}_{\mathsf{fund}}$ is written on $\mathcal{G}_{\mathsf{Ledger}}$ within a particular round, where $\mathcal{T}_{\mathsf{fund}} := (tid, (\mathcal{T}^{A}_{on}.tid, \mathcal{T}^{B}_{on}.tid), \gamma.cash, \gamma.st)$. If yes, let $\mathcal{S}$ define $\overrightarrow{tid}$ and send $(\mathsf{clock-Read}, \mathsf{sid}) \rightarrow \mathcal{G}_{\mathsf{Ledger}}$. If receive $(\mathsf{clock-Read}, \mathsf{sid}, \mathsf{n}) \leftarrow \mathcal{G}_{\mathsf{Ledger}}$, then set $\gamma.id := id$, $\gamma.ctList := \gamma.ctList \cup (\overrightarrow{tid}, n)$. After that, append $\Gamma(\gamma.id) = (\gamma, \mathcal{T}_{\mathsf{fund}})$ and send $(\textsc{created}, \gamma.id) \rightarrow \gamma.us$. Otherwise, abort and stop.
2) Else, record $(\textsc{create}, sid, \gamma, \mathcal{T}^{A}_{on}.tid, A)$, and wait for $(\textsc{create}, id, \mathcal{T}^{B}_{on}.tid) \leftarrow B$. If no such message is received within a particular round, then stop.

Update: Upon $(\textsc{update}, id, \boldsymbol{\theta}) \leftarrow A$, parse $(\gamma), \mathcal{T}_{\mathsf{fund}}) := \Gamma(\gamma.id)$, set $\gamma' := \gamma$, and $\gamma'.state := \boldsymbol{\theta}$, and proceeds as follows:

1) Within a particular round, let $\mathcal{S}$ define $\overrightarrow{tid}$. Then, send $(\textsc{update-Req}, sid, \boldsymbol{\theta}, \overrightarrow{tid}) \rightarrow B$. Then, send $(\textsc{audit-PreReq}, id) \rightarrow A$.
2) If $(\textsc{audit-PreOK}, id, \tau) \leftarrow A$, then send $(\textsc{audit-PreOK}, id) \rightarrow B$ and $(\textsc{audit-Req}, id) \rightarrow A$. Else, stop.
3) If $(\textsc{audit-OK}, id) \leftarrow A$, then send $(\textsc{audit-OK}, id) \rightarrow B$ and $(\textsc{setup}, sid) \rightarrow A$. Else, distinguish as follows:
   - If B is honest or instructed by $\mathcal{S}$, stop (reject audit information).
   - Else, execute $\mathsf{ForceClose}(\gamma.\mathsf{id})$ and stop.
4) If $(\textsc{setup-OK}, id) \leftarrow A$, then send $(\textsc{setup-OK}, sid) \rightarrow B$. Else, stop.
5) If $(\textsc{update-OK}, sid) \leftarrow B$, then send $(\textsc{update-OK}, sid) \rightarrow A$. Else, check:
   - If B is honest or instructed by $\mathcal{S}$, stop (reject update).
   - Else, execute $\mathsf{ForceClose}(\gamma.\mathsf{id})$ and stop.
6) If $(\textsc{revoke}, sid) \leftarrow A$, then send $(\textsc{revoke-Req}, sid) \rightarrow B$. Else, execute $\mathsf{ForceClose}(\gamma.\mathsf{id})$ and stop.
7) If receive $(\textsc{revoke}, sid) \leftarrow B$, then send $(\mathsf{clock-Read}, \mathsf{sid}) \rightarrow \mathcal{G}_{\mathsf{Ledger}}$. If receive $(\mathsf{clock-Read}, \mathsf{sid}, \mathsf{n}) \leftarrow \mathcal{G}_{\mathsf{Ledger}}$, then send $(\textsc{updated}, sid, \gamma') \rightarrow \gamma.us$ and set $\gamma.ctList := \gamma.ctList \cup (\overrightarrow{tid}, n)$ and stop. Else, check:
   - If B is honest, execute $\mathsf{ForceClose}(\gamma.\mathsf{id})$ and stop.
   - Else, i.e., B is corrupt, wait for a particular rounds and check if $\mathcal{T}_{\mathsf{fund}}$ is unspent. If yes, set $\boldsymbol{\theta}_{old} := \gamma.state$, $\gamma.state := (\boldsymbol{\theta}_{old}, \boldsymbol{\theta})$ $\Gamma(\gamma.id) := (\gamma, \mathcal{T}_{\mathsf{fund}})$. Execute $\mathsf{ForceClose}(\gamma.\mathsf{id})$ and stop.

Close: Upon $(\textsc{close}, id) \leftarrow A$, then proceeds as follows:

1) If already record $(\textsc{close}, id, B)$ (i.e., both parties agree to close channel) within a particular round. Then, parse $(\gamma, \mathcal{T}_{\mathsf{fund}}) := \Gamma(\gamma.id)$ and proceeds as follows:
   - If a transaction $\mathcal{T}'_{\mathsf{com}}$ is written on $\mathcal{G}_{\mathsf{Ledger}}$ within a certain round, where $\mathcal{T}'_{\mathsf{com}}.in = \mathcal{T}_{\mathsf{fund}}.tid$ and $\mathcal{T}'_{\mathsf{com}}.out = \gamma.st$. Then set $\Gamma(id) := (\gamma', \mathcal{T}_{\mathsf{fund}})$, where $\gamma' := (\gamma.id, \perp, \perp, \perp, \gamma.ctList)$, and send $(\textsc{closed}, sid) \rightarrow \gamma.us$ and stop.
   - Else, if at least one of parties behaves dishonestly, execute $\mathsf{ForceClose}(\gamma.\mathsf{id})$. Otherwise, send $(\mathsf{ERROR}) \leftarrow \gamma.\mathsf{us}$ and stop.
2) Else, record $(\textsc{close}, id, A)$, and wait for $(\textsc{close}, id) \leftarrow B$ from B at most within a particular rounds (defined by $\mathcal{S}$). If not receive such message, execute $\mathsf{ForceClose}(\gamma.\mathsf{id})$.

Audit: Upon $(\textsc{audit}, id) \leftarrow \mathcal{D}$, parse $(\gamma, \mathcal{T}_{\mathsf{fund}}) := \Gamma(id)$. If $\mathcal{T}_{\mathsf{fund}}$ is still unspent on $\mathcal{G}_{\mathsf{Ledger}}$, then send $(\textsc{audit-Rej}, id) \rightarrow \mathcal{D}$ and stop. Otherwise, parse $\gamma := (\gamma.id, \perp, \perp, \perp, \gamma.ctList)$ and send $(\textsc{audit-stReq}, id) \rightarrow p$ where $p \in \{A, B\}$. If within a particular round, receive $(\textsc{audit-ST}, id, \widetilde{ctList}) \leftarrow p$, then distinguish as follows:

1) If p is honest and $\mathsf{Compare}(\gamma.\mathsf{ctList}, \widetilde{\mathsf{ctList}}) = 1$, send $(\textsc{audited}, id, 1) \rightarrow \mathcal{D}$ and stop.
2) Otherwise, send $(\textsc{audit-Comp}, id, \widetilde{ctList}) \rightarrow \mathcal{S}$, upon receiving $(\textsc{audit-Comped}, id, b) \leftarrow \mathcal{S}$, send $(\textsc{audited}, id, b) \rightarrow \mathcal{D}$ and stop.

Punish: At the end of each round, check if a transaction $\mathcal{T}_{\mathsf{com}}$ is written to $\mathcal{G}_{\mathsf{Ledger}}$, where $\mathcal{T}_{\mathsf{com}}.in := \Gamma(\gamma.id).\mathcal{T}_{\mathsf{fund}}$. If yes, then proceeds as follows:

- (**Punish**): For party $p \in \{A, B\}$ honest, if in a particular round, a transaction $\mathcal{T}''_{\mathsf{com}}$, where $\mathcal{T}''_{\mathsf{com}}.in := \mathcal{T}'_{\mathsf{com}}.tid$ and $\mathcal{T}''_{\mathsf{com}}.out := (\gamma.v, \gamma.\theta)$, is written on $\mathcal{G}_{\mathsf{Ledger}}$. Then, send $(\textsc{punished}, sid) \rightarrow p$, set $\Gamma(\gamma.id) := \perp$ and stop.
- (**Close**): Check if $\Gamma(\gamma.id) := (\perp, \mathcal{T}_{\mathsf{fund}})$ before round $t_0 + \Delta$, or within a particular round, a transaction $\mathcal{T}''_{\mathsf{com}}$, where $\mathcal{T}''_{\mathsf{com}}.in := \mathcal{T}'_{\mathsf{com}}.tid$ and $\mathcal{T}''_{\mathsf{com}}.out \in \gamma.st$, is written on $\mathcal{G}_{\mathsf{Ledger}}$. If it is the latter case, set $\Gamma(\gamma.id) := (\perp, \mathcal{T}_{\mathsf{fund}})$ and send $(\textsc{closed}, sid) \rightarrow \gamma.us$.
- (**Error**): Otherwise, send $(\textsc{error}) \rightarrow \gamma.us$.

Subprocedure **ForceClose(id)**: Let $(\gamma, \mathcal{F}_{\mathsf{fund}}) := \Gamma(sid)$. If $\mathcal{F}_{\mathsf{fund}}$ is still unspent on $\mathcal{G}_{\mathsf{Ledger}}$ within a particular round, then send $(\textsc{error}) \rightarrow \gamma.us$ and stop. Otherwise, the latest in a predefined round, $m \in \{\textsc{punished}, \textsc{closed}, \textsc{extract}, \textsc{error}\}$ is output via Punish.

[a] A particular round means several execution rounds defined before the protocol starts.

Fig. 5: The ideal functionality $\mathcal{F}_{\mathsf{IvyAPC}}$ modeling an auditable PC scheme.

commitment transaction lacking signatures from both parties does not qualify for audit. After the completion of each revocation phase, $\mathcal{F}_{\mathsf{IvyAPC}}$ records the commitment transaction in $\gamma.ctList$.

Close. The channel γ is considered closed once a commitment transaction that spends the funding transaction has appeared on the blockchain.

Audit. The operation is triggered by the auditor, $\mathcal{D}$. When an AUDIT request is received from the environment, $\mathcal{F}_{\mathsf{IvyAPC}}$ first assesses the channel state, γ. If γ remains open, an AUDIT-REJ message is dispatched to $\mathcal{D}$. In contrast, if γ is closed, $\mathcal{F}_{\mathsf{IvyAPC}}$ solicits the list of commitment transactions, $\widetilde{ctList}$, from either party involved in the transaction. $\mathcal{F}_{\mathsf{IvyAPC}}$ then undertakes a comparison between the obtained commitment transactions and those previously recorded, focusing on three key parameters: (i) the aggregate transaction amount, (ii) the sequence of transactions, and (iii) the timestamp associated with each transaction. Consistency across these dimensions indicates a successful audit, leading $\mathcal{F}_{\mathsf{IvyAPC}}$ to return a value of 1. Conversely, should the submitting party be compromised, $\mathcal{F}_{\mathsf{IvyAPC}}$ redirects the audit inquiry to the adversary, $\mathcal{S}$, granting them the discretion to determine the outcome. A return value of 0 from $\mathcal{F}_{\mathsf{IvyAPC}}$ signals a breach of the audit's completeness criterion within the APC framework.

UC-security definition. Let Π be the real world protocol with access to $\mathcal{G}_{pRO}$ and $\mathcal{G}_{\mathsf{Ledger}}$. Let $REAL_{\pi,\mathcal{A},\mathcal{E}}(\lambda, inps)$ be the output of an environment $\mathcal{E}$ interacting with Π and an adversary $\mathcal{A}$, where $inps$ is the inputs of $\mathcal{E}$. On the other hand, let $IDEAL_{\mathcal{F},\mathcal{S},\mathcal{E}}(\lambda, inps)$ be the ideal world protocol that is related to the execution of the ideal protocol. Then, the security definition of Π is as follows.

Definition 8 (GUC Security of Π). *Let λ be security parameter, Π be a protocol in the $(\mathcal{G}_{pRO}, \mathcal{G}_{\mathsf{Ledger}}, \mathcal{F}_{\mathsf{AA}})$-hybrid world. Then Π UC-*realizs *an ideal functionality $\mathcal{F}_{\mathsf{IvyAPC}}$ in the $(\mathcal{G}_{pRO}, \mathcal{G}_{\mathsf{Ledger}})$-hybrid world if for any PPT adversary $\mathcal{A}$, i.e., there exists a simulator $\mathcal{S}$ that $\mathcal{E}$ cannot distinguish whether it is interacting with the real world Π and $\mathcal{A}$ or the ideal world $\mathcal{F}_{\mathsf{IvyAPC}}$ and $\mathcal{S}$, i.e.,*

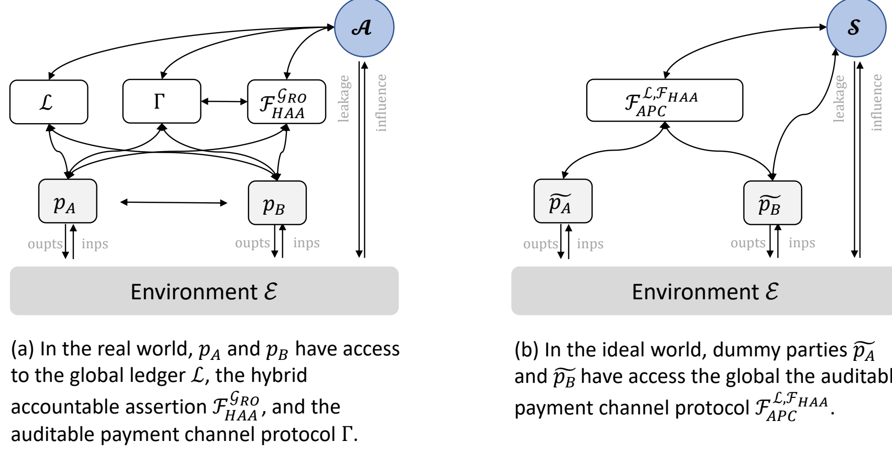
$$\begin{aligned} &\forall \mathcal{E},\ REAL^{\mathcal{G}_{\mathsf{Ledger}},\mathcal{F}_{\mathsf{AA}}}_{\Pi,\mathcal{G}_{pRO},\mathcal{A},\mathcal{E}}(\lambda, inps) \approx_c \\ &IDEAL^{\mathcal{F}_{\mathsf{IvyAPC}},\mathcal{F}_{AA},\mathcal{G}_{pRO},\mathcal{G}_{\mathsf{Ledger}}}_{\mathcal{S},\mathcal{E}}(\lambda, inps) \end{aligned} \tag{5}$$

D.3 Formal Protocol Specification

The formal protocol specification of IvyAPC is shown in Fig. 7, Fig. 8, and Fig. 9. Following [5], we utilize lowercase letters to denote the communication between transacting parties, and uppercase to denote the communication between transacting parties and environment. We let $\mathsf{Multi}-\mathsf{Sig}_{\mathsf{pk}_1,\mathsf{pk}_2}$ denote a multiple signature $\mathsf{One}-\mathsf{Sig}_{\mathsf{pk}_1}$ and $\mathsf{One}-\mathsf{Sig}_{\mathsf{pk}_2}$.

D.4 Performance Discussion

We define the following succinctness property for characterizing any auditable payment channel protocol in terms of data size published on the blockchain.



(a) In the real world, p_A and p_B have access to the global ledger $\mathcal{L}$, the hybrid accountable assertion $\mathcal{F}_{HAA}^{\mathcal{G}_{RO}}$, and the auditable payment channel protocol Γ.

(b) In the ideal world, dummy parties $\widetilde{p_A}$ and $\widetilde{p_B}$ have access the global the auditable payment channel protocol $\mathcal{F}_{APC}^{\mathcal{L},\mathcal{F}_{HAA}}$.

Fig. 6: An illustration of a simulation with honest transacting parties in IvyAPC.

Definition 9 (Succinctness). *An auditable payment channel protocol IvyAPC* := (Create, Update, Close, Punish, Audit) *is succinct if for auditing each payment channel the size of data published on blockchain as required by the protocol is no greater than $\mathcal{O}(1)$.*

Theorem 3. *The IvyAPC protocol is succinct.*

Proof. It is straightforward that the size of data published on the blockchain is not relevant to the number of commitment transactions through the chain-linking structure in the payment channel.

D.5 UC Security Proof

Here, we prove that the IvyAPC protocol UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{IvyAPC}}$.

Theorem 4. *Let $\widetilde{\sum}$ be a complete and unforgeable accountable assertion with a flexible public key, $\mathcal{G}_{\mathsf{Ledger}}$ be a global blockchain functionality, $\mathcal{G}_{pRO}$ be a programmable random oracle. Then the IvyAPC protocol UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{IvyAPC}}$.*

Proof. The security proof is shown in the Appendix §D.6.

D.6 Formal Security Proof

In this subsection, we provide the security proof for Theorem 4. Our proof strategy is to construct a simulator $\mathcal{S}$ that can simulate the behaviors of corrupted in the real world and interact with the functionality $\mathcal{F}_{\mathsf{IvyAPC}}$ in the ideal

Auditable Payment Channels

Here, we highlight the main differences to the previous generalized channels [5]. We abbreviate $B := \gamma.\mathsf{otherParty(A)}$ for $A \in \gamma.us$.

Initialize

Party A upon $(\text{INIT}, id) \leftarrow \mathcal{E}$:

- Generate a blockchain key pair (pk_A, sk_A), set the accountable assertions key pair $ask_A := sk_A$ and $apk_A := (pk_A, z)$ and the auxiliary secret information $auxk_A$ and randomly choose $\omega_A \in \mathbb{Z}_q$, and send $(\mathsf{keyGened}, \mathsf{id}, \mathsf{apk_A}, \omega_\mathsf{A}) \to \mathcal{B}$.
- If $(\mathsf{keyGen}, \mathsf{id}, \mathsf{apk_B}, \omega_\mathsf{B}) \leftarrow \mathcal{B}$, then send $(\mathsf{keyGened}, \mathsf{id}) \to \mathcal{E}$.

Create

Party A upon $(\text{CREATE}, id, \gamma, \mathcal{T}^A_{\mathsf{on}}.tid) \leftarrow \mathcal{E}$:

1) Generate $(r_A, r_A) \leftarrow GenA$ and $(Y_A, y_A) \leftarrow GenA$ and send $(\mathsf{createInfo}, \mathsf{id}, \mathsf{h_A}, \mathsf{Y_A}) \to \mathsf{B}$.
2) If $(\mathsf{createInfo}, \mathsf{id}, \mathsf{h_B}, \mathsf{Y_B}) \leftarrow \mathsf{B}$, create $[\mathcal{T}_{\mathsf{fund}}]$, $[\mathcal{T}_{\mathsf{com}}]$, and $[\mathcal{T}_{\mathsf{spl}}]$. Else, stop.
3) Compute $\tau^A \to \mathsf{Assert}(\mathsf{ask_A}, \mathsf{auxk_A}, [\mathcal{T}_{\mathsf{fund}}], [\mathcal{T}_{\mathsf{com}}])$ and send $(\mathsf{Assert-req}, \mathsf{id}, \tau^\mathsf{A}) \to \mathsf{B}$.
4) If $(\mathsf{Asserted-ok}, \mathsf{id}, \tau^\mathsf{B}, \varsigma^\mathsf{B}) \leftarrow \mathsf{B}$ for $\varsigma^B := H(\tau_A || r_B)$, send $(\text{CLOCK-READ}, id) \to \mathcal{G}_{\mathsf{Ledger}}$. Else, stop.
5) If $(\text{CLOCK-READ}, id, ts) \leftarrow \mathcal{G}_{\mathsf{Ledger}}$, then set $\eta^A_0 := \tau_A || \tau_B || \varsigma^A || \varsigma^B || ts$ for $\varsigma^A := H(\tau_B || r_A)$, and send $(\mathsf{audit-Info}, \mathsf{id}, \eta^\mathsf{A}_0) \to \mathsf{B}$ and $(\text{AUDIT-INFO}, id) \to \mathcal{E}$. Else, stop.
6) If receive $(\text{AUDIT-OK}, id) \leftarrow \mathcal{E}$, then set $[\mathcal{T}_{\widetilde{\mathsf{com}}}] := \{[\mathcal{T}_{\mathsf{com}}], \eta^A_0\}$, and compute $s^A_{\mathsf{com}} \to \mathsf{pSign}_{\mathsf{sk_A}}([\mathcal{T}_{\widetilde{\mathsf{com}}}], \mathsf{Y_B})$ and $s^A_{\mathsf{spl}} \to \mathsf{Sign}_{\mathsf{sk_A}}([\mathcal{T}_{\mathsf{spl}}])$, and send $(\mathsf{createCom}, \mathsf{id}, \mathsf{s^A_{com}}, \mathsf{s^A_{spl}}) \to \mathsf{B}$. Else, stop.
7) If $(\mathsf{createCom}, \mathsf{id}, \mathsf{s^B_{com}}, \mathsf{s^B_{spl}}) \leftarrow \mathsf{B}$, verify $s^B_{\mathsf{com}}, s^B_{\mathsf{spl}}$ and compute $s^A_{\mathsf{fund}} \to \mathsf{Sign}_{\mathsf{sk_A}}([\mathcal{T}_{\mathsf{fund}}])$, and send $(\mathsf{createFund}, \mathsf{id}, \mathsf{s^A_{fund}}) \to \mathsf{B}$. Else, stop.
8) If $(\mathsf{createFund}, \mathsf{id}, \mathsf{s^B_{fund}}) \leftarrow \mathsf{B}$, verify s^B_{fund} and compute $\mathcal{T}_{\mathsf{fund}} := \{[\mathcal{T}_{\mathsf{fund}}], s^A_{\mathsf{fund}}, s^B_{\mathsf{fund}}\}$, and send $(\text{WRITE}, id, \mathcal{T}_{\mathsf{fund}}) \to \mathcal{G}_{\mathsf{Ledger}}$. Else, stop.
9) If $\mathcal{T}_{\mathsf{fund}}$ is written on the blockchain within a particular round, compute $\mathcal{T}_{\mathsf{com}} := \{[\mathcal{T}_{\widetilde{\mathsf{com}}}], \mathsf{Sign}_{\mathsf{sk_A}}([\mathcal{T}_{\mathsf{com}}]), \mathsf{Adapter}(\mathsf{s^B_{com}}, \mathsf{y_A})\}$, $\mathcal{T}_{\mathsf{spl}} := \{[\mathcal{T}_{\mathsf{spl}}], s^A_{\mathsf{spl}}, s^B_{\mathsf{spl}}\}$. Then set $comTX := \mathcal{T}_{\mathsf{com}}$ and $\Gamma^A(id) := (\gamma, \mathcal{T}_{\mathsf{fund}}, (\mathcal{T}_{\mathsf{com}}, r_A, h_B, Y_B, s^A_{\mathsf{com}}), comTX, \mathcal{T}_{\mathsf{spl}})$, and send $(\text{CREATED}, id) \to \mathcal{E}$.

Update

Party A upon $(\text{UPDATE}, id, \boldsymbol{\theta}) \leftarrow \mathcal{E}$:

Parse $(\gamma, \mathcal{T}_{\mathsf{fund}}) := \Gamma(\gamma.id)$, and set $n := len(\gamma.ctList) + 1$.

1) Generate $(h_A, r_A) \leftarrow GenA$ and $(Y_A, y_A) \leftarrow GenA$ and send $(\mathsf{createInfo}, \mathsf{id}, \mathsf{h_A}, \mathsf{Y_A}) \to \mathsf{B}$.

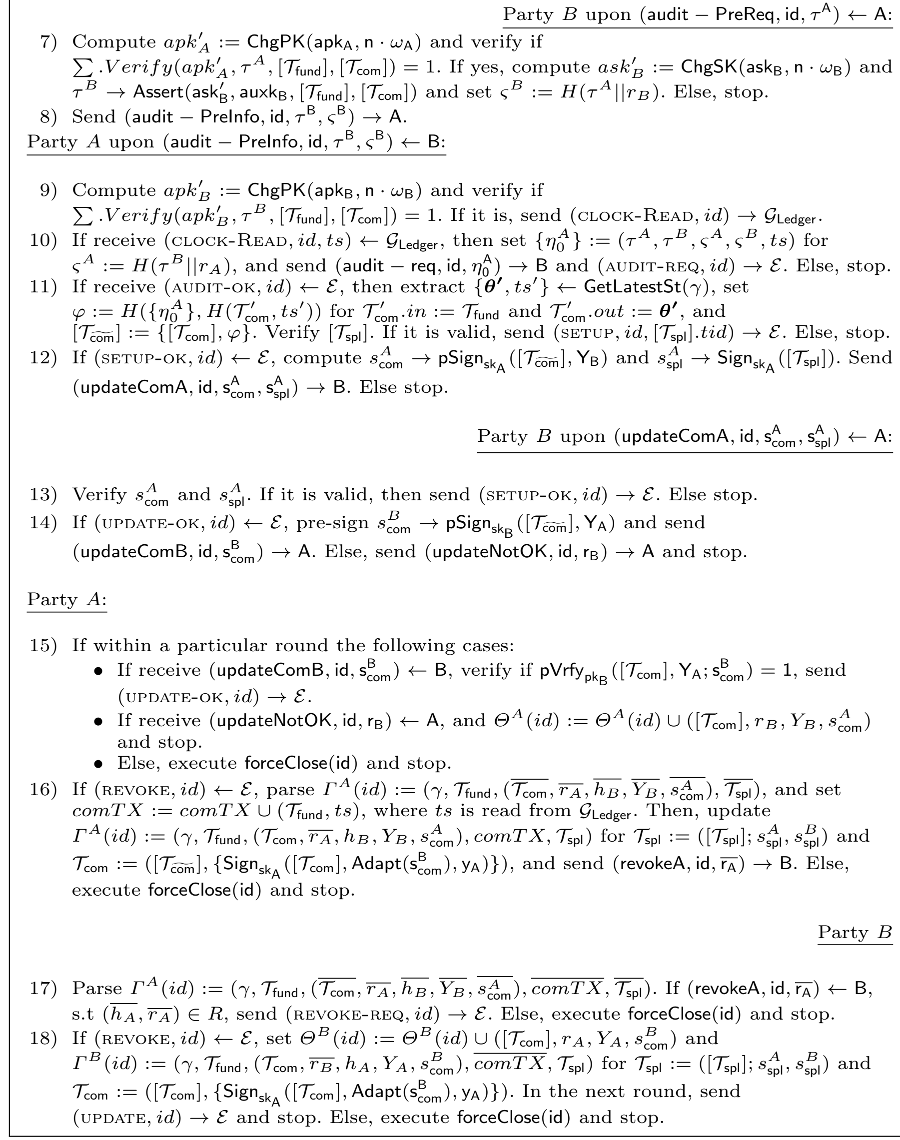
Party B upon $(\mathsf{createInfo}, \mathsf{id}, \mathsf{h_A}, \mathsf{Y_A}) \leftarrow \mathsf{A}$:

2) Generate $(h_B, r_B) \leftarrow GenB$ and $(Y_B, y_B) \leftarrow GenB$.
3) Extract $[\mathcal{T}_{\mathsf{fund}}]$ from $\Gamma^B(id)$, and compute $[\mathcal{T}_{\mathsf{com}}]$, $[\mathcal{T}_{\mathsf{spl}}]$.
4) Sign $s^B_{\mathsf{spl}} \leftarrow \mathsf{Sign}_{\mathsf{sk_B}}([\mathcal{T}_{\mathsf{spl}}])$, and send $(\mathsf{updateInfo}, \mathsf{id}, \mathsf{h_B}, \mathsf{Y_B}, \mathsf{s^B_{spl}}) \to \mathsf{A}$, $(\text{UPDATE-REQ}, id, \boldsymbol{\theta}, [\mathcal{T}_{\mathsf{spl}}].tid) \to \mathcal{E}$.

Party A upon $(\mathsf{updateInfo}, \mathsf{id}, \mathsf{h_B}, \mathsf{Y_B}, \mathsf{s^B_{spl}}) \leftarrow \mathsf{B}$:

5 Extract $[\mathcal{T}_{\mathsf{fund}}]$ from $\Gamma^B(id)$, and compute $[\mathcal{T}_{\mathsf{com}}]$, $[\mathcal{T}_{\mathsf{spl}}]$. If verify $\mathsf{Vrfy}_{\mathsf{pk_B}}([\mathcal{T}_{\mathsf{spl}}], \mathsf{s^B_{spl}}) = 1$, then send $(\text{AUDIT-PREREQ}, id) \to \mathcal{E}$. Else, stop.

6) If $(\text{AUDIT-PREOK}, id) \leftarrow \mathcal{E}$, then compute $ask'_A := \mathsf{ChgSK}(\mathsf{ask_A}, \mathsf{n} \cdot \omega_\mathsf{A})$ and $\tau^A \to \mathsf{Assert}(\mathsf{ask'_A}, \mathsf{auxk_A}, [\mathcal{T}_{\mathsf{fund}}], [\mathcal{T}_{\mathsf{com}}])$ and send $(\mathsf{audit-PreReq}, \mathsf{id}, \tau^\mathsf{A}) \to \mathsf{B}$.



Party B upon $(\mathsf{audit-PreReq}, \mathsf{id}, \tau^{\mathsf{A}}) \leftarrow \mathsf{A}$:

7) Compute $apk'_A := \mathsf{ChgPK}(\mathsf{apk_A}, \mathsf{n} \cdot \omega_{\mathsf{A}})$ and verify if $\sum.Verify(apk'_A, \tau^A, [\mathcal{T}_{\mathsf{fund}}], [\mathcal{T}_{\mathsf{com}}]) = 1$. If yes, compute $ask'_B := \mathsf{ChgSK}(\mathsf{ask_B}, \mathsf{n} \cdot \omega_{\mathsf{B}})$ and $\tau^B \rightarrow \mathsf{Assert}(\mathsf{ask'_B}, \mathsf{auxk_B}, [\mathcal{T}_{\mathsf{fund}}], [\mathcal{T}_{\mathsf{com}}])$ and set $\varsigma^B := H(\tau^A || r_B)$. Else, stop.
8) Send $(\mathsf{audit-PreInfo}, \mathsf{id}, \tau^{\mathsf{B}}, \varsigma^{\mathsf{B}}) \rightarrow \mathsf{A}$.

Party A upon $(\mathsf{audit-PreInfo}, \mathsf{id}, \tau^{\mathsf{B}}, \varsigma^{\mathsf{B}}) \leftarrow \mathsf{B}$:

9) Compute $apk'_B := \mathsf{ChgPK}(\mathsf{apk_B}, \mathsf{n} \cdot \omega_{\mathsf{B}})$ and verify if $\sum.Verify(apk'_B, \tau^B, [\mathcal{T}_{\mathsf{fund}}], [\mathcal{T}_{\mathsf{com}}]) = 1$. If it is, send $(\textsc{clock-Read}, id) \rightarrow \mathcal{G}_{\mathsf{Ledger}}$.
10) If receive $(\textsc{clock-Read}, id, ts) \leftarrow \mathcal{G}_{\mathsf{Ledger}}$, then set $\{\eta_0^A\} := (\tau^A, \tau^B, \varsigma^A, \varsigma^B, ts)$ for $\varsigma^A := H(\tau^B || r_A)$, and send $(\mathsf{audit-req}, \mathsf{id}, \eta_0^{\mathsf{A}}) \rightarrow \mathsf{B}$ and $(\textsc{audit-req}, id) \rightarrow \mathcal{E}$. Else, stop.
11) If receive $(\textsc{audit-ok}, id) \leftarrow \mathcal{E}$, then extract $\{\boldsymbol{\theta'}, ts'\} \leftarrow \mathsf{GetLatestSt}(\gamma)$, set $\varphi := H(\{\eta_0^A\}, H(\mathcal{T}'_{\mathsf{com}}, ts'))$ for $\mathcal{T}'_{\mathsf{com}}.in := \mathcal{T}_{\mathsf{fund}}$ and $\mathcal{T}'_{\mathsf{com}}.out := \boldsymbol{\theta'}$, and $[\mathcal{T}_{\widetilde{\mathsf{com}}}] := \{[\mathcal{T}_{\mathsf{com}}], \varphi\}$. Verify $[\mathcal{T}_{\mathsf{spl}}]$. If it is valid, send $(\textsc{setup}, id, [\mathcal{T}_{\mathsf{spl}}].tid) \rightarrow \mathcal{E}$. Else, stop.
12) If $(\textsc{setup-ok}, id) \leftarrow \mathcal{E}$, compute $s^A_{\mathsf{com}} \rightarrow \mathsf{pSign}_{\mathsf{sk_A}}([\mathcal{T}_{\widetilde{\mathsf{com}}}], \mathsf{Y_B})$ and $s^A_{\mathsf{spl}} \rightarrow \mathsf{Sign}_{\mathsf{sk_A}}([\mathcal{T}_{\mathsf{spl}}])$. Send $(\mathsf{updateComA}, \mathsf{id}, \mathsf{s^A_{com}}, \mathsf{s^A_{spl}}) \rightarrow \mathsf{B}$. Else stop.

Party B upon $(\mathsf{updateComA}, \mathsf{id}, \mathsf{s^A_{com}}, \mathsf{s^A_{spl}}) \leftarrow \mathsf{A}$:

13) Verify s^A_{com} and s^A_{spl}. If it is valid, then send $(\textsc{setup-ok}, id) \rightarrow \mathcal{E}$. Else stop.
14) If $(\textsc{update-ok}, id) \leftarrow \mathcal{E}$, pre-sign $s^B_{\mathsf{com}} \rightarrow \mathsf{pSign}_{\mathsf{sk_B}}([\mathcal{T}_{\widetilde{\mathsf{com}}}], \mathsf{Y_A})$ and send $(\mathsf{updateComB}, \mathsf{id}, \mathsf{s^B_{com}}) \rightarrow \mathsf{A}$. Else, send $(\mathsf{updateNotOK}, \mathsf{id}, \mathsf{r_B}) \rightarrow \mathsf{A}$ and stop.

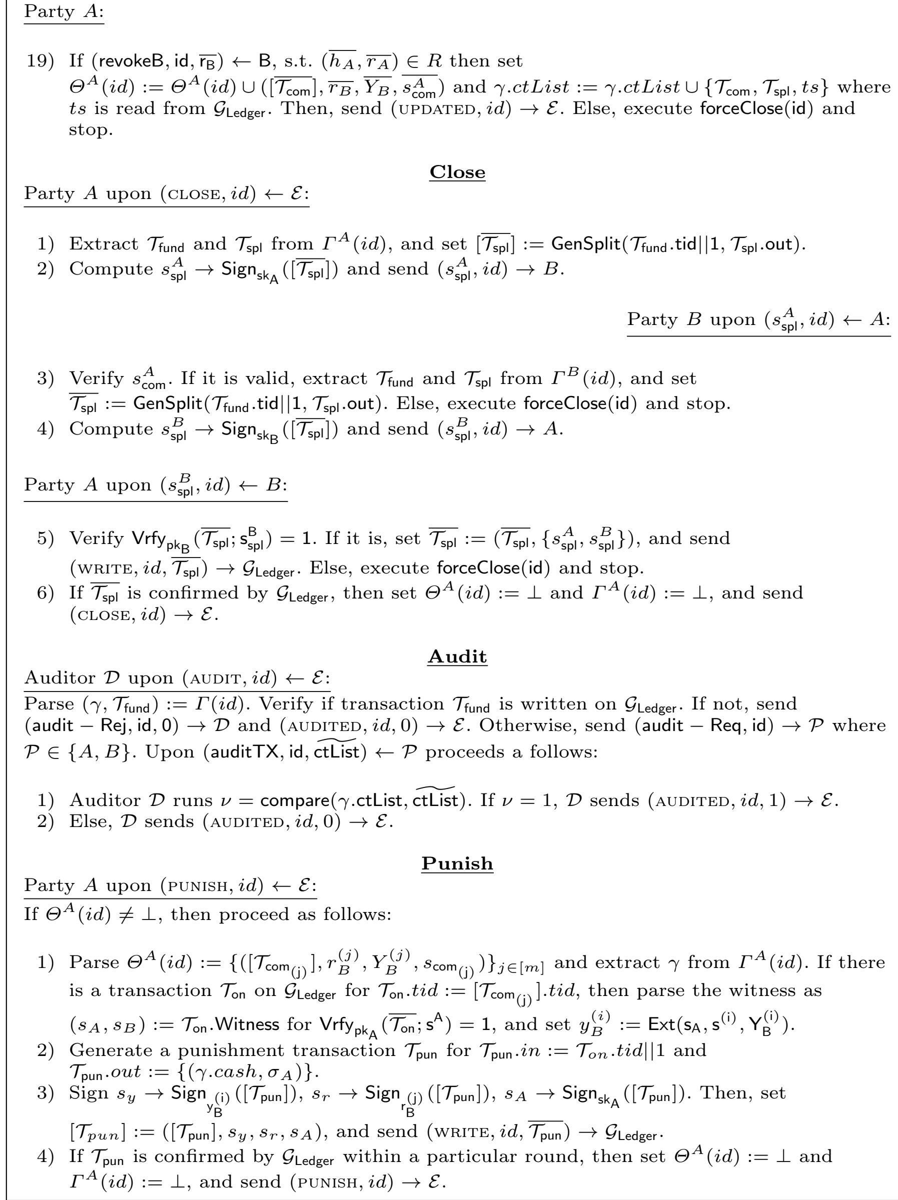
Party A:

15) If within a particular round the following cases:
   - If receive $(\mathsf{updateComB}, \mathsf{id}, \mathsf{s^B_{com}}) \leftarrow \mathsf{B}$, verify if $\mathsf{pVrfy}_{\mathsf{pk_B}}([\mathcal{T}_{\mathsf{com}}], \mathsf{Y_A}; \mathsf{s^B_{com}}) = 1$, send $(\textsc{update-ok}, id) \rightarrow \mathcal{E}$.
   - If receive $(\mathsf{updateNotOK}, \mathsf{id}, \mathsf{r_B}) \leftarrow \mathsf{A}$, and $\Theta^A(id) := \Theta^A(id) \cup ([\mathcal{T}_{\mathsf{com}}], r_B, Y_B, s^A_{\mathsf{com}})$ and stop.
   - Else, execute $\mathsf{forceClose(id)}$ and stop.
16) If $(\textsc{revoke}, id) \leftarrow \mathcal{E}$, parse $\Gamma^A(id) := (\gamma, \mathcal{T}_{\mathsf{fund}}, (\overline{\mathcal{T}_{\mathsf{com}}}, \overline{r_A}, \overline{h_B}, \overline{Y_B}, \overline{s^A_{\mathsf{com}}}), \overline{\mathcal{T}_{\mathsf{spl}}})$, and set $comTX := comTX \cup (\mathcal{T}_{\mathsf{fund}}, ts)$, where ts is read from $\mathcal{G}_{\mathsf{Ledger}}$. Then, update $\Gamma^A(id) := (\gamma, \mathcal{T}_{\mathsf{fund}}, (\mathcal{T}_{\mathsf{com}}, \overline{r_A}, h_B, Y_B, s^A_{\mathsf{com}}), comTX, \mathcal{T}_{\mathsf{spl}})$ for $\mathcal{T}_{\mathsf{spl}} := ([\mathcal{T}_{\mathsf{spl}}]; s^A_{\mathsf{spl}}, s^B_{\mathsf{spl}})$ and $\mathcal{T}_{\mathsf{com}} := ([\mathcal{T}_{\widetilde{\mathsf{com}}}], \{\mathsf{Sign}_{\mathsf{sk_A}}([\mathcal{T}_{\mathsf{com}}], \mathsf{Adapt}(\mathsf{s^B_{com}}), \mathsf{y_A})\})$, and send $(\mathsf{revokeA}, \mathsf{id}, \overline{\mathsf{r_A}}) \rightarrow \mathsf{B}$. Else, execute $\mathsf{forceClose(id)}$ and stop.

Party B

17) Parse $\Gamma^A(id) := (\gamma, \mathcal{T}_{\mathsf{fund}}, (\overline{\mathcal{T}_{\mathsf{com}}}, \overline{r_A}, \overline{h_B}, \overline{Y_B}, \overline{s^A_{\mathsf{com}}}), \overline{comTX}, \overline{\mathcal{T}_{\mathsf{spl}}})$. If $(\mathsf{revokeA}, \mathsf{id}, \overline{\mathsf{r_A}}) \leftarrow \mathsf{B}$, s.t $(\overline{h_A}, \overline{r_A}) \in R$, send $(\textsc{revoke-req}, id) \rightarrow \mathcal{E}$. Else, execute $\mathsf{forceClose(id)}$ and stop.
18) If $(\textsc{revoke}, id) \leftarrow \mathcal{E}$, set $\Theta^B(id) := \Theta^B(id) \cup ([\mathcal{T}_{\mathsf{com}}], r_A, Y_A, s^B_{\mathsf{com}})$ and $\Gamma^B(id) := (\gamma, \mathcal{T}_{\mathsf{fund}}, (\mathcal{T}_{\mathsf{com}}, \overline{r_B}, h_A, Y_A, s^B_{\mathsf{com}}), \overline{comTX}, \mathcal{T}_{\mathsf{spl}})$ for $\mathcal{T}_{\mathsf{spl}} := ([\mathcal{T}_{\mathsf{spl}}]; s^A_{\mathsf{spl}}, s^B_{\mathsf{spl}})$ and $\mathcal{T}_{\mathsf{com}} := ([\mathcal{T}_{\mathsf{com}}], \{\mathsf{Sign}_{\mathsf{sk_A}}([\mathcal{T}_{\mathsf{com}}], \mathsf{Adapt}(\mathsf{s^B_{com}}), \mathsf{y_A})\})$. In the next round, send $(\textsc{update}, id) \rightarrow \mathcal{E}$ and stop. Else, execute $\mathsf{forceClose(id)}$ and stop.

Fig. 7: The formal specification for the IvyAPC protocol.



Party A:

19) If $(\mathsf{revokeB}, \mathsf{id}, \overline{r_B}) \leftarrow \mathsf{B}$, s.t. $(\widetilde{h_A}, \overline{r_A}) \in R$ then set $\Theta^A(id) := \Theta^A(id) \cup ([\overline{\mathcal{T}_{\mathsf{com}}}], \overline{r_B}, \overline{Y_B}, \overline{s^A_{\mathsf{com}}})$ and $\gamma.ctList := \gamma.ctList \cup \{\mathcal{T}_{\mathsf{com}}, \mathcal{T}_{\mathsf{spl}}, ts\}$ where ts is read from $\mathcal{G}_{\mathsf{Ledger}}$. Then, send $(\textsc{updated}, id) \to \mathcal{E}$. Else, execute $\mathsf{forceClose}(\mathsf{id})$ and stop.

Close

Party A upon $(\textsc{close}, id) \leftarrow \mathcal{E}$:

1) Extract $\mathcal{T}_{\mathsf{fund}}$ and $\mathcal{T}_{\mathsf{spl}}$ from $\Gamma^A(id)$, and set $[\overline{\mathcal{T}_{\mathsf{spl}}}] := \mathsf{GenSplit}(\mathcal{T}_{\mathsf{fund}}.\mathsf{tid}||1, \mathcal{T}_{\mathsf{spl}}.\mathsf{out})$.
2) Compute $s^A_{\mathsf{spl}} \to \mathsf{Sign}_{\mathsf{sk_A}}([\overline{\mathcal{T}_{\mathsf{spl}}}])$ and send $(s^A_{\mathsf{spl}}, id) \to B$.

Party B upon $(s^A_{\mathsf{spl}}, id) \leftarrow A$:

3) Verify s^A_{com}. If it is valid, extract $\mathcal{T}_{\mathsf{fund}}$ and $\mathcal{T}_{\mathsf{spl}}$ from $\Gamma^B(id)$, and set $\overline{\mathcal{T}_{\mathsf{spl}}} := \mathsf{GenSplit}(\mathcal{T}_{\mathsf{fund}}.\mathsf{tid}||1, \mathcal{T}_{\mathsf{spl}}.\mathsf{out})$. Else, execute $\mathsf{forceClose}(\mathsf{id})$ and stop.
4) Compute $s^B_{\mathsf{spl}} \to \mathsf{Sign}_{\mathsf{sk_B}}([\overline{\mathcal{T}_{\mathsf{spl}}}])$ and send $(s^B_{\mathsf{spl}}, id) \to A$.

Party A upon $(s^B_{\mathsf{spl}}, id) \leftarrow B$:

5) Verify $\mathsf{Vrfy}_{\mathsf{pk_B}}(\overline{\mathcal{T}_{\mathsf{spl}}}; \mathsf{s^B_{spl}}) = 1$. If it is, set $\overline{\mathcal{T}_{\mathsf{spl}}} := (\overline{\mathcal{T}_{\mathsf{spl}}}, \{s^A_{\mathsf{spl}}, s^B_{\mathsf{spl}}\})$, and send $(\textsc{write}, id, \overline{\mathcal{T}_{\mathsf{spl}}}) \to \mathcal{G}_{\mathsf{Ledger}}$. Else, execute $\mathsf{forceClose}(\mathsf{id})$ and stop.
6) If $\overline{\mathcal{T}_{\mathsf{spl}}}$ is confirmed by $\mathcal{G}_{\mathsf{Ledger}}$, then set $\Theta^A(id) := \perp$ and $\Gamma^A(id) := \perp$, and send $(\textsc{close}, id) \to \mathcal{E}$.

Audit

Auditor $\mathcal{D}$ upon $(\textsc{audit}, id) \leftarrow \mathcal{E}$:
Parse $(\gamma, \mathcal{T}_{\mathsf{fund}}) := \Gamma(id)$. Verify if transaction $\mathcal{T}_{\mathsf{fund}}$ is written on $\mathcal{G}_{\mathsf{Ledger}}$. If not, send $(\mathsf{audit-Rej}, \mathsf{id}, 0) \to \mathcal{D}$ and $(\textsc{audited}, id, 0) \to \mathcal{E}$. Otherwise, send $(\mathsf{audit-Req}, \mathsf{id}) \to \mathcal{P}$ where $\mathcal{P} \in \{A, B\}$. Upon $(\mathsf{auditTX}, \mathsf{id}, \widetilde{\mathsf{ctList}}) \leftarrow \mathcal{P}$ proceeds a follows:

1) Auditor $\mathcal{D}$ runs $\nu = \mathsf{compare}(\gamma.\mathsf{ctList}, \widetilde{\mathsf{ctList}})$. If $\nu = 1$, $\mathcal{D}$ sends $(\textsc{audited}, id, 1) \to \mathcal{E}$.
2) Else, $\mathcal{D}$ sends $(\textsc{audited}, id, 0) \to \mathcal{E}$.

Punish

Party A upon $(\textsc{punish}, id) \leftarrow \mathcal{E}$:
If $\Theta^A(id) \neq \perp$, then proceed as follows:

1) Parse $\Theta^A(id) := \{([\mathcal{T}_{\mathsf{com_{(j)}}}], r^{(j)}_B, Y^{(j)}_B, s_{\mathsf{com_{(j)}}})\}_{j \in [m]}$ and extract γ from $\Gamma^A(id)$. If there is a transaction $\mathcal{T}_{\mathsf{on}}$ on $\mathcal{G}_{\mathsf{Ledger}}$ for $\mathcal{T}_{\mathsf{on}}.tid := [\mathcal{T}_{\mathsf{com_{(j)}}}].tid$, then parse the witness as $(s_A, s_B) := \mathcal{T}_{\mathsf{on}}.\mathsf{Witness}$ for $\mathsf{Vrfy}_{\mathsf{pk_A}}(\overline{\mathcal{T}_{\mathsf{on}}}; \mathsf{s^A}) = 1$, and set $y^{(i)}_B := \mathsf{Ext}(\mathsf{s_A}, \mathsf{s^{(i)}}, \mathsf{Y^{(i)}_B})$.
2) Generate a punishment transaction $\mathcal{T}_{\mathsf{pun}}$ for $\mathcal{T}_{\mathsf{pun}}.in := \mathcal{T}_{on}.tid||1$ and $\mathcal{T}_{\mathsf{pun}}.out := \{(\gamma.cash, \sigma_A)\}$.
3) Sign $s_y \to \mathsf{Sign}_{\mathsf{y^{(i)}_B}}([\mathcal{T}_{\mathsf{pun}}])$, $s_r \to \mathsf{Sign}_{\mathsf{r^{(j)}_B}}([\mathcal{T}_{\mathsf{pun}}])$, $s_A \to \mathsf{Sign}_{\mathsf{sk_A}}([\mathcal{T}_{\mathsf{pun}}])$. Then, set $[\mathcal{T}_{pun}] := ([\mathcal{T}_{\mathsf{pun}}], s_y, s_r, s_A)$, and send $(\textsc{write}, id, \overline{\mathcal{T}_{\mathsf{pun}}}) \to \mathcal{G}_{\mathsf{Ledger}}$.
4) If $\mathcal{T}_{\mathsf{pun}}$ is confirmed by $\mathcal{G}_{\mathsf{Ledger}}$ within a particular round, then set $\Theta^A(id) := \perp$ and $\Gamma^A(id) := \perp$, and send $(\textsc{punish}, id) \to \mathcal{E}$.

Fig. 8: The formal specification for the IvyAPC protocol.

Subprocedures

$\mathsf{GetLatestSt}(\gamma)$:

- Parse $\gamma.ctList := \{\boldsymbol{\theta_j}, ts_j\}_{j\in[n]}$ and return $\{\boldsymbol{\theta_j}, ts_j\}$, where ts_j is the maximum timestamp in the $\gamma.ctList$.

$\mathsf{Extract}(\mathsf{apk}, [\mathcal{T}_{\mathsf{fund}}], [\mathcal{T}_{\mathsf{com}}], [\mathcal{T}'_{\mathsf{com}}], \tau, \tau')$:

- Verify if $\sum.\mathsf{Verify}(\mathsf{apk}, [\mathcal{T}_{\mathsf{fund}}], [\mathcal{T}_{\mathsf{com}}], \tau) = 1$ and $\sum.\mathsf{Verify}(\mathsf{apk}, [\mathcal{T}_{\mathsf{fund}}], [\mathcal{T}'_{\mathsf{com}}], \tau') = 1$. Else, return $\perp$.
- Compute $ask := \sum.\mathsf{Extract}([\mathcal{T}_{\mathsf{fund}}], [\mathcal{T}_{\mathsf{com}}], [\mathcal{T}'_{\mathsf{com}}], \tau, \tau')$ and return ask.

$\mathsf{GenFund}(\mathbb{T}, \gamma)$:

- Return $[\mathcal{T}]$, where $\mathcal{T}.in := \mathbb{T}, \mathcal{T}_{\mathsf{f}}.out := \{(\gamma.cash, \mathsf{Multi-Sig}_{\gamma.\mathsf{us}})\}$.

$\mathsf{GenCommit}([\mathcal{T}_{\mathsf{fund}}], (\mathsf{pk_A}, \mathsf{h_A}, \mathsf{Y_A}), (\mathsf{pk_B}, \mathsf{h_B}, \mathsf{Y_B}), \mathsf{tlock})$:
Let $(c, \mathsf{Multi-Sig}_{\mathsf{pk_A},\mathsf{pk_B}}) := \mathcal{T}_{\mathsf{fund}}.\mathsf{out}[1]$ and denote:

$$\begin{aligned} \varphi_1 &:= \mathsf{Multi-Sig}_{\mathsf{ToKey(h_B)},\mathsf{ToKey(Y_B)},\mathsf{pk_A}}, \\ \varphi_2 &:= \mathsf{Multi-Sig}_{\mathsf{ToKey(h_A)},\mathsf{ToKey(Y_A)},\mathsf{pk_B}}, \\ \varphi_3 &:= \mathsf{CheckRelative}_{\Delta}||\mathsf{Multi-Sig}_{\mathsf{pk_A},\mathsf{pk_B}}. \end{aligned} \tag{6}$$

Return $[\mathcal{T}]$, where $\mathcal{T}.in := \mathcal{T}_{\mathsf{fund}}.out||1, \mathcal{T}.out := \{(c, \varphi_1 \vee \varphi_2 \vee \varphi_3)\}$ and set $\mathcal{T}.tl := tlock$ if $tlock > now$ or $\mathcal{T}.tl := 0$ otherwise.

$\mathsf{GenSplit}(\mathcal{T}_{\mathsf{on}}.\mathsf{tid}, \overrightarrow{\theta})$:

- Return $[\mathcal{T}]$, where $\mathcal{T}.in := \mathcal{T}_{\mathsf{on}}.tid, \mathcal{T}_{\mathsf{f}}.out := \overrightarrow{\theta}$.

---

forceClose(id): Let $(\gamma, \mathcal{F}_{\mathsf{fund}}) := \Gamma(sid)$. If $\mathcal{F}_{\mathsf{fund}}$ is still unspent on $\mathcal{L}$ within a particular round, then send (Error) $\rightarrow \gamma$.us and stop. Otherwise, latest in a predefined round, $m \in \{$PUNISHED,CLOSED,ERROR$\}$ is output via Punish.

Fig. 9: The formal specification for the IvyAPC protocol.

world, enabling the environment $\mathcal{E}$ unable to distinguish whether it is interacting with $\mathcal{A}$ and transacting parties in the real world or $\mathcal{S}$ and dummy transacting parties in the ideal world.

Proof. The security of $\mathcal{F}_{\mathsf{IvyAPC}}$ relies on the unforgeability of the AAFPK scheme and the security of adapter signature that has been formally proved in the GC [5]. Thus, here we only focus on corrupted parties during the generation of audit trails in the Create, Update, and Audit operation. During the generation of audit trails, if the environment $\mathcal{E}$ can generate an assertion on behalf of an honest assertor, it represents that we can construct an adversary that can break the unforgeability of the AAFPK scheme. We can use this restriction to simulate $\mathcal{S}$. Specifically, the simulated operations for IvyAPC can be described as follows:

Simulator $\mathcal{S}$ for creating auditable payment channel. Let a transacting party A initialize to create a payment channel. We describe the simulation of $\mathcal{S}$ when a transaction party A is honest and the other party B is corrupted as follows:

– When A sends the channel creation message, if party B does not send a CREATE request to functionality $\mathcal{F}_{\mathsf{IvyAPC}}$, then check if party B sends channel creation message createInfo to party A. If yes, $\mathcal{S}$ sends channel creation message CREATE on behalf of B. Otherwise, stop. Then, $\mathcal{S}$ simulates to generate a revocation public/secret key pair (h_A, r_A), payment public/secret key pair (apk_A, ask_A) (used in AAFPK), and send channel creation message back to party B. Upon receiving channel creation message from B, $\mathcal{S}$ generate the payment body of $[\mathcal{T}_{\mathsf{fund}}]$, $[\mathcal{T}_{\mathsf{com}}]$, and $[\mathcal{T}_{\mathsf{spl}}]$. After that, $\mathcal{S}$ simulates to generate an audit trial τ_A^0 with $\widetilde{\sum}.\mathsf{Assert}$ and send τ_A^0 to B. Upon receiving τ_B^0, $\mathcal{S}$ verifies τ_B^0 and simulates to generate an exchange proof ς_A^0. Then $\mathcal{S}$ pre-signs $[\mathcal{T}_{\mathsf{com}}]$ and signs $[\mathcal{T}_{\mathsf{spl}}]$ and $[\mathcal{T}_{\mathsf{fund}}]$, and exchanges the signatures with party B.

Simulator $\mathcal{S}$ for updating auditable payment channel. Let a transacting party A initialize to update the created payment channel. We describe the simulation of $\mathcal{S}$ with two cases as follows:

– *Party A is honest and B is corrupted.* If party A sends a UPDATE request to functionality $\mathcal{F}_{\mathsf{IvyAPC}}$, then $\mathcal{S}$ proceeds to generate payment body of $[\mathcal{T}_{\mathsf{fund}}]$, $[\mathcal{T}_{\mathsf{com}}]$, and $[\mathcal{T}_{\mathsf{spl}}]$. Consider that B is corrupted that B may not respond to messages upon receiving a request from party A, then honest party A can send the corresponding message to $\mathcal{F}_{\mathsf{IvyAPC}}$ on behalf of B. In addition, if A receives a reject message from B, then A can instruct $\mathcal{F}_{\mathsf{IvyAPC}}$ to stop the protocol.
  Particularly, when party A generates an assertion τ_A^n and sends an audit prerequirement message $\mathsf{Assert-req}$ to party B, if A receives pre-requirement message $\mathsf{Assert-ok}$ from B while B does not send ASSERT-PREOK to environment $\mathcal{E}$, then A can send ASSERT-PREOK to $\mathcal{E}$ on behalf of B.

- *Party A is corrupted and B is honest.* This situation is similar to the above simulation procedures, S only simulates the behaviors of party B and respond according to the behaviors of party A. $\mathcal{S}$ can behave on half of A to send a message to $\mathcal{F}_{\mathsf{IvyAPC}}$.

Simulator $\mathcal{S}$ for auditing auditable payment channel. Let an auditor $\mathcal{D}$ initialize to audit the created payment channel by requiring a list of commitment transactions from a transaction party A. We describe the simulation of $\mathcal{S}$ with the corrupted party A as follows:

- Upon $\mathcal{D}$ sending AUDIT message to $\mathcal{F}_{\mathsf{IvyAPC}}$, if A does not respond audit-related message AUDITED to environment $\mathcal{E}$ in particular round while A sends $\mathsf{auditTX}$ to $\mathcal{D}$, then simulator $\mathcal{S}$ sends AUDITED on behalf of party A. According to the instruction of environment E to $\mathcal{A}$ in the real world, $\mathcal{S}$ simulates to output the auditing result to $\mathcal{D}$ on behalf of party A.