Unlinkability and Interoperability in Account-Based Universal Payment Channels

Mohsen Minaei[1], Panagiotis Chatzigiannis[1], Shan Jin[1], Srinivasan Raghuraman[1], Ranjit Kumaresan[1], Mahdi Zamani[1], and Pedro Moreno-Sanchez[2]

[1] Visa Research
{mominaei,pchatzig,shajin,srraghur,rakumare,mzamani}@visa.com
[2] IMDEA Software Institute
pedro.moreno@imdea.org

Abstract. Payment channels allow a sender to do multiple transactions with a receiver without recording each single transaction on-chain. While most of the current constructions for payment channels focus on UTXO-based cryptocurrencies with reduced scripting capabilities (e.g., Bitcoin or Monero), little attention has been given to the possible benefits of adapting such constructions to cryptocurrencies based on the account model and offering a Turing complete language (e.g., Ethereum).
The focus of this work is to implement efficient payment channels tailored to the capabilities of account-based cryptocurrencies with Turing-complete language support in order to provide scalable payments that are interoperable across different cryptocurrencies and unlinkable for third-parties (e.g., payment intermediaries). More concretely, we continue the line of research on cryptocurrency universal payment channels (UPC) which facilitate interoperable payment channel transactions across different ledgers in a hub-and-spoke model, by offering greater scalability than point-to-point architectures. Our design proposes two different versions, UPC and AUPC. For UPC we formally describe the protocol ideas sketched in previous work and evaluate our proof-of-concept implementation. Then, AUPC further extends the concept of universal payment channels by payment unlinkability against the intermediary server.

1 Introduction

Payment channels [2,11] aim at scaling blockchain payments throughput and latency, by "off-loading" payment transactions to an off-chain communication channel between the sender and the receiver of the payment. The channel is "opened" through an on-chain funding transaction followed by any number off-chain transactions. Eventually, when one or both parties agree, the channel is "closed" through another on-chain transaction. This design mitigates both the costs and the latency associated with on-chain operations, effectively amortizing the overhead of on-chain transactions over many off-chain ones. However, basic payment channels lack *universality* since they only enable transactions within

the same ledger; and *connectivity* since only payments between the two parties that share the channel are possible.

Payment channel networks/hubs. To tackle the connectivity challenge, a payment channel network (PCN) can be treated as a graph in which nodes are senders and receivers, and edges are the payment channels between them. Several proposals improve upon the initial design of payment channels to support multi-hop payments [32] (e.g., for improving scalability [12], security [23,24], programmability [3], privacy [16] and collateral efficiency [28]). PCNs however come with additional challenges: (a) minimizing the overhead of finding paths between the users and maintaining the network topology and (b) privacy concerns when payments are performed through sequential channel updates between senders and receivers, especially though a single intermediary. Tumblers or payment channel hubs (PCHs) were introduced to address the above challenges, which act as gateways to receive the payments from the senders and route them to the corresponding receivers. Recent proposals such as Tumblebit [18], A^2L [35] and A^2L+ [15] are examples of such PCHs. While these proposals reduce the storage overhead on the underlying blockchain, they are designed for the UTXO model. It would be interesting to borrow these designs to the account-based model.

More recently, Universal Payment Channels (UPC) were proposed [10] as a protocol relying on hashed time-lock contracts (HTLCs) (which are common throughout the cryptocurrency ecosystem) and on generic accumulator data structures, tied to a hub-and-spoke model, where end users need to register with a UPC hub in order to send and receive payments. This protocol enabled *universality*, i.e. enabling transactions across different ledgers, and *concurrency*, i.e. parallelizing the internal flow of transactions to maximize throughput.

Confidentiality and anonymity. Many blockchain-based payment systems such as Bitcoin [31] provide a (false) sense of privacy by using "pseudo-anonymous" addresses. However, academic efforts [27,34] and the surveillance industry [20] have demonstrated that it is possible to associate those addresses with real identities, for instance using clustering techniques [27]). To protect user privacy, systems were specifically designed with privacy in mind such as Zcash [7] or Monero [36]. Note that "privacy" in financial transactions typically implies both the aspect of *confidentiality* and *anonymity* (or unlinkability), which imply preventing the leakage of information about the transaction value and the transacting parties from external observers respectively [9]. Therefore, it is expected that the payment channel hub used in UPC will also provide such privacy guarantees, without being able to learn information about transactions routed through it. A^2L [35] and A^2L+ [15] focus on this issue and provide a solution for a secure PCH which preserves anonymity. In addition, they only rely on digital signatures and timelock functionalities, making it interoperable across different ledgers.

In summary, we are currently missing an approach for universal, efficient, privacy-preserving scalable payments with small blockchain storage overhead for cryptocurrencies based on the account model. Such a proposal would be of interest to the blockchain community since it would be possible to be deployed in many blockchains, e.g., those based on the Ethereum Virtual Machine (EVM).

Our contributions. In our work, we start from the basic idea of Universal Payment Channels [10,29] (UPC). We first formalize the UPC ecosystem by providing a complete set of protocols that describe the system as a whole. Then, being inspired from A^2L [35] and A^2L+ [15], we augment it with anonymity properties (resulting in AUPC), while preserving the core UPC properties, namely universality and concurrency. Finally, we evaluate UPC through a proof-of-concept implementation, which showcases its feasibility in a practical deployment, while providing insights towards a fully private and auditable payment hub.

2 Preliminaries and building blocks

In this section, we provide an overview of the cryptographic primitives, the background and the related works necessary for building our protocols.

Standard cryptographic building blocks. We consider a digital signature scheme consisting of algorithms $\mathsf{KeyGen}()$, $\mathsf{Sign}()$ and $\mathtt{SigVerify}()$ and a commitment scheme $\mathsf{P}_{\mathtt{COM}}$. We also consider non-interactive zero-knowledge proof scheme $\mathtt{NIZK} := (\mathsf{P}_{\mathtt{NIZK}}, \mathsf{V}_{\mathtt{NIZK}})$ where $\pi \leftarrow \mathsf{P}_{\mathtt{NIZK}}(x, w)$ and $\mathsf{V}_{\mathtt{NIZK}}(x, \pi) := \{0, 1\}$ are the prover's algorithm and verifier's algorithm respectively for statement x and witness w and NP relation $R(x, w)$. We refer to Appendix A for formal definitions of the above primitives.

Blinded randomizable signature (BRS) scheme. A BRS scheme consists of algorithms: (i) $\widetilde{\sigma} \leftarrow \mathtt{BlindSign}(\mathtt{com}, \mathsf{sk})$, that generates a blinded signature given a commitment $\mathtt{com}$ to a message m; (ii) $\sigma := \mathtt{UnBlindSign}(\widetilde{\sigma}, \mathtt{decom})$, that unblinds $\widetilde{\sigma}$ to produce a valid signature σ based on the decommitment information $\mathtt{decom}$; (iii) and $\sigma' \leftarrow \mathtt{RandSign}(\sigma)$, that generates a randomized signature σ' based on a valid signature σ.

Adaptor signatures. Let statement/witness pair $(x, w) \in R$ where R is a hard relation, and secret/public key pair $(\mathsf{sk}, \mathsf{vk})$. At a high level, an adaptor signature scheme [5] allows a party to pre-sign a message m w.r.t. some statement x of a hard relation R, while that pre-signature $\hat{\sigma}$ can be adapted into a full valid signature σ by any party knowing the witness w. Also, the adaptor signature scheme makes possible to extract the witness w by any party which knows both the pre-signature and the adapted full signature. We refer to Appendix A for a formal definition of adaptor signatures.

Randomizable puzzles A randomizable puzzle scheme $\mathtt{RnP}$ with a solution space $\mathcal{S}$ and a function ϕ which acts on $\mathcal{S}$, consists of the following algorithms: (a) $(\mathtt{pp}, \mathtt{td}) \leftarrow \mathtt{PSetup}(1^\lambda)$ where $\mathtt{pp}$ are the public parameters and $\mathtt{td}$ is the trapdoor. (b) $Z \leftarrow \mathtt{PGen}(\mathtt{pp}, \zeta)$ where ζ is a puzzle solution, and Z is the generated puzzle (c) $\zeta := \mathtt{PSolve}(\mathtt{td}, Z)$ (d) $(Z', r) \leftarrow \mathtt{PRand}(\mathtt{pp}, Z)$ where r is a randomization factor and Z' is a randomized puzzle with the solution as $\phi(\zeta, r)$.

Note that it is assumed that there exists a deterministic function ϕ such that for a puzzle Z with the corresponding solution ζ, given its randomized version Z' with the randomization factor r, it has $\phi(\zeta, r) \in \mathcal{S}$ is a solution to Z'.

Hash-time lock contract (HTLC) A HTLC is a smart contract built upon *timelocks* and *hashlocks*. A timelock implements the "locking" of funds on a transaction until a predetermined time is reached, when the funds will return to the sender. A hashlock implements the "locking" of funds on a transaction until the hash preimage is revealed, where the funds are released to the receiver.

Accumulators. An accumulator acc enables a succinct and binding representation of a set of elements S and supports constant-size proofs of (non) membership on S. We consider *trapdoorless* accumulators to prevent the need for a trusted party that holds a trapdoor and could potentially create fake (non)membership proofs. An accumulator acc typically consists of the following algorithms [6]: $(\mathsf{pp}, D_0) \leftarrow \mathsf{AccSetup}(n_{acc})$ generates the public parameters pp and instantiates the accumulator initial state D_0; $\mathsf{Add}(D_t, x) := (D_{t+1}, \mathsf{upmsg})$ adds element x to accumulator D_t, outputting D_{t+1} and upmsg such that witness holders can update their witnesses; $\mathsf{MemWitCreate}(D_t, x, S_t) := w_x^t$ Creates a membership proof w_x^t for element x where S_t is the set of elements accumulated in D_t; $\mathsf{MemWitUp}(D_t, w_x^t, x, \mathsf{upmsg}) := w_x^{t+1}$ Updates membership proof w_x^t for element x after it is added to the accumulator; $\mathsf{VerMem}(D_t, x, w_x^t) := \{0, 1\}$ Verifies membership proof w_x^t of x in D_t.

Notation. We present the notations that are used through the rest of the paper using Table 1. In Section 3 we begin introducing the UPC protocol and later modify the protocol to achieve AUPC, we denote with blue color the added variables and functions and with red color the removals to modify UPC to AUPC.

3 Universal Payment Channels (UPC)

The goal of Universal Payment Channels (UPC) is to facilitate digital token transfers of funds across different ledgers between two parties A and B, thus achieving interoperability between those ledgers. UPC follows a hub-and-spoke design for scalability purposes, where a trustless UPC hub H plays a central role in the system. In this section, we provide an overview of the basic UPC system as well as its core protocols which serve as foundations towards constructing its privacy-preserving version AUPC. Next, we provide a high level description of the UPC system, and provide a detailed description for all of its functionalities and formal descriptions of the respective protocols in Appendix B.

Registration and UPC contract. After both parties have registered their public keys with H (Appendix B.1), an instance of the UPC contract (described in Appendix B.2 and Figure 3) is deployed between each party and the hub on the party's respective ledger (using protocols described in Appendix B.3 and Figure 6)[3].

Payment channel funding and monitoring. After the contract deployment by each party and the hub, the next step is to open a payment channel be-

[3] For ease of notation we have considered that the each instance of the UPC contract is between the Hub and a single party, however, it can easily be extended to consider all the parties within a single blockchain to use the same contract.

Table 1: Details of the variables used in the UPC smart contract, Receipt (R), Promise (P), and Channel(C) objects.

Contract Variables	
chanId	channel identifier
$\mathsf{vk_H}$&$\mathsf{vk_C}$	public key of the server and client respectively
claimDuration	duration to submit claims before channel termination
status	channel's status ("`Active`", "`Closing`", "`Closed`")
$\mathsf{deposit_H}$&$\mathsf{deposit_C}$	server's and client's deposit amounts on-chain
$\mathsf{lock_C}$	client's locked amounts on-chain
$\mathsf{credit_H}$&$\mathsf{credit_C}$	server's and client's aggregated amounts received off-chain
chanExpiry	channel's expiry time, to be set at the time of channel closing
$\mathsf{acc_H}$&$\mathsf{acc_C}$	accumulators storing pending transactions for server and client
Secrets	mapping of claimed promises and corresponding secrets
Solutions	mapping of solved puzzle and corresponding solution
closeRequester	first party finalizing the closure of the channel
Channel (C)	
cid	channel identifier
contract	contract object from the contract deployment
params	parameters initialized at contract deployment
$\mathsf{credit_{in}}$&$\mathsf{credit_{out}}$	(in)outgoing credits respectively
$\mathsf{Promises_{in}}$&$\mathsf{Promises_{out}}$	(in)outgoing promises respectively
$\mathsf{acc_{in}}$&$\mathsf{acc_{out}}$	accumulator for (in)outgoing pending promises
$\mathsf{accAux_{in}}$&$\mathsf{accAux_{out}}$	auxiliary info used by (in)outgoing accumulators
$\mathsf{netProm_{in}}$& $\mathsf{netProm_{out}}$	aggregate of (in)outgoing pending promise amounts
receipt	latest receipt received from the counter party
ledger	ledger that the channel resides on.
Receipt (R)	
credit	total promises' amounts for which a secret has been received
acc	accumulator for tracking pending promises
σ	valid signature on the values cid, credit and acc
Promise (P)	
credit	total promises' amounts for which a secret has been received
amount	amount of coins to be transferred in this transaction
hash&secret	hash value of a preimage secret for this transaction
$Z := (A_\alpha, c_\alpha)$	puzzle with statement A_α and encrypted solution c_α for this transaction
α	puzzle solution for this transaction
expiry	timestamp that this promise expires
proof	membership proof of this promise in an accumulator
$\hat{\sigma}$	pre-signature on the values cid, credit, amount, A_α, expiry
σ	signature on the values cid, credit, amount, hash (or A_α), expiry

tween them. This is achieved through the deposit on-chain protocol described in Appendix B.3 and Figure 6 which in turn invokes the UPC contract's deposit function. UPC contract also includes on-chain protocols for closing a payment channel, as well as a continuous process being run by the UPC parties to monitor the state of the contract and take any on-chain action as needed.

UPC payments. After each party A and B has established a payment channel through the payment hub H and have agreed on the transaction parameters, (i.e., transaction amounts and expiry), the receiving party B samples a secret value and creates its $\mathsf{hash} = h(x)$. Then it requests payment from A by sending the payment details, which includes $\mathsf{amount_B}$, a time expiry and the hash value. Then, A creates a *promise*, which is a signed message containing $\mathsf{amount_A}$, hash and expiry, and sends it to H (using `CreatePromise` in Figure 4). After H verifies the promise (using `VerifyPromise` in Figure 4), it sends a similar promise

to B that consists of amount_B, hash and expiry (using `CreatePromise` in Figure 4). Finally, B sends secret to H, which is also forwarded to A. The transfer of amount_A from A to H and amount_B from H to B is completed by updating the channel parameters and finalizing through a signed Receipt message (using `UpdateChannel` and `CreateReceipt` in Figure 4). The payment flow described above is depicted in Figure 1 and described in details in Appendix B.4.

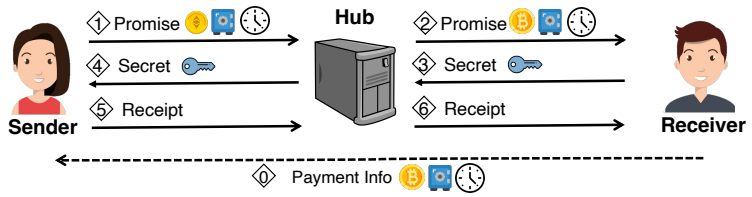


Fig. 1: Overview of off-chain steps taken by the parties to send payments from a sender to a receiver via the intermediary server

Concurrent payments. To provide maximum parallelization for a receiver that can process multiple promises simultaneously, UPC allows the sender to submit multiple promises without waiting for each promise to be processed (we refer to this property as *non-blocking/concurrent* payments). The UPC provides this feature by asking the parties to commit to the set of pending promises along with every receipt to prevent double-spending (promises are added to the pending list in `CreatePromise` and committed to when `CreateReceipt` is called in Figure 4).

As the list of pending promises grows linearly, it is inefficient to send the entire list in every receipt exchange. To address this issue, UPC uses cryptographic accumulators (e.g., Merkle tree and RSA accumulator). This allows to reduce the asymptotic bandwidth/fee overhead of inclusion proofs to a logarithm (e.g., for a Merkle tree) or a constant (e.g., for an RSA accumulator) in the number of pending promises.

Channel closing. When a party decides to close the channel, they can initiate the closure by invoking the UPC contract (Figure 3). We can consider two main scenarios. In the optimistic case, after a promise is sent from the sender, the receiver releases the secret and consequently, the sender sends a corresponding receipt to the receiver (the receipt has the aggregated amount of all previously completed promises). In such a scenario, the receiving party invokes the `ReceiptClaim` function of the UPC contract using the latest *receipt* object (Figure 3). However, in the pessimistic case, where the receiving party releases the secret but does not receive a receipt, it first invokes the `ReceiptClaim` function to present its latest receipt and then submits the corresponding *promise* object using the `PromiseClaim` function.

4 Privacy-preserving AUPC

In UPC, the payment hub learns all payment details routed through it: sender, receiver and transaction amounts. As discussed in the introduction, such a significant exposure of the transacting parties' privacy towards the hub can be problematic in many cases. Therefore, being inspired from A^{2}L and A^{2}L+, we describe how to modify UPC discussed in the previous section which enables transacting parties to maintain their anonymity against the hub, by making them unlinkable by the hub when a large number of parties transact through it. We first provide below a short high-level description of A^{2}L, then we discuss the modifications in the smart contract and both the on-chain and off-chain protocols at a high level, and we provide the detailed protocols in Appendix C. We use color coding for the changes in the respective figures imported from UPC.

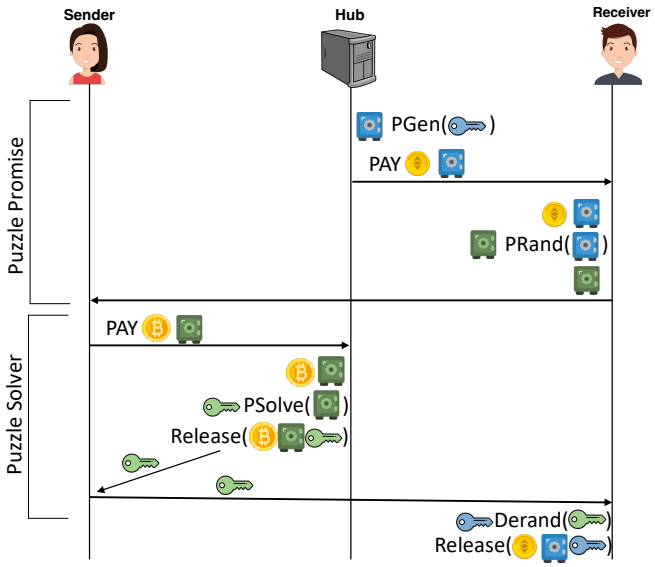


Fig. 2: Off-chain steps in A^{2}L to send payments via an intermediary server.

A^{2}L overview. The goal of A^{2}L [35] is to improve on existing solution (Tumblebit [19]) which implemented a protocol to facilitate privacy-preserving payments between parties through an untrusted payment channel hub (PCH). A^{2}L improves over Tumblebit by enabling the protocol to be *interoperable* across different cryptocurrencies. In addition, it improves on communication costs and addresses potential denial of service (DoS) attacks, where an attacker could potentially ask the PCH to initiate a large number of transactions without intending to complete them.

A^{2}L follows the paradigm of Tumblebit: it consists of two phases, a "Puzzle Promise" and a "Puzzle Solver" phase, as shown in Figure 2.The Puzzle Promise

phase takes place between PCH H and B, where H computes an adaptor signature $\hat{\sigma}_G$ and a *re-randomizeable* puzzle holding a secret value k that can be solved using some ephemeral secret. Then B re-randomizes the puzzle (denoted by `PRand`()) using randomness r_B and forwards the new puzzle to A.

Next in the"Puzzle Solver" phase, A re-randomizes the puzzle again using randomness r_A, and computes an adaptor signature $\hat{\sigma}_A$ which is sent to H. Now H can solve A's puzzle using its emphemeral secret (denoted by `PSolve`()) and extract the product of $r_A \cdot r_B \cdot k$ and use it to complete the adaptor signature $\hat{\sigma}_A$ and get paid by A. Now A due to the properties of adaptor signatures, can extract the randomized solution and send after removing r_A from the solution, can obtain $r_B \cdot k$ and send it to B. Lastly, B removes r_B from the solution and recovers k which can be used to complete $\hat{\sigma}_G$ and get paid by H. Note that the re-randomizations described throughout this process are crucial to prevent H from being able to link payments between parties.

A^{2}L+. Follow-up work [15] addresses potential attack vectors to A^{2}L which would result into recovering the hub's private key or into stealing coins from the hub. These are addressed by two additional steps, the first being a NIZK proof during the first interaction of the parties with the hub, proving that its public key is in the support of the public-key encryption scheme, and the second being a check by the hub during the puzzle solver phase, that A's verification key is in the support of the adaptor signature scheme.

Registration and AUPC contract. The first major change in order to follow the A^{2}L paradigm is to replace hash values by "puzzle promises". In addition, because the flow of the protocol first requires the establishment of a "promise" to the intended received on behalf of the payment hub, the sender is first required to lock some funds before initiating the protocol to prevent "griefing" attacks by malicious actors, which would make the payment hub establish such promises without the initiating sender having the intent to complete the protocol. The registration process is described in detail in Figure 9 in the Appendix.

Payment channel funding and monitoring. After the sender has locked the needed funds in the contract, the payment channel is opened between each party and the hub in a similar fashion as in UPC. Also, a second modification is required in the contract state monitoring process, which now tracks the existence of puzzle solutions and puzzle expiries instead of hash preimages (or "secrets"). The changes are discussed in Appendix B.2 and shown in Figure 3 using our color-coding (i.e, by removing the steps in blue color and adding those in red).

AUPC payments. The off-chain payment protocols for AUPC are similar to UPC payments described in Section 3 with the following differences: (a) There is no need to agree on the transaction amount, as this is fixed and pre-determined for any transaction facilitated through that particular hub. (b) HTLCs are replaced by rerandomizeable puzzles. (c) In `CreatePromise`(), the signed message is replace by an adaptor signature as in A^{2}L. (d) A `PreVerifyPromise`() function used by B to pre-verify the validity of the pre-signature and the proof of knowledge of puzzle solution received from H. (e) `VerifySecret`() is replaced by

`VerifySolution`() function to verify the rerandomizeable puzzle solution. Those differences are highlighted with color code in Figure 4.

Summary. We observe that no extensive changes are required to modify the UPC protocol to construct the privacy-preserving AUPC, and those changes are mostly related to the corresponding primitives utilized by A^2L, i.e., rerandomizeable puzzles and adaptor signatures instead of hashes and standard signatures. Therefore, we can consider the two versions of UPC to be modular, where only a few functions (e.g., `PreVerifyPromise`, `VerifySolution`) and variables need to be modified. We thereby present the details of two systems that provide a tradeoff in terms of efficiency and privacy guarantees. We expand on this and other discussion points in Section 6.

5 Implementation and evaluation

We have created a simple proof of concept implementation of our UPC protocol detailed in Section 3. The platform we used for our evaluation is 2.6 GHz 6-Core Intel Core i7 laptop. In addition, we have developed a mobile client for the users to interact with the UPC protocol and show the feasibility of our solution.

5.1 Accumulators

In this work, we considered two types of accumulators namely the Merkle Tree and RSA accumulators. We compared the efficiency of the two by different operations in the Ethereum network using Solidity contracts, shown in Figure 5. We observe that the Merkle tree is the better choice as it has less run-time for a practical number of in-flight transactions per channel (less than 60K at a time), and less gas cost for membership proof verification (450K gas compared to 20K when $100,000$ promises are stored in the accumulator).

5.2 UPC smart contract implementation

The ledger contract presented in Figure 3 has been implemented using the programming language Solidity for the Ethereum blockchain. Users within the Ethereum network communicate with the network through the means of transactions. The finality of a transaction is dependent on the block creation rate (i.e., about 13 seconds in Ethereum) and the fee associated to the transaction. In this section, we will be focusing on the the fees associated to the transaction calls made to the smart contract, which is captured with the *gas* value. The gas refers to the unit that measures the amount of computational effort required to execute specific set of operations in the Ethereum network. Moreover, the final price of a transaction fee depends on the exchange rate between gas and Ether known as the *gas price*. The gas price is chosen by the sender of the transaction, however acceptable gas prices by the miners of the blocks would be dependent on the demand and network congestion. At the time of writing (Jan 25th 2023)

UPC Contract

`Init`$(\mathsf{cid}, \mathsf{vk}_1, \mathsf{vk}_2, T)$:

1. Set $(\mathsf{chanId}, \mathsf{vk}_\mathsf{H}, \mathsf{vk}_\mathsf{C}, \mathsf{claimDuration}) \leftarrow (\mathsf{cid}, \mathsf{vk}_1, \mathsf{vk}_2, T)$;
2. Set status $\leftarrow$ "`Active`";
3. Set $(\mathsf{deposit}_\mathsf{H}, \mathsf{deposit}_\mathsf{C}, \mathsf{lock}_\mathsf{C}, \mathsf{credit}_\mathsf{H}, \mathsf{credit}_\mathsf{C}, \mathsf{chanExpiry}) \leftarrow (0, 0, 0, 0, 0, 0)$;
4. Set $(\mathsf{acc}_\mathsf{H}, \mathsf{acc}_\mathsf{C}, \mathsf{Secrets}, \mathsf{Solutions}, \mathsf{closeRequester}) \leftarrow (\bot, \bot, \bot, \bot, \bot)$;

`GetParams`():

Output $[\mathsf{chanId}, \mathsf{vk}_\mathsf{H}, \mathsf{vk}_\mathsf{C}, \mathsf{claimDuration}]$.

`Deposit`(amount):

1. Abort if status $\neq$ "`Active`" or caller.vk $\notin \{\mathsf{vk}_\mathsf{H}, \mathsf{vk}_\mathsf{C}\}$;
2. If caller.vk $= \mathsf{vk}_\mathsf{H}$, then set $\mathsf{deposit}_\mathsf{H} \leftarrow \mathsf{deposit}_\mathsf{H} + \mathsf{amount}$;
3. If caller.vk $= \mathsf{vk}_\mathsf{C}$, then set $\mathsf{deposit}_\mathsf{C} \leftarrow \mathsf{deposit}_\mathsf{C} + \mathsf{amount}$.

`Lock`(amount):

1. Abort if status $\neq$ "`Active`" or caller.vk $\notin \{\mathsf{vk}_\mathsf{C}\}$;
2. If caller.vk $= \mathsf{vk}_\mathsf{C}$, then set $\mathsf{lock}_\mathsf{C} \leftarrow \mathsf{lock}_\mathsf{C} + \mathsf{amount}$.

`PromiseClaim`(P):

1. Abort if status $\neq \{$"`Active`", "`Closing`"$\}$ or caller.vk $\notin \{\mathsf{vk}_\mathsf{H}, \mathsf{vk}_\mathsf{C}\}$;
2. Abort if now $\geq P.\mathsf{expiry}$ or $\mathsf{Hash}(P.\mathsf{secret}) \neq P.\mathsf{hash}$ or $\mathsf{Secrets}[P.\mathsf{hash}] \neq \bot$ or $\mathsf{Solutions}[P.\mathsf{Z}] \neq \bot$ or $P.\sigma \neq \mathsf{Adapt}(P.\hat{\sigma}, P.\alpha)$;
3. If caller.vk $= \mathsf{vk}_\mathsf{H}$, then set vk $\leftarrow \mathsf{vk}_\mathsf{C}$, credit $\leftarrow \mathsf{credit}_\mathsf{H}$, and acc $\leftarrow \mathsf{acc}_\mathsf{H}$;
   Otherwise, set vk $\leftarrow \mathsf{vk}_\mathsf{H}$, credit $\leftarrow \mathsf{credit}_\mathsf{C}$, and acc $\leftarrow \mathsf{acc}_\mathsf{C}$;
4. Abort if `SigVerify`$(P.\sigma, [\mathsf{chanId}, P.\mathsf{credit}, P.\mathsf{amount}, P.\mathsf{expiry}], \mathsf{vk})$;
5. Abort if $P.\mathsf{credit} <$ credit and $\mathsf{ACC.VerifyProof}(\mathsf{acc}, P.\mathsf{hash}, P.\mathsf{Z}, P.\mathsf{proof}) = 0$;
6. If caller.vk $= \mathsf{vk}_\mathsf{H}$, then set $\mathsf{credit}_\mathsf{H} \leftarrow \mathsf{credit}_\mathsf{H} + P.\mathsf{amount}$;
   Otherwise, $\mathsf{credit}_\mathsf{C} \leftarrow \mathsf{credit}_\mathsf{C} + P.\mathsf{amount}$;
7. Set $\mathsf{Secrets}[P.\mathsf{hash}] \leftarrow P.\mathsf{secret}$ $\mathsf{Solutions}[P.\mathsf{Z}] \leftarrow P.\alpha$.
8. If status $=$ "`Active`", then set chanExpiry $\leftarrow$ now $+$ claimDuration, and status $\leftarrow$ "`Closing`".

`ReceiptClaim`(R):

1. Abort if status $\neq \{$"`Active`", "`Closing`"$\}$ or caller.vk $\notin \{\mathsf{vk}_\mathsf{H}, \mathsf{vk}_\mathsf{C}\}$;
2. If caller.vk $= \mathsf{vk}_\mathsf{H}$, then:
   (a) Abort if `SigVerify`$(R.\sigma, [\mathsf{chanId}, R.\mathsf{credit}, R.\mathsf{acc}], \mathsf{vk}_\mathsf{C}) = 0$;
   (b) Set $\mathsf{credit}_\mathsf{H} \leftarrow R.\mathsf{credit}$, and $\mathsf{acc}_\mathsf{H} \leftarrow R.\mathsf{acc}$;
   Otherwise:
   (a) Abort if `SigVerify`$(R.\sigma, [\mathsf{chanId}, R.\mathsf{credit}, R.\mathsf{acc}], \mathsf{vk}_\mathsf{H}) = 0$;
   (b) Set $\mathsf{credit}_\mathsf{C} \leftarrow R.\mathsf{credit}$, and $\mathsf{acc}_\mathsf{C} \leftarrow R.\mathsf{acc}$;
3. If status $=$ "`Active`", then set chanExpiry $\leftarrow$ now $+$ claimDuration, and status $\leftarrow$ "`Closing`".

`Close`():

1. Abort if status $\neq \{$"`Active`", "`Closing`"$\}$ or caller.vk $\notin \{\mathsf{vk}_\mathsf{H}, \mathsf{vk}_\mathsf{C}\}$;
2. If closeRequester $= \bot$, then set closeRequester $\leftarrow$ caller.vk;
3. If closeRequester $\neq$ caller.vk, then set status $\leftarrow$ "`Closed`";
4. If status $=$ "`Active`", then set chanExpiry $\leftarrow$ now $+$ claimDuration, and status $\leftarrow$ "`Closing`".

`Withdraw`():

1. Abort if status $\neq \{$"`Closing`", "`Closed`"$\}$;
2. Abort if status $=$ "`Closing`" and now $<$ chanExpiry;
3. Invoke ledger.`transfer`$(\mathsf{vk}_\mathsf{H}, \mathsf{deposit}_\mathsf{H} + \mathsf{credit}_\mathsf{H} - \mathsf{credit}_\mathsf{C})$ and ledger.`transfer`$(\mathsf{vk}_\mathsf{C}, \mathsf{deposit}_\mathsf{C} + \mathsf{credit}_\mathsf{C} - \mathsf{credit}_\mathsf{H})$;

`Unlock`():

1. Abort if status $\neq \{$"`Closing`", "`Closed`"$\}$;
2. Abort if status $=$ "`Closing`" and now $<$ chanExpiry;
3. Invoke ledger.`transfer`$(\mathsf{vk}_\mathsf{C}, \mathsf{lock}_\mathsf{C})$;

Fig. 3: UPC smart contract.

Off-Chain Functions

$\texttt{CreatePromise}(C, \mathsf{amount}, \mathsf{hash}, Z, \mathsf{expiry}, \mathsf{sk})$:

$\mathsf{cid} \leftarrow C.\mathsf{cid}$; $\mathsf{credit} \leftarrow C.\mathsf{credit}_{\mathsf{out}}$;

1. $P \leftarrow [\mathsf{credit}, \mathsf{amount}, \mathsf{hash}, \mathsf{expiry}, \sigma]$, where
   $\sigma \leftarrow \texttt{Sign}([\mathsf{cid}, \mathsf{credit}, \mathsf{amount}, \mathsf{hash}, \mathsf{expiry}], \mathsf{sk})$
   $P \leftarrow [m, Z := (A_\alpha, c_\alpha), \hat{\sigma}, \sigma]$,
   where $\hat{\sigma} \leftarrow \texttt{PreSign}(m, A_\alpha, \mathsf{sk})$, $\sigma = \perp$ and $m = [\mathsf{cid}, \mathsf{credit}, \mathsf{amount}, \mathsf{expiry}]$;
2. Add P to $C.\mathsf{Promises}_{\mathsf{out}}$;
3. $\mathsf{ACC.Insert}(C.\mathsf{acc}_{\mathsf{out}}, \mathsf{hash}, Z, C.\mathsf{accAux}_{\mathsf{out}})$;
4. Output P.

$\texttt{PreVerifyPromise}(P, \mathsf{promiseSender}, C, \mathsf{amount}, Z, \mathsf{expiry}, \mathsf{vk})$:

$\mathsf{cid} \leftarrow C.\mathsf{cid}$; $\mathsf{credit} \leftarrow C.\mathsf{credit}_{\mathsf{in}}$;

1. If $\mathsf{promiseSender} =$ "$\texttt{Server}$" then $\mathsf{deposit} \leftarrow C.\mathsf{contract.deposit}_{\mathsf{H}}$;
   else $\mathsf{deposit} \leftarrow C.\mathsf{contract.deposit}_{\mathsf{C}}$
2. If $[\texttt{PreVerify}(P.\hat{\sigma}, [\mathsf{cid}, \mathsf{credit}, \mathsf{amount}, \mathsf{expiry}], A_\alpha, \mathsf{vk}) = 0] \vee [C.\mathsf{credit}_{\mathsf{in}} + C.\mathsf{netProm}_{\mathsf{in}} + \mathsf{amount} > \mathsf{deposit} + C.\mathsf{credit}_{\mathsf{out}}]$ then output 0; else output 1.

$\texttt{VerifyPromise}(P, \mathsf{promiseSender}, C, \mathsf{amount}, \mathsf{hash}, Z, \mathsf{expiry}, \mathsf{vk})$:

$\mathsf{cid} \leftarrow C.\mathsf{cid}$; $\mathsf{credit} \leftarrow C.\mathsf{credit}_{\mathsf{in}}$;
If $\mathsf{promiseSender} =$ "$\texttt{Server}$" then $\mathsf{deposit} \leftarrow C.\mathsf{contract.deposit}_{\mathsf{H}}$;
Otherwise, $\mathsf{deposit} \leftarrow C.\mathsf{contract.deposit}_{\mathsf{C}}$

1. Output 0 if any of the following conditions is true:
   - $\texttt{SigVerify}(P.\sigma, [\mathsf{cid}, \mathsf{credit}, \mathsf{amount}, \mathsf{hash}, \mathsf{expiry}], \mathsf{vk}) = 0$
   - $C.\mathsf{credit}_{\mathsf{in}} + C.\mathsf{netProm}_{\mathsf{in}} + \mathsf{amount} > \mathsf{deposit} + C.\mathsf{credit}_{\mathsf{out}}$;
   - $P.\mathsf{expiry} - \mathsf{now} \leq C.\mathsf{params.claimDuration}$
2. Add P to $C.\mathsf{Promises}_{\mathsf{in}}$ and set $C.\mathsf{netProm}_{\mathsf{in}} \leftarrow C.\mathsf{netProm}_{\mathsf{in}} + \mathsf{amount}$;
3. $\mathsf{ACC.Insert}(C.\mathsf{acc}_{\mathsf{in}}, \mathsf{hash}, Z, C.\mathsf{accAux}_{\mathsf{in}})$;
4. Output 1.

$\texttt{VerifySecret}(C, P, \mathsf{secret})$:

Output 0 if any of the following conditions is true; otherwise output 1:
- $P \neq C.\mathsf{Promises}_{\mathsf{out}}$;
- Abort if $\texttt{Hash}(\mathsf{secret}) \neq P.\mathsf{hash}$;

$\texttt{VerifySolution}(C, P, \sigma, A_\alpha)$:

Output 0 if any of the following conditions is true; otherwise output 1:
- $P \neq C.\mathsf{Promises}_{\mathsf{out}}$;
- Abort if $\alpha = \perp$ where $\alpha := \texttt{Ext}(\sigma, P.\hat{\sigma}, A_\alpha)$;

$\texttt{UpdateChannel}(C, P, \mathsf{updateDirection})$:

If $\mathsf{updateDirection} =$ "$\texttt{outgoing}$", then
1. Set $C.\mathsf{credit}_{\mathsf{out}} \leftarrow C.\mathsf{credit}_{\mathsf{out}} + P.\mathsf{amount}$;
2. Remove P from $C.\mathsf{Promises}_{\mathsf{out}}$;
3. $\mathsf{ACC.Delete}(C.\mathsf{acc}_{\mathsf{out}}, P.\mathsf{hash}, P.Z, C.\mathsf{accAux}_{\mathsf{out}})$;

Otherwise,
1. Set $C.\mathsf{credit}_{\mathsf{in}} \leftarrow C.\mathsf{credit}_{\mathsf{in}} + P.\mathsf{amount}$;
2. Remove P from $C.\mathsf{Promises}_{\mathsf{in}}$;
3. $\mathsf{ACC.Delete}(C.\mathsf{acc}_{\mathsf{in}}, P.\mathsf{hash}, P.Z, C.\mathsf{accAux}_{\mathsf{in}})$

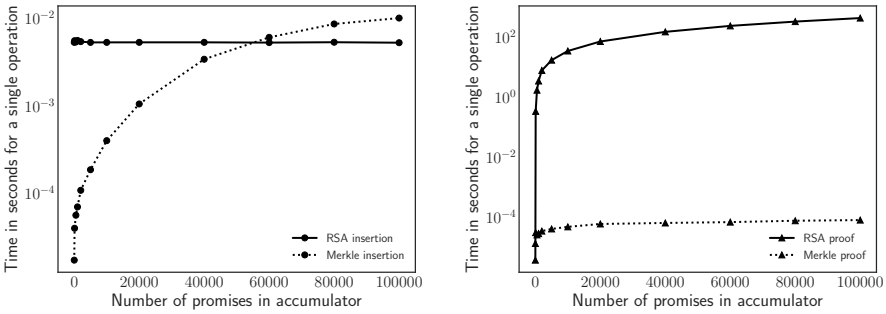
$\texttt{CreateReceipt}(C, \mathsf{sk})$:

$\mathsf{cid} \leftarrow C.\mathsf{cid}$; $\mathsf{credit} \leftarrow C.\mathsf{credit}_{\mathsf{out}}$; $\mathsf{acc} \leftarrow C.\mathsf{acc}_{\mathsf{out}}$;
Output $\sigma \leftarrow \texttt{Sign}([\mathsf{cid}, \mathsf{credit}, \mathsf{acc}], \mathsf{sk})$;

$\texttt{VerifyReceipt}(R, P, C, \mathsf{vk})$: $\mathsf{cid} \leftarrow C.\mathsf{cid}$; $\mathsf{credit} \leftarrow C.\mathsf{credit}_{\mathsf{in}}$; $\mathsf{acc} \leftarrow C.\mathsf{acc}_{\mathsf{in}}$; $\mathsf{accAux} \leftarrow C.\mathsf{accAux}_{\mathsf{in}}$;

1. $\mathsf{ACC.Delete}(\mathsf{acc}, P.\mathsf{hash}, P.Z, \mathsf{accAux})$;
2. Return $\texttt{SigVerify}(R.\sigma, [\mathsf{cid}, \mathsf{credit} + P.\mathsf{amount}, \mathsf{acc}], \mathsf{vk})$.

Fig. 4: Off-Chain Functions



(a) Accumulator Insertion (b) Membership Proof

Fig. 5: Performance comparisons between Merkle Tree and RSA accumulators

the average gas price for Ethereum is 24 Gwei and 51 Gwei for the Polygon network. Furthermore, ETH is priced at 1544 USD and MATIC[4] at 0.95 USD.

We begin by evaluating the gas needed for the deployment of the contract. The UPC contract requires $1,532,271$ gas (56.55 USD for Ethereum and 0.1 USD for Polygon) to be deployed on the respective blockchain. We emphasize that in our implementation we did not aim to optimize gas costs and further optimizations can reduce the gas. respectively. The gas usages for different functions of UPC contract are reported in Table 2.

In the case of HTLC payments, we can consider two main scenarios. In the optimistic case, after a promise is sent from the sender, the receiver releases the secret for the HTLC and consequently, the sender sends a corresponding receipt to the receiver. In such a scenario, the receiving party will submit the receipt to the contract and close accordingly. However, in the pessimistic case, where the receiving party releases the secret but does not receive a receipt, it goes on-chain and first submits its latest receipt. Next, it submits the promise for the HTLC where the party can reveal the secret of HTLC. Comparing the two scenarios and referencing the Table 2, we see that in the pessimistic case, about 70K more gas (2.50 USD) will be needed to resolve the promise.

Off-chain Evaluation. To evaluate the protocol's runtime, we conducted a test with ten clients simultaneously sending transactions to a single server. Each client transmitted 1,000 transactions, resulting in a total of 10,000 promises, secret reveals, and receipts. We achieved an end-to-end throughput of 110 TPS, encompassing random secret generation, secret hashing, promise creation, promise verification, secret revealing, secret verification, receipt creation, and receipt verification. We note that optimizing the off-chain code and using a more capable server machine could further enhance the performance.

[4] MATIC is the native token used in the Polygon blockchain

Table 2: UPC contract's functions Gas Prices. USD amounts reflect the date Jan 25th, 2023.

Functions	**Gas Units**	**USD-ETH**	**USD-Polygon**
Deposit	45,079	1.67	0.002
ReceiptClaim	75,336	2.80	0.004
Promise Claim	65,954 (w/o. proof)	2.45	0.003
	66,196 (Merkle-1 tx)	2.46	0.003
	74,755 (Merkle-1K txs)	2.78	0.004
	80,750 (Merkle-100K txs)	3.00	0.004
	524,378 (RSA)	19.48	0.026
Close	48,089 (initial)	1.79	0.002
	32,250 (final)	1.20	0.002
Withdraw	29,089	1.08	0.001

AUPC **Overhead.** As described in Section 4, inspired from the A^2L work we can modify UPC to achieve transactional unlinkability. We refer readers to section 6 of [15] for the overhead of A^2L and A^2L + for providing privacy.

6 Discussion

6.1 Security

The two intuitive security properties we require for both UPC and AUPC to satisfy are:

1. *Theft prevention*, meaning the funds of all honest participants in the system are protected despite adversarial actions.
2. *Balance*, meaning the total funds in the system do not increase with time (or in other words, an adversary cannot double-spend).

In addition, AUPC needs to satisfy *anonymity*, meaning a malicious payment hub which does not collude with other parties cannot infer any information between the sender and receiver of a transaction.

Informally, the properties of theft prevention and balance are satisfied through the sequence of hashed secrets and signed promises in UPC and the rerandomizeable puzzles and adaptor signatures in AUPC, which guarantee *atomicity*, i.e., either the payment is successful with the funds transfered simultaneously, or the funds are returned back to the sender, as discussed in the optimistic and pessimistic scenarios in Section 5.2. Also anonymity is achieved through the series of rerandomizeable puzzles, as for the case of A^2L and A^2L+ discussed in Section 4.

6.2 Tradeoffs between UPC and AUPC

We now discuss the advantages and disadvantages of AUPC over UPC (other than privacy, which was the intended goal for AUPC):

HTLCs vs timelocks. Since UPC relies on HTLCs (hashed time-lock contracts), it is only interoperable across chains that use the same hash function. In contrast, the conditions in AUPC only rely on time-locks, which allows it to be interoperable between ledgers that do not support the same hash function.

Fixed vs. variable amounts: While UPC supports arbitrary amounts, AUPC inherits a limitation from A^2L, where the amount for each transaction should be fixed (since arbitrary amounts could be used to link the sender and receiver, thus defeating privacy). Therefore, to send an arbitrary amount, multiple transactions may be required. For example, consider the case that the fixed amount for each payment is set to 0.1 ETH. To send a transaction of amount 0.7 ETH, seven transactions are required. As we can see, there is a tradeoff between transaction efficiency (e.g., tx fees, tx finality) and privacy. As a result, given this tradeoff, in some cases users may opt to use the UPC protocol instead of AUPC.

6.3 Related work

In addition to A^2L and A^2L+ which inspired AUPC, a work closely related to ours is Raiden [3], which like UPC, focuses on Ethereum and ERC-20 tokens, and additionally relies on HTLCs to route payments through the Raiden network.[5] Furthermore, Raiden supports concurrent transactions and makes use of a Merkle tree to commit to the set of pending payments. However, it faces a major limitation in the low number of in-flight payments (limited to 160 at a time [6][7][8]) Also, as pointed out earlier, Raiden off-chain payments are *simple* payments as opposed to *conditional* payments *a la* HTLCs. For this reason, Merkle paths corresponding to *all* pending payments are opened (and in particular, checked for expiry) before allowing the parties to withdraw from their Raiden channel at settlement time. In contrast, in UPC, pending payments that have expired do not need to be opened on-chain. While this may seem a minor improvement in the context of off-chain payments, this turns out to have a significant impact in larger applications (such as atomic swaps or secure computation with penalties [8,33]) that rely on HTLCs. If UPC is used to implement HTLCs in these applications, then a counterparty that aborts without claiming the HTLC payment (i.e., the HTLC payment refunds the money back to the sender), does not incur any additional (gas) cost to the honest sender. Finally, we note that to the best of our knowledge, Raiden has not been formally described.

[5] Notably they employ a separate SecretRegistry contract [28] to ensure that worst case delays are independent of the length of the payment route.

[6] As noted in [4], this is to avoid the risk of not being able to unlock the transfers, as the gas cost for this operation grows linearly with the number of the pending locks and thus the number of pending transfers.

[7] The limit, currently set to 160, is a rounded value that ensures the gas cost of unlocking will be less than 40% of Ethereum's traditional pi-million (3141592) block gas limit.

[8] Lightning has a similar limitation in that it supports 483 (unidirectional) concurrent payments owing to block size limits in Bitcoin [30,1].

Next, we discuss a work on virtual payment channels [12], which constructs the so called *ledger channels* between participants, such that parties that have a ledger channel with an intermediary party, can send payments to each other via a *virtual channel* that does not require interaction with the intermediary for every payment. Note that interaction with the intermediary is required for sending the first payment (and for closing the virtual channel), and thus the benefits are obtained only if parties send at least two or more payments over the virtual channel. Additionally, participants have to pre-allocate (i.e., lock) funds on the ledger channel for every virtual channel. This may cause a large payment on a given virtual channel to fail unless funds are re-allocated by closing multiple existing virtual channels (each of which would require interaction with server). On the one hand, UPC requires interaction with the UPC Hub for authorization of every off-chain payment. On the other hand, UPC clients do not have to perform any additional pooling of funds as long as they have sufficient balance on the UPC channel.

Other works focus on efficient cross-chain atomic swaps [37] or off-chain trading platforms [14], and require clients to hold accounts on both chains (required per definition of the atomic swap problem [13]). UPC, on the other hand, can allow payments between two clients who do not share a common blockchain. Also, several works in addition to A^2L focus on providing privacy and/or anonymity in payment channels [26,18,16,25,16,26,22] for blockchains based on the UTXO model. However, we are currently missing an approach for universal, efficient, privacy-preserving scalable payments account based blockchains.

Finally, there have been other scalability approaches like rollups or sharding [17,21] which are orthogonal to our work.

7 Conclusion and future work

We presented two implementations of UPC, or Universal Payment Channels: A first implementation is UPC which uses HTLC contracts to enable payment channels between a sender and a receiver through a payment hub, and AUPC which only uses time-locks and digital signatures compatible with adaptor signatures, in the respective contracts (making it interoperable across a broader family of ledgers) and offers unlinkability against the payment hub. However, the tradeoff in AUPC is that it does not support arbitrary amounts, which implies that in practice, payments utilizing AUPC would require more transactions to complete. Moreover, we could foresee that in practice, different denominations might be supported by different hubs and this would require users to interact with multiple hubs, each serving a different fixed amount.

One exciting research direction for future work is to consider auditability guarantees and how to add support to UPC and AUPC in order to be able to enforce policies such as anti money-laundering (AML).

Acknowledgements

This work has been partially supported by Madrid regional government as part of the program S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union; by grant IJC2020-043391-I/MCIN/AEI/10.13039/501100011033; by PRODIGY Project (TED2021-132464B-I00) funded by MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/PRTR.

Disclaimers

Case studies, comparisons, statistics, research and recommendations are provided "AS IS" and intended for informational purposes only and should not be relied upon for operational, marketing, legal, technical, tax, financial or other advice. Visa Inc. neither makes any warranty or representation as to the completeness or accuracy of the information within this document, nor assumes any liability or responsibility that may result from reliance on such information. The Information contained herein is not intended as investment or legal advice, and readers are encouraged to seek the advice of a competent professional where such advice is required. These materials and best practice recommendations are provided for informational purposes only and should not be relied upon for marketing, legal, regulatory or other advice. Recommended marketing materials should be independently evaluated in light of your specific business needs and any applicable laws and regulations. Visa is not responsible for your use of the marketing materials, best practice recommendations, or other information, including errors of any kind, contained in this document. All trademarks are the property of their respective owners, are used for identification purposes only, and do not necessarily imply product endorsement or affiliation with Visa.

References

1. Bolt #2: Peer protocol for channel management. `https://github.com/lightningnetwork/lightning-rfc/blob/master/02-peer-protocol.md#rationale-7`, (Accessed on 09/22/2020)
2. Payment channels - bitcoin wiki. `https://en.bitcoin.it/wiki/Payment_channels`, (Accessed on 05/05/2021)
3. Raiden. `https://raiden.network/`, (Accessed on 05/04/2021)
4. Raiden. `https://raiden-network-specification.readthedocs.io/en/latest/mediated_transfer.html#limit-to-number-of-simultaneously-pending-transfers`, (Accessed on 05/04/2021)
5. Aumayr, L., Ersoy, O., Erwig, A., Faust, S., Hostakova, K., Maffei, M., Moreno-Sanchez, P., Riahi, S.: Generalized bitcoin-compatible channels. Cryptology ePrint Archive, Report 2020/476 (2020)
6. Baldimtsi, F., Camenisch, J., Dubovitskaya, M., Lysyanskaya, A., Reyzin, L., Samelin, K., Yakoubov, S.: Accumulators with applications to anonymity-preserving revocation. In: IEEE EuroS&P (2017)

7. Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: 2014 IEEE Symposium on Security and Privacy. pp. 459–474. IEEE Computer Society Press (May 2014). https://doi.org/10.1109/SP.2014.36
8. Bentov, I., Kumaresan, R.: How to use bitcoin to design fair protocols. In: Crypto. pp. 421–439. Springer (2014)
9. Chatzigiannis, P., Baldimtsi, F., Chalkias, K.: SoK: Auditability and accountability in distributed payment systems. In: Sako, K., Tippenhauer, N.O. (eds.) ACNS 21, Part II. LNCS, vol. 12727, pp. 311–337. Springer, Heidelberg (Jun 2021). https://doi.org/10.1007/978-3-030-78375-4_13
10. Christodorescu, M., English, E., Gu, W.C., Kreissman, D., Kumaresan, R., Minaei, M., Raghuraman, S., Sheffield, C., Wijeyekoon, A., Zamani, M.: Universal payment channels: An interoperability platform for digital currencies. CoRR **abs/2109.12194** (2021), `https://arxiv.org/abs/2109.12194`
11. Decker, C., Wattenhofer, R.: A fast and scalable payment network with bitcoin duplex micropayment channels. In: Pelc, A., Schwarzmann, A.A. (eds.) Stabilization, Safety, and Security of Distributed Systems. pp. 3–18. Springer International Publishing, Cham (2015)
12. Dziembowski, S., Eckey, L., Faust, S., Malinowski, D.: Perun: Virtual payment hubs over cryptocurrencies. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 106–123. IEEE (2019)
13. Ethan Heilman, Sebastien Lipmann, S.G.: Atomic swaps. `https://en.bitcoin.it/wiki/Atomic_swap`, (Accessed on 05/04/2021)
14. Ethan Heilman, Sebastien Lipmann, S.G.: The arwen trading protocols. In: Financial Cryptography. pp. 156–173 (2020)
15. Glaeser, N., Maffei, M., Malavolta, G., Moreno-Sanchez, P., Tairi, E., Thyagarajan, S.A.: Foundations of coin mixing services. Cryptology ePrint Archive, Report 2022/942 (2022), `https://eprint.iacr.org/2022/942`
16. Green, M., Miers, I.: Bolt: Anonymous payment channels for decentralized currencies. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 473–489. ACM (2017)
17. Gudgeon, L., Moreno-Sanchez, P., Roos, S., McCorry, P., Gervais, A.: SoK: Layer-two blockchain protocols. In: Bonneau, J., Heninger, N. (eds.) FC 2020. LNCS, vol. 12059, pp. 201–226. Springer, Heidelberg (Feb 2020). https://doi.org/10.1007/978-3-030-51280-4_12
18. Heilman, E., Alshenibr, L., Baldimtsi, F., Scafuro, A., Goldberg, S.: TumbleBit: An untrusted bitcoin-compatible anonymous payment hub. In: NDSS (2017)
19. Heilman, E., Alshenibr, L., Baldimtsi, F., Scafuro, A., Goldberg, S.: TumbleBit: An untrusted bitcoin-compatible anonymous payment hub. In: NDSS 2017. The Internet Society (Feb / Mar 2017)
20. Inc, C.: Chainanalysis: Blockchain analysis, `https://www.chainalysis.com/`
21. Khalil, R., Zamyatin, A., Felley, G., Moreno-Sanchez, P., Gervais, A.: Commit-Chains: Secure, scalable off-chain payments. Cryptology ePrint Archive, Report 2018/642 (2018), `https://eprint.iacr.org/2018/642`
22. Le, D.V., Hurtado, L.T., Ahmad, A., Minaei, M., Lee, B., Kate, A.: A tale of two trees: One writes, and other reads: Optimized oblivious accesses to bitcoin and other utxo-based blockchains. Proceedings on Privacy Enhancing Technologies **2020**(2). https://doi.org/10.2478/popets-2020-0039, `https://par.nsf.gov/biblio/10200542`
23. Lind, J., Eyal, I., Pietzuch, P.R., Sirer, E.G.: Teechan: Payment channels using trusted execution environments (2016), `http://arxiv.org/abs/1612.07766`

24. Lind, J., Naor, O., Eyal, I., Kelbert, F., Sirer, E.G., Pietzuch, P.: Teechain: A secure payment network with asynchronous blockchain access. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. p. 63–79. SOSP '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3341301.3359627, https://doi.org/10.1145/3341301.3359627
25. Malavolta, G., Moreno-Sanchez, P., Kate, A., Maffei, M., Ravi, S.: Concurrency and privacy with payment-channel networks. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 455–471. ACM Press (Oct / Nov 2017). https://doi.org/10.1145/3133956.3134096
26. Malavolta, G., Moreno-Sanchez, P., Schneidewind, C., Kate, A., Maffei, M.: Anonymous multi-hop locks for blockchain scalability and interoperability. In: NDSS 2019. The Internet Society (Feb 2019)
27. Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., McCoy, D., Voelker, G.M., Savage, S.: A fistful of bitcoins: Characterizing payments among men with no names. Commun. ACM (2016)
28. Miller, A., Bentov, I., Bakshi, S., Kumaresan, R., McCorry, P.: Sprites and state channels: Payment networks that go faster than lightning. In: Goldberg, I., Moore, T. (eds.) Financial Cryptography and Data Security. pp. 508–526. Springer International Publishing, Cham (2019)
29. Minaei Bidgoli, M., Kumaresan, R., Zamani, M., Gaddam, S.: System and method for managing data in a database (Feb 2023), https://patents.google.com/patent/US11556909B2/
30. Mizrahi, A., Zohar, A.: Congestion attacks in payment channel networks. https://arxiv.org/pdf/2002.06564.pdf, (Accessed on 05/04/2021)
31. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2009), http://bitcoin.org/bitcoin.pdf
32. Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments. http://lightning.network/lightning-network-paper.pdf, (Accessed on 09/22/2020)
33. Ranjit Kumaresan, Vinod Vaikuntanathan, P.N.V.: Improvements to secure computation with penalties. In: CCS. pp. 406–417 (2016)
34. Ron, D., Shamir, A.: Quantitative analysis of the full bitcoin transaction graph. In: Financial Cryptography and Data Privacy (2013)
35. Tairi, E., Moreno-Sanchez, P., Maffei, M.: A^{2}l: Anonymous atomic locks for scalability in payment channel hubs. In: 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021. pp. 1834–1851. IEEE (2021). https://doi.org/10.1109/SP40001.2021.00111, https://doi.org/10.1109/SP40001.2021.00111
36. Van Saberhagen, N.: Cryptonote v 2.0 (2013), https://cryptonote.org/whitepaper.pdf
37. Zamyatin, A., Harz, D., Lind, J., Panayiotou, P., Gervais, A., Knottenbelt, W.: Xclaim: Trustless, interoperable, cryptocurrency-backed assets. IEEE Security and Privacy. IEEE (2019)

A Cryptographic background

Digital Signature Scheme. Typically, a digital signature scheme Σ with a message space $\mathcal{M}$ consists of three algorithms:

- $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
- $\sigma \leftarrow \mathsf{Sign}(m, \mathsf{sk})$ for any $m \in \mathcal{M}$
- $\mathtt{SigVerify}(\sigma, m, \mathsf{vk}) := \{0, 1\}$

Note, the digital signature scheme also holds that for every $m \in \mathcal{M}$, there always exists:

$$\mathtt{Pr}[\mathtt{SigVerify}(\mathsf{Sign}(m, \mathsf{sk}), m, \mathsf{vk}) = 1 | (\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(1^\lambda)] = 1 \quad ,$$

for a secret/public key pair $(\mathsf{sk}, \mathsf{vk})$.

Commitment Scheme. Typically, a commitment scheme $\mathtt{COM}$ consists of the commitment algorithm $\mathsf{P}_{\mathtt{COM}}$ and the verification algorithm $\mathsf{V}_{\mathtt{COM}}$:

- $(\mathtt{com}, \mathtt{decom}) \leftarrow \mathsf{P}_{\mathtt{COM}}(m)$
- $\mathsf{V}_{\mathtt{COM}}(\mathtt{com}, \mathtt{decom}, m) := \{0, 1\}$

A $\mathtt{COM}$ scheme can allow a *prover* to commit on a message m without revealing it. Later, the prover can convince a *verifier* that the message m was committed through the using of the commitment $\mathtt{com}$ and decommitment information $\mathtt{decom}$.

Non-interactive Zero-knowledge. Suppose R is an NP relation with statement/witness pairs (x, w) and L is a set of positive instances that corresponding to the relation R, such as

$$L = \{x | \exists w \text{ s.t } R(x, w) = 1\}$$

Generally, we say R is a *hard relation* if the following holds:

- There exists a $\mathtt{PPT}$ (probabilistic polynomial time) sampling algorithm $\mathtt{GenR}(1^\lambda)$, for the given input the security parameter λ, it outputs a statement/witness pair $(x, w) \in R$.
- The relation is poly-time decidable.
- For all $\mathtt{PPT}$ adversaries $\mathcal{A}$, there exists a negligible function $\mathtt{negl}(\cdot)$, such that:
$$\Pr\left[(x, w^*) \in R \;\middle|\; \begin{array}{c}(x, w) \leftarrow \mathtt{GenR}(1^\lambda), \\ w^* \leftarrow \mathcal{A}(x)\end{array}\right] \leq \mathtt{negl}(\lambda),$$
where the probability is taken over the randomness of $\mathtt{GenR}$ and $\mathcal{A}$.

Furthermore, A non-interactive zero-knowledge proof scheme $\mathtt{NIZK}$ consists of two algorithms $\mathtt{NIZK} := (\mathsf{P}_{\mathtt{NIZK}}, \mathsf{V}_{\mathtt{NIZK}})$ where

- $\pi \leftarrow \mathsf{P}_{\mathtt{NIZK}}(x, w)$ with $\mathsf{P}_{\mathtt{NIZK}}$ is prover algorithm
- $\mathsf{V}_{\mathtt{NIZK}}(x, \pi) := \{0, 1\}$ with $\mathsf{V}_{\mathtt{NIZK}}$ is the verification algorithm

Adaptor signatures. An adaptor signature scheme consists of the following algorithms:

- $\hat{\sigma} \leftarrow \mathtt{PreSign}(m, x, \mathsf{sk})$ where σ is a pre-signature
- $\mathtt{PreVerify}(\hat{\sigma}, m, x, \mathsf{vk}) := \{0, 1\}$
- $\mathsf{Adapt}(\hat{\sigma}, w) := \sigma$
- $\mathtt{Ext}(\sigma, \hat{\sigma}, x) := w$

The security properties of an adaptor signature schemes are i) *existential unforgeability under chosen message attack for adaptor signature* (aEUF–CMA) (which is similar to EUF-CMA for standard signatures) ii) *pre-signature adaptability*, which guarantees that any valid pre-signature w.r.t. x (possibly produced by a malicious signer) can be completed into a valid signature using a witness w, iii) *witness extractability*, which guarantees that a valid signature/pre-signatue pair $(\sigma, \tilde{\sigma})$ for message/statement (m, x) can always be used to extract the witness w.

Cryptographic Accumulators. An accumulator acc enables a succinct and binding representation of a set of elements S and supports constant-size proofs of (non) membership on S. For UPC we consider *dynamic* accumulators where elements can be both added and deleted over time into/from S and *positive* accumulators which allow for efficient proofs of membership. We consider *trapdoorless* accumulators in order to prevent the need for a trusted party that holds a trapdoor and could potentially create fake (non)membership proofs. An accumulator acc typically consists of the following algorithms [6]:

- $(\mathsf{pp}, D_0) \leftarrow \mathsf{AccSetup}(n_{acc})$ generates the public parameters pp and instantiates the accumulator initial state D_0.
- $\mathsf{Add}(D_t, x) := (D_{t+1}, \mathsf{upmsg})$ adds element x to accumulator D_t, outputting D_{t+1} and upmsg such that witness holders can update their witnesses.
- $\mathsf{MemWitCreate}(D_t, x, S_t) := w_x^t$ Creates a membership proof w_x^t for element x where S_t is the set of elements accumulated in D_t. $\mathsf{NonMemWitCreate}$ creates the equivalent non-membership proof u_x^t.
- $\mathsf{MemWitUp}(D_t, w_x^t, x, \mathsf{upmsg}) := w_x^{t+1}$ Updates membership proof w_x^t for element x after it is added to the accumulator.
- $\mathsf{VerMem}(D_t, x, w_x^t) := \{0, 1\}$ Verifies membership proof w_x^t of x in D_t.

B UPC functionalities and protocols

B.1 Client Registration

In the first step, every client needs to participate in a one-time registration procedure with the server H to register its verification key. The client starts by locally generating a pair of verification/secret keys, denoted by (vk, sk). Next, it transmits the verification key, vk, to H. Upon receiving the client's request, H adds vk to a registry list if the key has not been registered before. In the remaining of this section, we consider that the clients have registered with the server and omit the registration check in the remaining protocols for succinctness.

B.2 UPC basic smart contract

We now provide a detailed description of the basic UPC Protocol. We start from the basic UPC smart contract (shown in Figure 3), which is used to establish the payment channel between clients and the hub.

The contract consists of seven functions: three functions for initializing and funding the payment channel and four for closing the channel. In the following, we describe each function in detail.

Init Function. This function acts as the contract's constructor, and defines and initializes the variables used throughout the contract. The list of contract's variables and their description is provided in Table 1. The first set of variables (chanId, $\mathsf{vk_H}$, $\mathsf{vk_C}$, and $\mathsf{claimDuration}$), referred to as channel parameters, are passed to the contract during the contract's deployment phase. The next variable, status, maintains the channel's status and is initialized to "`Active`". This variable can take one of the following three values:

- "`Active`" indicating that the channel is open and transactions can be made off-chain;
- "`Closing`" specifying that the channel is about to close allowing the other end of the channel to submit its claims (if any) before the $\mathsf{claimDuration}$ expiry;
- "`Closed`" indicating that the channel is closed (i.e., parties' final balances have been determined and transferred to them).

The remaining variables are all initialized to default values (i.e., zero for numerical values and null for all other object types).

GetParams. This function is used to retrieve the channel parameters given at the initialization step of the contract. As we will see in Appendix B.3, a party can use it to verify the correct deployment of the contract by the other party.

Deposit. This function is used by the parties to increase their on-chain deposit balance (i.e., $\mathsf{deposit_H}$ or $\mathtt{Deposit_C}$), and can be invoked any number of times as long as the contract's status is "`Active`".

ReceiptClaim. When a party decides to close the channel, they can invoke this function to submit the last receipt they received from the other party. The input

to this function is a receipt object (R), detailed in Table 1. This function performs a series of verification checks before updating the on-chain credit values. The first check of the function is to ensure that the contract is not in a "`Closed`" status. Next, it ensures that the party invoking the function is the channel participant (i.e., their public key is $\mathsf{vk_H}$ or $\mathsf{vk_C}$). In the next step, depending on the caller of the function, the signature of the receipt object is verified and the appropriate on-chain credit (i.e., $\mathsf{credit_H}$ or $\mathsf{credit_C}$) and accumulator ($\mathsf{acc_H}$ or $\mathsf{acc_C}$) variables are set. For example, if the server H calls the function then the client's public key ($\mathsf{vk_C}$) will be used to verify the signature and $\mathsf{credit_H}$ and $\mathsf{acc_H}$ will be updated according to the receipt's values.

In the final step, if the contract's status is "`Active`" (i.e., this is the initial call for closing the channel), it is changed to "`Closing`" and the channel expiry (chanExpiry) time is set by adding the claimDuration to the current timestamp.

PromiseClaim. This function is used to claim a promise(s) for which the party holds its secret value[9]. The promise object (P) detailed in Table 1 is given to this function as input.

Similar to the `ReceiptClaim` function, a series of verification checks are performed before updating the on-chain credit values. Like the previous function, the first two checks are for ensuring that the status of contract is not "`Closed`" and the invoking party is a channel participant. Next, it ensures that the promise is not expired and that the provided secret value, hashes to promise's hash. Further, to prevent double spending attacks, the function ensures that each promise is submitted only once by tracking the claimed promises in the Secrets mapping (step 7).

In the next step, the signature of the promise (P) is verified (step 4) and the on-chain credit value (i.e., $\mathsf{credit_H}$ or $\mathsf{credit_C}$ depending on the invoking party) is updated in step 6 (i.e., adding the promise amount to the credit value). Note that, before updating the credit values the function checks that the promise is not a double spend via a previously `ReceiptClaim`. Therefore, all promises that have a credit value less than the contract's recorded credit, a valid membership proof of that promise inside the accumulator (pending promises) is required (step 5). Finally, similar to `ReceiptClaim`, contract's status and chanExpiry variables are set.

Close. The `Close` function can be used in two ways. One is for the case that the party does not have a receipt or promise and wants to close the channel without claiming any off-chain credits. The second case is for situations that the parties would like to close the channel cooperatively without waiting for the chanExpiry time to arrive. In this scenario, parties first claim their receipts and promises, then they invoke the `Close` function.

Similar to the previous two claim functions, the `Close` function first checks the status of the contract to not be "`Closed`", and that the caller of the function

[9] This function can be invoked at any time by the promise-holding party, however, in an optimistic scenario, parties will call this function only when the promise is about to expire and the counter party has not sent a receipt for it.

is a channel participant. Next, it checks to see if this is the first time that this function is being called using the closeRequester variable. If it is the first time, then closeRequester variable will be set by assigning the caller's public key. This is to keep track whether both parties do call the `Close` function to cooperatively close the channel or not. As a result, in step 3, if the caller of the function is different than the closeRequester (i.e., the initial closing party), the channel's status will be set to "`Closed`" which allows the parties to skip the claimDuration period and withdraw their funds sooner. Finally, similar to the previous two functions, contract's status and chanExpiry variables are set accordingly.

Withdraw. Once the channel has successfully been closed, the parties will call this function to withdraw their funds. First, the function ensures that the channel has been closed by checking the status. If the status is at a "`Closing`" state, then an extra validation of the timestamp against the channel's expiry time (chanExpiry) is performed. This is the case that one party did not call the `Close` function to cooperatively close the channel before the channel expiry time). Next, the function transfers funds according to the deposit values (via the `Deposit` function) and claimed credits (via the `ReceiptClaim` and `PromiseClaim` functions) of each channel participant[10]. Note that unlike the previous functions this function can be called by any party (not necessarily the channel participants); Therefore, the task of waiting for the channel to expire and calling `Withdraw` can be delegated to a third-party service (e.g., watchtowers).

B.3 On-Chain Protocols

We now provide details for the on-chain protocols.

Deploy Protocol The deploy protocol, presented in Figure 6 is responsible for creating a channel between client A and server H. The server begins by deploying the upcContCode (pseudocode of the contract is presented in Figure 3) onto the $\mathsf{ledger}_{\mathsf{A}}$ used by both parties. Next, it initializes the contract with the parties' verification keys and the agreed upon channel id (cid), and claimDuration values. The cid is a unique channel identifier between the two parties, which has the role of preventing replay attacks across different channels (in the case of having multiple channels between the client and server). Further, the claimDuration identifies the period that the parties have to submit any off-chain payments claims when a party invokes the closing of the channel.

After contract deployment, the server creates a local channel object C detailed in Table 1[11] and initializes it using the `InitializeChannel` function

[10] As we can see the aggregated amount transferred to both parties equal the sum of the deposit values.

[11] The server H, exclusively maintains an extra key-value channel variable denoted by PromiseMapping that links the sender's promise with its own promise to the receiver for each transaction. The keys of this variable are transaction hashs and the corresponding values are of the form $(\mathsf{A}, \mathsf{cid}_{\mathsf{A}}, P_{\mathsf{A}}, \mathsf{B}, \mathsf{cid}_{\mathsf{B}}, P_{\mathsf{H}})$.

in Figure 6. In the final step, server sends the contract object (e.g., the address of the deployed contract) to client A. Upon receiving a **ContractDeployed** message from the server, client A performs a verification check of the upcCont by confirming that the deployed contract contains the upcContCode as well as the agreed channel parameters. Next, similar to the server, client creates a local channel object (C) and initializes it using the `InitializeChannel` function in Figure 6.

Deposit Protocol After the deployment of the contract and its verification by the server and client respectively, each party can use the contract's `Deposit` function to increase its on-chain deposit balance. This balance will be used as the prefund for the off-chain transactions. Each party can invoke the deposit protocol many times with the arbitrary amount of coins as long as the status of the contract is "`Active`", allowing the parties to maintain a dynamic (monotonically increasing) prefund balances. In other words, the on-chain balances can be updated without creating new payment channels and halting off-chain transactions. The formal description of the deposit protocol is presented in Figure 6.

Close Protocol The close protocol, presented in Figure 6, dictates the steps for closing a channel between client A and server H. Either party can invoke the protocol at any time.The invoking party starts the closure by calling the contract's `ReceiptClaim` function (i.e., if it is holding a receipt object signed by the other party). Next it iterates through all the incoming promises and calls the `LocalPromiseClaim` function (shown in Figure 6) which invokes the contract's `PromiseClaim` function only if the party holds the secret to that promise. Finally, after claiming all the promises, the party invokes the contract's `Close` function to indicate that it has no other claims and the channel can be closed from its side.

As discussed previously, when a party is claiming a promise (i.e., via the contract's `PromiseClaim` function) that has been issued prior to the receipt that it has, then it would need to provide a membership proof of that promise inside its incoming accumulator ($\mathsf{acc}_{\mathsf{in}}$)[12]. To determine when a membership proof is needed, the party can compare the credit values of the promise (P) and receipt (R) objects. That is, if the credit value of the P is smaller than the one in R then it means that a membership proof is needed.

Event Handler The event handler is a process (thread) that *continuously* observes the state of the contract (i.e., as a new block is created on-chain) and the status of the incoming promises to see if any on-chain action is needed (i.e., claiming and closing the channel). The formal description of this protocol is presented in Figure 6.

[12] This is equivalent to showing the membership proof inside the other party's outgoing accumulator, as both are in sync with each other

The handler begins by updating the channel's contract object by retrieving the contract object from the ledger[13]. Next, based on the current status of the contract, different actions are triggered.

Active. While the contract is in an "`Active`" state, the handler only needs to track the expiry time of the incoming promises ($\mathsf{Promises}_{\mathsf{in}}$). First, it iterates through all the $\mathsf{Promises}_{\mathsf{in}}$ and removes all the expired promises. Next, it checks to see if there exists a promise that the party has its secret (i.e., a promise that the party awaits a receipt for) and is about to expire, for which it initiates the `Close` protocol in Appendix B.3. The threshold for waiting before claiming a promise is defined as, $\mathsf{now} + \delta + 2 \times C.\mathsf{ledger}.\Delta$, where δ is the time for executing the handler loop and $\mathsf{ledger}.\Delta$ is the time that it takes to process a claim (receipt or promise) on-chain. As explained in Appendix B.3, the party may need to submit its latest received receipt prior to submitting its promise claim. Therefore, a minimum of two $\mathsf{ledger}.\Delta$ is require to submit both claims[14].

Closing. When the channel is in a "`Closing`" state, the handler invokes the `Close` protocol (detailed in Appendix B.3). After submitting all the claims and calling the contract's `Close` function to finalize the closure, the handler checks to see if the current time is passed the contract's expiry (chanExpiry) time (i.e., either both parties cooperatively called the `Close` function or the claimDuration time has passed) to invoke the contract's `Withdraw` function.

Server Secret Forwarding. While the contract is in a "`Closing`" status, parties will be submitting their receipt and promises to the contract. In the case of a promise claim, the claiming party needs to provide the secret to the promise, which may have not been transferred off-chain (i.e., it is being revealed for the first time in the contract). Therefore, the server H needs to extract this secret from the contract and forward it to the original sender of the transaction. This secret extraction is a necessity for the server to not lose any coins. The receiving party can claim funds from the server's promise and if the server does not extract the secret the sender's promise will expire and would not be valid anymore. Moreover, it allows the channel between the server and the sending client to remain open regardless of the channel closure with the receiving client. Consequently, the handler, iterates through all its outgoing promises and checks the contract to see if any secret has been submitted for them. If so, then the corresponding information of the sender client is extracted from the PromiseMapping (a set that maps the promises of the sender with the promises of the server for each transaction) and the secret is forwarded to it. More details about the forwarding of the secret is provide in Appendix B.4.

[13] This update particularly helps with retrieving the status and client/server deposits.

[14] We note that both of these calls can be combined where the contract's `PromiseClaim` takes a receipt as input; However, to keep the contract and off-chain protocols simple and concise we forego this optimization and leave it as future work.

B.4 Off-Chain Protocols

In the previous section, we explained the on-chain protocols that facilitated the opening and closing of a channel between two parties. In this section, we detail the off-chain protocols that flow a transaction from client A to B through the server H. Note that parties would engage in the off-chain protocols only if the status of the contract is "`Active`".

Authorize Protocol The authorize protocol, presented in Figure 8, is a three party protocol, where client A wishes to make an offline payment to client B using the intermediary server [15]. Before engaging in this protocol, client A and B agree on the transaction parameters—(1) transaction amount txAmount, (2) transaction timeout txExpiry, and (3) txHash which is the hash value of a secret that is known by client B. Client B would only reveal the secret if it is sure that it will receive txAmount from the intermediary server. In what follows, we detail the steps of the authorize protocol.

Step 1. Client A begins by creating and sending a promise of the transaction (i.e., with the agreed parameters with client B) to the server and asking the server to make a similar promise (using the promise values) to client B. These promises are created by invoking the `CreatePromise` function in Figure 8. This function creates a promise object P (Table 1) and adds P to the channel's outgoing promise set ($\mathsf{Promises}_{\mathsf{out}}$) in addition to inserting the promise's hash value into channel's outgoing accumulator ($\mathsf{acc}_{\mathsf{out}}$) to keep track of pending transactions.

Step 2. Upon receiving a promise from client A, the server first verifies the promise and then creates a promise of its own to client B. The promise verification process is presented in Figure 4. The `VerifyPromise` function takes as input values for amount, hash, and expiry[16] and retrieves other promise values cid, and clientCredit from its local channel object (C). Next, using these promise variables, the verifier performs a series of validity checks:

- verifying the signature of the counter-party on these values (i.e., H checks the signature of A and B checks the signature of H);
- verifying that the counter-party has sufficient funds to cover for the amount stated in the promise;
- verifying that the promise timeout (expiry) is large enough (greater that the claimDuration identified in the contract) that in case of a dispute the party has sufficient time to go on-chain.

After validating the conditions, the promise is added to channel C's $\mathsf{Promises}_{\mathsf{in}}$ set and the hash value is inserted into the incoming accumulator ($\mathsf{acc}_{\mathsf{in}}$). Further, the aggregated incoming pending promise amount ($\mathsf{netProm}_{\mathsf{in}}$) is updated such that the next promise amounts do not exceed the collateral and credits that the

[15] This process is for scenarios where client A and B do not have a established payment channel with each other but each have a channel with the server.

[16] If the verifier is H, then these values are extracted from the received promise itself (since the server is not aware of the values agreed between client A and B beforehand). Otherwise, the values would be the agreed transaction values between A and B.

counter-party holds. Server H, also maintains a mapping between the promises that it receives (from A) and the ones that it creates (for B) to track the linkage between the promises. This mapping is denoted by PromiseMapping. We will observe the usage of this mapping in the Pay protocol (see Appendix B.4) and Event Handler (see Appendix B.3) where the secret token received from B is forwarded to A.

Step 3. Lastly, similar to step 2, client B verifies the promise it receives from the server. The only difference is that client B uses the values txAmount, txHash, and txExpiry which were agreed with client A to verify the server's promise.

One important thing to note is the expiry times of the two promises. The first promise sent by client A needs to have an expiry duration greater than the one sent by the server. The reason is that, in the case where client B claims the server's promise on-chain in the last seconds of its validity, the server would need to have sufficient time to take action to claim A's promise; Otherwise, it will be losing money by paying to B but not being able to claim the promise from A. Therefore, to resolve this potential issue, the promise sent by A has an additional ledger.Δ time on top of the txExpiry. This Δ time is the duration needed to include a transaction on the ledger used between client A and the server and is not needed to be known by B.

Pay Protocol The pay protocol, presented in Figure 8, follows the authorize protocol. Prior to this protocol, server and client B have obtained a promise from client A and the server, respectively. In what follows, we detail the steps of the pay protocol.

Step 1. Client B begins the protocol by sending (i.e., revealing) the secret of the promise (i.e., preimage of the promise hash) to H.

Step 2. After receiving a promise and its corresponding secret, server performs two checks by calling the `VerifySecret` function (presented in Figure 4)—(1) verifies that the promise exists in the $\mathsf{Promises}_{\mathsf{out}}$ set (i.e., it is not a double spending attack), (2) verifies that the secret is the preimage of the promise's hash. Knowing the preimage of the hash allows the server to claim the promise from client A.

Next, server updates its channel object (C) with client B using the `UpdateChannel` function shown in Figure 4.

In this function, based on the direction of the update (i.e., incoming or outgoing) the channel variables $\mathsf{credit}_{\mathsf{in}}$ or $\mathsf{credit}_{\mathsf{out}}$, promise set $\mathsf{Promises}_{\mathsf{in}}$ or $\mathsf{Promises}_{\mathsf{out}}$, and the accumulator $\mathsf{acc}_{\mathsf{in}}$ or $\mathsf{acc}_{\mathsf{out}}$ are updated. In this step of the protocol, the server H would be updating it channel with B in the outgoing direction by increasing the outgoing credit (client B's credit) and removing the corresponding promise from the $\mathsf{Promises}_{\mathsf{out}}$ set and $\mathsf{acc}_{\mathsf{out}}$.

After updating B's channel, server creates and sends a receipt to client B using the `CreateReceipt` function shown in Figure 4. The receipt is the signature of the party (here the server) on the updated channel variables, namely the channel identifier cid (to prevent replay attacks on other channels), outgoing

credit ($\mathsf{credit}_{\mathsf{out}}$), and the outgoing accumulator ($\mathsf{acc}_{\mathsf{out}}$) which attests to the list of pending promises till this point of time. Note that, unlike promises, receipts do not have an expiry time and are valid indefinitely.

Server continues by switching to client's A channel and retrieving the promise sent by A from the PromiseMapping map. The secret obtained from B will be the secret for A's promise as well, since the hash of both promises are the same. Finally, the server forwards the secret to client A.

Step 3. Upon receiving a promise and its corresponding secret from the server, client A performs the similar verification checks done by the server when it received the secret from B. First, the secret is checked via the `VerifySecret` function. Next, the channel is updated by increasing the outgoing credit (i.e., the server's credit) and removing the promise from the $\mathsf{Promises}_{\mathsf{out}}$ set and outgoing accumulator ($\mathsf{acc}_{\mathsf{out}}$). Finally, client A creates a receipt by signing on the updated channel variables and sending it to H.

Step 4 and 5. Both client B and server H receive a receipt from their counterparties (i.e., the server and client A respectively). Upon receiving the receipt, each party validates it by verifying the receipt signature (`VerifyReceipt` function presented in Figure 4). More specifically, each party retrieves their local channel identifier (cid), incoming credit ($\mathsf{credit}_{\mathsf{in}}$), and incoming accumulator ($\mathsf{acc}_{\mathsf{in}}$) and stores them in temporary variables. Next, it updates these temporary variables according to the promise values and checks to see if the receipt's signature verifies on these updated variables. Upon verification, the party will also update its channel variables in the incoming direction. This means that their $\mathsf{credit}_{\mathsf{in}}$ will be increased by the promise amount and the promise will be removed from their $\mathsf{Promises}_{\mathsf{in}}$ set and incoming accumulator ($\mathsf{acc}_{\mathsf{in}}$). Finally, a receipt object (details in Table 1) is created and stored in the channel's receipt variable.

On-Chain Protocols

Deploy Protocol
Registered client A and server H agree on contract parameters cid and $\mathsf{claimDuration}$. Both parties A and H use $\mathsf{ledger_A}$ to post transaction on-chain.

1. Server H does the following:
   (a) $\mathsf{upcCont} \leftarrow \mathsf{ledger_A}.\texttt{deployContract}(\mathsf{upcContCode})$;
   (b) Invoke $\mathsf{upcCont}.\texttt{Init}(\mathsf{cid}, \mathsf{vk_H}, \mathsf{vk_A}, \mathsf{claimDuration})$;
   (c) Invoke $\texttt{InitializeChannel}(\mathsf{cid}, \mathsf{upcCont})$;
   (d) Send $\mathsf{upcCont}$ to A.
2. Upon receiving $\mathsf{upcCont}$, client A does the following:
   (a) Abort if $\mathsf{ledger_A}.\texttt{verifyDeployment}(\mathsf{upcContCode}, \mathsf{upcCont}) = 0$;
   (b) Abort if $\mathsf{upcCont}.\texttt{GetParams}() \neq [\mathsf{cid}, \mathsf{vk_H}, \mathsf{vk_C}, \mathsf{claimDuration}]$;
   (c) Invoke $\texttt{InitializeChannel}(\mathsf{cid}, \mathsf{upcCont})$;

Deposit Protocol
Party $W \in \{\mathsf{A}, \mathsf{H}\}$ wishes to deposit amount to the channel C established between the two parties.

1. Abort if $C.\mathsf{contract.status} \neq$ "$\texttt{Active}$";
2. Invoke $C.\mathsf{contract}.\texttt{Deposit}(\mathsf{amount})$.

Lock Protocol
Party A wishes to lock amount to the channel C established between the two parties.

1. Abort if $C.\mathsf{contract.status} \neq$ "$\texttt{Active}$";
2. Invoke $C.\mathsf{contract}.\texttt{Lock}(\mathsf{amount})$.

Close Protocol
Party $W \in \{\mathsf{A}, \mathsf{H}\}$ wishes to close the channel C established between the two parties.

1. If $C.\mathsf{receipt} \neq \perp$ then invoke $C.\mathsf{contract}.\texttt{ReceiptClaim}(C.\mathsf{receipt})$;
2. For all $P_{\mathsf{in}} \in C.\mathsf{Promises_{in}}$, invoke $\texttt{LocalPromiseClaim}(C, P_{\mathsf{in}})$;
3. Invoke $C.\mathsf{contract}.\texttt{Close}()$;

Contract and Time Event Handler
For all parties $W \in \{\mathsf{A}, \mathsf{H}\}$, we assume that a processing thread is assigned to the loop defined below that is executed every δ seconds.

While $\texttt{True}$:
For each channel C in the party's list of channels:

1. $C.\mathsf{contract} \leftarrow C.\mathsf{ledger}.\texttt{getContractInfo}(\mathsf{contract})$;
2. If $C.\mathsf{contract.status} =$ "$\texttt{Active}$":
   For all $P_{\mathsf{in}} \in C.\mathsf{Promises_{in}}$:
   (a) If $P_{\mathsf{in}}.\mathsf{expiry} < \mathsf{now}$, then remove P_{in} from $C.\mathsf{Promises_{in}}$;
   (b) Otherwise, if $P_{\mathsf{in}}.\alpha \neq \perp$ and $P_{\mathsf{in}}.\mathsf{expiry} < (\mathsf{now} + \delta + 2 \times C.\mathsf{ledger}.\Delta)$, then initiate the $\texttt{Close}$ protocol;
3. If $C.\mathsf{contract.status} =$ "$\texttt{Closing}$":
   (a) Invoke the $\texttt{Close}$ protocol;
   (b) If $\mathsf{now} > C.\mathsf{contract.chanExpiry}$, call $C.\mathsf{contract}.\texttt{Withdraw}()$ and $C.\mathsf{contract}.\texttt{Unlock}()$;
4. If $C.\mathsf{contract.status} =$ "$\texttt{Closed}$", invoke $C.\mathsf{contract}.\texttt{Withdraw}()$ and $C.\mathsf{contract}.\texttt{Unlock}()$;
5. If $W = \mathsf{H}$ A and $C.\mathsf{contract.status} =$ "$\texttt{Closing}$":
   For all $P_{\mathsf{out}} \in C.\mathsf{Promises_{out}}$, if $C.\mathsf{contract.Secrets}[P_{\mathsf{out}}.\mathsf{hash}]$ $C.\mathsf{contract.Solutions}[P_{\mathsf{out}}.Z] \neq \perp$:
   (a) $\mathsf{secret} \leftarrow C.\mathsf{contract.Secrets}[P_{\mathsf{out}}.\mathsf{hash}]$ $\alpha^{\mathsf{A}} \leftarrow C.\mathsf{contract.Solutions}[P_{\mathsf{out}}.Z]$;
   (b) $(\mathsf{sender}, \mathsf{cid}, P_{\mathsf{sender}}, \cdot, \cdot, \cdot) \leftarrow \mathsf{PromiseMapping}[P_{\mathsf{out}}.\mathsf{hash}]$ $\alpha^{\mathsf{B}} = \alpha^{\mathsf{A}} \cdot \beta_{\mathsf{A}}^{-1}$
   (c) Sends $[P_{\mathsf{sender}}, \mathsf{secret}]$ to sender $[Z_{\mathsf{B}}, \alpha^{\mathsf{B}}]$ to receiver.

Fig. 6: On-Chain Protocols

On-Chain Protocols' Helper Functions

`InitializeChannel`$(\mathsf{cid}, \mathsf{upcCont})$:

1. Set $C[\mathsf{cid}] \leftarrow$ `NewChannel`();
2. Set $C[\mathsf{cid}].\mathsf{contract} \leftarrow \mathsf{upcCont}$;
3. Set $C[\mathsf{cid}].\mathsf{params} \leftarrow \mathsf{upcCont}.$`GetParams`();

`LocalPromiseClaim`(C, P):

1. If $P.\mathsf{secret} = \perp$ return;
2. If $P.\mathsf{credit} \geq C.\mathsf{receipt.credit}$, then call $C.\mathsf{contract}.$`PromiseClaim`$(P, \perp)$;
   Otherwise, invoke $C.\mathsf{contract}.$`PromiseClaim`(P, proof) where,
   $\mathsf{proof} \leftarrow \mathsf{ACC.GenProof}(C.\mathsf{acc}_{\mathsf{in}}, P.\mathsf{hash}, P.\mathsf{Z}, C.\mathsf{accAux}_{\mathsf{in}})$;
3. Remove P from $C.\mathsf{Promises}_{\mathsf{in}}$.

Fig. 7: On-Chain Protocols' Helper Functions (We denote with blue color the add-on functions to achieve privacy and with red color the parts of the contract not applicable for privacy).

UPC **Off-Chain Protocols**

Preliminaries.

- Let A wish to make a transaction to party B using the already established channels with server H. Both A and B agree on the values: $\mathsf{txAmount}$, $\mathsf{txExpiry}$, and txHash for which B to reveal secret where $\mathtt{Hash}(\mathsf{secret}) = \mathsf{txHash}$.
- Let cid_A denote the channel id that A will use for this payment. Likewise, let cid_B denote the channel id for B. We assume that H knows $\mathsf{cid}_\mathsf{A}, \mathsf{cid}_\mathsf{B}$.

Authorize Protocol

1. A begins by performing the following steps:
   (a) Abort if $C_\mathsf{A}[\mathsf{cid}_\mathsf{A}].\mathsf{contract.status} \neq$ "`Active`";
   (b) $P_\mathsf{A} \leftarrow \mathtt{CreatePromise}(C_\mathsf{A}[\mathsf{cid}_\mathsf{A}],\ \mathsf{txAmount}, \mathsf{txHash}, \mathsf{txExpiry} + C_\mathsf{A}.\mathsf{ledger}.\Delta,\ \mathsf{sk}_\mathsf{A})$;
   (c) Send $[P_\mathsf{A}, \mathsf{B}]$ to H.
2. Upon receiving $[P_\mathsf{A}, \mathsf{B}]$ from A, H does the following:
   (a) Abort if $C_\mathsf{H}[\mathsf{cid}_\mathsf{A}].\mathsf{contract.status} \neq$ "`Active`" or $C_\mathsf{H}[\mathsf{cid}_\mathsf{B}].\mathsf{contract.status} \neq$ "`Active`";
   (b) Abort if $\mathtt{VerifyPromise}(P_\mathsf{A},\ \text{"}\mathtt{Client}\text{"},\ C_\mathsf{H}[\mathsf{cid}_\mathsf{A}], P_\mathsf{A}.\mathsf{amount},\ P_\mathsf{A}.\mathsf{hash},\ P_\mathsf{A}.\mathsf{expiry},\ \mathsf{vk}_\mathsf{A}) = 0$;
   (c) $P_\mathsf{H} \leftarrow \mathtt{CreatePromise}(C_\mathsf{H}[\mathsf{cid}_\mathsf{B}],\ P_\mathsf{A}.\mathsf{amount},\ P_\mathsf{A}.\mathsf{hash}, P_\mathsf{A}.\mathsf{expiry} - C_\mathsf{H}[\mathsf{cid}_\mathsf{A}].\mathsf{ledger}.\Delta,\ \mathsf{sk}_\mathsf{H})$;
   (d) $\mathsf{PromiseMapping}[P_\mathsf{H}.\mathsf{hash}] \leftarrow (\mathsf{A}, \mathsf{cid}_\mathsf{A}, P_\mathsf{A}, \mathsf{B}, \mathsf{cid}_\mathsf{B}, P_\mathsf{H})$;
   (e) Send P_H to B.
3. Upon receiving P_H from H, client B does the following:
   (a) Abort if $C_\mathsf{B}[\mathsf{cid}_\mathsf{B}].\mathsf{contract.status} \neq$ "`Active`";
   (b) Abort if $\mathtt{VerifyPromise}(P_\mathsf{H},\ \text{"}\mathtt{Server}\text{"},\ C_\mathsf{B}[\mathsf{cid}_\mathsf{B}], \mathsf{txAmount},\ \mathsf{txHash},\ \mathsf{txExpiry},\ \mathsf{vk}_\mathsf{H}) = 0$.

Pay Protocol

1. Client B begins by performing the following steps:
   (a) Abort if $C_\mathsf{B}[\mathsf{cid}_\mathsf{B}].\mathsf{contract.status} \neq$ "`Active`";
   (b) $P_\mathsf{H}.\mathsf{secret} \leftarrow \mathsf{secret}$;
   (c) Send $[P_\mathsf{H}, \mathsf{secret}, \mathsf{A}]$ to H.

2. Upon receiving $[P_\mathsf{H}, \mathsf{secret}, \mathsf{A}]$ from B, server H does the following:
   (a) Abort if $C_\mathsf{H}[\mathsf{cid}_\mathsf{B}].\mathsf{contract.status} \neq$ "`Active`";
   (b) Abort if $\mathtt{VerifySecret}(C_\mathsf{H}[\mathsf{cid}_\mathsf{B}], P_\mathsf{H}, \mathsf{secret}) = 0$;
   (c) Invoke $\mathtt{UpdateChannel}(C_\mathsf{H}[\mathsf{cid}_\mathsf{B}], P_\mathsf{H}, \mathsf{secret}, \text{"}\mathtt{outgoing}\text{"})$;
   (d) $R_\mathsf{H} \leftarrow \mathtt{CreateReceipt}(C_\mathsf{H}[\mathsf{cid}_\mathsf{B}], \mathsf{sk}_\mathsf{H})$;
   (e) Send $[R_\mathsf{H},\ P_\mathsf{H}]$ to B;
   (f) Set $P_\mathsf{A} \leftarrow \mathsf{PromiseMapping}[P.\mathsf{hash}].P_\mathsf{A}$;
   (g) $P_\mathsf{A}.\mathsf{secret} \leftarrow \mathsf{secret}$;
   (h) Send $[P_\mathsf{A}, \mathsf{secret}]$ to A.

3. Upon receiving $[P_\mathsf{A}, \mathsf{secret}]$ from H, client A does the following:
   (a) Abort if $C_\mathsf{A}[\mathsf{cid}_\mathsf{A}].\mathsf{contract.status} \neq$ "`Active`";
   (b) Abort if $\mathtt{VerifySecret}(C_\mathsf{A}[\mathsf{cid}_\mathsf{A}], P_\mathsf{A}, \mathsf{secret}) = 0$;
   (c) Invoke $\mathtt{UpdateChannel}(C_\mathsf{A}[\mathsf{cid}_\mathsf{A}], P_\mathsf{A}, \mathsf{secret}, \text{"}\mathtt{outgoing}\text{"})$;
   (d) $R_\mathsf{A} \leftarrow \mathtt{CreateReceipt}(C_\mathsf{A}[\mathsf{cid}_\mathsf{A}], \mathsf{sk}_\mathsf{A})$;
   (e) Send $[R_\mathsf{A},\ P_\mathsf{A}]$ to H.

4. Upon receiving $[R_\mathsf{A}, P_\mathsf{A}]$ from A, server H does the following:
   (a) Abort if $\mathtt{VerifyReceipt}(R_\mathsf{A}, C_\mathsf{H}[\mathsf{cid}_\mathsf{A}], \mathsf{vk}_\mathsf{A}) = 0$;
   (b) Invoke $\mathtt{UpdateChannel}(C_\mathsf{H}[\mathsf{cid}_\mathsf{A}], P_\mathsf{A}, P_\mathsf{A}.\mathsf{secret}, \text{"}\mathtt{incoming}\text{"})$;
   (c) $C.\mathsf{receipt} \leftarrow [C.\mathsf{credit}_\mathsf{in}, C.\mathsf{acc}_\mathsf{in}, R_\mathsf{A}.\sigma]$.

5. Upon receiving $[R_\mathsf{H}, P_\mathsf{H}]$ from H, client B does the following:
   (a) Abort if $\mathtt{VerifyReceipt}(R_\mathsf{H}, C_\mathsf{B}[\mathsf{cid}_\mathsf{B}], \mathsf{vk}_\mathsf{H}) = 0$;
   (b) Invoke $\mathtt{UpdateChannel}(C_\mathsf{B}[\mathsf{cid}_\mathsf{B}], P_\mathsf{H}, P_\mathsf{H}.\mathsf{secret}, \text{"}\mathtt{incoming}\text{"})$;
   (c) $C.\mathsf{receipt} \leftarrow [C.\mathsf{credit}_\mathsf{in}, C.\mathsf{acc}_\mathsf{in}, R_\mathsf{H}.\sigma]$.

Fig. 8: Off-Chain Protocols for UPC

C Privacy-preserving AUPC

C.1 UPC Contract

The modifications to the UPC contract are shown in Figure 3. First, in order to incorporate the paradigm followed by A^2L, we first need to use promise puzzles instead of promise hash values (this is also naturally the case for the rest of the protocols in this section). Also, where the hub needs to make an "advance" promise to the receiver under the condition that the sender successfully pays the hub, we also need to add a new channel variable named $\mathsf{lock_C}$ and new functions named `Lock` and `Unlock`. These are used to prevent the "griefing" DoS-type attack, where a receiver could initiate many promise operations without intending to complete a transaction. Therefore, in the `Lock` function the sender locks additional funds in the variable $\mathsf{lock_C}$ before the payments begin, to prove that sender is willing to participate in the protocols (which effectively moves the "risk" back to the sender.) Once the channel has successfully been closed, the sender can call the `Unlock` function to retrieve the locked funds back to his own account. We also introduce some additional minor needed modifications: In the `Init` function we track the additional $\mathsf{lock_C}$ variable as well as Solutions instead of Secrets (which works as the mapping between the solved puzzle and its solution).

In addition, the `PromiseClaim` function is now called by the party which is able to obtain the solution of the puzzle that it holds. Also it now checks if each promise is submitted only once by tracking the claimed promises in the Solutions mapping (step 2 & step 7), instead of the Secrets mapping in original UPC. Besides, the `PromiseClaim` function needs to ensure that the obtained valid signature is the result of applying Adapt function on the promise's invalid signature with the corresponding puzzle solution (step 2)

Once the channel has successfully been closed, the clients will call the `Unlock`() function to withdraw their funds that were locked into the contract, and works in a similar way to the `Withdraw` function.

C.2 On-Chain Protocols

Generally, the on-chain protocols in AUPC are almost same as the ones in UPC. The major changes include the addition of Lock protocol and the modification on event handler, as shown in Figure 6.

Lock Protocol After the deployment of the contract and its verification by the server and client respectively, the client can use the contract's `Lock` function to increase its on-chain lock balance. This balance will be used as the proof for client to show his willing to participate in the privacy-preserving protocol as the sender. Same to the `Deposit` function, the client can invoke the lock protocol many times with the arbitrary amount of coins as long as the status of the contract is "`Active`".

Event Handler As in UPC, the event handler triggers different actions based on the current status of the contract. The differences are as follows:

"Active": The event handler now checks if there exists a promise that the party has its puzzle solution, and is about to expire.

"Closing": When the current time is past the contract's expiry, the handler will also invoke the contract's `Unlock` function. Also in the case of a promise claim, the claiming party needs to provide the solution to the promise, which may have not been transferred off-chain. Therefore, in AUPC the sender A needs to extract the puzzle solution from the contract, derandomize it, and forward this derandomized solution to the original receiver of the payment. This puzzle solution forwarding is necessary for the receiver to make sure he can eventually get the funds from the server. Consequently, the handler iterates through all its outgoing promises and checks the contract to see if any puzzle solution has been submitted for them. If so, the puzzle solution is forwarded from sender to the corresponding receiver.

Finally for the helper functions, the membership proof is generated based on the promise's puzzle instead of hash, as shown in Figure 7.

C.3 Off-Chain Protocols

Registration Protocol In order to maintain the unlinkability but also mitigate the griefing attack, AUPC needs to run a registration protocol before launching the payment, as in A^2L. This protocol shown in Figure 9, is a 3-party protocol, where client A proves the willingness to the server H for participating in the payment. The protocol also assumes that A has already locked funds with H in a escrow output oid, which is in fact the funds that locked in the contract (lock_C, in our protocol). The protocol also makes use of a (blinded) randomizable signature scheme $\widetilde{\Sigma}$ as described in Section 2. We now describe the steps in detail.

Step 1. Client A begins by generating a random token identifier tid. Then A creates a commitment `com` and decommitment `decom` to tid as well as a `NIZK` proof π of its opening, and sends $[\pi, \mathtt{com}]$ along with the oid to server H.

Step 2. Upon receiving the commitment, proof, and escrow output from client A, the server H verifies the π and aborts if the verification fails. Then H generates a blind signature $\widetilde{\sigma}$ on tid by using the commitment `com` and sends the signature to client A.

Step 3. Upon receiving the $\widetilde{\sigma}$ from H, client A first unblinds this signature by using the decommitment information `decom` to obtain a valid signature σ_{tid} for the token tid, then verifies the valid signature and aborts if the verification fails. Then client A randomizes σ_{tid} to obtain a randomized signature σ'_{tid}. Finally A sends the pair $[\sigma'_{\mathsf{tid}}, \mathsf{tid}]$ to client B, which finalizes the registration protocol.

Authorize Protocol The authorize protocol, presented in Figure 10, is a three party protocol, where client A wishes to make an offline payment to client B using the intermediary server H. At the beginning, client A and server H agree

on the transaction timeout $\mathsf{txExpiry}_{\mathsf{AH}}$ and server H and client A agree on the transaction timeout $\mathsf{txExpiry}_{\mathsf{HB}}$, for each of the channels, respectively. Note that in AUPC, the transaction amount $\mathsf{txAmount}$ is fixed for all transactions associated with a given hub. We now describe the steps in detail.

Step 1 and 2. Client B begins by sending the pair of $[\sigma'_{\mathsf{tid}}, \mathsf{tid}]$ which he obtained during the registration phase, to server H. Upon receiving this pair, H will check if the received token tid has been already presented by checking the list $\mathcal{T}$ which keeps all the previously seen token identifiers and will also verify the signature σ'_{tid}. Then H will insert tid into the list $\mathcal{T}$ if it was not in the list.

Next, H samples a statement/witness pair $(A^{\mathsf{H}}_{\alpha}, \alpha^{\mathsf{H}})$ through the `GenR` algorithm, and generates the randomizable puzzle $Z_{\mathsf{H}} := (A^{\mathsf{H}}_{\alpha}, c^{S}_{\alpha})$ by using the `PGen` algorithm. Then it produces a NIZK proof $\pi^{\mathsf{H}}_{\alpha}$ to prove that α^{H} is a valid solution of puzzle Z_{H}. Finally, H creates a promise of the transaction and send it to client B. The promise is created by invoking the `CreatePromise` function in Figure 4, where an adaptor signature is generated $\hat{\sigma}$ over the previously agreed transaction message m.

Note the `CreatePromise` for AUPC includes the following changes:

- **Puzzle:** A puzzle with an invalid signature is included into the promise, which replaces the hash value of the secret in UPC.
- **Pre-signature:** When each promise is created, only a pre-signature is generated on the payment message with the statement of the puzzle, within the promise.
- **Valid Signature:** After each party obtains the solution for his own puzzle, it can adapt its pre-signature to a full valid signature.

Step 3. Upon receiving a promise and proof from server H, client B first checks if the NIZK proof is valid. Then B will pre-verify the validation of the pre-signature in the received promise through the `PreVerifyPromise` helper function in Figure 4, then randomize the puzzle within the received promise through the using of the `PRand` function. Finally, client B shares this randomized puzzle with client A.

Step 4 and 5. Upon receiving the randomized puzzle Z_{B} from B, client A randomizes the received puzzle again to obtain his re-randomized puzzle Z_{A}. Then A creates a promise of the transaction with the agreed parameters with client B and send it to server H. Upon receiving the promise from A, H pre-verifies the validation of the pre-signature within the received promise, which finalizes the authorize protocol.

Pay Protocol The pay protocol, shown in Figure 11, follows the authorize protocol. Prior to this protocol, server H and client B have obtained a promise from client A and the server H, respectively. We now describe the steps in detail.

Step 1. Upon receiving the promise from A, H extracts the puzzle within the promise. Since H has the trapdoor td for the randomized puzzle scheme, it can easily obtain α^{A}, which is the doubly randomized version of the value α^{H} from

the received puzzle. Then, H can convert the invalid signature $\hat{\sigma}$ within the promise to a valid signature σ through the algorithm. Upon obtaining the valid signature, H verifies the promise P_{A} using the `VerifyPromise` helper function in Figure 4. Finally, H inserts the valid signature into the client A's promise and send the promise's valid signature back to A.

Step 2. After receiving the promise's valid signature back from H, A extracts the solution α^{A} based on the adaptor signature (invalid signature) and the valid signature, and then verifies the solution's validation through the function `VerifySolution` in Figure 4. Then A updates its channel object (C) with server H using the `UpdateChannel` function. After updating A's channel, A creates and sends a receipt to server H using the `CreateReceipt` function. Then A will remove one layer of randomization and obtain the solution α^{B} which corresponds to puzzle Z_{B}. Then A forwards the pair of $[\alpha^{\mathsf{B}}, Z_{\mathsf{B}}]$ to client B.

Step 3. After receiving the puzzle and solution from A, client B removes its own randomness from α^{B} and therefore obtain the original value of α^{H}. As a result, the valid signature can be generated by adapting the solution α^{H} to the invalid signature $\hat{\sigma}$ in P_{H} and is inserted into P_{H}. Finally, B sends the promise's valid signature $P_{\mathsf{H}}.\sigma$ back to H.

Step 4. H first verifies the received valid signature in P_{H} by verifying the validity of its corresponding solution. Then H updates the channel with client B through `UpdateChannel` function, then creates and sends a receipt to client B using the `CreateReceipt` function.

Step 5 and 6. H and B upon receiving the receipts, each verify the signature, then update its channel variables in the incoming direction. Finally, a receipt object is created and stored in the channel's receipt variable.

Off-Chain Protocol - Registration

Preliminaries.

- Let A wish to make a proof to server H that he is willing to participate in the protocol with B's forwarding.
- Both A and H agree on the locked funds in a escrow output oid
- H holds the key pair $(\widetilde{\mathsf{vk}_\mathsf{H}}, \widetilde{\mathsf{sk}_\mathsf{H}})$ used in a blinded randomizable signature scheme $\widetilde{\Sigma}$

Registration Protocol

1. A begins by performing the following steps:
   (a) Abort if $C_\mathsf{A}[\mathsf{cid}_\mathsf{A}].\mathsf{contract.status} \neq$ "`Active`";
   (b) Generate a random token identifier tid
   (c) $(\mathtt{com}, \mathtt{decom}) \leftarrow \mathsf{P}_{\mathtt{COM}}(\mathsf{tid})$
   (d) $\pi \leftarrow \mathsf{P}_{\mathtt{NIZK}}(\{\exists \mathtt{decom} \,|\mathsf{V}_{\mathtt{COM}}(\mathtt{com}, \mathtt{decom}, \mathsf{tid}) = 1\}, \mathtt{decom})$
   (e) $\mathsf{oid} := C_\mathsf{A}[\mathsf{cid}_\mathsf{A}].\mathsf{params.lock}_\mathsf{C}$
   (f) Send $[\pi, \mathtt{com}, \mathsf{oid}]$ to H

2. Upon receiving $[\pi, \mathtt{com}]$ and oid from A, H does the following:
   (a) Abort if $C_\mathsf{A}[\mathsf{cid}_\mathsf{A}].\mathsf{contract.status} \neq$ "`Active`";
   (b) Abort if $\mathsf{V}_{\mathtt{NIZK}}(\mathtt{com}, \pi) = 0$
   (c) $\widetilde{\sigma} \leftarrow \mathtt{BlindSign}(\mathtt{com}, \widetilde{\mathsf{sk}_\mathsf{H}})$
   (d) Send $[\widetilde{\sigma}]$ to A

3. Upon receiving $\widetilde{\sigma}$ from H, A does the following:
   (a) $\sigma_{\mathsf{tid}} := \mathtt{UnBlindSign}(\mathtt{decom}, \widetilde{\sigma})$
   (b) Abort if $\mathtt{SigVerify}(\sigma_{\mathsf{tid}}, \mathsf{tid}, \widetilde{\mathsf{vk}_\mathsf{H}}) = 0$
   (c) $\sigma'_{\mathsf{tid}} \leftarrow \mathtt{RandSign}(\sigma_{\mathsf{tid}})$
   (d) Send $[\sigma'_{\mathsf{tid}}, \mathsf{tid}]$ to B

Fig. 9: AUPC Off-Chain Registration Protocol

Off-Chain Protocol - Authorize

Preliminaries.

- Let A wish to make a transaction to party B. Both A and H agree on the value $\mathsf{txExpiry}_{\mathsf{AH}}$ and both H and B agree on the value $\mathsf{txExpiry}_{\mathsf{HB}}$ (note $\mathsf{txAmount}$ now is fixed for the server H).
- Let $\mathsf{cid}_{\mathsf{A}}$ denote the channel id that A will use for this payment. Likewise, let $\mathsf{cid}_{\mathsf{B}}$ denote the channel id for B. We assume that H knows $\mathsf{cid}_{\mathsf{A}}, \mathsf{cid}_{\mathsf{B}}$.

Protocol

1. B begins by performing the following steps:
   (a) Abort if $C_{\mathsf{B}}[\mathsf{cid}_{\mathsf{B}}].\mathsf{contract.status} \neq$ "`Active`";
   (b) Sends $[\sigma'_{\mathsf{tid}}, \mathsf{tid}]$ to H

2. Upon receiving $[\sigma'_{\mathsf{tid}}, \mathsf{tid}]$ from B, H does the following:
   (a) Abort if $C_{\mathsf{H}}[\mathsf{cid}_{\mathsf{B}}].\mathsf{contract.status} \neq$ "`Active`";
   (b) Abort if $\mathsf{tid} \in \mathcal{T}$ or $\mathtt{SigVerify}(\sigma'_{\mathsf{tid}}, \mathsf{tid}, \widetilde{\mathsf{vk}_{\mathsf{H}}}) = 0$;
   (c) $\mathcal{T}.\mathrm{insert}(\mathsf{tid})$
   (d) $(A^{\mathsf{H}}_{\alpha}, \alpha^{\mathsf{H}}) \leftarrow \mathtt{GenR}(1^{\lambda})$
   (e) $Z_{\mathsf{H}} := (A^{\mathsf{H}}_{\alpha}, c^{S}_{\alpha}) \leftarrow \mathtt{PGen}(\mathsf{pp}, \alpha^{\mathsf{H}})$
   (f) $\pi^{\mathsf{H}}_{\alpha} \leftarrow \mathsf{P}_{\mathtt{NIZK}}(\{\ \exists \alpha\ |\mathtt{PSolve}(\mathsf{td}, Z) = \alpha^{\mathsf{H}}\}, \alpha^{\mathsf{H}})$
   (g) $P_{\mathsf{H}} \leftarrow \mathtt{CreatePromise}(C_{\mathsf{H}}[\mathsf{cid}_{\mathsf{B}}],\ \mathsf{txAmount},\ Z_{\mathsf{H}},$ $\mathsf{txExpiry}_{\mathsf{HB}},\ \mathsf{sk}_{\mathsf{H}})$;
   (h) Send $[P_{\mathsf{H}}, \pi^{\mathsf{H}}_{\alpha}]$ to B.

3. Upon receiving $[P_{\mathsf{H}}, \pi_{\alpha}]$ from H, B does the following:
   (a) Abort if $C_{\mathsf{B}}[\mathsf{cid}_{\mathsf{B}}].\mathsf{contract.status} \neq$ "`Active`";
   (b) Abort if $\mathsf{V}_{\mathtt{NIZK}}(P_{\mathsf{H}}.Z, \pi^{\mathsf{H}}_{\alpha}) = 0$
   (c) Abort if $\mathtt{PreVerifyPromise}(P_{\mathsf{H}},$ "`Server`", $C_{\mathsf{B}}[\mathsf{cid}_{\mathsf{B}}],\ P_{\mathsf{H}}.\mathsf{amount},\ P_{\mathsf{H}}.Z,$ $P_{\mathsf{H}}.\mathsf{expiry},\ \mathsf{vk}_{\mathsf{H}}) = 0$.
   (d) $(Z_{\mathsf{B}} := (A^{\mathsf{B}}_{\alpha}, c^{\mathsf{B}}_{\alpha}),\ \beta_{\mathsf{B}}) \leftarrow \mathtt{PRand}(\mathsf{pp}, P_{\mathsf{H}}.Z)$
   (e) Send Z_{B} to A.

4. Upon receiving Z_{B} from B, A begins by performing the following steps:
   (a) Abort if $C_{\mathsf{A}}[\mathsf{cid}_{\mathsf{A}}].\mathsf{contract.status} \neq$ "`Active`";
   (b) $(Z_{\mathsf{A}} := (A^{\mathsf{A}}_{\alpha}, c^{\mathsf{A}}_{\alpha}),\ \beta_{\mathsf{A}}) \leftarrow \mathtt{PRand}(\mathsf{pp}, Z_{\mathsf{B}})$
   (c) $P_{\mathsf{A}} \leftarrow \mathtt{CreatePromise}(C_{\mathsf{A}}[\mathsf{cid}_{\mathsf{A}}],\ \mathsf{txAmount},\ Z_{\mathsf{A}},$ $\mathsf{txExpiry}_{\mathsf{AH}},\ \mathsf{sk}_{\mathsf{A}})$;
   (d) Send P_{A} to H.

5. Upon receiving P_{A} from A, H does the following:
   (a) Abort if $C_{\mathsf{H}}[\mathsf{cid}_{\mathsf{A}}].\mathsf{contract.status} \neq$ "`Active`";
   (b) Abort if $\mathtt{PreVerifyPromise}(P_{\mathsf{A}},$ "`Client`", $C_{\mathsf{H}}[\mathsf{cid}_{\mathsf{A}}],\ P_{\mathsf{A}}.\mathsf{amount},\ P_{\mathsf{A}}.Z,$ $P_{\mathsf{A}}.\mathsf{expiry},\ \mathsf{vk}_{\mathsf{A}}) = 0$

Fig. 10: AUPC Off-Chain Authorize Protocol

Off-Chain Protocol - Pay

Preliminaries. Let cid_A denote the channel id that A has with server H. Likewise, let cid_B denote the channel id between B and H. We assume that H knows $\mathsf{cid}_\mathsf{A}, \mathsf{cid}_\mathsf{B}$.

Protocol

1. Server H begins by performing the following steps:
   (a) Abort if $C_\mathsf{H}[\mathsf{cid}_\mathsf{A}].\mathsf{contract.status} \neq$ "`Active`";
   (b) $\alpha^\mathsf{A} := \mathtt{PSolve}(\mathsf{td}, P_\mathsf{A}.Z)$
   (c) $P_\mathsf{A}.\alpha = \alpha^\mathsf{A}$
   (d) $P_\mathsf{A}.\sigma = \mathsf{Adapt}(P_\mathsf{A}.\hat{\sigma}, \alpha^\mathsf{A})$
   (e) Abort if $\mathtt{VerifyPromise}(P_\mathsf{A},$ "`Client`", $C_\mathsf{H}[\mathsf{cid}_\mathsf{A}],$ $P_\mathsf{A}.\mathsf{amount},$ $P_\mathsf{A}.Z,$ $P_\mathsf{A}.\mathsf{expiry},$ $\mathsf{vk}_\mathsf{A}) = 0$
   (f) Send $P_\mathsf{A}.\sigma$ to A.

2. Upon receiving $P_\mathsf{A}.\sigma$ from H, A does the following:
   (a) Abort if $C_\mathsf{A}[\mathsf{cid}_\mathsf{A}].\mathsf{contract.status} \neq$ "`Active`";
   (b) Abort if $\mathtt{VerifySolution}(C_\mathsf{A}[\mathsf{cid}_\mathsf{A}], P_\mathsf{A}, P_\mathsf{A}.\sigma, P_\mathsf{A}.Z.A_\alpha) = 0$;
   (c) Invoke $\mathtt{UpdateChannel}(C_\mathsf{A}[\mathsf{cid}_\mathsf{A}], P_\mathsf{A},$ "`outgoing`");
   (d) $R_\mathsf{A} \leftarrow \mathtt{CreateReceipt}(C_\mathsf{A}[\mathsf{cid}_\mathsf{A}], \mathsf{sk}_\mathsf{A})$;
   (e) Send $[R_\mathsf{A}, P_\mathsf{A}]$ to H;
   (f) $\alpha^\mathsf{B} = \alpha^\mathsf{A} \cdot \beta_\mathsf{A}^{-1}$
   (g) Send $[Z_B, \alpha^\mathsf{B}]$ to B.

3. Upon receiving $[Z_B, \alpha^\mathsf{B}]$ from A, B does the following:
   (a) Abort if $C_\mathsf{B}[\mathsf{cid}_\mathsf{B}].\mathsf{contract.status} \neq$ "`Active`";
   (b) Abort if $\alpha^\mathsf{B} = \perp$
   (c) $\alpha^\mathsf{H} = \alpha^\mathsf{B} \cdot \beta_\mathsf{B}^{-1}$
   (d) $P_\mathsf{H}.\alpha = \alpha^\mathsf{H}$
   (e) $P_\mathsf{H}.\sigma = \mathsf{Adapt}(P_\mathsf{H}.\hat{\sigma}, \alpha^\mathsf{H})$
   (f) Abort if $\mathtt{VerifyPromise}(P_\mathsf{H},$ "`Server`", $C_\mathsf{B}[\mathsf{cid}_\mathsf{B}],$ $P_\mathsf{H}.\mathsf{amount},$ $P_\mathsf{H}.Z,$ $P_\mathsf{H}.\mathsf{expiry},$ $\mathsf{vk}_\mathsf{H}) = 0$
   (g) Send $P_\mathsf{H}.\sigma$ to H.

4. Upon receiving $P_\mathsf{H}.\sigma$ from B, H does the following:
   (a) Abort if $C_\mathsf{H}[\mathsf{cid}_\mathsf{B}].\mathsf{contract.status} \neq$ "`Active`";
   (b) Abort if $\mathtt{VerifySolution}(C_\mathsf{H}[\mathsf{cid}_\mathsf{B}], P_\mathsf{H}, P_\mathsf{H}.\sigma, P_\mathsf{H}.Z.A_\alpha) = 0$;
   (c) Invoke $\mathtt{UpdateChannel}(C_\mathsf{H}[\mathsf{cid}_\mathsf{B}], P_\mathsf{H},$ "`outgoing`");
   (d) $R_\mathsf{H} \leftarrow \mathtt{CreateReceipt}(C_\mathsf{H}[\mathsf{cid}_\mathsf{B}], \mathsf{sk}_\mathsf{H})$;
   (e) Send $[R_\mathsf{H}, P_\mathsf{H}]$ to B;

5. Upon receiving $[R_\mathsf{A}, P_\mathsf{A}]$ from A, H does the following:
   (a) Abort if $\mathtt{VerifyReceipt}(R_\mathsf{A}, C_\mathsf{H}[\mathsf{cid}_\mathsf{A}], \mathsf{vk}_\mathsf{A}) = 0$;
   (b) Invoke $\mathtt{UpdateChannel}(C_\mathsf{H}[\mathsf{cid}_\mathsf{A}], P_\mathsf{A},$ "`incoming`");
   (c) $C.\mathsf{receipt} \leftarrow [C.\mathsf{credit}_\mathsf{in}, C.\mathsf{acc}_\mathsf{in}, R_\mathsf{A}.\sigma]$.

6. Upon receiving $[R_\mathsf{H}, P_\mathsf{H}]$ from H, B does the following:
   (a) Abort if $\mathtt{VerifyReceipt}(R_\mathsf{H}, C_\mathsf{B}[\mathsf{cid}_\mathsf{B}], \mathsf{vk}_\mathsf{H}) = 0$;
   (b) Invoke $\mathtt{UpdateChannel}(C_\mathsf{B}[\mathsf{cid}_\mathsf{B}], P_\mathsf{H},$ "`incoming`");
   (c) $C.\mathsf{receipt} \leftarrow [C.\mathsf{credit}_\mathsf{in}, C.\mathsf{acc}_\mathsf{in}, R_\mathsf{H}.\sigma]$.

Fig. 11: AUPC Off-Chain Pay Protocol