Off-Chain Programmability at Scale

Yibin Yang[2⋆], Mohsen Minaei[1⋆], Srinivasan Raghuraman[1], Ranjit Kumaresan[1], and Mahdi Zamani[1]

[1] Visa Research
[2] Georgia Institute of Technology

Abstract. A typical approach for scaling blockchains is to create bilateral, off-chain channels, known as payment/state channels, that can protect parties against cheating via on-chain collateralization. While such channels have been studied extensively, not much attention has been given to off-chain programmability, where the parties can agree to enforce *arbitrary* conditions over their payments without going on-chain. Such ability is especially important for scaling off-chain channels via the hub-and-spoke model, where each party establishes a channel with a highly available (but untrusted) hub without a priori knowledge about the type and conditions of its off-chain transactions.

We introduce the notion of a *programmable payment channel* (PPC) that allows two parties to agree on a smart contract off-chain specifying the conditions on which the transactions can happen. If either party violates any of the terms, the other party can later deploy the contract on-chain to receive a remedy as agreed upon in the contract. Specifically, our PPC supports programmable payments where only one party deposits to the agreed off-chain contract, enabling lightweight payments. We further show that any two-party contract (even ones with two party deposits) can be implemented with PPC, by a compiler and associated protocol, allowing the parties to use their pre-deposited on-chain collaterals for *any* off-chain interaction potentially not anticipated at the time of channel setup. We formalize and prove the security and correctness of our protocol under the UC framework. We implement our protocol on Ethereum using accumulators to achieve efficient concurrent programmable transactions and measure the gas overhead of a hash-time-lock PPC contract to be < 100K which can be amortized over many off-chain payments.

1 Introduction

Today, financial institutions offer a variety of services such as payments, loans, trading, etc., to consumers and businesses. These products heavily rely on some metrics of trustworthiness that allow the institution and the clients to engage in transactions while minimizing the risk of financial loss in case either party fails to meet certain requirements that protect both parties' interests. These requirements are usually specified as a mutual contract agreed upon by both parties, allowing the suffering party to collect all or a portion of the other party's assets as a remedy in case of failure.

With the rise of decentralized services, financial products can be offered on blockchains with higher security and lower operational costs. With its ability to run arbitrary programs, called smart contracts, and direct access to assets, a blockchain can execute complex financial contracts and settle disputes automatically. Unfortunately, these benefits all come with a major scalability challenge due to the overhead of on-chain transactions, preventing the adoption of blockchain services as mainstream financial products.

Payment channels [2,13] are a class of mechanisms for scaling blockchain payments, by "off-loading" transactions to an off-chain communication channel between the two parties. The channel is "opened" via an on-chain transaction to fund the channel, followed by any number of off-chain transactions. Eventually, by a request from either or both parties, the channel is "closed" via another on-chain transaction.

⋆ Both authors contributed equally and are considered co-first authors. This work was done while at Visa Research.

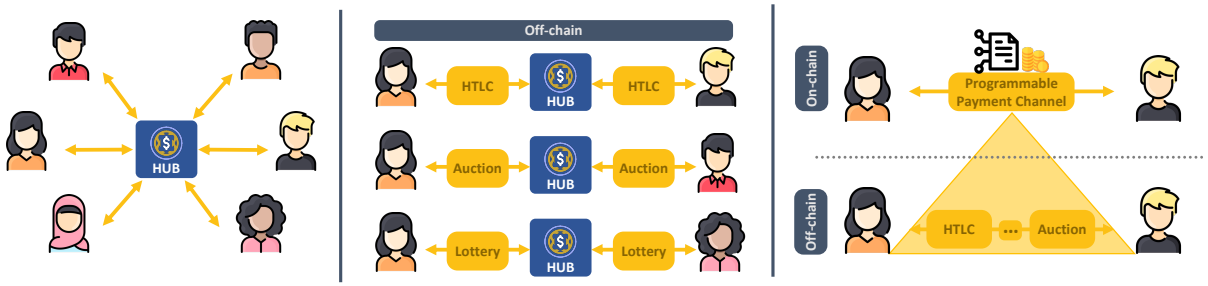


Fig. 1: **Left:** Hub-and-spoke model: Each party creates a single channel with the hub;
Middle: Every pair of parties reuse their channels with the hub to execute different contracts; **Right:** PPC between two parties supporting any off-chain application.

This design avoids the costs and the latency associated with on-chain operations, effectively amortizing the overhead of on-chain transactions over many off-chain ones. While several proposals improve the scalability of payment channels [30,22,3,16,24,18,25,31], they do not allow imposing arbitrary conditions on off-chain payments.

State channels [4,28,4,17,14,12,10] allow two parties to perform general-purpose computation off-chain by mutually tracking the current state of the program. Unfortunately, the existing state channel proposals have two major drawbacks in practice. First, they require the parties to fix the program, which they wish to run off-chain, at the time of channel setup. This means that no changes to the program are allowed after the parties go off-chain. This is especially problematic in off-chain scalability approaches based on the hub-and-spoke model [33,15,8], where each party establishes a general-purpose channel with a highly available (but untrusted) hub during setup to be able to later transact with many other parties without the need to establish an individual channel with each party (see Fig. 1 Left and Middle). In practice, parties usually have no a priori knowledge about the specific set of conditions required to transact with other parties that they have never established any relationship with.

Second, the complexity of the existing state channel proposals could be overkill for simple, programmable payments. The authorization of an off-chain transaction via a payment channel is significantly simpler as the flow of the payments is usually unidirectional (i.e., from a payer to a payee) while state channels need to track all state changes from both parties irrespective of the payment direction.

In this paper, we introduce the notion of *programmable payment channels* (PPC) that allows the parties to agree off-chain on the set of conditions (i.e., a smart contract) they wish to impose for each of their off-chain transactions (see Fig. 1 Right). A classic example of such a program is a hash-time-locked contract (HTLC) [1] which is foundational to the design of payment channels [30,3]. These payment channel networks rely on HTLCs to provide trustless routing of payments across the network.

While most current payment channels already embed HTLCs for routing, many useful applications remain difficult to build on top of payment channels. Consider the following simple programmable payment example. Alice wants to reserve a room through an established payment channel with the hotel. Alice would like to send a payment under the following conditions: (1) Alice is allowed to cancel the reservation within 48 hours of booking to get back all of her funds, and (2) Alice can get back half of her funds if she cancels the reservation within 24 hours of the stay date. Achieving this simple real-life example of a payment with current payment channels and HTLCs is either inefficient or impossible.

Our Contribution. This paper makes the following contributions:

– We propose the notion of a programmable payment channel (PPC) that is a payment channel allowing two parties to transact off-chain according to the collateral that they deposit on-chain and a smart contract that they agree on off-chain. PPC provides the following features:
  - *Scalability:* Only opening and closing the channel require on-chain access.
  - *Off-Chain Programmability*: The PPC protocol does not need be modified for new payment logic after the channel is opened.
  - *Completeness*: Any two-party contract can be taken off-chain and compiled into two interlocked programmable payments.
– We formalize PPC and prove its correctness and security in the Universal Composability (UC) framework using a global ledger.
– We evaluate PPC by instantiating it on Ethereum. We show how the PPC contract can deploy new contracts that embed the conditions of a payment. Furthermore, we use cryptographic accumulators to achieve efficient concurrent payments. Our results show that deploying the PPC contract needs about 3M gas, and further see that in the optimistic case (honest parties) of settling on-chain we need only 75K gas, while in the pessimistic case (malicious parties) 700K more gas is needed for a simple logic such as HTLC.

2 Related Work

Payment Channels. The key idea behind a payment channel is an on-chain contract: both parties instantiate this contract and transfer digital money to it. Later on, whenever one party wants to pay another, they simply signs on the other party's monotonically-increasing credit. When the two parties want to close the channel, they submit their final signed credits to rebalance the money in the channel. Before closing the channel, no execution has to happen on the blockchain; the payment between two parties relies only on sending digital signatures via a traditional communication network. Payment channels have been heavily studied and deployed before [2,13,17,26,18,28,9,29].

State Channels. A proposal for executing arbitrary contracts off-chain is state channels [28,4,17,14,12,10]. The key idea is as follows: (1) The contract can be executed off-chain by exchanging signatures, and (2) the contract can be executed on-chain from the agreed state to resolve disagreement. For example, considering a two-party contract between Alice and Bob, whenever Alice wants to update the current state, she simply signs the newer state. Then, she forwards her signature and requests for Bob's signature. While Bob may not reply with his signature, Alice can submit the pre-agreed state to the blockchain with the contract and execute it on-chain. This idea can be naturally extended to multi-party contracts (e.g., [28,14,11]). In the optimistic case, the execution of the contract will be finished completely off-chain. However, due to its complexity, state channels largely remain in the realm of theory today. We argue that the work [12] is closest to ours, but unlike us, they do not provide any formal proofs or guarantees. Also, our protocols take advantage of the Ethereum CREATE2 opcode (which was introduced subsequent to the work of [12]).

Channels Formalization. We follow [16,17,14,5] to formalize our channel using *universal composable* (UC) framework with a global ledger. We note that these works mainly focus on *channel virtualization*, which are *not* directly related to this work.

Other Related Work. An excellent systematization of knowledge that explores off-chain solutions can be found in [20]. Coinbase Commerce uses CREATE2 on the state channel to reuse a *fixed* contract

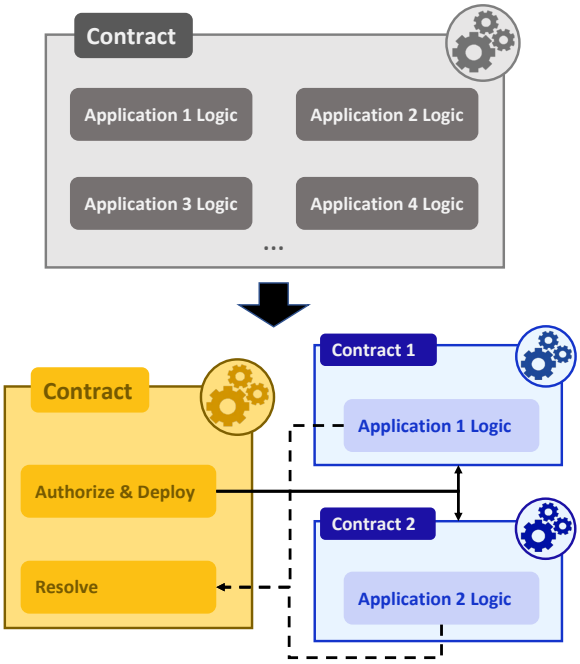


Fig. 2: Application-dependent approach (top) and on-the-fly promises as contracts (bottom)

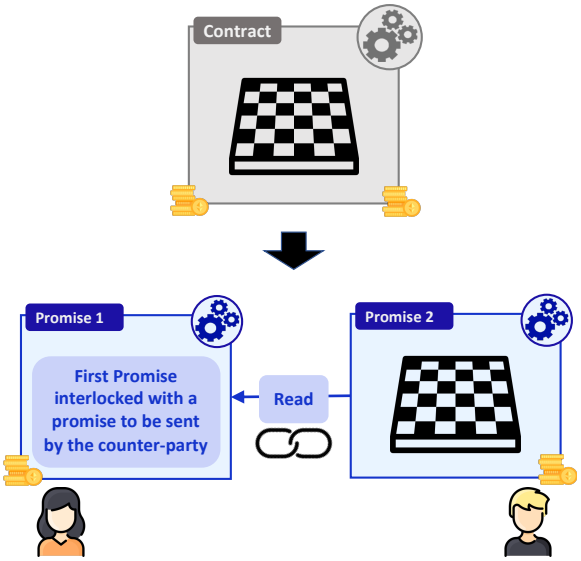


Fig. 3: Execute arbitrary two-party contract on PPC. Compiler compiles any two-party contract into two interlocked promises.

called Forwarder to process instant off-chain transactions[3]. This is different from our work where we utilize CREATE2 to achieve and commit general programmable payments, while Coinbase uses it to save gas fees by deploying the contract when the fees are less.

3 Technical Overview

Strawman Solution. Our starting point is to consider how programmability may be incorporated into a payment channel. The straightforward solution is to hard-code the logic of an application inside the protocol as a template. However, this approach is not desirable as every new application would require a protocol update that would also include changes to the existing on-chain contract. Typical state channels follow this application-based pattern. For instance, a state channel contract may enable off-chain chess but not a poker game.

Our Approach: Contracts Deploy Contracts. Contracts on blockchains are now allowed to deploy new contracts at deterministic addresses. For example, Ethereum introduced the opcode CREATE2 in EIP-1014[4]. Our PPC protocol crucially relies on the above feature to achieve *general* programmability.

In Ethereum, using the CREATE2 opcode, contracts deploy contracts whose address is set by $\mathcal{H}(\texttt{0xFF}, sender, salt, bytecode)$ (where $\mathcal{H}$ is a hash function; we replace it by a random oracle denoted by

[3] https://legacy.ethgasstation.info/blog/what-is-create2/
[4] EIP stands for Ethereum Improvement Proposals. EIP-1014 is available at https://eips.ethereum.org/EIPS/eip-1014.

RO in the rest of the paper). This capability implies that one can foresee the address of some yet-to-be-deployed contract. This property, which we dub *prescience*, will be crucial later.

In our system, a *promise* represents a single instance of a programmable payment which is written as a contract that can be deployed on-chain by the PPC contract. We rely on the PPC contract to determine whether a promise should be deployed. The PPC contract simply requires the payer's/sender's authentication via digital signatures, as funds flow unidirectionally in a programmable payment. After deploying the promise, to fast-forward to the last agreed off-chain execution state, we require both parties' signatures on the latest state. We directly embed this mechanism into the promise contract.

Promises are only deployed on-chain whenever a dispute arises, and the execution continues on-chain via the newly deployed contract. However, in the optimistic case, when both parties are honest, the entire execution is handled off-chain. Figure 2 presents our approach compared to the strawman solution.

To demonstrate that PPC is as expressive as two-party state channels, we also show how to compile any two-party contract into two *interlocked* promises. This compilation relies on the *prescience* property we described earlier. We briefly present this compiler by a toy example as shown in Figure 3: assume a two-party betting chess game. Party A instantiates a promise that relies on the existence of a promise from party B (cf. prescience). The promise authenticated by A can read the promise (yet to be) authenticated by B that will be located at a specific address, which plays a role in executing the chess game as well as settling the payment from B to A. B can only benefit from A's promise if it correctly forms its promise, which A is expecting. If a party deviates from the game, the other party can take both promises to the PPC contract and resolve the game.

4 Notations and Models

Contracts and Programmable Payment Channel. We define the object *contracts* and *channel* as in [17]. A *contract instance* consists of two attributes: *contract storage* (accessed by key $\mathsf{storage}$) and *contract code* (accessed by key code), where the contract storage σ is an attribute tuple containing at least the following attributes: $\sigma.user_L$, $\sigma.user_R$, $\sigma.locked$, $\sigma.cash$; and the contract code is a tuple $C := (\Lambda, g_1, \ldots, g_r, f_1, \ldots, f_s)$. A programmable payment channel is an attribute tuple $\gamma := (\gamma.id, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.cash, \gamma.\mathsf{pspace}, \gamma.duration)$. Note that the attribute $\gamma.duration$ was not part of prior channel formalizations (e.g., [17,14]); we will discuss the need for it in more details in Section 5. Appendix A.1 includes the complete descriptions for these two objects.

We further define two auxiliary functions: (1) $\gamma.\mathsf{endusers} := \{\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}\}$; and (2) $\gamma.\mathsf{otherparty}(x) := \gamma.\mathsf{endusers} \setminus \{x\}$ where $x \in \gamma.\mathsf{endusers}$.

Promises. We name a programmable payment a *promise*. Informally, a promise instance can be viewed as a special contract instance where only one party offers money. Formally, a *promise instance* consists of two attributes: *promise storage* (accessed by key $\mathsf{storage}$) and *promise code* (accessed by key code). Promise storage σ is an attribute tuple containing at least the following attributes: (1) $\sigma.payer$ denotes the party who sends money; (2) $\sigma.payee$ denotes the party who receives money; (3) $\sigma.final \in \mathbb{R}_{\geq 0}$ denotes the amount of money transferred from payer to payee; and (4) $\sigma.end \in \{0, 1\}$ denotes whether the payment is finalized. A promise code is a tuple $C := (\Lambda, \mathsf{Construct}, f_1, \ldots, f_s)$ similar to contract code with further restrictions: (1) the unique constructor function $\mathsf{Construct}$ will always set the caller to be the payer in the storage created; (2) the constructor function's output is independent of input argument t; and (3) any execution function f will not modify the *end* attribute in storage from 1 to 0.

GUC Model. We model and formalize PPC under *global universal composable* (GUC) framework [6,7]. UC is a general purpose framework for modeling and constructing secure protocols. The correctness and security of protocols rely on simulation-based proofs. We defer the formal description to Appendix A.2.

Adversary. We consider the adversary who can corrupt one party in the two-party channel. The corrupted party can deviate from the protocol arbitrarily and is also known as *malicious* or *byzantine*.

Network & Time. We assume a synchronous complete peer-to-peer authenticated communication network. Thus, the execution of protocol can be viewed as happening in rounds. The round is also used as global timestamp. We use $msg \overset{t\leq T}{\hookleftarrow} P$ to denote the message will be sent by party P before round T. Similarly, we use $msg \overset{t\leq T}{\hookrightarrow} P$ to denote that the message will be delivered to party P before round T. Specifically, the adversary is allowed to introduce delays for messages delivery but the delays should be bounded.

Cryptocurrency and Contracts. We follow [17,14] and model cryptocurrency as a global ledger functionality $\hat{\mathcal{L}}$ in the GUC framework. Parties can shift funds from/to the ledger functionality solely by invoking other ideal functionalities (including contracts) that can invoke the methods Add/Remove. However, any operation on the global ledger will happen within a delay of Δ rounds as decided by the adversary $\mathcal{A}$, capturing the fact that this is an on-chain transaction.

5 Realizing PPC

We propose our PPC protocol under the UC framework following [16,17,14]. We first define the ideal functionality $\mathcal{F}_{PPC}$ (with dummy parties) which summarizes all the features that our PPC protocol will provide. Our real PPC protocol π will be presented in a hybrid model, using a PPC contract functionality $\mathcal{G}_{PPC}$. The functionality $\mathcal{G}_{PPC}$ captures the on-chain contract functionality, which can be implemented on Ethereum via multiple smart contracts. The protocol π, on the other hand, captures the off-chain protocols between parties that will be executed via authenticated channels.

We make several reasonable simplifications for better presentation. Crucially, we assume the existence of a public key infrastructure (PKI) and work in the random oracle (RO) model to capture the CREATE2 opcode in Ethereum. We defer the justification of our simplifications to Appendix B, partial protocol boxes to Appendix C and the entire simulator-based proof to Appendix D.

5.1 The Ideal Functionality $\mathcal{F}_{PPC}$

$\mathcal{F}_{PPC}^{\hat{\mathcal{L}}}(PS)$ (see Fig. 4) is the UC ideal functionality achieved by our protocol. It will maintain a key-value data structure Γ to track all programmable payment channels between parties. The functionality has an argument promise set PS, which denotes all possible promise codes that can be instantiated and executed in channels. Note that this does not harm our general programmability and on-the-fly logic since PS can be any set of promise codes. In the following presentation, we will use $\mathcal{F}_{PPC}$ as an abbreviation in the absence of ambiguity. $\mathcal{F}_{PPC}$ mainly contains the following 4 procedures.

PPC **Creation.** Similar to all other channels, a party can instantiate a channel with another party by sending the channel creation information to $\mathcal{F}_{PPC}$. W.l.o.g., assume party P wants to construct a channel with party Q. Within Δ rounds, $\mathcal{F}_{PPC}$ will take corresponding coins specified by the channel instance from P's account from $\hat{\mathcal{L}}$. If Q agrees to the creation, within another Δ rounds, $\mathcal{F}_{PPC}$ will take Q's coins. Thus, the successful creation of a initial programmable payment channel takes at most 2Δ rounds.

Promise Initial Instance Creation. This procedure is used to create a promise, which is a programmable payment from payer P to the payee Q. The creation instance is specified by payer's choice of channel, contract code and arguments for the corresponding constructor function. Informally, this procedure captures the fact that the payee cannot refuse a payment from payer. Since payee always gains coins in any promise, we don't need an acknowledgement from the payee to instantiate a promise. Thus,

Functionality $\mathcal{F}_{PPC}^{\hat{\mathcal{L}}}(PS)$

Let RO be a random oracle.

Programmable payment channel creation

Upon $(\texttt{create}, \gamma) \overset{t_0}{\hookleftarrow} P$ where γ is a valid initial programmable payment channel ($P \in \gamma.\mathsf{endusers}, \gamma.\mathsf{cash}(\cdot) \geq 0, \gamma.\mathsf{pspace} = \bot$, denote $Q := \gamma.\mathsf{otherparty}(P)$):

1. Within Δ rounds remove $\gamma.\mathsf{cash}(P)$ from P's account on $\hat{\mathcal{L}}$.
2. If $(\texttt{create}, \gamma) \overset{t_1 \leq t_0+\Delta}{\hookleftarrow} Q$, remove within another Δ rounds $\gamma.\mathsf{cash}(Q)$ coins from Q's account on $\hat{\mathcal{L}}$, set $\Gamma(\gamma.id) := \gamma$, and send $(\texttt{created}, \gamma) \hookrightarrow \gamma.\mathsf{endusers}$ and stop.
3. Else, upon $(\texttt{refund}, \gamma) \overset{>t_0+2\Delta}{\hookleftarrow} P$, within Δ rounds add $\gamma.\mathsf{cash}(P)$ coins to P's account on $\hat{\mathcal{L}}$.

Promise initial instance creation

Upon $(\texttt{create-instance}, id, C, z) \overset{t_0}{\hookleftarrow} P$, let $\gamma := \Gamma(id)$ and calculate $pid := RO(id, P, C, z)$. If $\gamma = \bot$ or $P \notin \gamma.\mathsf{endusers}$ or $\gamma.\mathsf{pspace}(pid) \neq \bot$ or $C \notin PS$ then stop. Else proceed as follows:

- Let $\nu := \bot$ and $\sigma := C.\mathsf{Construct}(P, t_0, z)$. Stop if $\sigma = \bot$. Set $\nu.\mathsf{code} := C$ and $\nu.\mathsf{storage} := \sigma$. Set $\Gamma(id).\mathsf{pspace}(pid) := \nu$. Send $(\texttt{instance-created}, id, pid, \nu) \overset{t_0+1}{\hookrightarrow} \gamma.\mathsf{endusers}$.

Promise instance execution

Upon $(\texttt{execute}, id, pid, f, z) \overset{t_0}{\hookleftarrow} P$, let $\gamma := \Gamma(id)$. If $P \notin \gamma.\mathsf{endusers}$ or $\gamma.\mathsf{pspace}(pid) = \bot$ or $f \notin \gamma.\mathsf{pspace}(pid).\mathsf{code}$ then stop. Else proceed as follows:

- In the optimistic case when both parties in $\gamma.\mathsf{endusers}$ are honest, set $T := t_0 + 5$.
- In the pessimistic case when at least one party in $\gamma.\mathsf{endusers}$ is corrupt, set $T := t_0 + 4\Delta + 5$.

If both party are honest, set $t := t_0$, else t is set by the simulator (however $t \leq T$). Let $\nu := \gamma.\mathsf{pspace}(pid)$ and $\sigma := \nu.\mathsf{storage}$. Let $(\tilde{\sigma}, m) := f(\sigma, P, t, z)$. If $m = \bot$, then stop. Else set $\Gamma(id).\mathsf{pspace}(pid).\mathsf{storage} := \tilde{\sigma}$ and send $(\texttt{executed}, id, pid, P, f, t, z, \nu) \overset{t_1 \leq T}{\hookrightarrow} \gamma.\mathsf{endusers}$.

Programmable payment channel closure

Upon $(\texttt{close}, id) \overset{t_0}{\hookleftarrow} P$, let $\gamma := \Gamma(id)$. If $P \notin \gamma.\mathsf{endusers}$ then stop. Else block all future close invocations on γ. Wait at most 7Δ rounds and proceed as follows:

1. If there exists $pid \in \{0,1\}^*$ such that $\gamma.\mathsf{pspace}(pid).\mathsf{storage} \neq \bot$ and $\gamma.\mathsf{pspace}(pid).\mathsf{storage}.end = 0$, then wait $\gamma.duration$ rounds.
2. Calculate the following values:
   (a) Set $total := \gamma.\mathsf{cash}(\gamma.\mathsf{Alice}) + \gamma.\mathsf{cash}(\gamma.\mathsf{Bob})$.
   (b) Set $credit_A := \sum_{\gamma.\mathsf{pspace}(pid).\mathsf{storage.payer}=\gamma.\mathsf{Bob}}(\gamma.\mathsf{pspace}(pid).\mathsf{storage}.final)$.
   (c) Set $credit_B := \sum_{\gamma.\mathsf{pspace}(pid).\mathsf{storage.payer}=\gamma.\mathsf{Alice}}(\gamma.\mathsf{pspace}(pid).\mathsf{storage}.final)$.
3. Within Δ rounds, add $min\{total, max\{0, \gamma.\mathsf{cash}(\gamma.\mathsf{Alice}) + credit_A - credit_B\}\}$ coins to $\gamma.\mathsf{Alice}$'s and $min\{total, max\{0, \gamma.\mathsf{cash}(\gamma.\mathsf{Bob}) + credit_B - credit_A\}\}$ coins to $\gamma.\mathsf{Bob}$'s account.
4. Send $(\texttt{contract-close}, id) \overset{t_1 \leq t_0+8\Delta+\gamma.duration}{\hookrightarrow} \gamma.\mathsf{endusers}$.

Fig. 4: The ideal functionality $\mathcal{F}_{PPC}^{\hat{\mathcal{L}}}(PS)$ achieved by the PPC protocol.

the creation takes exactly 1 round. Note that this does not hold for state channels as formalized in [17] where an instance requires coins from both parties. The created instance's identifier is determined by a random oracle (denote by RO). The random oracle ensures that a promise instance pid can be created at most once. Since the channel identifier id is also an argument to the random oracle, statistically, there will not be two promise instances even across different channels that share the same pid.

Promise Instance Execution. This procedure is used to update the promise instance's storage. Specifically, party P can execute the promise pid in channel id as long as P is one of the participants of the channel. Note that the existence of pid implies that this instance is properly constructed by the payer via the promise instance creation procedure. If both parties are honest, the execution completes in $O(1)$ rounds, inferring no on-chain operation. Otherwise, if one of them is corrupt, it relies on on-chain operations which takes $O(\Delta)$ rounds. Note that, the adversary can select the function execution time, however, it cannot block a party from executing and updating it.

Programmable Payment Channel Closure. When one party of the channel instance γ wants to close the channel, the procedure checks whether there is still some promise that has not been finalized. If so, it will wait for $\gamma.duration$ rounds. The corresponding procedure in the state channel functionality of [17] requires that all contract instances in the channel are finalized in order to close the channel. We cannot imitate this approach because in our case, the creation of a promise instance need only be authenticated by the payer, and so requiring finality will allow a malicious party to block closing by simply creating some non-finalizable promise instance. Note that it could be the case that a malicious party overpays the other party, but our ideal functionality ensures that no extra coins are created. Informally, this implies that it is the users' responsibility to ensure that all the expected promises can be finalized in $\gamma.duration$ rounds.

5.2 Protocol Realizing $\mathcal{F}_{PPC}$

We present the programmable payment channel protocol in this subsection. The protocol $\pi(PS)$ is defined assuming access to an ideal programmable payment channel smart contract functionality $\mathcal{G}_{PPC}$. All specifications in $\pi(PS)$ should be viewed as the procedure executed on the party's local machine while all specifications in $\mathcal{G}_{PPC}$ should be viewed as on-chain smart contracts. Since $\mathcal{G}_{PPC}$ emulates smart contracts, coins in $\hat{\mathcal{L}}$ can be transferred to/from it.

Each party P will maintain a local key-value data structure Γ^P to monitor all channels belongings. P also locally monitors all promise instances executed on each channel. Besides the latest promise storage σ, P also maintains another key-value object Γ^P_{aux} to save auxiliary data including signatures, version, etc. We briefly explain the use of signatures and version below.

Signature. The signatures are used to authenticate the creation as well as the latest state of promise instances. Users need to provide a valid signature from the payer on the creation arguments in order for the PPC contract to deploy the authenticated promise contract. To execute the contract off-chain, two parties exchange their signatures on the latest state. For simplification, we directly allow the PPC contract to "deploy" the promise instance if either party provides a state (including *pid*) with both parties' signatures. Since one party is honest, *pid* is in line with creation process. Formally, the PPC contract deploys an initial promise contract. Then a party can submit the latest state to the promise contract.

Version. Integer number version is used in order to obtain a total ordering for the states of promise instance. We define that the initial promise storage created by constructor function is of version 0.

Register. A special sub-procedure (not interacting with environment) called Register is used to deploy a promise instance on-chain. The PPC protocol π will heavily use this sub-procedure to register a promise instance. The protocol requests both parties to submit a valid storage and will deploy the one with larger version. The entire procedure will be finished within 3Δ rounds with one corrupt party and within 2Δ rounds with two honest parties. We defer the specification and discussion of this sub-procedure to Appendix C.

We are now ready to describe the protocol that achieves the 4 procedures in $\mathcal{F}_{PPC}$. All UC-style protocol boxes can be found in Appendix C.

Create a Programmable Payment Channel. Party P sends the valid initial channel object to the PPC contract. If the other party agrees and they both have enough funds, coins will be transferred to $\mathcal{G}_{PPC}$ from $\hat{\mathcal{L}}$.

Create an Initial Promise Instance. Party P signs the constructor arguments (without time) and forward it to party Q. Both parties then can use the Register sub-procedure to deploy it as a contract on Layer-1 if needed.

Execute a Promise Instance. Assume party P wants to execute a promise function to update promise storage. If this promise instance is already on-chain, he directly calls the function on Layer-1. Otherwise, P will first try to peacefully (off-chain) execute the promise function. P will fetch the latest storage σ including version from his local memory, execute it locally and sign the newer storage $\tilde{\sigma}$ with (version $+1$). P forwards the signature to Q and requests her signature. If Q accepts the execution, she sends back her signature, and the execution will be finished off-chain in $O(1)$ rounds. If not, P needs to execute the function on-chain. He can register the latest promise instance on the Layer-1 blockchain. Note that since Q already has both parties' signatures on the newer version, she could register it directly on-chain to invalidate the version P wants to register. If after the registration sub-procedure, the promise instance on-chain is still P's version, P can then update it by on-chain function execution. We follow [17] to sequentialize the execution.

Closing a PPC. For a party P to close a PPC channel γ with Q, he first (in parallel) registers all the promise instances he has off-chain in γ^P.pspace within 3Δ rounds. Then P will notify the PPC contract $\mathcal{G}_{PPC}$ that he wants to close the channel. The contract will further notify Q about the closing and set up a 3Δ time window for Q to register her local instances. Note that if both parties are honest, all promise instances in γ^Q.pspace should already be registered by P. After the time window passes, $\mathcal{G}_{PPC}$ will check the status of all registered instances in γ. If all of them are finalized, the channel will be closed and corresponding final balances will be transferred back to the party's account on $\hat{\mathcal{L}}$ within Δ rounds. If there still exists a not finalized instance, it will wait for $\gamma.duration$ rounds and split the coins.

5.3 Theorems

We formally claim our theorems and features of PPC captured by $\mathcal{F}_{PPC}$. All of the theorems are proved in Appendix D.

Theorem 1 (Main). *Suppose the underlying signature scheme is existentially unforgeable against chosen message attacks. The protocol $\pi(PS)$ working in $\mathcal{G}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$-hybrid model emulates the ideal funcitonlity $\mathcal{F}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$ against any environment $\mathcal{E}$ in the random oracle model for every set of promise codes PS and every $\Delta \in \mathbb{N}$.*

Claim 1 (Single round instance creation)**.** *The creation of an initial promise instance takes* 1 *round.*

Claim 2 (Constant round off-chain execution)**.** *If both parties are honest, every call to instance execution procedure of $\mathcal{F}_{PPC}$ will finish in $O(1)$ round.*

Theorem 2 (Coins momentum). *Any channel γ cannot produce coins.*

Theorem 3 (Balance security). *Honest users will not lose coins in channels.*

5.4 Comparison with Previous Channels

Our formalization methodology follows the previous works on payment and state channels, i.e., [16,17,14]. Despite looking similar, there are some crucial differences between our work and prior works. In particular, PPC has certain features that other works do not, and we discuss these below.

1. **General Programmable Payment:** Our PPC protocol and the ideal $\mathcal{F}_{PPC}$ enable programmable payments by defining the object promise and the procedure execute. These are not part of the traditional non-programmable payment channels [16]. Note that PPC's programmability is general since all execution functions $f_1, \ldots, f_s$ associated with a contract can be written inside the promise as $C = (\cdot, \cdot, f_1, \ldots, f_s)$.

2. **Succinct Initialization:** Our PPC protocol and the ideal $\mathcal{F}_{PPC}$ initialize a promise using a single digital signature from payer to payee, i.e., the procedure create-instance. This is appropriate since only the payer will spend digital coins. State channels (e.g., [17,14]) do not have this feature. Formally, the procedure named update (in [17,14]) involves exchanging multiple signatures and crucially, potential on-chain interactions.
3. **On-the-fly Logic:** Our PPC protocol and the ideal $\mathcal{F}_{PPC}$ instantiate a promise as a Layer-1 contract, which will only be deployed in the pessimistic case. The execution of the promise is performed on the blockchain directly via the corresponding contract. This is captured by the instance-construct procedure in $\mathcal{G}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$, which is used to register an initial promise on-chain. The identifiers (i.e., the contract address) of promises are determined by a random oracle (emulating the CREATE2 opcode). The argument PS is just a formalization choice–it can contain all possible promises, and so, the protocol does not need to be updated based on the promise types.

6 Compiling A Contract to Promises

On the one hand, our programmable payment channel protocol subsumes regular payment channel protocols. A simple payment can be captured by payer P creating an initial promise instance directly constructed as finalized with the proper amount. On the other hand, it seems that our programmable payment channel protocol may not subsume protocols for state channels, i.e., execute a contract where two parties can both deposit coins in. Informally, a promise is somewhat a half state contract with the specified payer and payee where only the payee can put money in. Surprisingly, this turns out to be *false*. In this section, we provide a compiler that can compile any two-party contract into two promises. With an associated protocol to set up these two promises, we enable arbitrary two-party contract to be executed off-chain on the programmable payment channel.

Our compiler relies on two key observations:

- Since an on-chain promise is a contract, it can read/write other contracts.
- Since an on-chain promise is created by the CREATE2 opcode, a promise instance (even one that has not been created yet) is bound to one address.

Consider a contract code as $C := (\Lambda, g_1, \cdots, g_r, f_1, \cdots, f_s)$ where Alice (denoted by A) wants to start an instance (with Bob, denoted by B) created by some $g \in \{g_i\}$ with auxiliary inputs z' at time t'. That is, the first state will be $\sigma' := g(A, t', z')$ ($\sigma' \neq \perp$). At a high level, our compiler will compile this contract code into two promise codes, one from A to B, and one from B to A. All the logic will be wrapped into the promise from B to A. For example, if Alice and Bob want to play a chess game, the chess board logic will be coded inside the promise from B to A. This promise will be updated into a final payment if B indeed needs to pay A at the end of the game. The promise from A to B simply monitors the state of the other promise and settles the payment if A needs to pay B. The CREATE2 opcode plays an important role here as Alice can "play B in her head" and simulate the creation of B's promise. B's promise will be assigned to a specific address that A can precompute via playing B in her head as above. A can thus create a promise that reads B's non-existing promise.

The construction of the promise code from Bob to Alice, defined as $C_{B\to A} := (\cdot, \mathsf{Construct}, f_1, \cdots, f_s, \mathsf{Enable}, \mathsf{Finalize})$, is shown in Figure 5a. The construction of the promise code from Alice to Bob, defined as $C_{A\to B} := (\cdot, \mathsf{Construct}, \mathsf{Finalize})$, is shown in Figure 5b. We discuss some crucial points:

- The constructor function of $C_{B\to A}$ uses σ' as a white-box. Note that the constructor function is independent of input t. However, the contract address will be affected by the *salt* picked by Alice.

Promise Code $C_{B\to A}$

Construct$(P, t, salt)$:
1. if $P \notin \{\sigma'.user_L, \sigma'.user_R\}$, return $\perp$.
2. $\sigma \leftarrow \sigma'$.
3. σ.payer $:= P, \sigma.expiry := t' + 2\Delta + 5, \sigma$.payee $:= \{\sigma'.user_L, \sigma'.user_R\} \setminus P, \sigma.end := 0, \sigma.final := 0, \sigma.valid := 0$.
4. return σ.

Enable(σ, P, t, x):
1. if $P \neq \sigma$.payer, return $(\sigma, \perp)$.
2. if $t > \sigma.expiry$, return return $(\sigma, \perp)$.
3. if $\sigma.valid = 1$, return $(\sigma, \perp)$.
4. $\sigma.valid := 1$.
5. return $(\sigma, 1)$.

Finalize(σ, P, t, x):
1. if $P \neq \sigma$.payer and $P \neq \sigma$.payee, return $(\sigma, \perp)$.
2. if $\sigma.valid = 0$, return $(\sigma, \perp)$.
3. if $\sigma.locked > 0$, return return $(\sigma, \perp)$.
4. if $\sigma.end = 1$, return return $(\sigma, \perp)$.
5. $\sigma.end := 1$.
6. $\sigma.final := \sigma$.cash$(\sigma$.payer$) < 0$? σ.cash$(\sigma$.payer$) : 0$.
7. $\sigma.final := -\sigma.final$.
8. return $(\sigma, 1)$.

$f_1, \cdots, f_s$.

(a) Promise code $C_{B\to A}$ from Bob to Alice.

Promise Code $C_{A\to B}$

Construct$(P, t, pid, salt)$:
1. $\sigma \leftarrow \perp$.
2. σ.payer $:= A, \sigma$.payee $:= B, \sigma.end := 0, \sigma.final := 0, \sigma.pid := pid$.
3. return σ.

Finalize(σ, P, t, x):
1. if $\sigma.pid$ does not exist, return $(\sigma, \perp)$, else let $\sigma_{B\to A}$ be the storage of contract $\sigma.pid$.
2. if $\sigma_{B\to A}.end = 0$, return $(\sigma, \perp)$.
3. if $P \neq \sigma$.payer and $P \neq \sigma$.payee, return $(\sigma, \perp)$.
4. if $\sigma.end = 1$, return return $(\sigma, \perp)$.
5. $\sigma.end := 1$.
6. if $\sigma_{B\to A}$.cash$(\sigma$.payer$) < 0$, then $\sigma.final := \sigma_{B\to A}$.cash$(\sigma$.payer$)$.
7. $\sigma.final := -\sigma.final$.
8. return $(\sigma, 1)$.

(b) Promise code $C_{A\to B}$ from Alice to Bob.

Fig. 5: The compiled promises

– Informally, $C_{B\to A}$ is not valid when constructed, but can become valid before some expiry time by invoking the function Enable. This is crucial because as we will show later, Alice will first send her promise to Bob. We need to ensure that Bob cannot silently register this promise and punish Alice without legally initializing the game.
– The two-party contract can be executed in the same way one would execute $C_{B\to A}$ since we directly clone $f_1, \cdots, f_s$.
– $C_{A\to B}$ trivially programs a payment from Alice to Bob while reading the state of corresponding $C_{B\to A}$.
– $C_{A\to B}$ is independent from the contract code C.

We are now ready to present the protocol for executing application contract C with two compiled promises $C_{A\to B}$ and $C_{B\to A}$ within a programmable payment channel between A and B. The protocol has 7 steps and is shown in Figure 6. We directly present it in a hybrid model where A and B connected with some PPC channel γ with identifier id stored in $\mathcal{F}_{PPC}$ (defined and realized in Section 5).

While executing C using the PPC and the state channel both achieve the intended functionality, executing C using two promises on PPC has an extra benefit. We end this section by briefly discuss this benefit.

Free Cancellation. In state channels (e.g., [17]), Alice will sign the first state created by the constructor function and send it to Bob. While an honest Bob will send back a signature based on whether he is in or not, a malicious Bob could abort. In this case, Bob already has two signatures, so Alice needs to block the contract identifier on the Layer-1 contract to prevent Bob from registering the state later. Executing two promises on PPC does not need this on-chain blocking. To see this, note that Alice's promise relies on a valid promise from Bob. If the Bob does not want to execute the contract, Bob cannot create a valid promise $pid_{B\to A}$ after the expiry time $\sigma.expiry$.

1. At round t', Alice wants to instantiate a contract instance created by contract code C and constructor function g. Note that she currently knows corresponding $C_{B\to A}$. She picks a random $salt$ string. Alice plays "Bob in her head" and calculates the address of the promise Bob would create as $pid := RO(id, B, C_{B\to A}, salt)$.
2. Alice sends $(\texttt{create-instance}, id, C_{A\to B}, pid) \overset{t'}{\hookrightarrow} \mathcal{F}_{PPC}$ to instantiate the promise to handle the payment from A to B. This promise will read another promise instantiated by $C_{B\to A}$ with fixed arguments $(B, \cdot, salt)$ created by channel γ. Alice also sends $(C, salt)$ to Bob at round t'.
3. At round $t'+1$, Bob will receive the creation notification of $C_{A\to B}$ from $\mathcal{F}_{PPC}$ and also the $(C, salt)$ values from Alice. Assume the promise instance is assigned to $pid_{A\to B}$ address. Bob checks every value is correctly generated. If Bob wants to execute the contract C, he sends $(\texttt{create-instance}, id, C_{B\to A}, salt) \overset{t'+1}{\hookrightarrow} \mathcal{F}_{PPC}$. Otherwise, Bob can simply stop/abort.
4. If Bob wants to execute the instance, he will receive $(\texttt{instance-created}, id, pid_{B\to A}, \nu)$ from $\mathcal{F}_{PPC}$ at round $t'+2$. He immediately sends $(\texttt{execute}, id, pid_{B\to A}, \mathsf{Enable}, \cdot)$ to make the instance of $C_{B\to A}$ valid.
5. Alice and Bob execute the contract instance by executing corresponding f_i in $C_{B\to A}$'s promise instance stored at $pid_{B\to A}$.
6. Suppose the execution of C's logic (at $pid_{B\to A}$) is finished at round t_{end}. Alice will send $(\texttt{execute}, id, pid_{B\to A}, \mathsf{Finalize}, \cdot)$ to settle the payment of the C's execution from B to A.
7. If Bob gets a notification of $pid_{B\to A}$'s finalization from $\mathcal{F}_{PPC}$ before time $t_{end} + 4\Delta + 5$, he immediately sends $(\texttt{execute}, id, pid_{A\to B}, \mathsf{Finalize}, \cdot) \hookrightarrow \mathcal{F}_{PPC}$ to settle the payment from A to B. Otherwise, he will send $(\texttt{execute}, id, pid_{B\to A}, \mathsf{Finalize}, \cdot) \hookrightarrow \mathcal{F}_{PPC}$ to settle his payment to A first at round $t_{end} + 4\Delta + 6$. Within another $4\Delta + 5$ rounds, once he gets the notification of $pid_{B\to A}$'s finalization, he sends $(\texttt{execute}, id, pid_{A\to B}, \mathsf{Finalize}, \cdot) \hookrightarrow \mathcal{F}_{PPC}$ to settle the payment from A to B.

Fig. 6: The protocol for executing compiled promises $C_{A\to B}$ and $C_{B\to A}$.

7 Implementation and Evaluation

We instantiated PPC in the Ethereum network. Here we provide an overview of the implementation and evaluation and defer further details to Appendix F.

Receipts. To enable an aggregation of all the previously resolved promises we introduce the *receipt* object. The message is signed by the payer and includes a credit value that denotes the *aggregated amount* of all previously resolved promises. This message decrease the number of transactions taken on-chain.

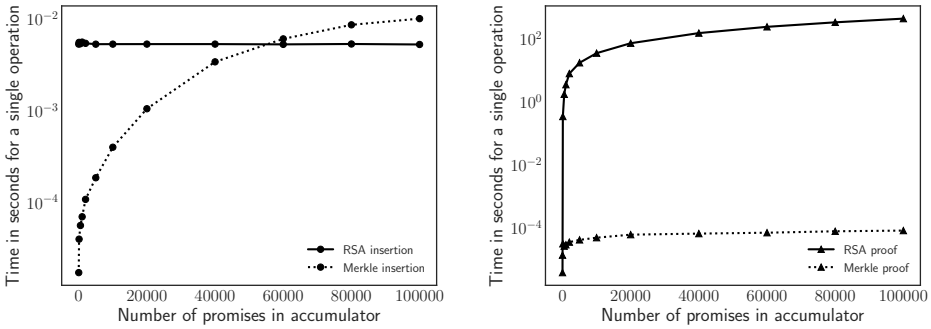
Accumulators. To provide maximum parallelization for a receiver that can process multiple promises simultaneously, the PPC allows the sender to submit multiple promises without waiting for each promise to be processed (we refer to this property as *non-blocking/concurrent* payments). The PPC implementation provides this feature by asking the parties to commit to the set of pending promises along with every receipt (to prevent double spending). To efficiently capture this set, we use cryptographic 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, shown in Fig. 7. We observe that the Merkle tree is the better choice as it has less overhead and much less gas cost for membership proof verification (450K gas compared to 20K when 100K promises are stored in the accumulator).

Smart Contracts. Figure 10 shows our PPC contract which we have implemented it via the Solidity language. Here we highlight three additional features to the protocols explained in Section 5.2 and defer the details to Appendix F. *Multiple Deposits:* The `Deposit` function in the contract can be used by the parties to increase their balance for off-chain payments. This function can be invoked any number of times as long as the channel is active (i.e., not closing).

Receipts and Accumulators: As mentioned earlier we added a *receipt* message to efficiently close the channel (i.e., consuming less amount of gas). The receipt object includes an *accumulator* to track the pending promises. Upon claiming a promise, it will be checked against the accumulator to prevent double spending.

Cooperative Closing: When parties submit receipts/promises to close a channel a channel expiry time will be set. As this time can be long, parties can finalize the closing sooner than the channel expiry time by invoking the `Close` function.



(a) Accumulator Insertion (b) Membership Proof

Fig. 7: Performance comparisons between Merkle Tree and RSA accumulators

Table 1: Gas units for invoking PPC contract's functions

Function	Deploy	Deposit	Receipt	Close	Withdraw	HTLC Promise
Gas units	3,243,988	43,010	75,336	44,324	71,572	611,296

Evaluation. We begin by evaluating the gas needed for the deployment of the contract. The PPC contract requires $3,243,988$ gas to be deployed on the Ethereum blockchain. We emphasize that in our implementation we did not aim to optimize gas costs and further optimizations can reduce the gas. However, within its current state, the PPC contract is comparable to other simple payment channel deployments 2M+ and 3M+ gas for Perun [16] and Raiden [3] respectively. The gas usage for functions of PPC contract are reported in Table 1.

PPC can support many general applications and programmable payments. To evaluate its feasibility, we tested it using the simple case of Hash Time Lock Contracts (HTLCs) [30,3,28] which due to space constraints we detail in Appendices F.3 and F.4. In summary, to deploy a HTLC promise on-chain a party will be consuming about 700K gas to resolve it.

We further evaluated the run-time of the protocol by performing a test on ten clients concurrently sending transactions to a single server. Each client sent $1,000$ transactions, which made the server process a total of $10,000$ promises, secret reveals, and receipts. We were able to achieve 110 TPS end-to-end on a commodity machine. We note that the performance can be further improved by optimizing the off-chain code and using a more powerful machine for the server.

References

1. Hash time locked contracts - bitcoin wiki. https://en.bitcoin.it/wiki/Hash_Time_Locked_Contracts, (Accessed on 10/18/2022)
2. Payment channels - bitcoin wiki. https://en.bitcoin.it/wiki/Payment_channels, (Accessed on 10/18/2022)
3. Raiden. https://raiden.network/, (Accessed on 10/18/2022)
4. State channels - ethereum.org. https://ethereum.org/en/developers/docs/scaling/state-channels/, (Accessed on 10/18/2022)
5. Aumayr, L., Maffei, M., Ersoy, O., Erwig, A., Faust, S., Riahi, S., Hostáková, K., Moreno-Sanchez, P.: Bitcoin-compatible virtual channels. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 901–918. IEEE (2021)

6. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: Proceedings 42nd IEEE Symposium on Foundations of Computer Science. pp. 136–145. IEEE (2001)
7. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: Theory of Cryptography Conference. pp. 61–85. Springer (2007)
8. 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 (2021). https://doi.org/10.48550/ARXIV.2109.12194, https://arxiv.org/abs/2109.12194
9. 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 (2021). https://doi.org/10.48550/ARXIV.2109.12194, https://arxiv.org/abs/2109.12194
10. Close, T.: Nitro protocol. Cryptology ePrint Archive (2019)
11. Close, T., Stewart, A.: Forcemove: an n-party state channel protocol. Magmo, White Paper (2018)
12. Coleman, J., Horne, L., Xuanji, L.: Counterfactual: Generalized state channels. Acessed: http://l4. ventures/-papers/statechannels. pdf **4**, 2019 (2018)
13. 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)
14. Dziembowski, S., Eckey, L., Faust, S., Hesse, J., Hostáková, K.: Multi-party virtual state channels. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 625–656. Springer (2019)
15. Dziembowski, S., Eckey, L., Faust, S., Malinowski, D.: Perun: Virtual payment hubs over cryptocurrencies. Cryptology ePrint Archive, Paper 2017/635 (2017), https://eprint.iacr.org/2017/635, https://eprint.iacr.org/2017/635
16. 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)
17. Dziembowski, S., Faust, S., Hostáková, K.: General state channel networks. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 949–966 (2018)
18. Erkan Tairi, Pedro Moreno-Sanchez, M.M.: a^2l: Anonymous atomic locks for scalability and interoperability in payment channel hubs. https://eprint.iacr.org/2019/589
19. Goldreich, O.: Foundations of cryptography: volume 2, basic applications. Cambridge university press (2009)
20. Gudgeon, L., Moreno-Sanchez, P., Roos, S., McCorry, P., Gervais, A.: Sok: Layer-two blockchain protocols. In: International Conference on Financial Cryptography and Data Security. pp. 201–226. Springer (2020)
21. Hearn, M., Corallo, M.: Connection Bloom filtering (2012), https://github.com/\bitcoin/bips/blob/master/bip-0037.mediawiki
22. Khalil, R., Gervais, A.: Nocust-a non-custodial 2nd-layer financial intermediary. (2018)
23. 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
24. Lind, J., Naor, O., Eyal, I., Kelbert, F., Sirer, E.G., Pietzuch, P.R.: Teechain: a secure payment network with asynchronous blockchain access. In: Brecht, T., Williamson, C. (eds.) Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019. pp. 63–79. ACM (2019)
25. Malavolta, G., Moreno-Sanchez, P., Kate, A., Maffei, M., Ravi, S.: Concurrency and privacy with payment-channel networks. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 455–471 (2017)
26. Malavolta, G., Moreno-Sanchez, P., Schneidewind, C., Kate, A., Maffei, M.: Anonymous multi-hop locks for blockchain scalability and interoperability. In: NDSS (2019)
27. Matetic, S., Wüst, K., Schneider, M., Kostiainen, K., Karame, G., Capkun, S.: BITE: Bitcoin lightweight client privacy using trusted execution. In: 28th USENIX Security Symposium (2019), https://www.usenix.org/conference/usenixsecurity19/presentation/matetic

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. Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments. http://lightning.network/lightning-network-paper.pdf (2016), (Accessed on 10/18/2022)
31. Roos, S., Moreno-Sanchez, P., Kate, A., Goldberg, I.: Settling payments fast and private: Efficient decentralized routing for path-based transactions. arXiv preprint arXiv:1709.05748 (2017)
32. Stefano Martinazzi, A.F.: The evolving topology of the lightning network: Centralization, efficiency, robustness, synchronization, and anonymity. Journal PLOS One
33. Todd, P.: [bitcoin-development] near-zero fee transactions with hub-and-spoke micropayments. https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2014-December/006988.html (2014), (Accessed on 10/19/2022)

Disclaimer

A Supplementary Material for Notations and Models

A.1 Definitions of Contracts, Channels

Contracts. A *contract instance* consists of two attributes: *contract storage* (accessed by key $\mathsf{storage}$) and *contract code* (accessed by key code). Contract storage σ is an attribute tuple containing at least the following attributes: (1) $\sigma.user_L$ and $\sigma.user_R$ denoting the two involved users; (2) $\sigma.locked \in \mathbb{R}_{\geq 0}$ denoting the total number of coins locked in the contract; (3) $\sigma.cash : \{\sigma.user_L, \sigma.user_R\} \to \mathbb{R}$ denoting the coins available to each user. A contract code is a tuple $C := (\Lambda, g_1, \ldots, g_r, f_1, \ldots, f_s)$ where (1) Λ denotes the admissible contract storage; (2) each g denotes a constructor function that takes (P, t, z) as inputs and provides as output an admissible contract storage or $\perp$ representing failure to construct, where P is the caller, t is the current time stamp and z denotes the auxiliary inputs; and (3) each f denotes an execution function that takes (σ, P, t, z) as inputs and provides as output an admissible contract storage (could be the same as the input) and an output message m, where $m = \perp$ represents failure.

Programmable Payment Channel. A programmable payment channel is an attribute tuple $\gamma := (\gamma.id, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.cash, \gamma.\mathsf{pspace}, \gamma.duration)$ where (1) $\gamma.id \in \{0,1\}^*$ is the identifier for the PPC instance; (2) $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ denote the two involved parties; (3) $\gamma.cash : \{\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}\} \to \mathbb{R}_{\geq 0}$ denotes the amount of money deposited by each participant; (4) $\gamma.\mathsf{pspace}$ stores all the promise instances opened in the channel–it takes a promise identifier *pid* and maps it to a promise instance; and (5) $\gamma.duration \geq 0$ denotes the time delay to closing a channel, when there exists a promise that has not yet been finalized.

A.2 Global Universal Composable Framework

UC models the execution of protocols as interactions of probabilistic polynomial-time (PPT) *Iterative Turing Machines* (ITMs) and attempts to argue that interactions between ITMs in the "real" world (by virtue of our defined real world protocols) are indistinguishable from the interactions between the ITMs in an "ideal" world (where whatever security property we are after would be satisfied).

Formally, let π be a protocol working in the $\mathcal{G}$-hybrid model with access to the global ledger $\hat{\mathcal{L}}(\Delta)$ (specified later). The output of an environment $\mathcal{E}$ interacting with the protocol π in the presence of an adversary $\mathcal{A}$ on input 1^λ and auxiliary input z is denoted as $\mathrm{EXEC}^{\hat{\mathcal{L}}(\Delta),\mathcal{G}}_{\pi,\mathcal{A},\mathcal{E}}(1^\lambda, z)$. We define another trivial protocol with ideal functionality $\mathcal{F}$, *dummy parties* and a simulator $\mathcal{S}$. We denote the output of the environment (similar to the above) in this scenario as $\mathrm{IDEAL}^{\hat{\mathcal{L}}(\Delta)}_{\mathcal{F},\mathcal{S},\mathcal{E}}(1^\lambda, z)$.

Definition 1. *We say that a protocol π working in a $\mathcal{G}$-hybrid model UC-emulates an ideal functionality $\mathcal{F}$ with respect to a global ledger $\hat{\mathcal{L}}(\Delta)$ i.f.f. for any PPT adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that for any environment $\mathcal{E}$ we have*

$$\{\mathrm{EXEC}^{\hat{\mathcal{L}}(\Delta),\mathcal{G}}_{\pi,\mathcal{A},\mathcal{E}}(1^\lambda, z)\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}} \overset{c}{\approx} [5]\{\mathrm{IDEAL}^{\hat{\mathcal{L}}(\Delta)}_{\mathcal{F},\mathcal{S},\mathcal{E}}(1^\lambda, z)\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}}$$

B Simplifications

In this section, we provide and justify the full simplifications we made in Section 5 for better presentation:

[5] "$\overset{c}{\approx}$" denotes computational indistinguishability of distribution ensembles, see [19].

- We omit session and sub-session identifiers $sid, ssid$.
- We assume the existence of a PKI. Note that this also implies the existence of a complete peer-to-peer authenticated communication network.
- We combine pairwise channel contracts into one single large hybrid functionality, i.e., in the implementation, this single hybrid can be separated into several pairwise contracts. This is permissible since the entire Layer-1 network can be modeled as a large publicly available trusted virtual machine. This combined hybrid will maintain a large set Γ to save all available PPC channels. Formally, the identifier id of each channel γ reflects the corresponding contract address.
- We work in the *random oracle* model. In particular, we use random oracles to capture the CREATE2 opcode in Ethereum. Basically, every promise instance will be associated with an identifier pid determined by random oracle applied to the the following inputs: (1) creator channel identifier id; (2) payer P; (3) promise code C; and (4) arguments for the constructor function z. Note that the arguments specified above are in line with the specification of CREATE2. We embed the salt value into the constructor arguments, which captures the fact the there might be several promise instances created by the same constructor (with different salts). Similar to the channel identifier, pid formally reflects the contract address of the promise contract created by the channel contract. Also note that the random oracle ensures that it is statistically hard to create a second contract whose address is also pid.
- Whenever we say we put a promise instance inside some channel contract's $\gamma.\mathsf{pspace}$, it means that this promise is deployed as a contract on Layer-1. We further combine processes to (1) create a promise using the sender's signature on the channel contract; and (2) bypass it to the latest state using both parties' signatures on the promise contract, into one process that saves the latest state into $\gamma.\mathsf{pspace}$.
- We allow parties to "register" promises *in parallel*. We will further discuss this in Appendix F.1 where we instantiate this parallelism using an accumulator.

Procedure Register(P, id, pid)

Denote $\mathcal{G}_{PPC} := \mathcal{G}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$.

Party P

1. Let $\gamma^P := \Gamma^P(id), \nu^P := \gamma^P.\mathsf{pspace}(pid), (s_P, s_Q, \mathsf{version}) := \Gamma^P_{aux}(id, pid)$, and let t_0 be the current round. Consider the following two scenarios:
   - If $\mathsf{version} = 0$ (i.e., this is an initial promise instance), fetch the constructing tuple (id, pid, C, z, A, s_A) from P's local memory, then send $(\texttt{instance-construct}, id, C, z, A, s_A) \overset{t_0}{\hookrightarrow} \mathcal{G}_{PPC}$.
   - If $\mathsf{version} \neq 0$ (i.e., this is a non-initial promise instance), send $(\texttt{instance-register}, id, pid, \nu^P, \mathsf{version}, s_P, s_Q) \overset{t_0}{\hookrightarrow} \mathcal{G}_{PPC}$.

Party Q upon $(\texttt{instance-registering}, id, pid) \overset{t_1}{\hookleftarrow} \mathcal{G}_{PPC}$

2. Let $\gamma^Q := \Gamma^Q(id), \nu^Q := \gamma^Q.\mathsf{pspace}(pid), (s_Q, s_P, \mathsf{version}) := \Gamma^Q_{aux}(id, pid)$. Consider the following two scenarios:
   - If $\nu^Q = \perp$ (i.e., P wants to initialize a promise on-chain), send $(\texttt{instance-register}, id, pid, \perp, -1, \perp, \perp) \overset{t_1}{\hookrightarrow} \mathcal{G}_{PPC}$.
   - If $\mathsf{version} = 0$ (i.e., this is an initial promise instance), fetch the constructing tuple (id, pid, C, z, A, s_A) from Q's local memory, then send $(\texttt{instance-construct}, id, C, z, A, s_A) \overset{t_1}{\hookrightarrow} \mathcal{G}_{PPC}$.
   - If $\mathsf{version} \neq 0$ (i.e., this is a non-initial promise instance), send $(\texttt{instance-register}, id, pid, \nu^Q, \mathsf{version}, s_Q, s_P) \overset{t_1}{\hookrightarrow} \mathcal{G}_{PPC}$.
3. Goto step 5.

Back to party P

4. If not $(\texttt{instance-registered}, id, pid, \nu) \overset{t_2 \leq t_0+2\Delta}{\hookleftarrow} \mathcal{G}_{PPC}$, then send $(\texttt{finalize-register}, id, pid) \overset{t_3=t_0+2\Delta+1}{\hookrightarrow} \mathcal{G}_{PPC}$.

For both parties T

5. Upon $(\texttt{instance-registered}, id, pid, \nu) \hookleftarrow \mathcal{G}_{PPC}$, mark (id, pid) as registered in Γ^T_{aux}. Set $\Gamma^T(id).\mathsf{pspace}(pid) := \nu$. Stop.

Functionality $\mathcal{G}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$: Promise instance registration

Let RO be a random oracle.

Upon $(\texttt{instance-construct}, id, C, z, A, s_A) \overset{t_0}{\hookleftarrow} P$, let $\gamma := \Gamma(id)$ and do:

1. Stop if one of the following conditions holds: $\gamma = \perp$; $P \notin \gamma.\mathsf{endusers}$; $A \notin \gamma.\mathsf{endusers}$; $C \notin PS$; $\gamma.\mathsf{pspace}(pid) \neq \perp$; $\mathsf{Vfy}_{pk_A}(id, C, z, A; s_A) = 0$.
2. Let $\nu := \perp$ and $pid := RO(id, A, C, z)$. Set $\nu.\mathsf{code} := C$. Let $\sigma := C.\mathsf{construct}(A, t_0, z)$, stop if $\sigma = \perp$ or $\sigma.\mathsf{payer} \neq A$. Else set $\nu.\mathsf{storage} := \sigma$.
3. Let $Q := \gamma.\mathsf{otherparty}(P)$ and consider the following three cases:
   - If the functionality's memory contains a tuple $(P, id, pid, \hat{\nu}, \hat{t_0}, \hat{\mathsf{version}})$, then stop.
   - If the functionality's memory contains a tuple $(Q, id, pid, \hat{\nu}, \hat{t_0}, \hat{\mathsf{version}})$, then set $\tilde{\nu} := \hat{\nu}$. Within Δ rounds, send $(\texttt{instance-registered}, id, pid, \tilde{\nu}) \overset{t_1 \leq t_0+\Delta}{\hookrightarrow} \gamma.\mathsf{endusers}$, set $\Gamma(id).\mathsf{pspace}(pid) := \tilde{\nu}$ and erase $(Q, id, pid, \hat{\nu}, \hat{t_0}, \hat{\mathsf{version}})$ from the memory.
   - Else save $(P, id, pid, \nu, t_0, 0)$ to the memory and send $(\texttt{instance-registering}, id, pid) \overset{t_1 \leq t_0+\Delta}{\hookrightarrow} Q$.

Upon $(\texttt{instance-register}, id, pid, \nu, \mathsf{version}, s_P, s_Q) \overset{t_0}{\hookleftarrow} P$, let $\gamma := \Gamma(id)$ and do:

1. Stop if one of the following conditions holds: $\gamma = \perp$; $P \notin \gamma.\mathsf{endusers}$; $\gamma.\mathsf{pspace}(pid) \neq \perp$.
2. Let $Q := \gamma.\mathsf{otherparty}(P)$. If $\mathsf{version} = -1$ and $\nu = \perp$ and the functionality's memory contains a tuple $(Q, id, pid, \cdots)$, goto step 4.
3. Stop if one of the following conditions holds: $\mathsf{Vfy}_{pk_P}(id, pid, \nu, \mathsf{version}; s_P) = false$; $\mathsf{Vfy}_{pk_Q}(id, pid, \nu, \mathsf{version}; s_Q) = false$; $\nu.\mathsf{code} \notin PS$; $\nu.\mathsf{storage} \notin \nu.\mathsf{code}.\Lambda$.
4. Consider the following three cases:
   - If the functionality's memory contains a tuple $(P, id, pid, \hat{\nu}, \hat{t_0}, \hat{\mathsf{version}})$, then stop.
   - If the functionality's memory contains a tuple $(Q, id, pid, \hat{\nu}, \hat{t_0}, \hat{\mathsf{version}})$, then compare the version number, i.e., $\tilde{\nu} := \mathsf{version} > \hat{\mathsf{version}} \ ? \ \nu : \hat{\nu}$. Within Δ rounds, send $(\texttt{instance-registered}, id, pid, \tilde{\nu}) \overset{t_1 \leq t_0+\Delta}{\hookrightarrow} \gamma.\mathsf{endusers}$, set $\Gamma(id).\mathsf{pspace}(pid) := \tilde{\nu}$ and erase $(Q, id, pid, \hat{\nu}, \hat{t_0}, \hat{\mathsf{version}})$ from the memory.
   - Else save $(P, id, pid, \nu, t_0, \mathsf{version})$ to the memory and send $(\texttt{instance-registering}, id, pid) \overset{t_1 \leq t_0+\Delta}{\hookrightarrow} Q$.

Upon $(\texttt{finalize-register}, id, pid) \overset{t_2}{\hookleftarrow} P$, let $\gamma := \Gamma(id)$ and do:

1. Stop if $\gamma = \perp$ or $P \notin \gamma.\mathsf{endusers}$.
2. If the functionality's memory contains a tuple $(P, id, pid, \hat{\nu}, \hat{t_0}, \hat{\mathsf{version}})$ such that $t_2 - \hat{t_0} \geq 2\Delta$, then set $\Gamma(id).\mathsf{pspace}(pid) := \hat{\nu}$, send $(\texttt{instance-registered}, id, pid, \hat{\nu}) \overset{t_3 \leq t_2+\Delta}{\hookrightarrow} \gamma.\mathsf{endusers}$, and erase the tuple.

Fig. 8: The sub-procedure Register to register a pormise instance on-chain.

C Formal PPC Protocols

In this section, we fully specify the PPC protocols described in Section 5.2. The protocols are formalized in the UC framework [6], which UC-emulate $\mathcal{F}_{PPC}$ (see Figure 4).

The special sub-procedure Register, which is used to deploy a promise instance on-chain, is specified in Figure 8. The online contracts functionality $\mathcal{G}_{PPC}$ will have 3 interfaces to handle this sub-procedure:

- `instance-construct`: This interface is used to initiate a promise instance. It requires an explicit initiator with the corresponding signature. Note that this will be viewed as a party submitting a promise instance of version $= 0$.
- `instance-register`: This interface is used to submit an agreed upon promise storage. It requires a promise instance specification with both parties' signatures. Intuitively, the honest party will always submit the largest version. Note that we have a special case for version $= -1$. This is used to capture the case where a corrupted sender could initiate an instance directly on-chain and the honest receiver will trivially accept it. Since RO is a random oracle, it is infeasible to create two different instances that have the same *pid*.
- `finalize-register`: This interface is used to enable the honest party to deploy a promise even when another party does not send any valid storage to the PPC contract. Informally, this means that the party can convince the contract that the other party aborted after a 2Δ dispute period.

Protocol $\Pi(PS)$: Create a programmable payment channel

Denote $\mathcal{G}_{PPC} := \mathcal{G}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$.

Party P upon $(\texttt{create}, \gamma) \overset{t_0}{\hookleftarrow} \mathcal{E}$

1. Send $(\texttt{construct}, \gamma) \overset{t_0}{\hookrightarrow} \mathcal{G}_{PPC}$ and wait.

Party Q upon $(\texttt{create}, \gamma) \overset{t_0}{\hookleftarrow} \mathcal{E}$

2. If $(\texttt{initializing}, \gamma) \overset{t_1 \leq t_0+\Delta}{\hookleftarrow} \mathcal{G}_{PPC}$, send $(\texttt{confirm}, \gamma) \overset{t_1}{\hookrightarrow} \mathcal{G}_{PPC}$ and wait. Else stop.
3. If $(\texttt{initialized}, \gamma) \overset{t_2 \leq t_0+2\Delta}{\hookleftarrow} \mathcal{G}_{PPC}$, then set $\Gamma^Q(\gamma.id) := \gamma$, output $(\texttt{created}, \gamma) \overset{t_2}{\hookrightarrow} \mathcal{E}$.

Back to party P

4. If $(\texttt{initialized}, \gamma) \overset{t_2 \leq t_0+2\Delta}{\hookleftarrow} \mathcal{G}_{PPC}$, then set $\Gamma^P(\gamma.id) := \gamma$, output $(\texttt{created}, \gamma) \overset{t_2}{\hookrightarrow} \mathcal{E}$ and stop. Otherwise, execute next step.
5. If $(\texttt{refund}, \gamma) \overset{t_3 > t_0+2\Delta}{\hookleftarrow} \mathcal{E}$, send $(\texttt{refund}, \gamma) \overset{t_3}{\hookrightarrow} \mathcal{G}_{PPC}$ and stop.

Functionality/Contract $\mathcal{G}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$

Upon $(\texttt{construct}, \gamma) \overset{t_0}{\hookleftarrow} P$**:**

1. Let $Q := \gamma.\mathsf{Bob}$, stop if one of the following conditions holds: there already exists a channel γ' such that $\gamma.\mathsf{id} = \gamma'.\mathsf{id}$; $\gamma.\mathsf{Alice} \neq P$; $\gamma.\mathsf{cash}(P) < 0$ or $\gamma.\mathsf{cash}(Q) < 0$; $\gamma.\mathsf{pspace} \neq \{\}$; $\gamma.duration < 0$.
2. Within Δ rounds remove $\gamma.\mathsf{cash}(P)$ coins from P's account on the ledger $\hat{\mathcal{L}}$. If it is impossible due to insufficient funds, then stop. Else $(\texttt{initializing}, \gamma) \hookrightarrow Q$ and store the pair $tamp := (t_0, \gamma)$.

Upon $(\texttt{confirm}, \gamma) \overset{t_1}{\hookleftarrow} Q$**:**

1. Stop if one of the following conditions holds: there is no pair $tamp := (t_0, \gamma)$ in the storage; $(t_1 - t_0) > \Delta$; $\gamma.\mathsf{Bob} \neq Q$.
2. Within Δ rounds remove $\gamma.\mathsf{cash}(Q)$ coins from Q's account on the ledger $\hat{\mathcal{L}}$. If it is impossible due to insufficient funds, then stop. Else set $\Gamma(\gamma.\mathsf{id}) := \gamma$ and delete $tamp$ from the memory. Thereafter send $(\texttt{initialized}, \gamma) \hookrightarrow \gamma.\mathsf{endusers}$.

Upon $(\texttt{refund}) \overset{t_2}{\hookleftarrow} P$**:**

1. Stop if one of the following conditions holds: there is no pair $tamp := (t_0, \gamma)$ in the storage; $(t_2 - t_0) \leq 2\Delta$; $P \neq \gamma.\mathsf{Alice}$.
2. Within Δ rounds add $\gamma.\mathsf{cash}(\gamma.\mathsf{Alice})$ coins to $\gamma.\mathsf{Alice}$'s account in ledger $\hat{\mathcal{L}}$ and delete $tamp$ from the storage.

Protocol $\Pi(PS)$: Create an initial promise instance

Denote $\mathcal{G}_{PPC} := \mathcal{G}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$.

Party P upon $(\texttt{create-instance}, id, C, z) \overset{t_0}{\hookleftarrow} \mathcal{E}$

1. Calculate $pid := RO(id, P, C, z)$.
2. Stop if one of the following conditions holds: $\Gamma^P(id) = \bot$; $P \notin \Gamma^P(id).\mathsf{endusers}$; $\Gamma^P(id).\mathsf{pspace}(pid) \neq \bot$; $C \notin PS$. Else let $\gamma := \Gamma^P(id)$.
3. Let $\nu := \bot$ and $\sigma := C.\mathsf{Construct}(P, t_0, z)$. Stop if $\sigma = \bot$. Else set $\nu.\mathsf{code} := C$ and $\nu.\mathsf{storage} := \sigma$. Set $\Gamma^P(id).\mathsf{pspace}(pid) := \nu$ and set $\Gamma^P_{aux}(id, pid) := (\bot, \bot, 0)$.
4. Compute $s_P := \mathsf{Sign}_{sk_P}(id, C, z, P)$, save (id, pid, C, z, P, s_P) and send $(\texttt{create-instance}, id, C, z, s_P) \hookrightarrow Q$.
5. Output $(\texttt{instance-created}, id, pid, \nu) \overset{t_0+1}{\hookrightarrow} \mathcal{E}$.

Party Q upon $(\texttt{create-instance}, id, C, z, s_P) \overset{t_1}{\hookleftarrow} P$

6. Calculate $pid := RO(id, P, C, z)$
7. Stop if one of the following conditions holds: $\Gamma^Q(id) = \bot$; $P \notin \Gamma^Q(id).\mathsf{endusers}$; $\Gamma^Q(id).\mathsf{pspace}(pid) \neq \bot$; $C \notin PS$. Else let $\gamma := \Gamma^Q(id)$.
8. Let $\sigma := C.\mathsf{Construct}(P, t_1 - 1, z)$. Stop if $\sigma = \bot$. Let $\nu := \bot$. Set $\nu.\mathsf{code} := C$ and $\nu.\mathsf{storage} := \sigma$. Stop if $\mathsf{Vfy}_{pk_P}(id, C, z, P; s_P) \neq 1$.
9. Set $\Gamma^Q(id).\mathsf{pspace}(pid) := \nu$ and set $\Gamma^Q_{aux}(id, pid) := (\bot, \bot, 0)$. Save (id, pid, C, z, P, s_P). Output $(\texttt{instance-created}, id, pid, \nu) \overset{t_1}{\hookrightarrow} \mathcal{E}$.

Protocol $\Pi(PS)$: Promise instance execution

Denote $\mathcal{G}_{PPC} := \mathcal{G}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$.

Party P upon $(\texttt{execute}, id, pid, f, z) \overset{t_0}{\hookleftarrow} \mathcal{E}$

1. Stop if $\Gamma^P(id) = \bot$, else let $\gamma^P := \Gamma^P(id)$. Stop if $P \notin \gamma^P.\mathsf{endusers}$ or $\gamma^P(pid) = \bot$, else let $\nu^P := \gamma^P.\mathsf{pspace}(pid)$. Stop if $f \notin \nu^P.\mathsf{code}$, else let $C^P := \nu^P.\mathsf{code}$ and $\sigma^P := \nu^P.\mathsf{storage}$. Let $Q := \gamma^P.\mathsf{otherparty}(P)$. Let $(\cdot, \cdot, \mathsf{version}^P) := \Gamma^P_{aux}(id, pid)$.

2. Set $t_1 := t_0 + x$, where x is the smallest offset such that $t_1 \equiv 1 (\text{mod } 4)$ if $P = \gamma^P.\mathsf{Alice}$ and $t_1 \equiv 3 (\text{mod } 4)$ if $P = \gamma^P.\mathsf{Bob}$.
3. If (id, pid) is marked as registered in Γ^P_{aux}, goto step 12 at round t_1.
4. Compute $(\tilde{\sigma}, m) := f(\sigma^P, P, t_0, z)$. Stop if $m = \perp$. Otherwise, set $\tilde{\mathsf{version}} := \mathsf{version}^P + 1$. Let $\tilde{\nu} := \perp$. Set $\tilde{\nu}.\mathsf{code} := C^P$ and $\tilde{\nu}.\mathsf{storage} := \tilde{\sigma}$. Compute $s_P := \mathsf{Sign}_{sk_P}(id, pid, \tilde{\nu}, \tilde{\mathsf{version}})$. Send $(\texttt{peaceful-request}, id, pid, f, z, s_P, t_0) \overset{t_1}{\hookrightarrow} Q$. Goto step 11.

Party Q upon $(\texttt{peaceful-request}, id, pid, f, z, s_P, t_0) \overset{t_Q}{\hookleftarrow} P$

5. Stop if $\Gamma^Q(id) = \perp$, else let $\gamma^Q := \Gamma^Q(id)$. Stop if $Q \notin \gamma^Q.\mathsf{endusers}$ or $P \notin \gamma^Q.\mathsf{endusers}$ or $\gamma^Q(pid) = \perp$, else let $\nu^Q := \gamma^Q.\mathsf{pspace}(pid)$. Stop if $f \notin \nu^Q.\mathsf{code}$, else let $C^Q := \nu^Q.\mathsf{code}$ and $\sigma^Q := \nu^Q.\mathsf{storage}$. Let $(\cdot, \cdot, \mathsf{version}^Q) := \Gamma^Q_{aux}(id, pid)$.
6. Stop if "$P = \gamma^Q.\mathsf{Alice}$ and $t_Q \not\equiv 2 (\text{mod } 4)$" or "$P = \gamma^Q.\mathsf{Bob}$ and $t_Q \not\equiv 0 (\text{mod } 4)$".
7. Stop if $t_0 \notin [t_Q - 4, t_Q - 1]$.
8. If (id, pid) is not marked as registered in Γ^Q_{aux}, do:
   (a) Compute $(\tilde{\sigma}, m) := f(\sigma^Q, P, t_0, z)$. Stop if $m = \perp$.
   (b) Set $\tilde{\mathsf{version}} := \mathsf{version}^Q + 1$. Let $\tilde{\nu} := \perp$. Set $\tilde{\nu}.\mathsf{code} := C^Q$ and $\tilde{\nu}.\mathsf{storage} := \tilde{\sigma}$.
   (c) If $\mathsf{Vfy}_{pk_P}(id, pid, \tilde{\nu}, \tilde{\mathsf{version}}; s_P) \neq 1$, then stop.
   (d) Compute $s_Q := \mathsf{Sign}_{sk_Q}(id, pid, \tilde{\nu}, \tilde{\mathsf{version}})$. Set $\Gamma^Q(id)$. $\mathsf{pspace}(pid) := \tilde{\nu}$ and $\Gamma^Q_{aux}(id, pid) := (s_Q, s_P, \tilde{\mathsf{version}})$. Send $(\texttt{peaceful-confirm}, id, pid, s_Q) \overset{t_Q}{\hookrightarrow} P$.
   (e) Send $(\texttt{executed}, id, pid, P, f, t_0, z, \tilde{\nu}) \overset{t_Q+1}{\hookrightarrow} \mathcal{E}$.

Back to party P

11. Distinguish the following two cases:
   – If $(\texttt{peaceful-confirm}, id, pid, s_Q) \overset{t_2 = t_1 + 2}{\hookleftarrow} Q$ such that $\mathsf{Vfy}_{pk_Q}(id, pid, \tilde{\nu}, \tilde{\mathsf{version}}; s_Q) = 1$, set $\Gamma^P(id)$. $\mathsf{pspace}(pid) := \tilde{\nu}$ and $\Gamma^P_{aux}(id, pid) := (s_P, s_Q, \tilde{\mathsf{version}})$.
   – Otherwise (i.e., Q aborts or replies with invalid signature). Execute $\mathsf{Register}(P, id, pid)$ to mark (id, pid) as registered in Γ^P_{aux}. Once the register procedure is executed (in round $t_3 \leq t_0 + 3\Delta + 5$), check if $\Gamma^P(id).\mathsf{pspace}(pid) = \tilde{\nu}$. If so (i.e., Q agrees the execution by registering the newest version onchain), output $(\texttt{executed}, id, pid, P, f, t_0, z, \tilde{\nu}) \overset{t_3}{\hookrightarrow} \mathcal{E}$ and stop.
12. Send $(\texttt{instance-execute}, id, pid, f, z) \hookrightarrow \mathcal{G}_{PPC}$.

For both parties T

13. If $(\texttt{executed-onchain}, id, pid, \mathsf{Caller}, f, t, z, \hat{\nu}) \overset{t_4 \leq t_0 + 4\Delta + 5}{\hookleftarrow} \mathcal{G}_{PPC}$, set $\Gamma^T(id)$. $\mathsf{pspace}(pid) := \hat{\nu}$ and output $(\texttt{executed}, id, pid, \mathsf{Caller}, f, t, z, \hat{\nu}) \overset{t_4}{\hookrightarrow} \mathcal{E}$.

Functionality/Contract $\mathcal{G}^{\hat{\mathcal{L}}(\Delta)}_{PPC}(PS)$

Upon $(\texttt{instance-execute}, id, pid, f, z) \overset{t}{\hookleftarrow} P$, proceed as follows:

1. Let $\gamma := \Gamma(id)$. Stop if $\gamma = \bot$.
2. Set $\nu := \gamma.\mathsf{pspace}(pid)$ and $\sigma := \nu.\mathsf{storage}$. Stop if one of the following conditions holds: $P \notin \gamma.\mathsf{endusers}$; $\nu = \bot$; $f \notin \nu.\mathsf{code}$.
3. Within Δ rounds, i.e., $t_1 \leq t + \Delta$. Compute $(\hat{\sigma}, m) := f(\sigma, P, t_1, z)$. Stop if $m = \bot$.
4. Set $\Gamma(id).\mathsf{pspace}(pid).\mathsf{storage} := \hat{\sigma}$ and send $(\texttt{executed- onchain}, \texttt{id}, \texttt{pid}, \texttt{P}, \texttt{f}, \texttt{t}, z, \hat{\nu}) \overset{t_1}{\hookrightarrow} \gamma.\mathsf{endusers}$.

Protocol $\Pi(PS)$: Close a programmable payment channel

Denote $\mathcal{G}_{PPC} := \mathcal{G}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$.

Party P upon $(\texttt{close}, id) \overset{t_0}{\hookleftarrow} \mathcal{E}$

1. Stop if $\Gamma^P(id) = \bot$, else let $\gamma^P := \Gamma^P(id)$. Stop if $P \notin \gamma^P.\mathsf{endusers}$. For each $\gamma^P.\mathsf{pspace}(pid) \neq \bot$ and (id, pid) is not marked as registered, execute (in parallel) $\mathsf{Register}(P, id, pid)$ immediately. Then send $(\texttt{contract-close}, id) \overset{t_1 \leq t_0 + 3\Delta}{\hookrightarrow} \mathcal{G}_{PPC}$.

Party Q upon $(\texttt{contract-closing}, id) \overset{t_2 \leq t_0 + 4\Delta}{\hookleftarrow} \mathcal{G}_{PPC}$

2. Let $\gamma^Q := \Gamma^Q(id)$. For each $\gamma^Q.\mathsf{pspace}(pid) \neq \bot$ and (id, pid) is not marked as registered, execute (in parallel) $\mathsf{Register}(Q, id, pid)$ immediately.

For both parties T

3. If $(\texttt{contract-close}, id) \overset{t_3 \leq t_0 + 8\Delta + \gamma^T.duration}{\hookleftarrow} \mathcal{G}_{PPC}$, output $(\texttt{closed}, id) \overset{t_3}{\hookrightarrow} \mathcal{E}$.

Functionality/Contract $\mathcal{G}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$

Upon $(\texttt{contract-close}, id) \overset{t_0}{\hookleftarrow} P$, let $\gamma := \Gamma(id)$ and proceed as follows:

1. Stop if $\gamma = \bot$ or $P \notin \gamma.\mathsf{endusers}$.
2. Block all the messages in the future related to close channel id.
3. Within Δ rounds send $(\texttt{contract-closing}, id) \overset{t_1 \leq t_0 + \Delta}{\hookrightarrow} \gamma.\mathsf{otherparty}(P)$.
4. Within another 3Δ rounds. If there a exists $pid \in \{0,1\}^*$ such that $\gamma.\mathsf{pspace}(pid) \neq \bot$ and $\gamma.\mathsf{pspace}(pid).\mathsf{storage}.ended = 0$, wait for next $\gamma.duration$ rounds.
5. At round $t_2 \leq t_0 + 4\Delta + \gamma.duration$:
   (a) Set $total := \gamma.\mathsf{cash}(\gamma.\mathsf{Alice}) + \gamma.\mathsf{cash}(\gamma.\mathsf{Bob})$.
   (b) Set $credit_A := \sum_{\gamma.\mathsf{pspace}(pid).\mathsf{storage.payer} = \gamma.\mathsf{Bob}} (\gamma.\mathsf{pspace}\ (pid).\mathsf{storage}.final)$.
   (c) Set $credit_B := \sum_{\gamma.\mathsf{pspace}(pid).\mathsf{storage.payer} = \gamma.\mathsf{Alice}} (\gamma.\mathsf{pspace}\ (pid).\mathsf{storage}.final)$.
   (d) Within Δ rounds, add $min\{total, max\{0, \gamma.\mathsf{cash}(\gamma.\mathsf{Alice}) + credit_A - credit_B\}\}$ coins to $\gamma.\mathsf{Alice}$'s account and $min\{total, max\{0, \gamma.\mathsf{cash}(\gamma.\mathsf{Bob}) + credit_B - credit_A\}\}$ coins to $\gamma.\mathsf{Bob}$'s account.
   (e) Send $(\texttt{contract-close}, \texttt{id}) \hookrightarrow \gamma.\mathsf{endusers}$.

D Security Proofs

In this section, we formally prove our theorems.

Theorem 1 (Main). *Suppose the underlying signature scheme is existentially unforgeable against chosen message attacks. The protocol $\pi(PS)$ working in $\mathcal{G}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$-hybrid model emulates the ideal funcitonlity*

$\mathcal{F}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$ against any environment $\mathcal{E}$ in the random oracle model for every set of promise codes PS and every $\Delta \in \mathbb{N}$.

Proof. We follow the framework of [17]. We will show that for any set of promise codes PS, $\pi(PS)$ UC-emulates the ideal functionality $\mathcal{F}_{PPC}^{\hat{\mathcal{L}}(\Delta)}$ in the $\mathcal{G}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$-hybrid model assuming random oracles. In other words, for any PPT adversary $\mathcal{A}$, we construct a simulator Sim that operates in the $\mathcal{G}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$-hybrid model and simulates the $\mathcal{F}_{PPC}^{\hat{\mathcal{L}}(\Delta)}$-hybrid world to any environment $\mathcal{E}$.

As in [17], since registration of a contract instance is defined as a separate procedure that can be called by parties of the protocol $\pi(PS)$, we define a "subsimulator" $\mathsf{SimRegister}(P, id, pid)$ which can be called as a procedure by the simulator Sim.

The technical details and approach to designing the simulator follow standard techniques (e.g., [17]), and hence we omit further description here due to lack of space. □

Simulator Sim: Create a programmable payment channel

Denote $\mathcal{F}_{PPC} := \mathcal{F}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$.
Let RO be a random oracle.

P is honest and Q is corrupt

Upon $(\texttt{create}, P, \gamma) \overset{t_1 \le t_0+\Delta}{\hookleftarrow} \mathcal{F}_{PPC}$ (the delay is set by simulator as real adversary):

1. Send $(\texttt{initializing}, \gamma) \overset{t_1}{\hookrightarrow} Q$.
2. If $(\texttt{confirm}, \gamma) \overset{t_1}{\hookleftarrow} Q$, then send $(\texttt{create}, \gamma) \overset{t_1}{\hookrightarrow} \mathcal{F}_{PPC}$ on behalf of Q.
3. If $(\texttt{created}, \gamma) \overset{t_2 \le t_1+\Delta}{\hookleftarrow} \mathcal{F}_{PPC}$, send $(\texttt{initialized}, \gamma) \overset{t_2}{\hookrightarrow} Q$ and set $\Gamma^P(id) := \gamma$, $\Gamma(id) := \gamma$.

P is corrupt and Q is honest

Upon $(\texttt{construct}, \gamma) \overset{t_0}{\hookleftarrow} P$:

1. Stop if one of the following conditions holds: there already exists a programmable payment channel γ' such that $\gamma.\mathsf{id} = \gamma'.\mathsf{id}$; $\gamma.\mathsf{Alice} \neq P$ or $\gamma.\mathsf{Bob} \neq Q$; $\gamma.\mathsf{cash}(P) < 0$ or $\gamma.\mathsf{cash}(Q) < 0$; $\gamma.\mathsf{pspace} \neq \{\}$; $\gamma.duration < 0$.
2. Send $(\texttt{create}, \gamma) \overset{t_0}{\hookrightarrow} \mathcal{F}_{PPC}$ on behalf of P.
3. Distinguish the following two situations:
   - If $(\texttt{created}, \gamma) \overset{t_1 \le t_0+2\Delta}{\hookleftarrow} \mathcal{F}_{PPC}$, send $(\texttt{initialized}, \gamma) \overset{t_1}{\hookrightarrow} P$. Set $\Gamma^Q(id) := \gamma$, $\Gamma(id) := \gamma$ and stop.
   - Else if $(\texttt{refund}, \gamma) \overset{t_2 > t_0+2\Delta}{\hookleftarrow} P$, send $(\texttt{refund}, \gamma) \overset{t_2}{\hookrightarrow} \mathcal{F}_{PPC}$.

Sub-simulator $\mathsf{SimRegister}(P, id, pid)$

Denote $\mathcal{F}_{PPC} := \mathcal{F}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$.
Let RO be a random oracle.

P is honest and Q is corrupt

1. Let $\gamma^P := \Gamma^P(id), \nu^P := \gamma^P.\mathsf{pspace}(pid), (s_P, s_Q, \mathsf{version}) := \Gamma_{aux}^P(id, pid)$.

2. Set t_0 be the current round. Send $(\texttt{instance-registering}, id, pid) \overset{t_1 \le t_0+\Delta}{\hookrightarrow} Q$.
3. Q can have two following reactions:
   - If $(\texttt{instance-construct}, id, C, z, A, s_A) \overset{t_1}{\hookleftarrow} Q$, ignore if $RO(id, A, C, z) \neq pid$ or the signature is not valid. Let $\sigma := C.\mathsf{Construct}(A, t_1, z)$. Ignore if $\sigma = \bot$ or $\sigma.\mathsf{payer} \neq A$. Send $(\texttt{instance-registered}, id, pid, \nu^P) \overset{\le t_1+\Delta}{\hookrightarrow} Q$ and goto step 5.
   - If $(\texttt{instance-register}, id, pid, \nu^Q, \hat{\mathsf{version}}, \hat{s_Q}, \hat{s_P}) \overset{t_1}{\hookleftarrow} Q$, ignore if some signature is not valid except $\hat{\mathsf{version}} = -1$. Else set $\tilde{\nu} := \mathsf{version} > \hat{\mathsf{version}} \ ? \ \nu^P : \nu^Q$. Set $\Gamma^P(id).\mathsf{pspace}(pid) := \tilde{\nu}$. Send $(\texttt{instance-registered}, id, pid, \tilde{\nu}) \overset{\le t_1+\Delta}{\hookrightarrow} Q$ and goto step 5.
4. Send $(\texttt{instance-registered}, id, pid, \nu^P) \overset{\le t_0+3\Delta}{\hookrightarrow} Q$ and goto step 5. (This captures the situation when honest P completes the registration alone).
5. Mark (id, pid) as register in Γ^P_{aux}.

P is corrupt and Q is honest

Upon $(\texttt{instance-construct}, id, C, z, A, s_A) \overset{t_0}{\hookleftarrow} P$, ignore if $\Gamma(id) = \bot$ or $P \notin \Gamma(id).\mathsf{endusers}$. Let $pid := RO(id, A, C, z)$. Ignore if (id, pid) is marked as registered in Γ^Q_{aux}. Let $\sigma := C.\mathsf{Construct}(A, t_0, z)$. Ignore if $\sigma = \bot$ or $\sigma.\mathsf{payer} \neq A$. Then distinguish the following two situations:

- If $\Gamma^Q(id).\mathsf{pspace}(pid) \neq \bot$: let $\nu := \Gamma^Q(id).\mathsf{pspace}(pid)$. Send $(\texttt{instance-registered}, id, pid, \nu) \overset{\le t_0+2\Delta}{\hookrightarrow} P$. Mark (id, pid) as registered in Γ^Q_{aux}.
- If $\Gamma^Q(id).\mathsf{pspace}(pid) = \bot$: let $\nu := \bot$ and set $\nu.\mathsf{code} := C, \nu.\mathsf{storage} := \sigma$. Set $\Gamma^Q(id).\mathsf{pspace}(pid) := \nu, \Gamma(id).\mathsf{pspace}(pid) := \nu$, mark (id, pid) as registered in Γ^Q_{aux}. Send $(\texttt{instance-registered}, id, pid, \nu) \overset{\le t_0+2\Delta}{\hookrightarrow} P$.

Upon $(\texttt{instance-register}, id, pid, \nu, \mathsf{version}, s_P, s_Q) \overset{t_0}{\hookleftarrow} P$, ignore if $\Gamma(id) = \bot$ or $P \notin \Gamma(id).\mathsf{endusers}$ or $\mathsf{version} = -1$ or at least one signature is not valid or (id, pid) is marked as registered in Γ^Q_{aux}. Let $\nu^Q := \Gamma^Q(id).\mathsf{pspace}(pid)$ and fetch $\mathsf{version}^Q$ of (id, pid) from Γ^Q_{aux}. Set $\tilde{\nu} := \mathsf{version} > \mathsf{version}^Q \ ? \ \nu : \nu^Q$. Mark (id, pid) as registered in Γ^Q_{aux}. Set $\Gamma^Q(id).\mathsf{pspace}(pid) := \tilde{\nu}$. Send $(\texttt{instance-registered}, id, pid, \tilde{\nu}) \overset{\le t_0+2\Delta}{\hookrightarrow} P$.

Simulator Sim: Create an initial promise instance

Denote $\mathcal{F}_{PPC} := \mathcal{F}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$.
Let RO be a random oracle.

P is honest and Q is corrupt

Upon $(\texttt{create-instance}, id, C, z) \overset{t_0}{\hookleftarrow} \mathcal{F}_{PPC}$:

1. Compute $pid := RO(id, P, C, z)$. Let $\nu := \bot$, compute $\sigma := C.\mathsf{Construct}(P, t_0, z)$. Set $\nu.\mathsf{code} := C$ and $\nu.\mathsf{storage} := \sigma$.
2. Compute $s_P := \mathsf{Sign}_{sk_P}(id, P, C, z)$.
3. Set $\Gamma^P(id).\mathsf{pspace}(pid) := \nu$ and set $\Gamma^P_{aux}(id, pid) := (\bot, \bot, 0)$.
4. Send $(\texttt{create-instance}, id, C, z, s_P) \hookrightarrow Q$ on behalf of P.

P is corrupt and Q is honest

Upon $(\texttt{create-instance}, id, C, z, s_P) \overset{t_0}{\hookleftarrow} P$:

1. Stop if one of the following conditions holds: $\Gamma^Q(id) = \bot$; $P \notin \Gamma^Q(id).\mathsf{endusers}$; $\Gamma^Q(id).\mathsf{pspace}(pid) \neq \bot$; $C \notin PS$. Else let $\gamma := \Gamma^Q(id)$.
2. Let $pid := RO(id, P, C, z)$. Let $\sigma := C.\mathsf{Construct}(P, t_0, z)$. Stop if $\sigma = \bot$. Let $\nu := \bot$. Set $\nu.\mathsf{code} := C$ and $\nu.\mathsf{storage} := \sigma$. Stop if $\mathsf{Vfy}_{pk_P}(id, P, C, z; s_P) \neq 1$.
3. Set $\Gamma^Q(id).\mathsf{pspace}(pid) := \nu$ and set $\Gamma^Q_{aux}(id, pid) := (\bot, \bot, 0)$. Send $(\texttt{create-instance}, id, C, z) \overset{t_0}{\hookrightarrow} \mathcal{F}_{PPC}$ on behalf of P.

Simulator Sim: Promise instance execution

Denote $\mathcal{F}_{PPC} := \mathcal{F}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$.
Let RO be a random oracle.

P is honest and Q is corrupt

Upon $(\texttt{execute}, id, pid, f, z) \overset{t_0}{\hookleftarrow} \mathcal{F}_{PPC}$, let $\gamma^P := \Gamma^P(id), \nu^P := \gamma^P.\mathsf{pspace}(pid), \sigma^P := \nu^P.\mathsf{storage}, (\cdot, \cdot, \mathsf{version}^P) := \Gamma^P_{aux}\ (id, pid)$:

1. Stop if $\gamma^P = \bot$ or $\nu^P = \bot$ or $P \notin \gamma^P.\mathsf{endusers}$ or $f \notin \gamma^P.\mathsf{code}$.
2. Set $t_1 := t_0 + x$, where x is the smallest offset such that $t_1 \equiv 1 (\mathrm{mod}\ 4)$ if $P = \gamma^P.\mathsf{Alice}$ and $t_1 \equiv 3 (\mathrm{mod}\ 4)$ if $P = \gamma^P.\mathsf{Bob}$.
3. If (id, pid) is *not* marked as registered in Γ^P_{aux}:
   (a) Compute $(\tilde{\sigma}, m) := f(\sigma^P, P, t_0, z)$. Stop if $m = \bot$. Otherwise, set $\tilde{\mathsf{version}} := \mathsf{version}^P + 1$. Let $\tilde{\nu} := \bot$. Set $\tilde{\nu}.\mathsf{code} := C^P$ and $\tilde{\nu}.\mathsf{storage} := \tilde{\sigma}$. Compute $s_P := \mathsf{Sign}_{sk_P}(id, pid, \tilde{\nu}, \tilde{\mathsf{version}})$.
   Send $(\texttt{peaceful-request}, id, pid, f, z, s_P, t_0) \overset{t_1+1}{\hookrightarrow} Q$.
   (b) If $(\texttt{peaceful-confirm}, id, pid, s_Q) \overset{t_1+1}{\hookleftarrow} Q$ such that $\mathsf{Vfy}_{pk_Q}(id, pid, \tilde{\nu}, \tilde{\mathsf{version}}) = 1$, then set $\Gamma^P_{aux}(id, pid) := (s_P, s_Q, \tilde{\mathsf{version}})$ and instruct the $\mathcal{F}_{PPC}$ to execute at time t_0 and output at time $t_1 + 2$ and stop.
   (c) Execute sub-simulator $\mathsf{SimRegister}(P, id, pid)$ (end at round $t_2 \leq t_0 + 5 + 3\Delta$). If $\Gamma^P_{aux}(id).\mathsf{pspace}(pid) = \tilde{\nu}$ (Q registered the latest state), instruct the $\mathcal{F}_{PPC}$ to execute at time t_0 and output at time t_2 and stop.
   (d) Let t_3 be the current round. Instruct the $\mathcal{F}_{PPC}$ to execute at time t_3 and output according to the onchain delay. Get $(\texttt{executed}, id, pid, P, f, t, z, \nu) \overset{t_4 \leq t_3 + \Delta}{\hookleftarrow} \mathcal{F}_{PPC}$. Send $(\texttt{executed-onchain}, id, pid, P, f, t, z, \nu) \overset{t_4}{\hookrightarrow} Q$.
4. If (id, pid) is marked as registered in Γ^P_{aux}:
   (a) Let t_3 be the current round. Instruct the $\mathcal{F}_{PPC}$ to execute at time t_3 and output according to the onchain delay. Get $(\texttt{executed}, id, pid, P, f, t, z, \nu) \overset{t_4 \leq t_3 + \Delta}{\hookleftarrow} \mathcal{F}_{PPC}$. Send $(\texttt{executed-onchain}, id, pid, P, f, t, z, \nu) \overset{t_4}{\hookrightarrow} Q$.

P is corrupt and Q is honest

Upon $(\texttt{peaceful-request}, id, pid, f, z, s_P, t_0) \overset{t_1}{\hookleftarrow} P$:

1. Stop if $\Gamma^Q(id) = \bot$, else let $\gamma^Q := \Gamma^Q(id)$. Stop if $Q \notin \gamma^Q.\mathsf{endusers}$ or $P \notin \gamma^Q.\mathsf{endusers}$ or $\gamma^Q(pid) = \bot$, else let $\nu^Q := \gamma^Q.\mathsf{pspace}(pid)$. Stop if $f \notin \nu^Q.\mathsf{code}$, else let $C^Q := \nu^Q.\mathsf{code}$ and $\sigma^Q := \nu^Q.\mathsf{storage}$. Let $(\cdot, \cdot, \mathsf{version}^Q) := \Gamma^Q_{aux}(id, pid)$.

2. Stop if "$P = \gamma^Q.\mathsf{Alice}$ and $t_1 \not\equiv 1 (\mathrm{mod}\ 4)$" or "$P = \gamma^Q.\mathsf{Bob}$ and $t_1 \not\equiv 3 (\mathrm{mod}\ 4)$".
3. Stop if $t_0 \notin [t_1 - 3, t_1]$.
4. Stop if (id, pid) is marked as registered, else do:
   (a) Compute $(\tilde{\sigma}, m) := f(\sigma^Q, P, t_0, z)$. Stop if $m = \perp$.
   (b) Set $\tilde{\mathsf{version}} := \mathsf{version}^Q + 1$. Let $\tilde{\nu} := \perp$. Set $\tilde{\nu}.\mathsf{code} := C^Q$ and $\tilde{\nu}.\mathsf{storage} := \tilde{\sigma}$.
   (c) If $\mathsf{Vfy}_{pk_P}(id, pid, \tilde{\nu}, \tilde{\mathsf{version}}; s_P) \neq 1$, then stop.
   (d) Compute $s_Q := \mathsf{Sign}_{sk_Q}(id, pid, \tilde{\nu}, \tilde{\mathsf{version}})$. Send $(\texttt{peaceful-confirm}, id, pid, s_Q) \overset{t_1+1}{\hookrightarrow} P$. Set $\Gamma^Q(id).\mathsf{pspace}(pid) := \tilde{\nu}$, $\Gamma^Q_{aux}(id, pid) := (s_Q, s_P, \tilde{\mathsf{version}})$.
   (e) Send $(\texttt{execute}, id, pid, f, z) \overset{t_1}{\hookrightarrow} \mathcal{F}_{PPC}$ and instruct $\mathcal{F}_{PPC}$ to execute at time t_0 and output at time $t_1 + 1$.

Upon $(\texttt{instance-execute}, id, pid, f, z) \overset{t_2}{\hookleftarrow} P$:

1. Stop if $\Gamma^Q(id) = \perp$, else let $\gamma^Q := \Gamma^Q(id)$. Stop if $Q \notin \gamma^Q.\mathsf{endusers}$ or $P \notin \gamma^Q.\mathsf{endusers}$ or $\gamma^Q.\mathsf{pspace}(pid) = \perp$, else let $\nu^Q := \gamma^Q.\mathsf{pspace}(pid)$. Stop if $f \notin \nu^Q.\mathsf{code}$, else let $C^Q := \nu^Q.\mathsf{code}$ and $\sigma^Q := \nu^Q.\mathsf{storage}$. Let $(\cdot, \cdot, \mathsf{version}^Q) := \Gamma^Q_{aux}(id, pid)$. Stop if (id, pid) is *not* marked as registered in Γ^Q_{aux}.
2. Send $(\texttt{execute}, id, pid, f, z) \overset{t_2}{\hookrightarrow} \mathcal{F}_{PPC}$ and instruct $\mathcal{F}_{PPC}$ to execute at time t_2 and output according to the onchain delay. Get $(\texttt{executed}, id, pid, P, f, t_2, z, \nu) \overset{t_3 \leq t_2 + \Delta}{\hookleftarrow} \mathcal{F}_{PPC}$. Send $(\texttt{executed-onchain}, id, pid, P, f, t, z, \nu) \overset{t_3}{\hookrightarrow} P$.

Simulator Sim: Close a programmable payment channel

Denote $\mathcal{F}_{PPC} := \mathcal{F}_{PPC}^{\hat{\mathcal{L}}(\Delta)}(PS)$.
Let RO be a random oracle.

P is honest and Q is corrupt

Upon $(\texttt{close}, P, id) \overset{t_0}{\hookleftarrow} \mathcal{F}_{PPC}$:

1. Stop if id is marked as closed. Otherwise, mark id as closed.
2. Stop if $\Gamma^P(id) = \perp$, else let $\gamma^P := \Gamma^P(id)$. For each $\gamma^P.\mathsf{pspace}(pid) \neq \perp$ and (id, pid) is not marked as registered in Γ^P_{aux}, execute (in parallel) sub-simulator $\mathsf{SimRegister}(P, id, pid)$ immediately. This execution will be finished in 3Δ rounds. Within another Δ rounds (set as real), send $(\texttt{contract-closing}, id) \overset{t_1 \leq t_0 + 4\Delta}{\hookrightarrow} Q$.
3. Execute sub-simulator $\mathsf{SimRegister}(Q, id, pid)$ if there exists some pid registered by Q. This will be finished in 3Δ rounds.
4. At round $t_2 \leq t_0 + 7\Delta$, instruct $\mathcal{F}_{PPC}$ to set the first waiting time till t_2.
5. If there exists some pid such that $\Gamma^P(id).\mathsf{pspace}(pid).ended = 0$ at t_2, wait for $\gamma^P.duration$ rounds. Within Δ rounds, send $(\texttt{contract-close}, id)$ $\overset{t_3 \leq t_0 + 8\Delta + \gamma^P.duration}{\hookrightarrow} Q$ and instruct $\mathcal{F}_{PPC}$ to send messages at round t_3.

P is corrupt and Q is honest

1. Execute sub-simulator $\mathsf{SimRegister}(P, id, pid)$ if there exists some pid registered by P at round t_0.
2. If $(\texttt{contract-close}, id) \overset{t_1 \leq t_0 + 2\Delta}{\hookleftarrow} P$ after the registration, send $(\texttt{close}, id) \overset{t_1}{\hookrightarrow} \mathcal{F}_{PPC}$ on behalf of P.
3. Stop if id is marked as closed. Otherwise, mark id as closed.

4. Within Δ rounds, for each $\gamma^Q.\mathsf{pspace}(pid) \neq \perp$ and (id, pid) is not marked as registered in Γ^Q_{aux}, execute (in parallel) sub-simulator $\mathsf{SimRegister}(Q, id, pid)$ immediately. This execution will be finished in 3Δ rounds (or at $t_2 \leq t_1 + 3\Delta$ round).
5. If there exists some pid such that $\Gamma^Q(id).\mathsf{pspace}(pid).ended = 0$ at t_2, wait for $\gamma^Q.duration$ rounds. Within Δ rounds, send $(\texttt{contract-close}, id)$ $\overset{t_3 \leq t_0 + 8\Delta + \gamma^Q.duration}{\hookrightarrow}$ P and instruct $\mathcal{F}_{PPC}$ to send messages at round t_3.

Theorem 2 (Coins momentum). *Any channel γ cannot produce coins.*

Proof. Alice will get $min\{total, max\{0, \gamma.\mathsf{cash}(\gamma.\mathsf{Alice}) + credit_A - credit_B\}\}$ coins and Bob will get $min\{total, max\{0, \gamma.\mathsf{cash}(\gamma.\mathsf{Bob}) + credit_B - credit_A\}\}$ coins while closing. Both are non-negative values and sum up to $total$. □

Theorem 3 (Balance security). *Honest users will not lose coins in channels.*

Proof. (sketch) For corrupted Q to steal coins from P, Q must have an initial promise from P, which requires an unforgeable signature from P. □

E The Global Ledger Functionality

Functionality $\hat{\mathcal{L}}(\Delta)$

Functionality $\hat{\mathcal{L}}$, running with parties $P_1, ..., P_n$ and several ideal functionalities $\mathcal{F}_1, ..., \mathcal{F}_m$, maintains a vector $(x_1, ..., x_n) \in \mathbb{R}^n_{\geq 0}$ representing the balances (in *coins*) of parties. $\hat{\mathcal{L}}$ is also parameterized by a positive integer Δ, representing the delay (controlled by the adversary) in updating its state.

Adding money

Upon receiving a message $(\texttt{add}, P_i, y)$ from $\mathcal{F}_j$ $(j \in [m], y \in \mathbb{R}_{\geq 0})$, set $x_i := x_i + y$ within Δ rounds. We say that *y coins are added to P_i's account in $\hat{\mathcal{L}}$.*

Removing money

Upon receiving a message $(\texttt{remove}, P_i, y)$ from $\mathcal{F}_j$ $(j \in [m], y \in \mathbb{R}_{\geq 0})$:

- Stop if $x_i < y$,
- Otherwise, set $x_i := x_i - y$ within Δ rounds. We say that *y coins are removed from P_i's account in $\hat{\mathcal{L}}$.*

Fig. 9: The global ledger functionality $\hat{\mathcal{L}}$.

F Implementation and Evaluation

We instantiated PPC in the Ethereum network. In this section, first, we introduce some building blocks for creating an efficient PPC protocol in the Ethereum. Next, we evaluate our implementation in terms of Ethereum gas usage. Finally, we measure the transaction throughput.

F.1 Extra Building Blocks

Receipts. We introduce a new type of message that corresponds to a particular promise. A *receipt* is sent by the sender of the promise after it is resolved between the two parties. The message is signed by the payer and includes a credit value that denotes the *aggregated amount* of all previously resolved promises. This message enables the parties to only present the *receipt* message to the on-chain contract, resulting in less number of transactions posted on-chain.

Monotonically Increasing Credits. With the introduction of *receipt* messages, a malicious party can deceive the contract by presenting a (promise, receipt) pair for which the amount of the promise is already included in the receipt. This problem is known as the double spending attack. Such an attack can simply be mitigated by including an index/counter for the promises and receipts, i.e., old promises are invalidated by higher index values included in every receipt.

In order to avoid introducing new variables to order transactions, each party in PPC implementation maintains two credit values for the aggregated amounts that they have sent and received via off-chain transactions. Since the credit values are monotonically increasing they can be used to serve the purpose of indexes. Futhermore, since at least one of the credit values is incremented for every off-chain transaction, their combination could be used to uniquely identify each message, and thus protect against replay attacks.

Accumulators. One downside of using monotonically increasing values is that they force off-chain transactions to happen sequentially, and so, no new promise can be accepted unless the previous one is satisfied (i.e., the corresponding receipt is received). To provide the maximum parallelization for a receiver that can process multiple promises simultaneously, the PPC implementation 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 PPC implementation provides this feature by asking the parties to commit to the set of pending promises along with every receipt.

As the list of pending promises grows linearly, it is inefficient to include the entire list in every receipt, as significant fees will be associated to claiming a receipt on-chain. To address this issue, PPC implementation uses cryptographic accumulators such as Merkle tree and RSA accumulator. Usage of accumulators reduces 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.

F.2 Smart Contract of PPC Implementation

In this subsection, we present the smart contract of the implementation presented in Fig. 10, which were implemented via the Solidity language. Here, we provide an overview of the contract's functionalities and the added features to the protocol presented in Fig. 8 and Appendix C.

Multiple Deposits. The `Deposit` function in the contract can be used by the parties to increase their balance for off-chain payments. This function can be invoked any number of times as long as the channel is active (i.e., not closing).

Receipts and Accumulators. As mentioned earlier we added a *receipt* message to efficiently close the channel (i.e., consuming less amount of gas). The receipt object R is formally defined as a tuple $(\mathsf{idx}, \mathsf{credit}, \mathsf{acc})$, where (1) idx is the receipt's index[6]; (2) credit is the aggregated amounts of resolved promises off-chain; and (3) acc is an accumulator for tracking the pending promises. Upon claiming a promise, the index in the *promise* object is checked to see if it is less than the index (idx) of the *receipt*

[6] Here we use index for ease of understanding, but as mentioned in Appendix F.1, we can use the credit values as indexes for more efficiency

PPC **Contract**

`Init`($\mathsf{cid}', \mathsf{vk}'_A, \mathsf{vk}'_B, \mathsf{claimDuration}'$):

1. Set $(\mathsf{cid}, \mathsf{claimDuration}) \leftarrow (\mathsf{cid}', \mathsf{claimDuration}')$;
2. Set $\mathsf{status} \leftarrow$ "`Active`"; $\mathsf{chanExpiry} \leftarrow 0$; $\mathsf{unresolvedPromises} \leftarrow \bot$;
3. Set $\mathsf{A} \leftarrow \{\mathsf{addr} : \mathsf{vk}'_A, \mathsf{deposit} : 0, \mathsf{rid} : 0, \mathsf{credit} : 0, \mathsf{acc} : \bot, \mathsf{closed} : \mathtt{F}\}$
4. Set $\mathsf{B} \leftarrow \{\mathsf{addr} : \mathsf{vk}'_B, \mathsf{deposit} : 0, \mathsf{rid} : 0, \mathsf{credit} : 0, \mathsf{acc} : \bot, \mathsf{closed} : \mathtt{F}\}$

`Deposit`(amount):

1. Require $\mathsf{status} =$ "`Active`" and $\mathsf{caller.vk} \in \{\mathsf{A.addr}, \mathsf{B.addr}\}$;
2. If $\mathsf{caller.vk} = \mathsf{A.addr}$, then set $\mathsf{A.deposit} \leftarrow \mathsf{A.deposit} + \mathsf{amount}$;
3. If $\mathsf{caller.vk} = \mathsf{B.addr}$, then set $\mathsf{B.deposit} \leftarrow \mathsf{B.deposit} + \mathsf{amount}$.

`RegisterReceipt`(R):

1. Require $\mathsf{status} \in \{$"`Active`", "`Closing`"$\}$;
2. Require $\mathsf{caller.vk} \in \{\mathsf{A.addr}, \mathsf{B.addr}\}$;
3. If $\mathsf{caller.vk} = \mathsf{A.addr}$, then:
   (a) Require `SigVerify`($R.\sigma$, [cid, R.idx, R.credit, R.acc], B.addr);
   (b) Set $\mathsf{A.rid} \leftarrow R.\mathsf{idx}$, $\mathsf{A.credit} \leftarrow R.\mathsf{credit}$, and $\mathsf{A.acc} \leftarrow R.\mathsf{acc}$;
   Otherwise:
   (a) Abort if `SigVerify`($R.\sigma$, [cid, R.idx, R.credit, R.acc], A.addr);
   (b) Set $\mathsf{B.rid} \leftarrow R.\mathsf{idx}$, $\mathsf{B.credit} \leftarrow R.\mathsf{credit}$, and $\mathsf{B.acc} \leftarrow R.\mathsf{acc}$;
4. If $\mathsf{status} =$ "`Active`" then set $\mathsf{chanExpiry} \leftarrow \mathsf{now} + \mathsf{claimDuration}$ and $\mathsf{status} \leftarrow$ "`Closing`".

`RegisterPromise`(P):

1. Require $\mathsf{status} \in \{$"`Active`", "`Closing`"$\}$;
2. Require $\mathsf{caller.vk} \in \{\mathsf{A.addr}, \mathsf{B.addr}\}$;
3. Require $[P.\mathsf{addr}, P.\mathsf{receiver}] \notin \mathsf{unresolvedPromises}$;
4. If $P.\mathsf{sendr} = \mathsf{A.addr}$, set $\mathsf{sendr} \leftarrow \mathsf{A}$ and $\mathsf{receiver} \leftarrow \mathsf{B}$;
   Otherwise set $\mathsf{sendr} \leftarrow \mathsf{B}$ and $\mathsf{receiver} \leftarrow \mathsf{A}$;
5. Require `SigVerify`($P.\sigma$, [cid, P.rid, P.sendr, P.receiver, P.addr], sendr.addr);
6. If $\mathsf{caller.vk} = \mathsf{receiver.addr}$ and $P.\mathsf{rid} < \mathsf{receiver.rid}$,
   Require ACC.VerifyProof(acc, P.addr, P.proof);
7. Invoke `Deploy` (P.byteCode, P.salt);
8. Set unresolvedPromises.`push`([P.addr, receiver])
9. If $\mathsf{status} =$ "`Active`", then set $\mathsf{chanExpiry} \leftarrow \mathsf{now} + \mathsf{claimDuration}$, and $\mathsf{status} \leftarrow$ "`Closing`".

`Close`():

1. Require $\mathsf{caller.vk} \in \{\mathsf{A.addr}, \mathsf{B.addr}\}$;
2. If $\mathsf{caller.vk} = \mathsf{A.addr}$, set $\mathsf{A.closed} \leftarrow \mathtt{T}$; Otherwise set $\mathsf{B.closed} \leftarrow \mathtt{T}$;
3. If A.closed and B.closed, set $\mathsf{status} \leftarrow$ "`Closed`";
4. If $\mathsf{status} =$ "`Active`", then set $\mathsf{chanExpiry} \leftarrow \mathsf{now} + \mathsf{claimDuration}$, and $\mathsf{status} \leftarrow$ "`Closing`".

`Withdraw`():

1. Require $\mathsf{status} \in \{$"`Closing`", "`Closed`"$\}$;
2. If $\mathsf{status} =$ "`Closing`", Require $\mathsf{now} > \mathsf{chanExpiry}$;
3. For each $(\mathsf{addr}, \mathsf{receiver}) \in \mathsf{unresolvedPromises}$:
   $\mathsf{receiver.credit} \leftarrow \mathsf{receiver.credit} + \mathsf{addr.resolve}()$;
4. Invoke `transfer`(A.addr, A.deposit + A.credit − B.credit) and
   `transfer`(B.addr, B.deposit + B.credit − A.credit).

Fig. 10: UPC Contract

submitted previously. In such a case, the claiming party should provide an extra inclusion proof to show that the promise is included in the accumulator (of the submmitted *receipt*) to avoid double spending.

Cooperative Closing. Parties can close the channel by submitting a receipt and/or a set of promises. In any case of closure, the contract status changes to "`Closing`" and the channel expiry is set (i.e., the timestamp that the parties have to submit any receipt and/or promises). However, as this time can be long, parties can finalize the closing sooner than the channel expiry time by invoking the `Close` function. Once both parties have finalized closing, the status of the channel changes to "`Closed`" and parties can withdraw their balances.

F.3 Hash-Time-Locked Contracts on PPC

As stated in the previous sections, many general applications and programmable payments can be defined and implemented on PPC. Here, we present a simple case of HTLC and due to space constraints leave other applications for future work.

HTLCs are one of the most used contracts on Layer-2 networks for routing payments between different parties [30,3,28]. Consider the case that there exists (1) a payment channel between parties A and B (A↔B); and (2) another channel between parties B and C (B↔C). A can utilize the B↔C channel to transfer funds to C off-chain without establishing a new channel. To remove any trust in party B, HTLCs can be used, by making the payment conditional on providing a preimage to a hash value specified in the HTLC before some expiry time. Fig. 11 shows the smart contract code for the HTLC. The compiled code along with the constructor values will construct the bytecodes of the promise, which can be deployed by the PPC contract (see Fig. 10).

HTLC Contract

`Init`(amount′, hash′, expiry′):
1. Set (amount, hash, expiry) ← (amount′, hash′, expiry′);
2. Set secretRevealed ← `F`.

`RevealSecret`(secret):
1. Require now < expiry;
2. Require `Hash`(secret) = hash;
3. Set secretRevealed ← `T`;

`Resolve`():
1. If secretRevealed, then return amount;
2. Require now ≥ expiry;
3. Return 0.

Fig. 11: HTLC Contract

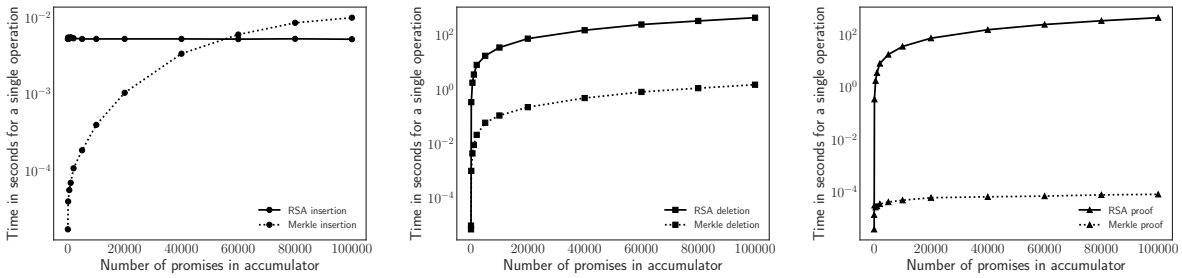
In a similar manner, the routing can be extended to multi-hops where each pair of parties in the route will construct their own HTLC contract. We can also consider the following optimization factor, where only one instance of HTLC code can be used[7] for all the promises within a route. In other words, all the promises will be referencing the same address, and the HTLC contract will be only deployed once (saving gas in the pessimistic case where multiple parties would need to go on-chain). A similar concept exists in [28] where a registry contract that all parties can reference stores the secret value of an HTLC payment.

[7] Contract changes to consider different timings for each party in the route.

Many related works [32,3,26,28,16] have extensively studied the use of HTLCs for routing. Due to space constraints, we refer the readers to the related work for detailed explanations of the off-chain protocol and in the following subsection, we present the performance results.

F.4 Evaluation

We have created a proof of concept implementation of our PPC protocol with promises containing HTLCs. The ledger contract presented in Fig. 10 has been implemented using the Solidity programming language for Ethereum. As discussed in Appendix F.1, we have used accumulators to allow for sending and processing concurrent promises. 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, shown in Fig. 12. We observe that the Merkle tree is the better choice as it has less overhead. We further investigated the gas consumption of the two and saw that membership proof verification of RSA consumes significantly more gas on-chain (450K gas compared to 20K when 100K promises are stored in the accumulator).



(a) Accumulator Insertion (b) Accumulator Deletion (c) Membership Proof

Fig. 12: Performance comparisons between Merkle Tree and RSA accumulators

We begin by evaluating the gas needed for the deployment of the contract. The PPC contract requires 3,243,988 gas to be deployed on the Ethereum blockchain. We emphasize that in our implementation we did not aim to optimize gas costs and further optimizations can reduce the gas. However, within its current state, the PPC contract is comparable to other simple payment channel deployments 2M+ and 3M+ gas for Perun [16] and Raiden [3][8] respectively. The gas usage for the remaining functions of the contract are reported in Table 2.

Table 2: Gas prices for invoking PPC contract's functions.

Function	**Gas Units**	**HTLC Specific**	**Gas Units**
Deploy	3,243,988	Promise	611,296 (w/o. proof)
Deposit	43,010	Promise	626,092 (Merkle-100K txs)
Receipt	75,336	RevealKey	66,340
Close	44,324	Withdraw	71,572

[8] https://tinyurl.com/etherscanRaiden

In the case of HTLC application, 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 submit its latest receipt. Next, it submits the promise for the HTLC which will be deployed by PPC 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 700K more gas will be needed to resolve the promise.

We further evaluated the run-time of the protocol by performing a test on ten clients concurrently sending transactions to a single server. Each client sent $1,000$ transactions, which made the server process a total of $10,000$ promises, secret reveals, and receipts. We were able to achieve 110 TPS end-to-end on a laptop running 2.6 GHz 6-Core Intel Core i7. The end-to-end process included random secret creation, hashing of secret, promise creation/verification, secret reveal/verification, and receipt creation/verification. We note that the performance can be further improved by optimizing the off-chain code and using a more powerful machine for the server.

Finally, we would like to highlight that the HTLC payment demonstrated between two parties with established payment channels can be expanded to parties that are not directly connected but are linked through intermediaries, also referred to as payment channel networks [30,16,18,3]. This method can also be utilized to facilitate cross-chain payments involving an intermediary on multiple chains or with the aid of relayers and light clients [23,27,21].