Programmable Payment Channels

Ranjit Kumaresan[1], Duc V. Le[1], Mohsen Minaei[1], Srinivasan Raghuraman[2], Yibin Yang[3]**, and Mahdi Zamani[1]

[1] Visa Research, USA
{rakumare,duc.le,mominaei,mzamani}@visa.com
[2] Visa Research and MIT, USA
srini131293@gmail.com
[3] Georgia Institute of Technology, USA
yyang811@gatech.edu

Abstract. One approach for scaling blockchains is to create bilateral, offchain channels, known as payment/state channels, that can protect parties against cheating via onchain collateralization. While such channels have been studied extensively, not much attention has been given to *programmability*, where the parties can agree to *dynamically enforce arbitrary* conditions over their payments without going onchain.

We introduce the notion of a *programmable payment channel* (PPC) that allows two parties to do exactly this. In particular, our notion of programmability enables the sender of a (unidirectional) payment to *dynamically* set the terms and conditions for each individual payment using a smart contract. Of course, the verification of the payment conditions (and the payment itself) happens offchain as long as the parties behave honestly. If either party violates any of the terms, then the other party can deploy the smart contract onchain to receive a remedy as agreed upon in the contract. In this paper, we make the following contributions:

- We formalize PPC as an ideal functionality $\mathcal{F}_{\mathsf{PPC}}$ in the universal composable framework, and build lightweight implementations of applications such as hash-time-locked contracts (HTLCs), "reverse HTLCs", and rock-paper-scissors in the $\mathcal{F}_{\mathsf{PPC}}$-hybrid model;
- We show how $\mathcal{F}_{\mathsf{PPC}}$ can be easily modified to capture the state channels functionality $\mathcal{F}_{\mathsf{SC}}$ (described in prior works) where two parties can execute *dynamically chosen* arbitrary two-party contracts (including those that take deposits from both parties) offchain, i.e., we show how to efficiently realize $\mathcal{F}_{\mathsf{SC}}$ in the $\mathcal{F}_{\mathsf{PPC}}$-hybrid model;
- We implement $\mathcal{F}_{\mathsf{PPC}}$ on blockchains supporting smart contracts (such as Ethereum), and provide several optimizations to enable *concurrent* programmable transactions—the gas overhead of an HTLC PPC contract is < 100K, amortized over many offchain payments.

We note that our implementations of $\mathcal{F}_{\mathsf{PPC}}$ and $\mathcal{F}_{\mathsf{SC}}$ depend on the CREATE2 opcode which allows one to compute the deployment address of a contract (without having to deploy it).

Keywords: Blockchain · Layer-2 channels · Programmable payments.

** Work done in part while at Visa Research.

1 Introduction

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 onchain transactions, preventing the adoption of blockchain services as mainstream financial products.

Payment Channels. A well-known class of mechanisms for scaling blockchain payments are payment channels [2, 15]. Payment channels "off-load" transactions to an offchain communication channel between two parties. The channel is "opened" via an onchain transaction to fund the channel, followed by any number of offchain transactions. Eventually, by a request from either or both parties, the channel is "closed" via another onchain transaction. This design avoids the costs and the latency associated with onchain operations, effectively amortizing the overhead of onchain transactions over many offchain ones. While several proposals improve the scalability of payment channels [3, 17, 21–23, 28–30], they do not allow imposing arbitrary conditions on offchain payments, which prohibit fruitful applications requiring programmability.

State Channels. From a feasibility standpoint, the conditions on offchain payments can be achieved by a stronger notion called state channels. State channels [4,12,14,16,18,26] allow two parties to perform general-purpose computation offchain by mutually tracking the current state of the program. The existing state channel proposals have two major drawbacks in practice.

First, with the exception of [14], state channel constructions require the parties to fix the program, which they wish to run offchain, at the time of channel setup. This means that no changes to the program are allowed after the parties go offchain. This is especially problematic in offchain scalability approaches based on the hub-and-spoke model [10, 17, 32], 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 Figure 1 Left and Middle). In practice, parties usually have no a priori knowledge about the specific set of conditions required to transact with other (unknown) parties.

Second, the complexity of the existing state channel proposals could be overkill for simple, programmable payments. The authorization of an offchain transaction via a payment channel is significantly simpler as the flow of the payments is unidirectional while state channels need to track all state changes from both parties irrespective of the payment direction. Namely, the state channel is not a practical solution for achieving programmable payments.

Our Focus. In this paper, we introduce the notion of *programmable payment channels* (PPC) that allows the parties to agree offchain on the set of conditions (i.e., a smart contract) they wish to impose for each of their offchain payments (see Figure 1 Right). That is, we achieve lightweight offchain programmable

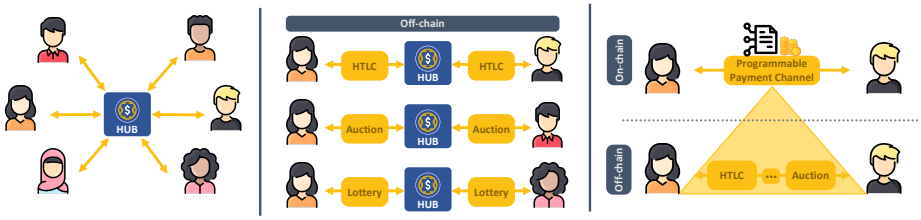


Fig. 1: **Left:** Hub-and-spoke model: Each party creates a single channel with the hub; **Middle:** Every pair of parties reuse their channels with the hub to execute different contracts; **Right:** PPC between two parties supporting any offchain application.

payments denoted as *promises* where the logic can be determined *on-the-fly* after the channel has been opened.

A classic programmable payment covered by PPC is a hash-time-locked contract (HTLC) [1], which is foundational to the design of (multihop) payment channels [3, 28]. Indeed, most current payment channels already embed HTLCs for routing. However, many useful applications remain difficult to build on top of payment channels using HTLCs. Consider the following 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 payment with PPC is simple and straightforward.

1.1 Our Contributions

– We propose the notion of a *programmable payment channel* (PPC) that is a payment channel allowing two parties to transact offchain according to a collateral that they deposit onchain and a smart contract that they agree on offchain. PPC provides the following features:

  - *Scalability:* Only opening and closing the channel require Layer-1 access.
  - *Offchain Programmability*: The PPC protocol stays identical for new payment logic after the channel is opened.

– We formalize PPC and prove its correctness and security in the universal composable (UC) framework using a global ledger. In particular, we provide an ideal functionality $\mathcal{F}_{\mathsf{PPC}}$. We then show how to build lightweight implementations of simple applications such as HTLCs, "reverse HTLCs," on-chain betting (and also rock-paper-scissors) in the $\mathcal{F}_{\mathsf{PPC}}$-hybrid world.

– We show how PPC can be modified to capture the state channels functionality where two parties can execute *dynamically chosen arbitrary two-party contracts* (including those that take deposits from both parties) offchain, namely, to realize $\mathcal{F}_{\mathsf{SC}}$ in the $\mathcal{F}_{\mathsf{PPC}}$-hybrid world. In particular, to launch an offchain

contract, parties only need to make three calls to $\mathcal{F}_{\mathsf{PPC}}$ to instantiate two programmable payments.

– We evaluate PPC by instantiating it on Ethereum. We show how the PPC contract deploys new contracts that embed the conditions of payments. Our results show that deploying the PPC contract needs about 3M gas, and that settling onchain in the optimistic case (honest parties) needs only 75K gas. In the pessimistic case (malicious parties), 700K more gas is needed for a simple logic such as HTLC.

We note that our implementations of $\mathcal{F}_{\mathsf{PPC}}$ and $\mathcal{F}_{\mathsf{SC}}$ depend on the CREATE2 opcode which allow one to compute the deployment address of a contract (without having to deploy it). This opcode is available on any EVM (Ethereum Virtual Machine) based chain (including Ethereum, Polygon, etc.).

Compared to prior formalizations of payment and state channels, our work shows a practical way to implement a state channel that enables arbitrary offchain smart contract applications. Additionally, our abstractions of $\mathcal{F}_{\mathsf{PPC}}$ and $\mathcal{F}_{\mathsf{SC}}$ make it more natural to design protocols for applications whose states depend on the states of other contracts on the blockchain.

We also note that our implementations of $\mathcal{F}_{\mathsf{PPC}}$ and $\mathcal{F}_{\mathsf{SC}}$ allow for flexible reuse of established channels. Exploiting this fact, one can use the abstractions of $\mathcal{F}_{\mathsf{PPC}}$ and $\mathcal{F}_{\mathsf{SC}}$ to efficiently build complex multiparty applications. For instance, every pair of parties need not establish a PPC channel with each other, and can instead reuse their existing PPC channels with, say, an untrusted hub.

Similar to payment and state channels, relay nodes (in particular, hub nodes) in PPC also face scalability concerns, as the money has to be locked for several rounds. There are known incentivization techniques to mitigate similar issues that arise in DeFi lending protocols. The same techniques can be applied in our case as well.

1.2 Related Work

Payment Channels. The key idea behind a payment channel is an onchain contract: both parties instantiate this contract and transfer digital money to it. Whenever one party wants to pay another, they simply sign 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. No execution happens on the blockchain before closing the channel; the payment between two parties relies only on exchanging digital signatures. Payment channels have been heavily studied [2, 11, 15, 18, 24, 26, 27, 30].

State Channels. A proposal for executing arbitrary contracts offchain is state channels [4, 12, 14, 16, 18, 26]. The key idea is as follows: (1) the contract can be executed offchain by exchanging signatures, and (2) the contract can be executed onchain from the last agreed state to resolve any disagreements. For example, consider 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 onchain. This idea can be naturally extended to multi-party contracts (e.g., [13, 16, 26]).

The works of [14] and [18] are closest to ours. Unlike us, [14] do not provide any formal proofs or guarantees. As mentioned in [18], their work lacks features useful for practical implementation. Also, our protocols take advantage of the CREATE2 opcode which was introduced subsequent to the work of [14]. We follow [5, 16–18] to formalize our channel using *universal composable* (UC) framework with a global ledger. However, these works focus on *channel virtualization*[4], and are *not* directly related to this work.

Other Related Work. An excellent systematization of knowledge that explores offchain solutions can be found in [20]. See Appendix B for the comparison with rollups, another popular Layer-2 scaling solution [25,31,33]. See Appendix C for other works that use the CREATE2 opcode.

2 Preliminaries

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.

GUC Model. We model and formalize PPC under *global universal composable* (GUC) framework [8, 9]. 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 D.1. We acknowledge that we restrict the distinguisher to a subclass of environments to simplify the formalizations. This restriction is standard (e.g., [17, 18]) and can be easily removed using straightforward checks.

Cryptocurrency/Contract Functionalities. We follow [16, 18] and model cryptocurrency as a global ledger functionality $\hat{\mathcal{L}}(\Delta)$ in the GUC framework (cf. Fig. 10 in Appendix D.2). Parties can move funds from/to the ledger functionality by invoking other ideal functionalities that can invoke the methods $\mathsf{Add}/\mathsf{Remove}$. Any operation on the global ledger will happen within a delay of Δ rounds, capturing that this is an onchain transaction.

Adversary. We consider an adversary who can corrupt one party in the two-party channel. The corrupted party is byzantine and can deviate from the protocol arbitrarily. As is standard in the GUC model, the objective of an adversary is to distinguish the real world from the ideal world. In applications such as ours,

[4] Virtual channels focus on designing protocols between parties who do not have a direct channel, but both have a channel with a (common) intermediary.

such behaviors could involve stealing funds from a party or a channel, violating channel restrictions, overriding application logic, state rollback, etc.

3 Programmable Payment Channels

3.1 Defining $\mathcal{F}_{\mathsf{PPC}}$

To incorporate programmability into a payment channel, one might hard-code the logic of an application inside the protocol as a template. However, this approach is not desirable as every new application requires a protocol update that would also include changes to the existing onchain contract. Motivated by this, our definition of $\mathcal{F}_{\mathsf{PPC}}$ allows for on-the-fly programmability as we explain below.

Recall that we call a programmable payment a *promise*. Concretely, our ideal functionality $\mathcal{F}_{\mathsf{PPC}}$ allows the following operations: (1) opening a payment channel, (2) creating a promise, (3) executing a promise, and (4) closing a payment channel. Our central observation is that a promise can be viewed as a smart contract. Specifically, the storage of the promise is captured by the storage of the contract, and the execution logic of the promise is captured by functions in the smart contract. The logic in different promises can be different or related, thereby capturing on-the-fly programmability. Also, importantly, the promise smart contract itself can be deployed from an appropriately designed payment channel contract.

Any number of promises can be created by an open channel and may be concurrently executed. Either party can create a promise to the other party. Since the payment is unidirectional, we refer to the creating party as the *sender* of a promise, and the other party as the *receiver* of a promise.

Promises can be related to each other in the sense that the state and the execution logic of a promise can depend on the state and execution logic of other promises. We capture this by allowing the functions of the promise have access to *its own storage,* read *access to the storage and functions of other promises in this channel, and more generally,* read *access to the storage and functions of other onchain contracts.*[5] Note that the *execution environment* of promises is quite rich, and we will show various examples of how to use this and certain caveats associated with what is implementable.

This type of dependence is common in onchain smart contracts especially in the *Decentralized Finance* applications. However, capturing this dependence (in the implementation of $\mathcal{F}_{\mathsf{PPC}}$) needs to be done carefully since promises executions are normally executed offchain, and may sometimes need to be executed onchain (and the dependence must be preserved even while the execution environment is changing). Care must be taken to ensure that this change of the execution environment (i.e., from offchain to onchain) does not affect function output.

[5] In Solidity (a high level language for EVM) parlance, promises can also call *pure* or *view* functions in onchain contracts or other promises.

Promises are executed onchain only if requested by the parties (following which, further executions related to that promise are carried out onchain).[6] Following prior work (e.g., [18]), we differentiate between onchain and offchain executions in $\mathcal{F}_{\mathsf{PPC}}$ by the amount of time it takes $\mathcal{F}_{\mathsf{PPC}}$ to respond to execution requests. That is, onchain executions are slower and take $O(\Delta)$ rounds where Δ is a blockchain parameter representing the amount of time it takes for the miners/validators to deliver a new block to the chain.

Each promise *resolves* to an unsigned integer value denoting the amount that needs to be transferred from the sender to the receiver. This resolved value is calculated at the time of payment channel closing, and then the resolved values of all promises are aggregated to determine the final settlements.

3.2 PPC Preliminaries

Contracts. We define *contracts* as in [18]. 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 $C := (\Lambda, \mathsf{Construct}, f_1, \ldots, f_s)$ where (1) Λ denotes the admissible contract storage; (2) $\mathsf{Construct}$ denotes a constructor function that takes (P, t, y) 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 y 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 unchanged) and an output message m, where $m = \perp$ represents failure.

PPC Parameters. A programmable payment channel is parameterized by 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 (think of this as the address of the PPC contract); (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.

Note that the attribute $\gamma.duration$ was not part of prior channel formalizations (e.g., [16,18]); we will further clarify it in Section 3.3. 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*

[6] In our implementation, we make the simplifying assumption that once a promise is executed onchain, all the remaining promise executions happen onchain as well.

(accessed by key 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; and (3) $\sigma.resolve \in \mathbb{R}_{\geq 0}$ denotes the amount of money transferred from payer to payee. 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 Construct will always set the caller to be the payer in the storage created; and (2) the constructor function's output is independent of input argument t, which is a time parameter capturing the current time of the blockchain. We add these restrictions to ensure that, even when the promise is registered onchain by CREATE2, the initial state remains identical.

Diverging from [16, 18], we assume that each f_i has access to the code and storage of other promises in the *same* channel, as well as the code and storage of all Layer-1 onchain contracts. Formally, we capture this by providing oracle access to the ideal functionalities. This is why we use the notation $f^{\mathcal{G},\gamma}$ in the definition of $\mathcal{F}_{\mathsf{PPC}}$ (see Figure 2), i.e., f has oracle access to the storage and the functions of onchain smart contracts and to the promises in the channel.

3.3 Ideal Functionality $\mathcal{F}_{\mathsf{PPC}}$

We propose our PPC protocol under the UC framework following [16–18]. We first define the ideal functionality $\mathcal{F}_{\mathsf{PPC}}^{\hat{\mathcal{L}}(\Delta)}$ (with dummy parties) which summarizes all the features that our PPC protocol will provide. We use $\mathcal{F}_{\mathsf{PPC}}$ as an abbreviation in the absence of ambiguity. See Figure 2 for the definition of $\mathcal{F}_{\mathsf{PPC}}$. The functionality will maintain a key-value data structure Γ to track all programmable payment channels between parties. $\mathcal{F}_{\mathsf{PPC}}$ contains the following 4 procedures.

(1) PPC *Creation.* Assume party P wants to construct a channel with party Q. Within Δ rounds, $\mathcal{F}_{\mathsf{PPC}}$ 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, $\mathcal{F}_{\mathsf{PPC}}$ will take Q's coins. Thus, the successful creation of a initial programmable payment channel takes at most 2Δ rounds. Note that if Q does not want to create the channel, P can take her money back after 2Δ rounds.

(2) *Promise Creation.* This procedure is used to create a programmable payment aka promise (offchain) from payer P to the payee Q. The promise instance is specified by payer's choice of channel γ, contract code C and arguments for the constructor function y, and a salt z that is used to identify this promise instance. Among other things, the ideal functionality ensures that $pid := (id, C, y, z)$ does not exist in $\gamma.\mathsf{pspace}$. Since payee always gains coins in any promise, we do not need an acknowledgment from the payee to instantiate a promise. Thus, the creation takes exactly 1 round.[7]

(3) *Promise 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

[7] Note that this does not hold for state channels as formalized in [18] where an instance requires coins from both parties.

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

Programmable payment channel opening

Upon $(\texttt{open}, \gamma) \overset{t_0}{\hookleftarrow} P$ where γ is a valid initial programmable payment channel, i.e., $P \in \gamma.\mathsf{endusers}, \gamma.\mathsf{cash}(\cdot) \geq 0, \gamma.\mathsf{pspace} = \bot$, 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 $(\texttt{open}, \gamma) \overset{t_1 \leq t_0+\Delta}{\hookleftarrow} Q$, 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 $(\texttt{opened}, \gamma) \hookrightarrow \gamma.\mathsf{endusers}$ 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 $(\texttt{create}, id, C||y, z) \overset{t_0}{\hookleftarrow} P$, let $\gamma := \Gamma(id)$ and let $pid := (id, C, P, y, z)$. If $\gamma = \bot$ or $P \notin \gamma.\mathsf{endusers}$ or $\gamma.\mathsf{pspace}(pid) \neq \bot$ then stop. Else proceed as follows:

- Let $\nu := \bot$ and $\sigma := C.\mathsf{Construct}(P, t_0, y)$. Stop if $\sigma = \bot$. Set $\nu.\mathsf{code} := C$ and $\nu.\mathsf{storage} := \sigma$. Set $\Gamma(id).\mathsf{pspace}(pid) := \nu$ and $\Gamma(id).\mathsf{pspace}(pid).\mathsf{flag} := 0$. 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:

- If $\gamma.\mathsf{pspace}(pid).\mathsf{flag} = 0$, and both parties are honest, then set $T := t_0 + 5$ and $t := t_0$.
- Else if $\gamma.\mathsf{pspace}(pid).\mathsf{flag} = 0$, and one party is malicious, the simulator is allowed to specify a message $\texttt{msg}$:
  - If $\texttt{msg} = \texttt{continue}$, set $T := t_0 + 5$ and $t := t_0$. This captures the situation where the adversary wants to execute offchain.
  - If $\texttt{msg} = \texttt{onchain}$, set $T := t_0 + 4\Delta + 5$ and $\gamma.\mathsf{pspace}(\cdot).\mathsf{flag} := 1$. t is further specified by the simulator. This captures the situation where the adversary wants to execute onchain.
- Else if $\gamma.\mathsf{pspace}(pid).\mathsf{flag} = 1$, one party must be malicious, then set $T := t_0 + \Delta + 5$, t is set by the simulator.

Let $\nu := \gamma.\mathsf{pspace}(pid)$ and $\sigma := \nu.\mathsf{storage}$. Let $(\tilde{\sigma}, m) := f^{\mathcal{G},\gamma}(\sigma, P, t, z)$. Set $\Gamma(id).\mathsf{pspace}(pid).\mathsf{storage} := \tilde{\sigma}$ and send $(\texttt{executed}, id, pid, P, f, t, z, \nu) \overset{t_1 \leq T}{\hookrightarrow} \gamma.\mathsf{endusers}$. Note that the adversary can only postpone the execution but *cannot* block it.

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 $\gamma.duration + 7\Delta$ rounds and proceed as follows:

1. Calculate the following values (Note that either Alice or Bob could be P):
   (a) Set $total := \gamma.\mathsf{cash}(\gamma.\mathsf{Alice}) + \gamma.\mathsf{cash}(\gamma.\mathsf{Bob})$.
   (b) Set $credit_A := \sum_{\gamma.\mathsf{pspace}(pid).\mathsf{storage}.\mathsf{payer}=\gamma.\mathsf{Bob}}(\gamma.\mathsf{pspace}(pid).\mathsf{storage}.resolve)$.
   (c) Set $credit_B := \sum_{\gamma.\mathsf{pspace}(pid).\mathsf{storage}.\mathsf{payer}=\gamma.\mathsf{Alice}}(\gamma.\mathsf{pspace}(pid).\mathsf{storage}.resolve)$.
2. 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 on $\hat{\mathcal{L}}$.
3. Send $(\texttt{contract-close}, id) \overset{t_1 \leq t_0+8\Delta+\gamma.duration}{\hookrightarrow} \gamma.\mathsf{endusers}$.

Fig. 2: The ideal functionality $\mathcal{F}_{\mathsf{PPC}}^{\hat{\mathcal{L}}(\Delta)}$ achieved by the PPC protocol.

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 onchain operation (i.e., *optimistic case*). Otherwise, if one of them is corrupt, it relies on onchain operations which takes $O(\Delta)$ rounds (i.e., *pessimistic case*). Note that, the adversary can postpone the function execution time, but it *cannot* block the honest party from executing it.

In particular, $\mathcal{F}_{\mathsf{PPC}}$ uses an attribute flag for each promise to trace the onchain/offchain status. Note that when the promise goes onchain for the first time, it takes at most 3Δ rounds to put the promise onchain. Once the promise is onchain, the execution will be taken on Layer-1 in Δ rounds. We follow [18] to break ties when both parties want to simultaneously execute the same promise, which includes at most 5 rounds delay.

(4) PPC *Closure.* When a party of the channel γ wants to close the channel, $\mathcal{F}_{\mathsf{PPC}}$ will wait for $\gamma.duration$ rounds to execute the remaining promises that have not been finalized. The corresponding procedure in the state channel functionality of [18] 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 in this case it will be the malicious sender who is locking up its money.) Waiting for $\gamma.duration$ can be avoided if both parties agree to cooperatively close the channel.

3.4 Concrete Implementation of $\mathcal{F}_{\mathsf{PPC}}$.

We show a pseudocode implementation of programmable payment channels contract in Figure 3. In this subsection, we will detail the methods in the programmable payment channels contract, and along the way we will discuss the offchain protocol that is executed to implement $\mathcal{F}_{\mathsf{PPC}}$.

The programmable payment channel contract is initialized with a channel id id, the parties' public keys $\mathsf{vk_A}$ and $\mathsf{vk_B}$, and an expiry time $\mathsf{claimDuration}$ by which the channel settles the amounts deposited. We track the deposit amount and the credit amount (which will be monotonically increasing) for the two parties. We also track a *receipt* id (i.e., rid) and an accumulator value acc. We will describe what these are for below, but for now think of receipts as keeping track of received promises that have been resolved, and the accumulator as keeping track of received promises that have not yet resolved.

Remark. Since promise executions may take some time (e.g., HTLC, chess), it is important to support *concurrency*. Promises issued by a sender are immediately added to an accumulator associated with the sender (which is maintained by both parties), and then are removed from the accumulator when they get resolved.

Just as a regular payment channel, we also provide methods for the parties to deposit an amount (the pseudocode supports multiple deposits), and also for initiating the closing of a channel via the `Close` method. A call to the `Close` method will ensure that the channel status is set to "`Closing`" or "`Closed`", and further, sets the channel expiry time.

PPC Contract

`Init`($\mathsf{id}', \mathsf{vk}'_\mathsf{A}, \mathsf{vk}'_\mathsf{B}, \mathsf{claimDuration}'$):

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

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

`RegisterPromise`(P):

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

`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 $\mathsf{A.closed}$ and $\mathsf{B.closed}$, 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`$(\mathsf{A.addr}, min(total, max(0, \mathsf{A.deposit} + \mathsf{A.credit} - \mathsf{B.credit}))$ and `transfer`$(\mathsf{B.addr}, min(total, max(0, \mathsf{B.deposit} + \mathsf{B.credit} - \mathsf{A.credit}))$, where $total = \mathsf{A.deposit} + \mathsf{B.deposit}$.

Fig. 3: PPC Contract

During the time that a channel is "`Active`" parties exchange any number of payment promises offchain. Each promise P is essentially the smart contract code describing the logic of the payment. Note that the promise contract logic may involve multiple steps and parties may concurrently send and receive any number of promises.

At a high level, the lifecycle of a promise is as follows: the sender sends the promise offchain, then the sender and the receiver execute the promise contract offchain. When both parties agree to the value of the final output of the resolve method on the promise, the sender of the promise signs a receipt signaling the fulfillment of the promise that reflects the updated credit balance of the receiver.

In more detail, a receipt from a sender consists of

- a monotonically increasing index, which keeps track of the number of fulfilled promises from the sender,
- a monotonically increasing credit, which keeps track of the sum of all resolved amounts in the fulfilled promises originating from the sender,
- an accumulator, which keeps track of all the pending promises issued by the sender, and
- a signature from the sender on all the above values with the channel id.

If the receiver obtains a faulty receipt (or did not receive the receipt, or is just malicious), then the receiver can deploy the promise onchain via the PPC contract. Note that in some cases (e.g., promises which involve multiple steps), it is possible that the sender (as opposed to the receiver) may need to deploy the promise onchain via the PPC contract.

This brings us to another important detail concerning the offchain execution of the promises that involve multiple steps (e.g., chess). In honest cases, parties will need to additionally exchange signatures with each other to commit to the storage of the promise contract after the offchain execution of individual steps. If some malicious behavior happens (e.g., some party aborts), to continue the promise execution onchain (we assume that the party also wishes to subsequently close the channel), the party calls `RegisterReceipt` with the latest receipt (along with the signature from the counterparty) that it possesses, and then calls `RegisterPromise` with the promise P.

Consistency Between Offchain and Onchain Executions. It is crucial to ensure that the switching between offchain and onchain is consistent. This is achieved by allowing parties to submit the latest state to the deployed promise (as a smart contract). Namely, the smart contract created by the PPC contract in Figure 3 using CREATE2 needs to have a function interface to "bypass" its state to the latest one. This can be trivially realized by including a monotonically increasing version number to the state, which is signed by both parties during the offchain execution. (We remark that Item 8 in Figure 3 will only deploy a smart contract (as a promise) on its initial state (e.g., an empty chess board).)

We now detail the components of a promise P:

- $P.\mathsf{sender}$ (resp. $P.\mathsf{receiver}$) denotes the sender (resp. receiver) of a promise,

– $P.\mathsf{byteCode}$ denotes the smart contract corresponding to the payment logic,
– $P.\mathsf{salt}$ denotes a one-time salt chosen by the sender,
– $P.\mathsf{addr}$ denotes the address at which the promise will be deployed by the PPC contract; note that $P.\mathsf{addr}$ is derived deterministically from $P.\mathsf{byteCode}$ and $P.\mathsf{salt}$ using a collision resistant hash function (e.g., CREATE2 opcode),
– $P.\mathsf{rid}$ denotes the latest receipt index at the time of generating this promise,
– $P.\mathsf{proof}$ denotes the proof that the promise is contained in the accumulator (i.e., is unresolved at the time the latest receipt was generated), and
– $P.\sigma$ denotes the signature of sender on $(\mathsf{id}, P.\mathsf{rid}, P.\mathsf{sender}, P.\mathsf{receiver}, P.\mathsf{addr})$.

When `RegisterPromise` is called (when malicious behaviors happen) with a valid promise, the PPC contract deploys $P.\mathsf{byteCode}$ (i.e., the smart contract associated with the payment logic of promise P) at a predetermined address. The fact that the contract is deployed at a predetermined address is what makes it possible to have promises depend on each other (cf. Section 4). Here, we assume that the PPC contract uses CREATE2 opcode to deploy the contract. In Ethereum, using the CREATE2 opcode (EIP-1014), contracts can deploy contracts whose address is set by $\mathcal{H}(\texttt{0xFF}, \mathit{sender}, \mathit{salt}, \mathit{bytecode})$ (where $\mathcal{H}$ is a collision resistant hash function). This capability implies that one can foresee the address of some yet-to-be-deployed contract.

Following deployment, parties can interact with the deployed promise independent of the PPC contract. (Again, they "bypass" to the last agreed state.) However, note that when a party calls the function `RegisterPromise`, the channel automatically goes into a closing state, and then after claimDuration time has passed, either party can withdraw funds. Thus, it is critical that the promises exchanged by the parties also meaningfully resolve within claimDuration time.

When a party calls the `Withdraw` method, the resolve method is called for each unresolved promise that is registered with the PPC contract. That is, these promises should be some onchain smart contracts. The value returned by the resolve method is then added to the credit of the corresponding receiver. Finally, each party gets transferred an amount that corresponds to its initial deposit and the difference of the credit that it is owed and the credit that it owes.

We formally state our theorem below. The formal protocols are described in Appendix E and its proof is in Appendix H.

Theorem 1 (Main). *Suppose the underlying signature scheme is existentially unforgeable against chosen message attacks. There exists a protocol working in $\mathcal{G}^{\hat{\mathcal{L}}(\Delta)}$-hybrid model that emulates $\mathcal{F}_{\mathsf{PPC}}^{\hat{\mathcal{L}}(\Delta)}$ for every $\Delta \in \mathbb{N}$ such that (1) the creation of the initial promise instance takes 1 round, and (2) if both parties are honest, every call to instance execution procedure takes $O(1)$ rounds.*

3.5 Lightweight Applications of Programmable Payments

We use programmable payments on PPC to implement many lightweight applications and report the evaluations in Section 3.6. Here, we focus on discussing how PPC helps us implement these applications *as smart contracts*.

HTLC Contract

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

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

`Resolve`():
1. If secretRevealed, then return amount, else return 0.

Fig. 4: HTLC Contract

HTLC. See Figure 4 for an implementation of HTLC promises. The constructor specifies the amount this HTLC is for, and the hash image for which the preimage is requested, and the expiry time by which the preimage must be provided. Observe that these values are specified by the sender of the promise. On sending the preimage to the sender, the receiver will expect a receipt reflecting the updated credit (i.e., an increase by amount). If such a receipt was not provided, then the receiver will deploy the HTLC promise contract onchain[8] and then execute the `RevealSecret` function to lock the final resolved amount to the HTLC amount. On the other hand, if the secret was not revealed, then when the PPC channel closes (which we assume happens after the HTLC expiry), the resolve function will return zero.

Reverse HTLC. See Figure 5 for an implementation of the reverse HTLC promise. In reverse HTLC, the sender commits to revealing a hash preimage within a given expiry time or else stands to lose the promise amount to the receiver. (Note that the roles are somewhat reversed in a regular HTLC promise.) This is a useful promise in, e.g., committing a reservation.

To implement reverse HTLC promise, the sender initializes the promise with the amount, the hash image, the expiry time, and the address of the receiver. Then the sender would reveal the hash preimage to the receiver offchain, and provide a receipt amount (reflecting a zero increase in credit). However, unlike a HTLC promise, here the sender additionally expects an acknowledgment from the receiver that they received the preimage (in the form of a signature on the preimage). If the acknowledgment is received, then the sender is assured that the promise will resolve to zero (since it can always call `SubmitAck` if the promise gets deployed onchain after the expiry time), and concludes the promise execution. Otherwise, the sender continues the promise execution onchain by deploying the reverse HTLC promise via the PPC contract, and then calling the `RevealSecret` method. This ensures that the promise will resolve to zero. Thus, reverse HTLC is an example (different from HTLC) where the sender might have to deploy the promise onchain.

[8] Note that the deployment byteCode already contains the constructor arguments hardcoded in it.

Reverse HTLC Contract

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

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

`SubmitAck`(secret, sig):
1. Require `Hash`(secret) = hash and `SigVerify`(sig, secret, receiver);
2. Set ackSubmitted ← T;

`Resolve`():
1. If secretRevealed or ackSubmitted, then return 0;
2. Return amount.

Fig. 5: Reverse HTLC Contract

On-chain Event Betting

`Init`(amt′, threshold′, tMin′, tMax′):
1. Set (amount, threshold, tMin, tMax) ← (amt′, threshold, tMin′, tMax′);
2. Set roundID ← 0

`SetRoundID`(roundID′):
1. Require tMax ≥ `getTimestamp`(roundID′) ≥ tMin;
2. Set roundID ← roundID′

`Resolve`():
1. If roundID = 0, return 0
2. (price, timestamp) ← `eth-usd.data.eth.getRoundData`(roundID)
3. If price > threshold and timestamp > 0

Fig. 6: Onchain event betting

On-chain Event Betting. See Figure 6 for an example promise where the sender is betting that the price of Ethereum will not go above a certain threshold (say, $2,000) within a certain time period. In such a scenario, the party can send a promise that reads the price of Ethereum on-chain from an oracle (e.g., `eth-usd.data.eth`). This is an example of a promise that depends on the state of external onchain contracts. In such cases, it is important to design the promise carefully as the external contract may change state and cause offchain and onchain execution of promises to be different. Thus we use the function `getRoundData` (say, instead of `latestPrice`). This way, suppose the receiver does not send an acknowledgment that the price was indeed above the threshold (i.e., a receipt reflecting the updated credit), then the sender can deploy the promise onchain (without worrying about the exact block in which its promise will appear). In the example, we assume that the roundID values are calculated offchain and correspond to a time duration that both parties agree on.

Table 1: 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	Reveal	66,340
Close	44,324	Withdraw	71,572

Table 2: The gas usage of the different functions of various applications. *:For Resolve functions we report the execution costs as these functions are view functions. +: The Reveal functions in the RockPaperScissor contracts need to be called twice to reveal the commitments for both parties.

HTLC		**ReverseHTLC**		**OnchainBetting**	
Deploy	222,795	Deploy	423,265	Deploy	442,479
Reveal	28,391	Reveal	28,413	checkPrice	48,093
Resolve*	4,582	SubmitAck	30,247	Resolve*	4,632
		Resolve*	2,499		
RockPaperScissor		**RockPaperScissor-P1**		**RockPaperScissor-P2**	
Deploy	534,167	Deploy	598,088	Deploy	381,537
Reveal$^+$	34,887	Reveal$^+$	34,773	Resolve*	16,937
Resolve*	9,571	Resolve*	6,573		

3.6 Implementation and Evaluation

PPC Gas Usage Costs. We implemented the PPC contract presented in Figure 3 in Solidity. We evaluate our implementation in terms of Ethereum gas usage. The PPC contract requires $3,243,988$ gas to be deployed on the Ethereum blockchain. While we did not aim to optimize gas costs. the PPC contract is already comparable to other simple payment channel deployments 2M+ and 3M+ gas for Perun [17] and Raiden [3][9] respectively. The gas usage for the remaining functions of the contract are reported in Table 1.

HTLC Application. 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 onchain 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 (cf. Table 1), we see that the pessimistic case costs about 700K more gas to resolve the promise. We were able to achieve 110 TPS for the HTLC application end-to-end on a laptop running 2.6 GHz 6-Core Intel Core i7. The end-to-end process included

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

random secret creation, hashing of secret, promise creation/verification, secret reveal/verification, and receipt creation/verification.

Other Applications. For the sake of completeness, we include gas usage costs for other applications presented in Section 3.5, i.e., reverse HTLC, onchain event betting, and rock-paper-scissors (cf. Appendix K) in Table 2. For the rock-paper-scissors, we provide two implementations: one using the compiler (cf. Section 4), and one without (i.e., the ad-hoc implementation in Appendix K). This is to emphasize that our SC from PPC compiler that we present next is highly efficient. Note that all this (i.e., gas cost) is relevant only when one of the parties is malicious. When both parties are honest, the executions are always offchain, and the application-specific onchain deployment costs are zero.

Comparing with Prior State Channels. Prior works on state channels (e.g., [4, 18, 26]) do not provide concrete implementations, performance numbers, or benchmarks. However, we note that, at the very least, state channel implementations typically require explicit signature verification on the application contract—something we avoid in most of our applications above. Furthermore, in multiparty applications where each party has a PPC channel with an untrusted hub, the onchain complexity in the worst case is only proportional to the number of malicious parties as opposed to the total number of parties as in the case with state channels.

4 State Channels from $\mathcal{F}_{\mathsf{PPC}}$

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. In this section, we first formalize a variant of state channels that we call $\mathcal{F}_{\mathsf{SC}}$ that is very similar to PPC. Then we provide a construction that compiles a contract instance input to $\mathcal{F}_{\mathsf{SC}}$ into two promises that can be input to $\mathcal{F}_{\mathsf{PPC}}$. That is, we show how to efficiently realize $\mathcal{F}_{\mathsf{SC}}$ in the $\mathcal{F}_{\mathsf{PPC}}$-hybrid model.

4.1 Modifying $\mathcal{F}_{\mathsf{PPC}}$ to Capture State Channels

Our formalization of programmable payment channels is heavily inspired by the formalization of state channels in [18]. In fact, $\mathcal{F}_{\mathsf{PPC}}$ can be easily modified to yield a *variant* of state channel functionality $\mathcal{F}_{\mathsf{SC}}$, which can be used to execute any two-party contract offchain. We call these contracts *covenant*s. Note that the ideal functionality for state channels $\mathcal{F}_{\mathsf{SC}}$ allows the following operations: (1) opening a (state) channel, (2) creating a covenant instance, (3) executing a covenant instance, and (4) closing the channel. Covenant instances, unlike promise instances, do not have a designated sender or receiver. Like $\mathcal{F}_{\mathsf{PPC}}$, any number of covenant instances can be created and executed using $\mathcal{F}_{\mathsf{SC}}$. Unlike

$\mathcal{F}_{\mathsf{PPC}}$ though, the ideal functionality $\mathcal{F}_{\mathsf{SC}}$ accepts a covenant creation operation from a party only if the other party consents to it. The covenant instances allowed by $\mathcal{F}_{\mathsf{SC}}$ resolve to two integer values (that corresponds to the payout of each party). Again, this resolved value is calculated at the time of channel closing, and then the resolved values of all contract instances are aggregated to determine the final settlements.

4.2 Defining $\mathcal{F}_{\mathsf{SC}}$

Just as how $\mathcal{F}_{\mathsf{PPC}}$ creates and executes promise instances, we will have $\mathcal{F}_{\mathsf{SC}}$ create and execute *covenant* instances.

Covenant Instance. A covenant instance can be viewed as a special contract instance consisting of two attributes: *covenant storage* (accessed by key $\mathsf{storage}$) and *covenant code* (accessed by key code). Covenant storage σ is an attribute tuple containing at least the following attributes: (1) $\sigma.resolve_A \in \mathbb{R}_{\geq 0}$ denotes the amount of money transferred from party B to party A; and (2) $\sigma.resolve_B \in \mathbb{R}_{\geq 0}$ denotes the amount of money transferred from party A to party B. Covenant code is a tuple $C := (\Lambda, \mathsf{Construct}, f_1, \ldots, f_s)$ similar to contract code. W.l.o.g., we assume $\mathsf{Construct}$ does not take caller as inputs but it can be incorporated into y. We note that we do not restrict the independence of the constructor.

See Figure 7 for the definition of the ideal functionality that captures state channels. Like $\mathcal{F}_{\mathsf{PPC}}$, the functionality $\mathcal{F}_{\mathsf{SC}}$ contains the following 4 procedures.

(1) *State channel creation.* Similar to $\mathcal{F}_{\mathsf{PPC}}$, a party can instantiate a channel with another party by sending the channel creation information to $\mathcal{F}_{\mathsf{SC}}$. The operation of this procedure is identical to that of $\mathcal{F}_{\mathsf{PPC}}$.

(2) *Covenant Creation.* The covenant instance is specified by choice of channel γ, contract code C and arguments for the constructor function y, and a salt z that is used to identify this promise instance. Among other things, the ideal functionality ensures that $cid := (id, C, y, z)$ does not exist in $\gamma.\mathsf{cspace}$. Note that unlike $\mathcal{F}_{\mathsf{PPC}}$, we need an acknowledgment from the counterparty before creating a covenant instance. Thus, the creation takes more rounds but optimistically remains $O(1)$.

(3) *Covenant Execution.* This procedure is used to update the covenant instance's storage. The operation of this procedure is identical to that of $\mathcal{F}_{\mathsf{PPC}}$.

(4) *State Channel Closure.* When a party of the channel instance γ wants to close the channel, $\mathcal{F}_{\mathsf{SC}}$ will wait for $\gamma.duration$ rounds to execute the remaining covenants that have not been finalized. The crucial difference from $\mathcal{F}_{\mathsf{PPC}}$ is in the way in which the credits are calculated (simply because of the difference in the final values of covenant instances vs. promise instances). We note that the closure requires extra $O(\Delta)$ rounds. Looking ahead, this is because we "compile" a covenant into two promises on $\mathcal{F}_{\mathsf{PPC}}$, and require an extra function call to settle down the resolved values of them.

Remarks. Our state channel ideal functionality differs from prior formalizations in many ways. Crucially, it makes explicit the dependence of covenant instances

Functionality $\mathcal{F}_{\mathsf{SC}}^{\hat{\mathcal{L}}(\Delta)}$

State channel opening

Identical to programmable payment channel opening in $\mathcal{F}_{\mathsf{PPC}}^{\hat{\mathcal{L}}(\Delta)}$ but with a state channel of covenants (saved in cspace) as inputs.

Covenant creation

Upon $(\texttt{create}, id, C||y, z) \overset{t_0}{\hookleftarrow} P$, let $\gamma := \Gamma(id)$ and let $cid := (id, C, y, z)$. If $\gamma = \bot$ or $P \notin \gamma.\mathsf{endusers}$ or $\gamma.\mathsf{cspace}(pid) \neq \bot$ then stop. Else let $Q := \gamma.\mathsf{otherparty}(P)$.

- If $(\texttt{create}, id, C||y, z) \overset{t_0}{\hookleftarrow} Q$ and P, Q are honest or the the simulator behaves honestly, then let $\nu := \bot$ and $\sigma := C.\mathsf{Construct}(t_0, y)$. Stop if $\sigma = \bot$. Within 7 rounds, set $\nu.\mathsf{code} := C$ and $\nu.\mathsf{storage} := \sigma$. Set $\Gamma(id).\mathsf{cspace}(cid).\mathsf{flag} = 0$. Set $\Gamma(id).\mathsf{cspace}(cid) := \nu$. Send $(\texttt{instance-created}, id, cid, \nu) \overset{t \leq t_0+7}{\hookrightarrow} \gamma.\mathsf{endusers}$.
- If $(\texttt{create}, id, C||y, z) \overset{t_0}{\hookleftarrow} Q$ and one party is malicious, then let $\nu := \bot$ and $\sigma := C.\mathsf{Construct}(t_0, y)$. Stop if $\sigma = \bot$. Within $4\Delta + 7$ rounds, set $\nu.\mathsf{code} := C$ and $\nu.\mathsf{storage} := \sigma$. Set $\Gamma(id).\mathsf{cspace}(cid).\mathsf{flag}$ by the simulator. Set $\Gamma(id).\mathsf{cspace}(cid) := \nu$. Send $(\texttt{instance-created}, id, cid, \nu) \overset{t \leq t_0+4\Delta+7}{\hookrightarrow} \gamma.\mathsf{endusers}$.

Covenant execution

Identical to promise instance execution in $\mathcal{F}_{\mathsf{PPC}}^{\hat{\mathcal{L}}(\Delta)}$ but with a state channel identity and a covenant identity as inputs.

State 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 $2\gamma.duration + 11\Delta + 5$ rounds and proceed as follows: (Note that either Alice or Bob could be P)

1. Calculate

$$total := \gamma.\mathsf{cash}(\gamma.\mathsf{Alice}) + \gamma.\mathsf{cash}(\gamma.\mathsf{Bob})$$
$$credit_A := \sum(\gamma.\mathsf{pspace}(pid).\mathsf{storage}.resolve_A)$$
$$credit_B := \sum(\gamma.\mathsf{pspace}(pid).\mathsf{storage}.resolve_B)$$

2. 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.
3. Send $(\texttt{contract-close}, id) \overset{t_1 \leq t_0+12\Delta+2\gamma.duration+5}{\hookrightarrow} \gamma.\mathsf{endusers}$.

Fig. 7: The ideal functionality $\mathcal{F}_{\mathsf{SC}}^{\hat{\mathcal{L}}(\Delta)}$.

on other onchain contracts. Also, a covenant instance can depend on other covenant instances (this is something not considered in prior works).

4.3 Implementing $\mathcal{F}_{\mathsf{SC}}$ in the $\mathcal{F}_{\mathsf{PPC}}$-hybrid World

Perhaps surprisingly, $\mathcal{F}_{\mathsf{PPC}}$ can be used to implement $\mathcal{F}_{\mathsf{SC}}$. In particular, a covenant can be compiled into two promises on $\mathcal{F}_{\mathsf{PPC}}$ that can be used to execute the covenant offchain.

To implement a covenant creation of a contract c in $\mathcal{F}_{\mathsf{SC}}$, we use two promises p_0, p_1, one from each endpoint of $\mathcal{F}_{\mathsf{PPC}}$. The promise p_0 contains all the logic of the covenant instance c. Note that c will resolve to either $(k, 0)$ or $(0, k)$ (or any other intermediate value), where k is non-negative. In particular, $(k, 0)$ denotes that the first party needs to pay k to the second party and $(0, k)$ denotes that the second party needs to pay k to the first party. Note that the resolved state of c will be saved in p_0 as well. Accordingly, p_0 will resolve to 0 in the case of $(0, k)$, otherwise as k. The resolve method of promise p_1 will instead read the state of p_0, and resolves in the opposite direction. That is, p_1 resolves to 0 in the case of $(k, 0)$, otherwise as k. That both parties consent to the contract instance is captured by requiring each party to provide its promise.

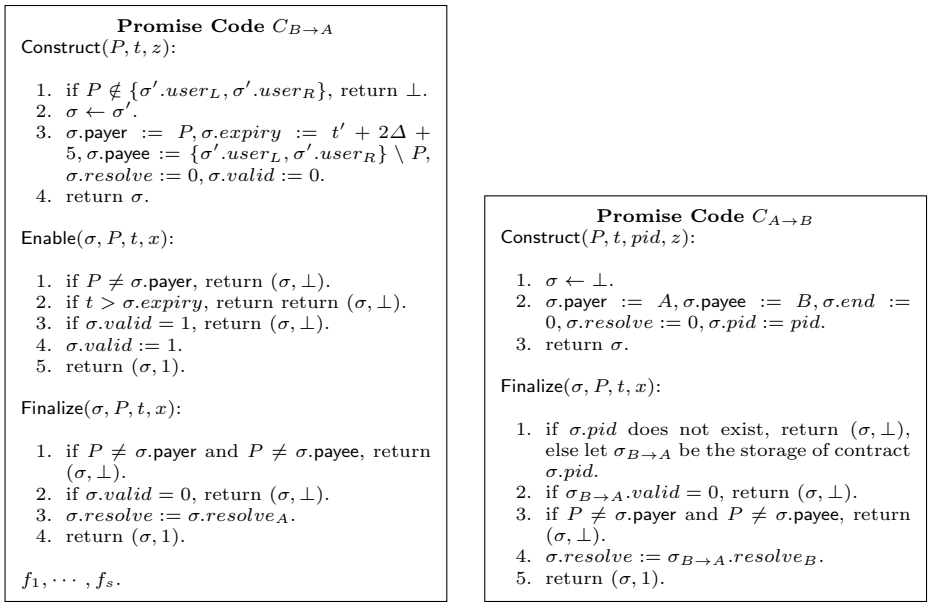
We illustrate this with an example of two-party contract for chess. We assume that each party puts in \$50, and the winner gets \$100. Assume that there exists a smart contract c that contains the entire logic of chess (i.e., checking validity of a move, checking whether the game has ended, who has won the game, and the payout to each party, etc.).

To play a game of chess offchain, parties each first create a promise. The promise from Bob contains all the logic in c and additionally has a resolve method which will depend on the payout logic in c in the following way: if the winner is Alice, then the resolve method returns \$50, else it returns zero. The promise from Alice is such that the resolve method invokes the resolve method of Bob's promise to get value v and returns $\$50 - v$ as the resolved amount.

There exists a protocol that can implement $\mathcal{F}_{\mathsf{SC}}$ in the $\mathcal{F}_{\mathsf{PPC}}$-hybrid model. The essential step is to compile a covenant into two associated promises (cf. Figure 8) and then execute them on $\mathcal{F}_{\mathsf{PPC}}$. We present this formally as follows.

Theorem 2. *There exists protocol Π_{SC} working in $\mathcal{F}_{\mathsf{PPC}}$-hybrid model that emulates the ideal functionality $\mathcal{F}_{\mathsf{SC}}^{\hat{\mathcal{L}}(\Delta)}$ for every $\Delta \in \mathbb{N}$. Note furthermore that the the protocol Π_{SC} requires only three invocations of $\mathcal{F}_{\mathsf{PPC}}$ to create a covenant.*

Similar to Theorem 1, Theorem 2 can be formally proved by constructing straightforward simulators to translate between covenant and associated promises. Note that the crucial point is to argue the rounds taken by the two worlds are identical. We provide the formal description of the protocol and its analysis in Appendix I and the protocol analysis in Appendix J.



Promise Code $C_{B\to A}$

Construct(P, t, z):

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 $:= \{\sigma'.user_L, \sigma'.user_R\} \setminus P$, $\sigma.resolve := 0, \sigma.valid := 0$.
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. $\sigma.resolve := \sigma.resolve_A$.
4. return $(\sigma, 1)$.

$f_1, \cdots, f_s$.

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

Promise Code $C_{A\to B}$

Construct(P, t, pid, z):

1. $\sigma \leftarrow \perp$.
2. σ.payer $:= A, \sigma$.payee $:= B, \sigma.end := 0, \sigma.resolve := 0, \sigma.pid := pid$.
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 $\sigma_{B\to A}.valid = 0$, return $(\sigma, \perp)$.
3. if $P \neq \sigma$.payer and $P \neq \sigma$.payee, return $(\sigma, \perp)$.
4. $\sigma.resolve := \sigma_{B\to A}.resolve_B$.
5. return $(\sigma, 1)$.

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

Fig. 8: The compiled promises from a covenant code C at time t' and constructor inputs y, where $\sigma' := C.\mathsf{Construct}(t', y)$. $C_{B\to A}$ will hard-code σ'.

5 Conclusions

In this paper we present programmable payment channels (PPC), a new abstraction that enables payment channels to support lightweight applications encoded in the form of smart contracts. We show the usefulness of PPC by constructing several example applications. Our gas cost estimates show us that the application implementations are indeed practical on Ethereum (or other EVM chains). Finally, we also present a modified version of state channels and show how PPC can also implement state channel applications efficiently.

Acknowledgments

We thank Pedro Moreno-Sanchez for many useful discussions and insightful comments.

References

1. Hash time locked contracts - bitcoin wiki. https://en.bitcoin.it/wiki/Hash_Time_Locked_Contracts, (Accessed on 10/20/2023)

2. Payment channels - bitcoin wiki. https://en.bitcoin.it/wiki/Payment_channels, (Accessed on 10/20/2023)
3. Raiden. https://raiden.network/, (Accessed on 10/20/2023)
4. State channels - ethereum.org. https://ethereum.org/en/developers/docs/scaling/state-channels/, (Accessed on 10/20/2023)
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. Breidenbach, L.: libsubmarine. https://github.com/lorenzb/libsubmarine (2018)
7. Breidenbach, L., Daian, P., Tramèr, F., Juels, A.: Enter the hydra: Towards principled bug bounties and Exploit-Resistant smart contracts. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 1335–1352. USENIX Association, Baltimore, MD (Aug 2018), https://www.usenix.org/conference/usenixsecurity18/presentation/breindenbach
8. 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)
9. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: Theory of Cryptography Conference. pp. 61–85. Springer (2007)
10. Christodorescu, M., English, E., Gu, W.C., Kreissman, D., Kumaresan, R., Minaei, M., Raghuraman, S., Sheffield, C., Wijeyekoon, A., Zamani, M.: Universal payment channels: An interoperability platform for digital currencies (2021). https://doi.org/10.48550/ARXIV.2109.12194, https://arxiv.org/abs/2109.12194
11. 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
12. Close, T.: Nitro protocol. Cryptology ePrint Archive (2019)
13. Close, T., Stewart, A.: Forcemove: an n-party state channel protocol. Magmo, White Paper (2018)
14. Coleman, J., Horne, L., Xuanji, L.: Counterfactual: Generalized state channels. Accessed: http://l4.ventures/papers/statechannels.pdf **4**, 2019 (2018)
15. 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)
16. 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)
17. 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)
18. 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)
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. Khalil, R., Gervais, A.: Nocust-a non-custodial 2nd-layer financial intermediary. (2018)
22. 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)
23. 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)
24. Malavolta, G., Moreno-Sanchez, P., Schneidewind, C., Kate, A., Maffei, M.: Anonymous multi-hop locks for blockchain scalability and interoperability. In: NDSS (2019)
25. McCorry, P., Buckland, C., Yee, B., Song, D.: Sok: Validating bridges as a scaling solution for blockchains. Cryptology ePrint Archive (2021)
26. 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)
27. 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/
28. Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments. http://lightning.network/lightning-network-paper.pdf (2016), (Accessed on 10/20/2023)
29. Roos, S., Moreno-Sanchez, P., Kate, A., Goldberg, I.: Settling payments fast and private: Efficient decentralized routing for path-based transactions. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018. The Internet Society (2018), http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-3_Roos_paper.pdf
30. Tairi, E., Moreno-Sanchez, P., Maffei, M.: a^2l: Anonymous atomic locks for scalability in payment channel hubs. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1834–1851 (2021). https://doi.org/10.1109/SP40001.2021.00111
31. Thibault, L.T., Sarry, T., Hafid, A.S.: Blockchain scaling using rollups: A comprehensive survey. IEEE Access **10**, 93039–93054 (2022). https://doi.org/10.1109/ACCESS.2022.3200051
32. 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 10/20/2023)
33. Yee, B., Song, D., McCorry, P., Buckland, C.: Shades of finality and layer 2 scaling. arXiv preprint arXiv:2201.07920 (2022)

SUPPLEMENTARY MATERIAL

A More Details on the Compiler

See Figure 9 for how to compile a chess application into two interlocked promises.

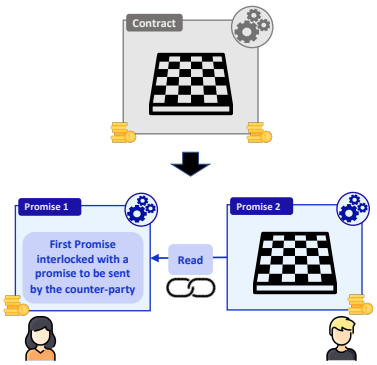


Fig. 9: Execute arbitrary two-party contract on PPC. Compiler compiles the contract into two interlocked promises.

B Comparsion with Rollups

Rollups are the most popular Layer-2 scaling solutions right now (on Ethereum). The high-level idea is that there is a special entity (or entities) called *sequencers* that aggregate transactions from multiple users on a separate Layer-2 system and then submits batched transactions every once in a while to Layer-1. Rollups come in two flavors: optimistic rollups or zk rollups. Both variants share many similarities with (programmable) payment channels (or state channels) such as:

- there is a smart contract on Layer-1 on which parties deposit money, and this smart contract controls the balances available on Layer-2 ;
- there are forced transactions that happen on Layer-1 if there is misbehavior or unavailability of parties (e.g., *sequencers* on rollups) on Layer-2.

On the other hand, there are significant differences too, such as:

- the sequencer in an optimistic rollup needs to keep submitting *all* the transactions (albeit in some compressed form) that it obtained (in order to prove correctness and also for *data availability*), whereas in a payment channel or state channel only the final states need to be submitted;

– the sequencer in a zk rollup needs to provide time-consuming zk proofs about validity of state changes that it submits (and in some cases the proof itself spans multiple Layer-1 blocks), whereas no such overheads exist for payment channels (barring signature verification).

We highlight that PPC allows for Layer-2 contracts to interact/depend directly with Layer-1 contracts but such a thing is not possible for rollups since they usually function as an independent blockchain systems. That said, rollups can have Layer-2 contracts which interact with each other, but there are some additional restrictions on the size of the contracts due to the use of some special opcodes which need to be translated into contract calls on Layer-1.

C Other Works Using CREATE2

Coinbase Commerce uses CREATE2 on the state channel to reuse a *fixed* contract called Forwarder to process instant offchain transactions[10]. 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. Breidenbach et al. also introduced the use of CREATE2 opcode to ensure fairness in the Hydra system [7], a new model for auto-payout bug bounty system for finding vulnerabilities in smart contracts. In their work, CREATE2 functions as a commit then reveal mechanism (i.e., libsubmarine [6]) that prevents front-running adversaries from stealing the bounty opportunities from honest users.

D Supplementary Material for Notations and Models

D.1 Global Universal Composable Framework

UC models the execution of protocols as interactions of probabilistic polynomial-time (PPT) *Interactive 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 desired security property 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 $\text{EXEC}^{\hat{\mathcal{L}}(\Delta),\mathcal{G}}_{\pi,\mathcal{A},\mathcal{E}}(1^\lambda, z)$. 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 $\text{IDEAL}^{\hat{\mathcal{L}}(\Delta)}_{\mathcal{F},\mathcal{S},\mathcal{E}}(1^\lambda, z)$.

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*

[10] https://legacy.ethgasstation.info/blog/what-is-create2/

$$\{\text{EXEC}^{\hat{\mathcal{L}}(\Delta),\mathcal{G}}_{\pi,\mathcal{A},\mathcal{E}}(1^\lambda, z)\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}} \overset{c}{\approx} [11]\{\text{IDEAL}^{\hat{\mathcal{L}}(\Delta)}_{\mathcal{F},\mathcal{S},\mathcal{E}}(1^\lambda, z)\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}}$$

D.2 The Global Ledger Functionality

The global ledger functionality is shown in Figure 10.

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

Functionality $\hat{\mathcal{L}}$, running with parties $P_1, ..., P_n$ and several ideal functionalities $\mathcal{G}_1, ..., \mathcal{G}_\mathsf{m}$, 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 $\mathcal{F}_\mathsf{j}$ $(j \in [m], y \in \mathbb{R}_{\geq 0})$, 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 $\mathcal{F}_\mathsf{j}$ $(j \in [m], y \in \mathbb{R}_{\geq 0})$:

- 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. 10: The global ledger functionality $\hat{\mathcal{L}}(\Delta)$.

E Implementing $\mathcal{F}_{\mathsf{PPC}}$

We detail the programmable payment channel protocol in this subsection. In particular, we focus on describing the protocol intuitions and defer the formal description to Appendix G. The protocol Π is defined assuming access to an ideal programmable payment channel smart contract functionality $\mathcal{G}_{\mathsf{PPC}}$. Note $\mathcal{G}_{\mathsf{PPC}}$ is the PPC smart contract. All specifications in Π should be viewed as the procedure executed on the party's local machine while all specifications in $\mathcal{G}_{\mathsf{PPC}}$ should be viewed as onchain smart contracts. Since $\mathcal{G}_{\mathsf{PPC}}$ emulates smart contracts, coins in $\hat{\mathcal{L}}(\Delta)$ 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

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

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.

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 offchain, 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 to obtain a total ordering for the states of promise instance. We define that the initial promise storage is of version 0.

Register. A special sub-procedure (not interacting with environment) called Register is used to deploy a promise instance onchain. 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 G.

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

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 $\mathcal{G}_{\mathsf{PPC}}$ 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 onchain, he directly calls the function on Layer-1. Otherwise, P will first try to peacefully (offchain) 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 offchain in $O(1)$ rounds. If not, P needs to execute the function onchain. 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 onchain to invalidate the version P wants to register. If after the registration sub-procedure, the promise instance onchain is still P's version, P can then update it by onchain function execution. We follow [18] 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 offchain in $\gamma^P.\mathsf{pspace}$ within 3Δ rounds. Then P will notify the PPC contract $\mathcal{G}_{\mathsf{PPC}}$ 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 $\gamma^Q.\mathsf{pspace}$ should already be registered by P. After the time window passes, $\mathcal{G}_{\mathsf{PPC}}$ 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.

F Simplifications

In this section, we provide and justify the full simplifications we made in Section 3 for better presentation:

- 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 contract address.
- We use collision-resistant hash functions to capture the CREATE2 opcode in Ethereum. Basically, every promise instance will be associated with an identifier pid determined by a collision-resistant hash function applied to the the following inputs: (1) creator channel identifier id; (2) payer P; (3) promise code C; (4) arguments for the constructor function y; and (5) a salt value z. Note that the arguments specified above are in line with the specification of CREATE2. The salt captures the fact that 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 collision resistance ensures that it is computationally *infeasible* to create a second contract whose address is also pid. This is also why we can avoid making any references to such hash functions and use just their corresponding input tuples in the descriptions of our ideal functionalities and the UC protocols realizing them.
- 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 instantiate this parallelism using an accumulator.

Procedure $\mathsf{Register}(P, id, pid)$

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

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 $(id, pid, C, y, z, A, s_A)$ from P's local memory, then send $(\texttt{instance-construct}, id, C, y, z, A, s_A) \overset{t_0}{\hookrightarrow} \mathcal{G}_{\mathsf{PPC}}$.
   - If $\mathsf{version} \neq 0$ (i.e., this is a non-initial promise instance), send $(\texttt{instance-register}, id, pid, \nu^P, \mathsf{version}, s_P, s_Q) \overset{t_0}{\hookrightarrow} \mathcal{G}_{\mathsf{PPC}}$.

   Party Q upon $(\texttt{instance-registering}, id, pid) \overset{t_1}{\hookleftarrow} \mathcal{G}_{\mathsf{PPC}}$
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 onchain), send $(\texttt{instance-register}, id, pid, \perp, -1, \perp, \perp) \overset{t_1}{\hookrightarrow} \mathcal{G}_{\mathsf{PPC}}$.
   - If $\mathsf{version} = 0$ (i.e., this is an initial promise instance), fetch the constructing tuple $(id, pid, C, y, z, A, s_A)$ from Q's local memory, then send $(\texttt{instance-construct}, id, C, y, z, A, s_A) \overset{t_1}{\hookrightarrow} \mathcal{G}_{\mathsf{PPC}}$.
   - If $\mathsf{version} \neq 0$ (i.e., this is a non-initial promise instance), send $(\texttt{instance-register}, id, pid, \nu^Q, \mathsf{version}, s_Q, s_P) \overset{t_1}{\hookrightarrow} \mathcal{G}_{\mathsf{PPC}}$.
3. Goto step 5.

Back to party P

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

For both parties T

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

Fig. 11: The sub-procedure Register to register a promise instance onchain.

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

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

1. Let $pid := (id, C, A, y, z)$. Stop if one of the following conditions holds: $\gamma = \bot$; $P \notin \gamma.\mathsf{endusers}$; $A \notin \gamma.\mathsf{endusers}$; $\gamma.\mathsf{pspace}(pid) \neq \bot$; $\mathsf{Vfy}_{pk_A}(id, C, y, z, A; s_A) = 0$.
2. Let $\nu := \bot$. Set $\nu.\mathsf{code} := C$. Let $\sigma := C.\mathsf{construct}(A, t_0, y)$, stop if $\sigma = \bot$ 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: $\gamma = \bot$; $P \notin \gamma.\mathsf{endusers}$; $\gamma.\mathsf{pspace}(pid) \neq \bot$.
2. Let $Q := \gamma.\mathsf{otherparty}(P)$. If $\mathsf{version} = -1$ and $\nu = \bot$ and the functionality's memory contains a tuple $(Q, id, pid, \cdots)$, goto step 4.
3. Stop if one of the following conditions holds: $\mathsf{Vfy}_{pk_P}(id, pid, \nu, \mathsf{version}; s_P) = 0$; $\mathsf{Vfy}_{pk_Q}(id, pid, \nu, \mathsf{version}; s_Q) = 0$.
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 $\gamma = \bot$ 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. 12: The contract associated with the sub-procedure Register.

G Formal PPC Protocols

In this section, we fully specify the PPC protocols described in Appendix E. The protocols are formalized in the UC framework [8], which UC-emulate $\mathcal{F}_{\mathsf{PPC}}$ (see Figure 2).

The special sub-procedure Register, which is used to deploy a promise instance onchain, is specified in Figures 11 and 12. The online contracts functionality $\mathcal{G}_{\mathsf{PPC}}$ 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 onchain and the honest receiver will trivially accept it. Note that 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 Π: Open a programmable payment channel

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

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

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

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

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

Back to party P

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

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

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 $\gamma.\mathsf{cash}(Q) < 0$; $\gamma.\mathsf{pspace} \neq \{\}$; $\gamma.\mathit{duration} < 0$.
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 $\mathit{tamp} := (t_0, \gamma)$.

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

1. Stop if one of the following conditions holds: there is no pair $\mathit{tamp} := (t_0, \gamma)$ 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 $\mathit{tamp} := (t_0, \gamma)$ 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 Π: Create an initial promise instance

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

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

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

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

6. Set $\mathit{pid} := (id, C, P, y, z)$
7. Stop if one of the following conditions holds: $\Gamma^Q(id) = \bot$; $P \notin \Gamma^Q(id).\mathsf{endusers}$; $\Gamma^Q(id).\mathsf{pspace}(\mathit{pid}) \neq \bot$. Else let $\gamma := \Gamma^Q(id)$.
8. Let $\sigma := C.\mathsf{Construct}(P, t_1 - 1, y)$. Stop if $\sigma = \bot$. Let $\nu := \bot$. Set $\nu.\mathsf{code} := C$ and $\nu.\mathsf{storage} := \sigma$. Stop if $\mathsf{Vfy}_{pk_P}(id, C, y, z, P; s_P) \neq 1$.

9. Set $\Gamma^Q(id).\mathsf{pspace}(pid) := \nu$ and set $\Gamma^Q_{aux}(id, pid) := (\bot, \bot, 0)$. Save $(id, pid, C, y, z, P, s_P)$. Output $(\texttt{instance-created}, id, pid, \nu) \overset{t_1}{\hookrightarrow} \mathcal{E}$.

Protocol Π: Promise instance execution

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

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 $t_1 \equiv 1 (\mathrm{mod}\ 4)$ 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 marked as registered in Γ^P_{aux}, goto step 12 at round t_1.
4. Compute $(\tilde{\sigma}, m) := f^{\mathcal{G}, \gamma^P}(\sigma^P, P, t_0, z)$. 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 $(\texttt{peaceful-request}, id, pid, f, z, s_P, t_0) \overset{t_1}{\hookrightarrow} Q$. 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 $\Gamma^Q(id) = \bot$, else let $\gamma^Q := \Gamma^Q(id)$. Stop if $Q \notin \gamma^Q.\mathsf{endusers}$ or $P \notin \gamma^Q.\mathsf{endusers}$ or $\gamma^Q(pid) = \bot$ or (id, pid) is marked as registered, 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 $t_Q \not\equiv 2 (\mathrm{mod}\ 4)$" or "$P = \gamma^Q.\mathsf{Bob}$ and $t_Q \not\equiv 0 (\mathrm{mod}\ 4)$".
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 $(\tilde{\sigma}, m) := f^{\mathcal{G}, \gamma^Q}(\sigma^Q, P, t_0, z)$. Stop if $m = \bot$.
   (b) Set $\tilde{\mathsf{version}} := \mathsf{version}^Q + 1$. Let $\tilde{\nu} := \bot$. 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). For all γ^P.$\mathsf{pspace}(pid')$ where pid' is not registered, execute $\mathsf{Register}(P, id, pid')$ (in parallel) 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 $(\texttt{instance-execute}, id, pid, f, z) \hookrightarrow \mathcal{G}_{\mathsf{PPC}}$.

For both parties T

13. If $(\texttt{executed-onchain}, id, pid, \mathsf{Caller}, f, t, z, \hat{\nu}) \overset{t_4 \leq t_0+4\Delta+5}{\hookleftarrow} \mathcal{G}_{\mathsf{PPC}}$, 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 $\mathcal{G}^{\hat{\mathcal{L}}(\Delta)}_{\mathsf{PPC}}$

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 $(\hat{\sigma}, m) := f^{\mathcal{G},\gamma}(\sigma, P, t_1, z)$. Stop if $m = \bot$.
4. Set $\Gamma(id).\mathsf{pspace}(pid).\mathsf{storage} := \hat{\sigma}$ and send $(\texttt{executed- onchain}, id, pid, P, f, t_1, z, \hat{\nu}) \overset{t_1}{\hookrightarrow} \gamma.\mathsf{endusers}$.

Protocol Π: Close a programmable payment channel

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

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 $(\texttt{contract-close}, id) \overset{t_1 \leq t_0+3\Delta}{\hookrightarrow} \mathcal{G}_{\mathsf{PPC}}$.

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

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

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

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

1. Stop if $\gamma = \perp$ 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. 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 $credit_A := \sum_{\gamma.\mathsf{pspace}(pid).\mathsf{storage.payer}=\gamma.\mathsf{Bob}}(\gamma.\mathsf{pspace}\ (pid).\mathsf{storage}.resolve)$.
   (c) Set $credit_B := \sum_{\gamma.\mathsf{pspace}(pid).\mathsf{storage.payer}=\gamma.\mathsf{Alice}}(\gamma.\mathsf{pspace}\ (pid).\mathsf{storage}.resolve)$.
   (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 $(\texttt{contract-close}, \texttt{id}) \hookrightarrow \gamma.\mathsf{endusers}$.

H 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. There exists a protocol working in $\mathcal{G}_{\mathsf{PPC}}^{\hat{\mathcal{L}}(\Delta)}$-hybrid model that emulates $\mathcal{F}_{\mathsf{PPC}}^{\hat{\mathcal{L}}(\Delta)}$ for every $\Delta \in \mathbb{N}$ such that (1) the creation of the initial promise instance takes 1 round, and (2) if both parties are honest, every call to instance execution procedure takes $O(1)$ rounds.*

Proof. We follow the framework of [18]. We will show that Π UC-emulates the ideal functionality $\mathcal{F}_{\mathsf{PPC}}^{\hat{\mathcal{L}}(\Delta)}$ in the $\mathcal{G}_{\mathsf{PPC}}^{\hat{\mathcal{L}}(\Delta)}$-hybrid model. In other words, for any PPT adversary $\mathcal{A}$, we construct a simulator Sim that operates in the $\mathcal{G}_{\mathsf{PPC}}^{\hat{\mathcal{L}}(\Delta)}$-hybrid model and simulates the $\mathcal{F}_{\mathsf{PPC}}^{\hat{\mathcal{L}}(\Delta)}$-hybrid world to any environment $\mathcal{E}$.

As in [18], since registration of a contract instance is defined as a separate procedure that can be called by parties of the protocol Π, 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., [18]), and hence we omit further description here due to lack of space. □

Simulator Sim: Open a programmable payment channel

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

P is honest and Q is corrupt

Upon $(\texttt{open}, P, \gamma) \overset{t_0}{\hookleftarrow} \mathcal{F}_{\mathsf{PPC}}$:

1. Wait until round $t_1 \leq t_0 + \Delta$, send $(\texttt{initializing}, \gamma) \overset{t_1}{\hookrightarrow} Q$.
2. If $(\texttt{confirm}, \gamma) \overset{t_1 \leq t_1' \leq t_0+\Delta}{\hookleftarrow} Q$, then send $(\texttt{open}, \gamma) \overset{t_1'}{\hookrightarrow} \mathcal{F}_{\mathsf{PPC}}$ on behalf of Q.
3. If $(\texttt{opened}, \gamma) \overset{t_2 \leq t_1'+\Delta}{\hookleftarrow} \mathcal{F}_{\mathsf{PPC}}$, 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 $(\texttt{open}, \gamma) \overset{t_0}{\hookrightarrow} \mathcal{F}_{\mathsf{PPC}}$ on behalf of P.
3. Distinguish the following two situations:
   - If $(\texttt{opened}, \gamma) \overset{t_1 \leq t_0+2\Delta}{\hookleftarrow} \mathcal{F}_{\mathsf{PPC}}$, 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 $(\texttt{refund}, \gamma) \overset{t_2}{\hookrightarrow} \mathcal{F}_{\mathsf{PPC}}$.

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

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

P is honest and Q is corrupt

1. Let $\gamma^P := \Gamma^P(id), \nu^P := \gamma^P.\mathsf{pspace}(pid), (s_P, s_Q, \mathsf{version}) := \Gamma^P_{aux}(id, pid)$.
2. Set t_0 be the current round. Send $(\texttt{instance-registering}, id, pid) \overset{t_1 \leq t_0+\Delta}{\hookrightarrow} Q$.
3. Q can have two following reactions:
   - If $(\texttt{instance-construct}, id, C, y, z, A, s_A) \overset{t_1}{\hookleftarrow} Q$, ignore if $(id, C, A, y, z) \neq pid$ or the signature is invalid. Let $\sigma := C.\mathsf{Construct}(A, t_1, y)$. Ignore if $\sigma = \bot$ or $\sigma.\mathsf{payer} \neq A$. Send $(\texttt{instance-registered}, id, pid, \nu^P) \overset{\leq t_1+\Delta}{\hookrightarrow} Q$, 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 $(\texttt{instance-registered}, id, pid, \tilde{\nu}) \overset{\leq t_1+\Delta}{\hookrightarrow} Q$ and goto step 5.
4. Send $(\texttt{instance-registered}, id, pid, \nu^P) \overset{\leq t_0+3\Delta}{\hookrightarrow} Q$ 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 $(\texttt{instance-construct}, id, C, y, z, A, s_A) \overset{t_0}{\hookleftarrow} P$, ignore if $\Gamma(id) = \bot$ or $P \notin \Gamma(id).\mathsf{endusers}$. Let $pid := (id, C, A, y, z)$. Ignore if (id, pid) is marked as registered

in $\Gamma^{\mathcal{Q}}_{aux}$. Let $\sigma := C.\mathsf{Construct}(A, t_0, y)$. Ignore if $\sigma = \bot$ or $\sigma.\mathsf{payer} \neq A$. Then distinguish the following two situations:

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

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 $\Gamma^{\mathcal{Q}}_{aux}$. Let $\nu^{\mathcal{Q}} := \Gamma^{\mathcal{Q}}(id).\mathsf{pspace}(pid)$ and fetch $\mathsf{version}^{\mathcal{Q}}$ of (id, pid) from $\Gamma^{\mathcal{Q}}_{aux}$. Set $\tilde{\nu} := \mathsf{version} > \mathsf{version}^{\mathcal{Q}}\ ?\ \nu : \nu^{\mathcal{Q}}$. Mark (id, pid) as registered in $\Gamma^{\mathcal{Q}}_{aux}$. Set $\Gamma^{\mathcal{Q}}(id).\mathsf{pspace}(pid) := \tilde{\nu}$. Send $(\texttt{instance-registered}, id, pid, \tilde{\nu}) \overset{\leq t_0+2\Delta}{\hookrightarrow} P$.

Simulator Sim: Create an initial promise instance

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

P is honest and Q is corrupt

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

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

P is corrupt and Q is honest

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

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

Simulator Sim: Promise instance execution

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

P is honest and Q is corrupt

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

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 $(\tilde{\sigma}, m) := f^{\mathcal{G}, \gamma^P}(\sigma^P, P, t_0, z)$. 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 $\mathcal{F}_{\mathsf{PPC}}$ to (keep the promise offchain) execute at time t_0 and output at time $t_1 + 2$ and stop.
   (c) Execute sub-simulator $\mathsf{SimRegister}(P, id, pid')$ for all pid' (in parallel, 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 $\mathcal{F}_{\mathsf{PPC}}$ to (make the promise onchain) execute at time t_0 and output at time t_2 and stop.
   (d) Let t_3 be the current round. Instruct the $\mathcal{F}_{\mathsf{PPC}}$ to (make the promise onchain) execute at time t_3 and output according to the delay. Get $(\texttt{executed}, id, pid, P, f, t, z, \nu) \overset{t_4 \leq t_3 + \Delta}{\hookleftarrow} \mathcal{F}_{\mathsf{PPC}}$. Send $(\texttt{executed-onchain}, id, pid, P, f, t, z, \nu) \overset{t_4}{\hookrightarrow} Q$. Stop.
4. If (id, pid) is marked as registered in Γ^P_{aux}:
   (a) (The promise is already onchain.) Instruct the $\mathcal{F}_{\mathsf{PPC}}$ to execute at time t_1 and output according to the onchain delay. Get $(\texttt{executed}, id, pid, P, f, t, z, \nu) \overset{t_5 \leq t_1 + \Delta}{\hookleftarrow} \mathcal{F}_{\mathsf{PPC}}$. Send $(\texttt{executed-onchain}, id, pid, P, f, t, z, \nu) \overset{t_5}{\hookrightarrow} Q$.

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 $\gamma^Q(pid) = \bot$ or (id, pid) is marked as registered, 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 $(\tilde{\sigma}, m) := f^{\mathcal{G}, \gamma^Q}(\sigma^Q, P, t_0, z)$. Stop if $m = \bot$.
   (b) Set $\tilde{\mathsf{version}} := \mathsf{version}^Q + 1$. Let $\tilde{\nu} := \bot$. 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 $(\texttt{execute}, id, pid, f, z) \overset{t_1}{\hookrightarrow} \mathcal{F}_{\mathsf{PPC}}$ and instruct $\mathcal{F}_{\mathsf{PPC}}$ 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 $\Gamma^Q(id) = \bot$, else let $\gamma^Q := \Gamma^Q(id)$. Stop if $Q \notin \gamma^Q.\mathsf{endusers}$ or $P \notin \gamma^Q.\mathsf{endusers}$ or $\gamma^Q.\mathsf{pspace}(pid) = \bot$, 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 $(\cdot, \cdot, \mathsf{version}^Q) := \Gamma^Q_{aux}(id, pid)$. Stop if (id, pid) is *not* marked as registered.
2. Send $(\texttt{execute}, id, pid, f, z) \overset{t_2}{\hookrightarrow} \mathcal{F}_{\mathsf{PPC}}$ and instruct $\mathcal{F}_{\mathsf{PPC}}$ to execute at time t_2 and output according to the onchain delay. Get $(\texttt{executed}, id, pid, P, f, t_2, z, \nu) \overset{t_3 \le t_2 + \Delta}{\hookleftarrow} \mathcal{F}_{\mathsf{PPC}}$. Send $(\texttt{executed-onchain}, id, pid, P, f, t_2, z, \nu) \overset{t_3}{\hookrightarrow} P$.

Simulator Sim: Close a programmable payment channel

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

P is honest and Q is corrupt

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

1. Stop if id is marked as closed. Otherwise, mark id as closed.
2. Stop if $\Gamma^P(id) = \bot$, else let $\gamma^P := \Gamma^P(id)$. For each $\gamma^P.\mathsf{pspace}(pid) \neq \bot$ 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 $(\texttt{contract-closing}, id) \overset{t_1 \le t_0 + 4\Delta}{\hookrightarrow} Q$.
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 $t_2 \le t_0 + 7\Delta$, instruct $\mathcal{F}_{\mathsf{PPC}}$ to set the first waiting time till t_2.
5. Wait for $\gamma^P.duration$ rounds. Within Δ rounds, send $(\texttt{contract-close}, id) \overset{t_3 \le t_0 + 8\Delta + \gamma^P.duration}{\hookrightarrow} Q$ and instruct $\mathcal{F}_{\mathsf{PPC}}$ 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 $(\texttt{contract-close}, id) \overset{t_1 \le t_0 + 2\Delta}{\hookleftarrow} P$ after the registration, send $(\texttt{close}, id) \overset{t_1}{\hookrightarrow} \mathcal{F}_{\mathsf{PPC}}$ on behalf of P.
3. Stop if id is marked as closed. Otherwise, mark id as closed.
4. Within Δ rounds, for each $\gamma^Q.\mathsf{pspace}(pid) \neq \bot$ 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 $t_2 \le t_1 + 3\Delta$ round).

5. Wait for $\gamma^Q.duration$ rounds. Within Δ rounds, send $(\texttt{contract-close}, id) \overset{t_3 \le t_0+8\Delta+\gamma^Q.duration}{\hookrightarrow} P$ and instruct $\mathcal{F}_{\mathsf{PPC}}$ to send messages at round t_3.

I Protocol Π_{SC}

Protocol Π_{SC}: Implement state channel in the $\mathcal{F}_{\mathsf{PPC}}^{\hat{\mathcal{L}}(\Delta)}$-hybrid

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

State channel opening

Open a programmable payment channel between two parties.

Covenant creation

Party A upon $(\texttt{create}, id, C||y, z) \overset{t_0}{\hookleftarrow} \mathcal{E}$

1. Let $\sigma' := C.\mathsf{Construct}(t_0, y)$. Using σ' to construct associated promise code $C_{B\to A}$, then calculate $pid := (id, C_{B\to A}, \bot, z)$. Send $(\texttt{create}, id, C_{A\to B}||pid, \cdot) \overset{t_0}{\hookrightarrow} \mathcal{F}_{\mathsf{PPC}}$.

Party B upon $(\texttt{create}, id, C||y, z) \overset{t_0}{\hookleftarrow} \mathcal{E}$

2. Let $\sigma' := C.\mathsf{Construct}(t_0, y)$. Using σ' to construct associated promise code $C_{B\to A}$, then calculate $pid := (id, C_{B\to A}, \bot, z)$. Upon receiving $(\texttt{instance-created}, id, pid', \nu) \overset{t_0+1}{\hookleftarrow} \mathcal{F}_{\mathsf{PPC}}$ where $pid' = (id, C_{A\to B}, A, pid, \cdot)$ and ν is the valid $C_{A\to B}$, send $(\texttt{create}, id, C_{B\to A}, z) \overset{t_0+1}{\hookrightarrow} \mathcal{F}_{\mathsf{PPC}}$.
3. Upon receiving $(\texttt{instance-created}, id, pid'', \nu) \overset{t_0+2}{\hookleftarrow} \mathcal{F}_{\mathsf{PPC}}$ where $pid'' = (id, C_{B\to A}, B, \bot, z)$ and ν is the valid $C_{B\to A}$, send $(\texttt{execute}, id, pid'', \mathsf{Enable}, \cdot) \overset{t_0+2}{\hookrightarrow} \mathcal{F}_{\mathsf{PPC}}$.
4. Upon receiving $(\texttt{executed}, \cdots) \overset{t_0+2<t_1\le t_0+4\Delta+7}{\hookleftarrow} \mathcal{F}_{\mathsf{PPC}}$ which indicates that the promise related to $C_{B\to A}$ is enabled. Let $cid := (id, C, y, z)$ and save $\Psi^B(cid) := (pid'', pid')$. Output $(\texttt{instance-created}, id, cid, \nu) \overset{t_1}{\hookrightarrow} B$ where $\nu.\mathsf{code} := C$ and $\nu.\mathsf{storage} := \sigma'$.

Back to party A

5. Upon receiving $(\texttt{executed}, \cdots) \overset{t_0+2<t_1\le t_0+4\Delta+7}{\hookleftarrow} \mathcal{F}_{\mathsf{PPC}}$ which indicates that the promise related to $C_{B\to A}$ is enabled. Let $C_{A\to B}$ is saved in pid' and $C_{B\to A}$ is saved in pid''. Let $cid := (id, C, y, z)$ and save $\Psi^A(cid) := (pid'', pid')$. Output $(\texttt{instance-created}, id, cid, \nu) \overset{t_1}{\hookrightarrow} A$ where $\nu.\mathsf{code} := C$ and $\nu.\mathsf{storage} := \sigma'$.

Covenant execution

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

1. Fetch $(pid'', pid') := \Psi^T(cid)$. Send $(\texttt{execute}, id, pid'', f, z) \overset{t_0}{\hookrightarrow} \mathcal{F}_{\mathsf{PPC}}$. Within $4\Delta+5$ rounds, output accordingly.

State channel closure

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

1. Wait for $\gamma.duration$ rounds.
2. Fetch (in parallel) each $(pid'', pid') := \Psi^T(\cdot)$. Send $(\texttt{execute}, id, pid'', \mathsf{Finalize}, z) \overset{t_0}{\hookrightarrow} \mathcal{F}_{\mathsf{PPC}}$ and $(\texttt{execute}, id, pid', \mathsf{Finalize}, z) \overset{t_0}{\hookrightarrow} \mathcal{F}_{\mathsf{PPC}}$. This will be finished within $4\Delta + 5$.

Close the programmable payment channel between two parties.

J Protocol Analysis

Recall that our compiler takes advantage of the following two features provided by promises.

- Promises in the same channel can read each other.
- A promise instance (even one that has not been deployed) is uniquely identifiable (recall this is because promises are deployed to a deterministic address thanks to the CREATE2 opcode), and hence one promise can refer to functions defined in other *future* promises.

Consider a covenant code as $C := (\Lambda, \mathsf{Construct}, f_1, \cdots, f_s)$ where Alice (denoted by A) wants to start a covenant instance (with Bob, denoted by B) created via call to $\mathsf{Construct}$ with auxiliary inputs y at time t_0. As mentioned in the overview, our compiler (see Figure 8) 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. 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 8a. The construction of the promise code from Alice to Bob, defined as $C_{A\to B} := (\cdot, \mathsf{Construct}_1, \mathsf{Finalize})$, is shown in Figure 8b. We discuss some crucial points:

- The constructor function of $C_{B\to A}$ uses σ' as a white-box, where σ' can be computed by both parties using y and t_0. More importantly, the address of $C_{B\to A}$ can be fixed by two parties using the same z (identical inputs from $\mathcal{E}$). To create a covenant, the protocol starts by constructing these two promises in $\mathcal{F}_{\mathsf{PPC}}$ (2 rounds in total).
- 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 we

want to the bound the rounds needed to create a covenant. Recall in $\mathcal{F}_{\mathsf{SC}}$, even a malicious Bob can only create a covenant instance within $4\Delta+7$ rounds (and he cannot create it after this). Without this, once Alice sends her promise, a malicious Bob can create this covenant at any time later. Thus, the final step to create a covenant is to call Enable by Bob via $\mathcal{F}_{\mathsf{PPC}}$. Note that a malicious player can only delay this execution by at most $4\Delta + 5$ rounds. The overall creation uses at most $4\Delta + 7$ rounds, so it is well-simulated.

– The two-party contract can be executed in the same way one would execute $C_{B\to A}$ (created in $\mathcal{F}_{\mathsf{PPC}}$) since we directly clone $f_1, \cdots, f_s$. Thus, the round delay stays identical.
– $C_{A\to B}$ trivially programs a payment from Alice to Bob while reading the state of corresponding $C_{B\to A}$. However, we need to move $resolve_A$ and $resolve_B$ into $resolve$ of two promises. We achieve this by another execution of function Finalize at the beginning of the closure. This incurs another $\gamma.duration+4\Delta+5$ rounds delay. The overall closure uses at most $2\gamma.duration + 12\Delta + 5$ rounds.

K Rock-Paper-Scissors Application

Rock-Paper-Scissors Promise

`Init`$(\mathsf{amt}', C_1', C_2', \mathsf{receiver}', \mathsf{expiry}')$:
1. Set $(\mathsf{amt}, C_1, C_2, \mathsf{receiver}, \mathsf{expiry}) \leftarrow (\mathsf{amt}', C_1', C_2', \mathsf{receiver}', \mathsf{expiry}')$;
2. Set $(\mathsf{revealed}_1, \mathsf{revealed}_2) \leftarrow (\mathtt{F}, \mathtt{F})$, and $(\mathsf{ch}_1, \mathsf{ch}_2) \leftarrow (\perp, \perp)$.

`Reveal`(i, m, r):
1. Require $i \in \{1, 2\}$, $\mathsf{now} < \mathsf{expiry}$ and $\mathtt{Hash}(m, r) = C_i$;
2. Revert if $i = 1$ and $\mathsf{revealed}_2 = \mathtt{F}$;
3. If $\mathsf{revealed}_i = \mathtt{F}$, set $\mathsf{revealed}_i \leftarrow \mathtt{T}$, $\mathsf{ch}_i \leftarrow m$, and $\mathsf{expiry} \leftarrow \mathsf{expiry} + \Delta$;

`Resolve`():
1. Return amt if
   – $\mathsf{revealed}_1 \wedge \mathsf{revealed}_2 \wedge \mathsf{didReceiverWin}(\mathsf{ch}_1, \mathsf{ch}_2)$, or:
   – $\mathsf{revealed}_{\mathsf{receiver}} = \mathtt{T}$ and $\mathsf{revealed}_{3-\mathsf{receiver}} = \mathtt{F}$.
2. Return 0.

Fig. 13: Rock-Paper-Scissors Promise

Rock-Paper-Scissors. Next, we show that two promises can be used to implement two-party contracts where both parties provide money. See Figure 13 for an implementation of a promise that can be used by two parties to deposit amt coins to play a game of rock-paper-scissors such that the winner of the contest will get the counterparty's deposit. In the offchain protocol, first party A sends commitment C_1 to party B. Next, party B sends the promise in Figure 13 with $\mathsf{receiver}' = 1$ (denoting A as the receiver) along with the commitment C_1 from A and its own commitment C_2 hardcoded in the constructor with *an extra*

requirement: C_1 can only be revealed after C_2 is revealed. This extra requirement ensures A will send the promise as follows. Upon receiving the promise offchain from B with the correct C_1 value hardcoded in it, party A sends a promise in Figure 13 with $\mathsf{receiver}' = 2$ (denoting B as the receiver) along with the commitments C_1 and C_2. As before, parties will abort the offchain protocol if the commitment hardcoded in the promises are inconsistent with what they expect. If the commitments are consistent, then the parties go ahead and reveal the openings of the commitments offchain (B first, and then A) and expect to receive updated receipts reflecting the outcome of the game. Now if the winner does not get the correct receipt (before expiry), then it can deploy the promise from the loser, onchain, and then open its commitment on the onchain promise. This process will ultimately ensure that the onchain promise will get the winning amount. On the other hand, a malicious loser may deploy the winner promise onchain and submit its opening. Here, note that the expiry time is increased by an additional Δ rounds, so that the winner has sufficient time to submit its opening, and ensures that its promise resolves to zero.[12]

Indeed, the above rock-paper-scissors protocol is ad-hoc and application-specific. To see why it is secure, we discuss some critical points:

1. After B sends the promise to A, we require B to open his commitment first. This is because if A can open her commitment, a malicious A can directly open it on the promise from B without making a promise to B. In this case, A will *not* pay B even if she loses since there is no promise from A to B.
2. A and B will abort if they do *not* see their commitments in the incoming promises. E.g., if B receives a promise where a malicious A *changes* C_1 or C_2, he can abort. Similarly, if a malicious B sets $C_2 = C_1$[13], A can abort.
3. Anyone can open the commitment as long as it is opened correctly. For example, as long as B opens his commitment in one promise, even if he does not open it on another promise, A can open it as she learns the answer from B, or in the *public* blockchain.

We emphasize that our generic compiler (see Section 4) can directly compile out a Rock-Paper-Scissors *without* performing the above tedious design and analysis. More importantly, our experiment shows that the compiled version is more efficient, in particular, requires less gas fee (see Section 3.6).

[12] Increasing the expiry time is just a simple technique that works in this setting. More generally, one would use the acknowledgments of the most recent offchain state like we used in the reverse HTLC example.

[13] Indeed, this is harmless as this will result in a draw.

Disclaimer

Case studies, comparisons, statistics, research and recommendations are provided "AS IS" and intended for informational purposes only and should not be relied upon for operational, marketing, legal, technical, tax, financial or other advice. Visa Inc. neither makes any warranty or representation as to the completeness or accuracy of the information within this document, nor assumes any liability or responsibility that may result from reliance on such information. The information contained herein is not intended as investment or legal advice, and readers are encouraged to seek the advice of a competent professional where such advice is required. All trademarks are the property of their respective owners, are used for identification purposes only, and do not necessarily imply product endorsement or affiliation with Visa.

These materials and best practice recommendations are provided for informational purposes only and should not be relied upon for marketing, legal, regulatory or other advice. Recommended marketing materials should be independently evaluated in light of your specific business needs and any applicable laws and regulations. Visa is not responsible for your use of the marketing materials, best practice recommendations, or other information, including errors of any kind, contained in this document.