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, Palo Alto, USA
{rakumare,duc.le,mominaei,mzamani}@visa.com
[2] Visa Research and MIT, Cambridge, USA
[3] Georgia Institute of Technology, Atlanta, 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

Y. Yang— Work done in part while at Visa Research.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-031-54776-8_3.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. Pöpper and L. Batina (Eds.): ACNS 2024, LNCS 14585, pp. 51–73, 2024.
https://doi.org/10.1007/978-3-031-54776-8_3

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 [2,14]. 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 [3,16,20–22,27–29], 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 [4,11,13,15,17,25] 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 [13], 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 [10,16,31], where each party establishes a general-purpose channel with a highly available (but untrusted) hub during setup to be able to later transact with many other parties without the need to establish an individual channel with each party (see Fig. 1 Left and Middle). In practice, parties usually have no a priori knowledge about the specific set of conditions required to transact with other (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

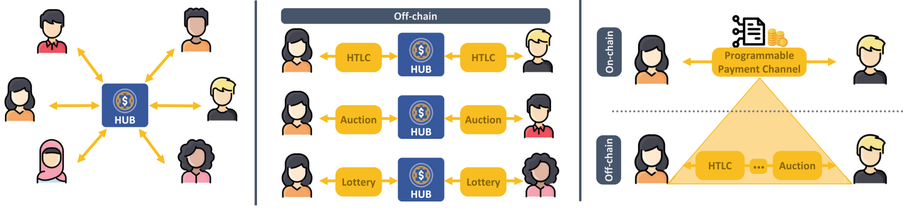


Fig. 1. Left: Hub-and-spoke model: Each party creates a single channel with the hub; **Middle:** Every pair of parties reuse their channels with the hub to execute different contracts; **Right:** PPC between two parties supporting any offchain application.

(see Fig. 1 Right). That is, we achieve lightweight offchain programmable 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 [3,27]. 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.

Full Version. The full version of this paper is [32].

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 [2,14,17,23,25,26,29].

State Channels. A proposal for executing arbitrary contracts offchain is state channels [4,11,13,15,17,25]. 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., [12,15,25]).

The works of [13,17] are closest to ours. Unlike us, [13] do not provide any formal proofs or guarantees. As mentioned in [17], 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 [13]. We follow [5,15–17] to formalize our channel using *universal composable* (UC) framework with a global ledger. However, these works focus on *channel virtualization*[1], and are *not* directly related to this work.

Other Related Work. An excellent systematization of knowledge that explores offchain solutions can be found in [19]. See Appendix A for the comparison with rollups, another popular Layer-2 scaling solution [24,30,33]. See Appendix B for other works that use the CREATE2 opcode.

2 Preliminaries

Network and 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 C.1. We acknowledge that we restrict the distinguisher to a subclass of environments to simplify the formalizations. This restriction is standard (e.g., [16,17]) and can be easily removed using straightforward checks.

Cryptocurrency/Contract Functionalities. We follow [15,17] and model cryptocurrency as a global ledger functionality $\hat{\mathcal{L}}(\Delta)$ in the GUC framework (cf. Fig. 9 in Appendix C.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

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

is to distinguish the real world from the ideal world. In applications such as ours, 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.*[2] 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.

[2] 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).[3] Following prior work (e.g., [17]), 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 [17]. 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) $\sigma.cash : \{\sigma.user_L, \sigma.user_R\} \rightarrow \mathbb{R}$ 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) $\gamma.cash : \{\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}\} \rightarrow \mathbb{R}_{\geq 0}$ 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., [15,17]); we will further clarify it in Sect. 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

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

money. Formally, a *promise instance* consists of two attributes: *promise storage* (accessed by key $\mathsf{storage}$) and *promise code* (accessed by key code). Promise storage σ is an attribute tuple containing at least the following attributes: (1) $\sigma.payer$ denotes the party who sends money; (2) $\sigma.payee$ denotes the party who receives money; 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.

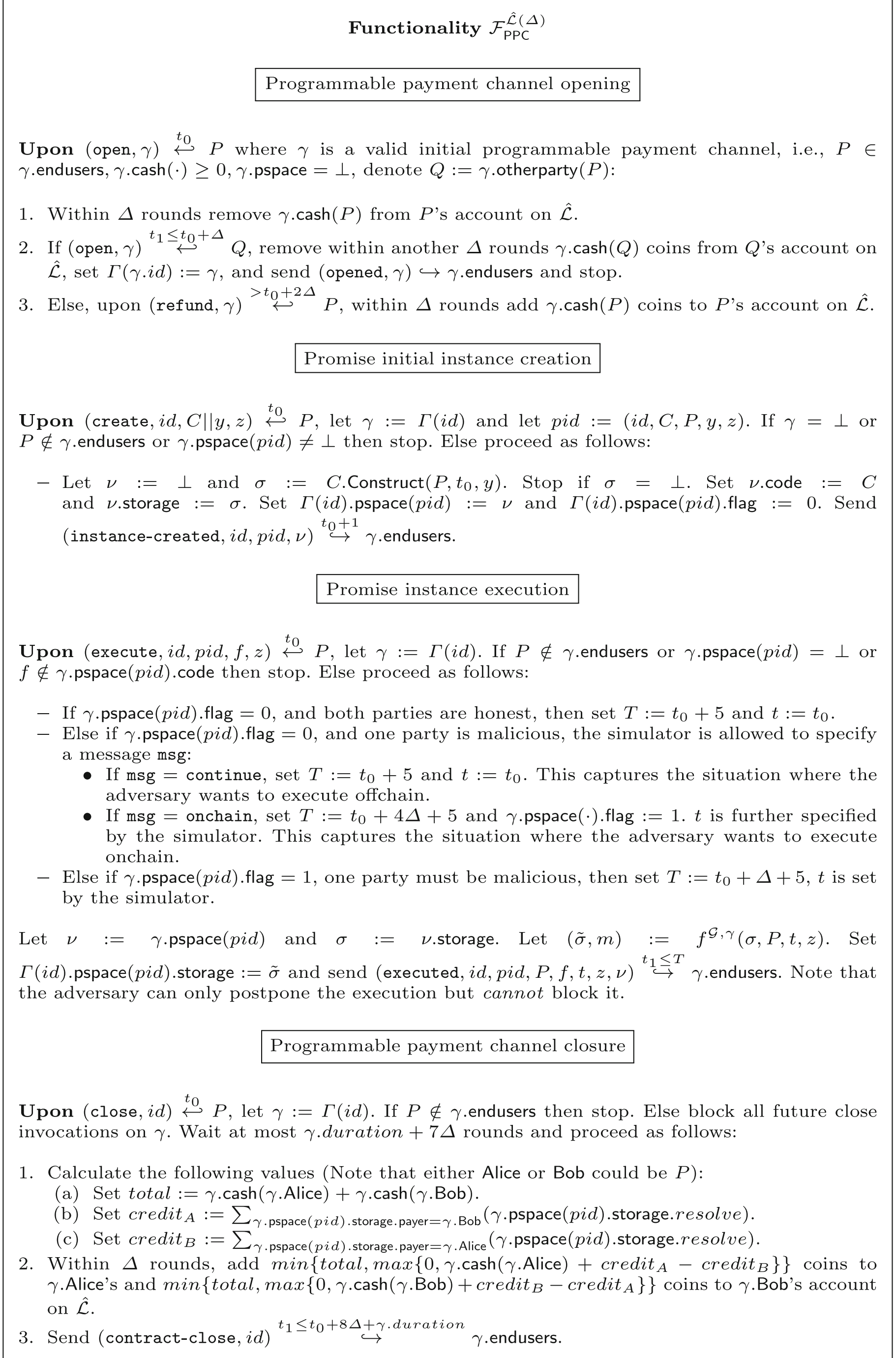
Diverging from [15,17], we assume that each f_i has access to the code and storage of other promises in the *same* channel, as well as the code and storage of all Layer-1 onchain contracts. Formally, we capture this by providing oracle access to the ideal functionalities. This is why we use the notation $f^{\mathcal{G},\gamma}$ in the definition of $\mathcal{F}_{\mathsf{PPC}}$ (see Fig. 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 [15–17]. 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 Fig. 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.[4]

[4] Note that this does not hold for state channels as formalized in [17] 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., $P \in \gamma.\mathsf{endusers}, \gamma.\mathsf{cash}(\cdot) \geq 0, \gamma.\mathsf{pspace} = \perp$, 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 $\gamma = \perp$ or $P \notin \gamma.\mathsf{endusers}$ or $\gamma.\mathsf{pspace}(pid) \neq \perp$ then stop. Else proceed as follows:

- Let $\nu := \perp$ and $\sigma := C.\mathsf{Construct}(P, t_0, y)$. Stop if $\sigma = \perp$. 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 $\gamma.\mathsf{pspace}(pid) = \perp$ 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 $credit_A := \sum_{\gamma.\mathsf{pspace}(pid).\mathsf{storage.payer}=\gamma.\mathsf{Bob}} (\gamma.\mathsf{pspace}(pid).\mathsf{storage}.resolve)$.
   (c) Set $credit_B := \sum_{\gamma.\mathsf{pspace}(pid).\mathsf{storage.payer}=\gamma.\mathsf{Alice}} (\gamma.\mathsf{pspace}(pid).\mathsf{storage}.resolve)$.
2. Within Δ rounds, add $min\{total, max\{0, \gamma.\mathsf{cash}(\gamma.\mathsf{Alice}) + credit_A - credit_B\}\}$ coins to γ.Alice's and $min\{total, max\{0, \gamma.\mathsf{cash}(\gamma.\mathsf{Bob}) + credit_B - credit_A\}\}$ coins to γ.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.

(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 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 [17] 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 [17] requires that all contract instances in the channel are finalized in order to close the channel. We cannot imitate this approach because in our case, the creation of a promise instance need only be authenticated by the payer, and so requiring finality will allow a malicious party to block closing by simply creating some non-finalizable promise instance. (Note that 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 Fig. 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.

PPC **Contract**

Init(id′, vk'_A, vk'_B, claimDuration′):

1. Set (id, claimDuration) ← (id′, claimDuration′);
2. Set status ← "Active"; chanExpiry ← 0; unresolvedPromises ← ⊥;
3. Set A ← {addr : vk'_A, deposit : 0, rid : 0, credit : 0, acc : ⊥, closed : F}
4. Set B ← {addr : vk'_B, deposit : 0, rid : 0, credit : 0, acc : ⊥, closed : F}

Deposit(amount):

1. Require status = "Active" and caller.vk ∈ {A.addr, B.addr};
2. If caller.vk = A.addr, then set A.deposit ← A.deposit + amount;
3. If caller.vk = B.addr, then set B.deposit ← B.deposit + amount.

RegisterReceipt(R):

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

RegisterPromise(P):

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

Close():

1. Require caller.vk ∈ {A.addr, B.addr};
2. If caller.vk = A.addr, set A.closed ← T; Otherwise set B.closed ← T;
3. If A.closed and B.closed, set status ← "Closed";
4. If status = "Active", then set chanExpiry ← now + claimDuration, and status ← "Closing".

Withdraw():

1. Require status ∈ {"Closing", "Closed"};
2. If status = "Closing", Require now > chanExpiry;
3. For each (addr, receiver) ∈ unresolvedPromises:
   receiver.credit ← receiver.credit + addr.resolve();
4. Invoke transfer(A.addr, $min(total, max(0,$ A.deposit + A.credit − B.credit$))$ and transfer(B.addr, $min(total, max(0,$ B.deposit + B.credit − A.credit$))$, where $total$ = A.deposit + B.deposit.

Fig. 3. PPC Contract

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.

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 Fig. 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 Fig. 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 $\mathcal{H}(\texttt{0xFF}, sender, salt, bytecode)$ (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 the full version of our work.

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 Sect. 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.

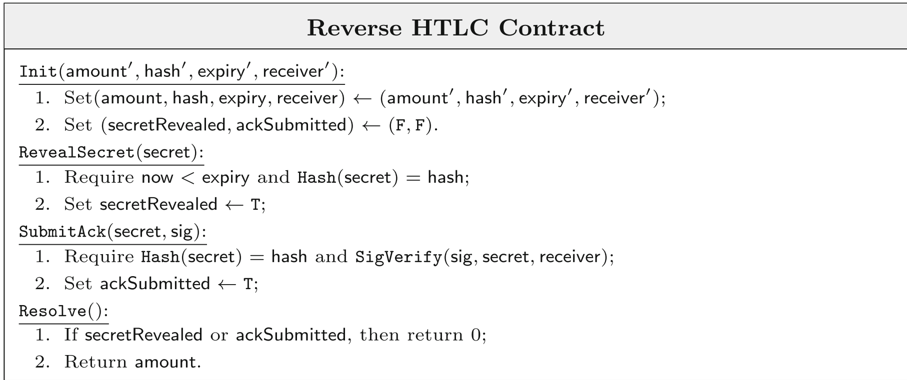
Fig. 4. HTLC Contract

HTLC. See Fig. 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[5] 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 Fig. 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.

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



Reverse HTLC Contract

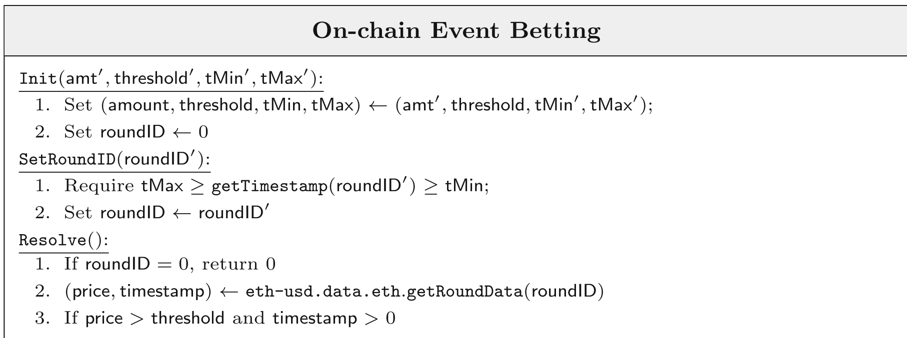
`Init`(amount′, hash′, expiry′, receiver′):
1. Set(amount, hash, expiry, receiver) ← (amount′, hash′, expiry′, receiver′);
2. Set (secretRevealed, ackSubmitted) ← (F, F).

`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 Fig. 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 Fig. 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 [16] and Raiden [3][6] 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.

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

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 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 Sect. 3.5, i.e., reverse HTLC, onchain event betting, and rock-paper-scissors (cf. Appendix D) in Table 2. For the rock-paper-scissors, we provide two implementations: one using the compiler (cf. Sect. 4), and one without (i.e., the ad-hoc implementation in Appendix D). 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., [4,17,25]) 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 [17]. 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 Fig. 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

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

- If $(\mathtt{create}, id, C||y, z) \overset{t_0}{\hookleftarrow} Q$ and P, Q are honest or the the simulator behaves honestly, then let $\nu := \perp$ and $\sigma := C.\mathsf{Construct}(t_0, y)$. Stop if $\sigma = \perp$. 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 $(\mathtt{instance\text{-}created}, id, cid, \nu) \overset{t \leq t_0+7}{\hookrightarrow} \gamma.\mathsf{endusers}$.
- If $(\mathtt{create}, id, C||y, z) \overset{t_0}{\hookleftarrow} Q$ and one party is malicious, then let $\nu := \perp$ and $\sigma := C.\mathsf{Construct}(t_0, y)$. Stop if $\sigma = \perp$. 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 $(\mathtt{instance\text{-}created}, id, cid, \nu) \overset{t \leq t_0+4\Delta+7}{\hookrightarrow} \gamma.\mathsf{endusers}$.

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 $(\mathtt{close}, id) \overset{t_0}{\hookleftarrow} P$, 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 γ.Alice's and $min\{total, max\{0, \gamma.\mathsf{cash}(\gamma.\mathsf{Bob}) + credit_B - credit_A\}\}$ coins to γ.Bob's account.
3. Send $(\mathtt{contract\text{-}close}, id) \overset{t_1 \leq t_0+12\Delta+2\gamma.duration+5}{\hookrightarrow} \gamma.\mathsf{endusers}$.

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

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 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.

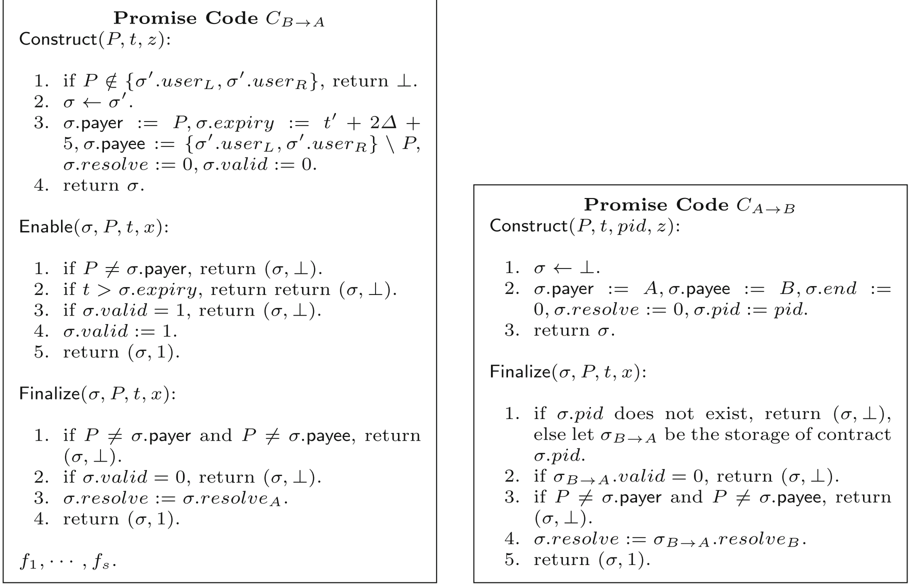
We illustrate this with an example of two-party contract for chess. We assume that each party puts in \$50, and the winner gets \$100. Assume that there exists a smart contract c that contains the entire logic of chess (i.e., checking validity of a move, checking whether the game has ended, who has won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



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

promises. Note that the crucial point is to argue the rounds taken by the two worlds are identical. Due to space limitations, we provide the formal description of the protocol and its analysis in the full version of our work.

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 Oct 20 2023

2. Payment channels - bitcoin wiki. https://en.bitcoin.it/wiki/Payment_channels. Accessed Oct 20 2023
3. Raiden. https://raiden.network/. Accessed Oct 20 2023
4. State channels - ethereum.org. https://ethereum.org/en/developers/docs/scaling/state-channels/. Accessed Oct 20 2023
5. Aumayr, L., et al.: 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: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 61–85. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-70936-7_4
10. Christodorescu, M., et al.: 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. Close, T.: Nitro protocol. Cryptology ePrint Archive (2019)
12. Close, T., Stewart, A.: Forcemove: an n-party state channel protocol. Magmo, White Paper (2018)
13. Coleman, J., Horne, L., Xuanji, L.: Counterfactual: generalized state channels. Accessed. https://l4.ventures/papers/statechannels.pdf 4 2019 (2018)
14. 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)
15. Dziembowski, S., Eckey, L., Faust, S., Hesse, J., Hostáková, K.: Multi-party virtual state channels. In: Ishai, Y., Rijmen, V. (eds.) Advances in Cryptology – EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part I, pp. 625–656. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-17653-2_21
16. Dziembowski, S., Eckey, L., Faust, S., Malinowski, D.: Perun: virtual payment hubs over cryptocurrencies. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 106–123. IEEE (2019)
17. Dziembowski, S., Faust, S., Hostáková, K.: General state channel networks. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 949–966 (2018)
18. Goldreich, O.: Foundations of cryptography: volume 2, basic applications. Cambridge University Press (2009)
19. Gudgeon, L., Moreno-Sanchez, P., Roos, S., McCorry, P., Gervais, A.: SoK: Layer-two blockchain protocols. In: Bonneau, J., Heninger, N. (eds.) Financial Cryptography and Data Security: 24th International Conference, FC 2020 , Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers, pp. 201–226. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-51280-4_12

20. Khalil, R., Gervais, A.: Nocust-a non-custodial 2nd-layer financial intermediary (2018)
21. 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)
22. 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)
23. Malavolta, G., Moreno-Sanchez, P., Schneidewind, C., Kate, A., Maffei, M.: Anonymous multi-hop locks for blockchain scalability and interoperability. In: NDSS (2019)
24. McCorry, P., Buckland, C., Yee, B., Song, D.: Sok: Validating bridges as a scaling solution for blockchains. Cryptology ePrint Archive (2021)
25. 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: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers, pp. 508–526. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-32101-7_30
26. 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/
27. Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments. https://lightning.network/lightning-network-paper.pdf (2016) Accessed Oct 20 2023
28. 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). https://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-3_Roos_paper.pdf
29. 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
30. 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
31. 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 Oct 20 2023
32. Yang, Y., Minaei, M., Raghuraman, S., Kumaresan, R., Le, D.V., Zamani, M.: Programmable payment channels. Cryptology ePrint Archive, Paper 2023/347 (2023). https://eprint.iacr.org/2023/347
33. Yee, B., Song, D., McCorry, P., Buckland, C.: Shades of finality and layer 2 scaling. arXiv preprint arXiv:2201.07920 (2022)