Sprites and State Channels: Payment Networks that Go Faster than Lightning

Andrew Miller[1], Iddo Bentov[2], Surya Bakshi[1], Ranjit Kumaresan[3] and Patrick McCorry[4]

[1] University of Illinois at Urbana-Champaign
[2] Cornell Tech
[3] VISA Research
[4] King's College London

Abstract. Bitcoin, Ethereum and other blockchain-based cryptocurrencies, as deployed today, cannot support more than several transactions per second. Off-chain payment channels, a "layer 2" solution, are a leading approach for cryptocurrency scaling. They enable two mutually distrustful parties to rapidly send payments between each other and can be linked together to form a payment network, such that payments between any two parties can be routed through the network along a path that connects them.

We propose a novel payment channel protocol, called Sprites. The main advantage of Sprites compared with earlier protocols is a reduced "collateral cost," meaning the amount of *money* × *time* that must be locked up before disputes are settled. In the Lightning Network and Raiden, a payment across a path of ℓ channels requires locking up collateral for $\Theta(\ell\Delta)$ time, where Δ is the time to commit an on-chain transaction; every additional node on the path forces an increase in lock time. The Sprites construction provides a *constant* lock time, reducing the overall collateral cost to $\Theta(\ell + \Delta)$. Our presentation of the Sprites protocol is also modular, making use of a generic state channel abstraction. Finally, Sprites improves on prior payment channel constructions by supporting partial withdrawals and deposits without any on-chain transactions.

1 Introduction

Popular cryptocurrencies such as Bitcoin and Ethereum have at times reached their capacity limits, leading to transaction congestion and higher fees. A limit to scalability seems inherent in their model, since they are designed for security through replication, every node validates every transaction.

A leading proposal for improving the scalability of cryptocurrencies is to form a network of "off-chain" rapid payment channels. Payment channels require initial deposits of on-chain currency, but once established can support an unbounded number of payments in a session using only off-chain messages. Payments can be routed through a network of such channels, with changes in balance flowing from one intermediary to the next. Only when the channel must

be settled is blockchain interaction required. The protocol is centered around a smart contract, which handles deposits and withdrawals and defines the rules for handling disputes.

In this paper we introduce the "collateral cost" of a payment channel, which roughly corresponds to the amount of time that an amount of money is locked up in the smart contract, (*money* × *time*). The main result of our paper is a new payment channel protocol called Sprites that improves on the state-of-the-art in worst-case collateral cost.

Collateral Costs in Payment Channels. A chief concern for the feasibility of payment channel networks is whether or not enough collateral will be available for payments to be routed at high throughput. For every pending payment, some money in the channel must be reserved and held aside as collateral until the payment is completed, called the "locktime." Even though off-chain payments complete quickly in the typical case, if parties fail (or act to maliciously impose a delay), the collateral can be locked up for longer, until a dispute handler can be activated on-chain.

We characterize the performance of a payment channel protocol as its "collateral cost," which we think of as the lost time value of money held in reserve (i.e., in units of *money* × *time*) during the locktime.[5] For a linked payment, the longer the payment path, the more total collateral must be reserved: for a payment of size $\$X$ across a path of ℓ channels, a total of $\theta(\ell\$X)$ money must be reserved. Payment channel protocols depend on a worst-case delay bound, Δ for the underlying blockchain. Essentially, Δ is a safe bound on how long it takes to observe a transaction committed on the blockchain and commit one new transaction in response, i.e., one blockchain round trip. In practical terms, Δ is roughly 1 day.

In the Lightning Network and in Raiden, the two most well-known payment networks, Δ is incorporated into the locktime parameter. However, a payment on a path of length ℓ requires an additional Δ delay added to the locktime for each link. Thus the worst-case total collateral cost of a $\$X$ payment over a path of length ℓ is $\Theta(\ell^2\$X\Delta)$. The diameter of the Lightning network is 8, and with a payment of \$10, the collateral costs for Lightning and Sprites are 360 dollar-days and 116 dollar-days, respectively. Therefore, Sprites has an approximately 3x collateral cost improvement over Lightning.

Sprites: Constant-Locktime Payment Channels. Sprites improves on Lightning and other linked-payments by avoiding the need to add an additional Δ delay for each payment on the path, reducing the collateral cost by a factor of ℓ with a constant locktime. The key insight behind this improvement is the use of a globally accessible smart contract that provides shared state between individual payment channels. As such, this is expressible in Ethereum, but does not appear possible in Bitcoin.

Although the Sprites protocol builds on prior payment channel designs, we present it from scratch in a simplified and modular way. Our presentation is based

[5] The rational investor's preference is to obtain and use money now rather than later.

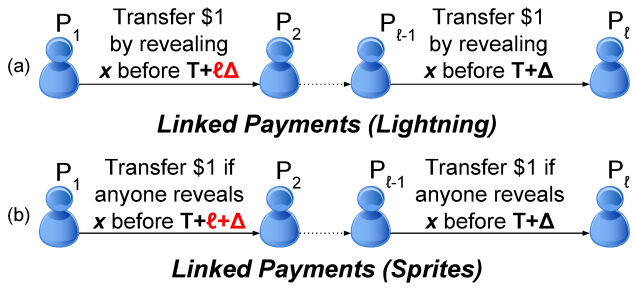

Fig. 1: The underlying currency serves as collateral for a payment network [19, 4]. A payment channels allow rapid payment to another party, requiring on-chain transactions in case of disputes. Payments can be routed through multiple channels based on a condition (a). We improve the worse case delay for ℓ-hop payments, (b), to $\Theta(\ell + \Delta)$.

on a generic abstraction, the state channel, which serves two roles: First, it neatly encapsulates the necessary cryptography (mainly exchanging digital signatures), separating concerns in the protocol presentation. Second, it provides a flexible interface bridging the off-chain and on-chain worlds. Sprites makes use of this interface in several ways, both to define its constant-locktime dispute handler, but also to support incremental deposits and withdrawals without interruption. Our security and worst-case performance analysis ensures that intermediaries are never at risk of losing money, and that the protocol provides real time guarantees even in spite of Byzantine failures. Finally, we implemented a proof of concept of Sprites, and deployed it on the Ropsten Ethereum Testnet.[6] We found that the transaction fees required to resolve a dispute on-chain are around $\approx$ \$0.20 USD as of November 2018, comparable to the Lightning Network.

2 Background and Preliminaries

2.1 Blockchains and Smart Contracts

At a high level, a blockchain is a distributed ledger of balances. The primary use of blockchains are as decentralized cryptocurrencies, which allow users to exchange a native token without trusted intermediaries. Transactions are made by users (addressed by pseudonyms) and published on the blockchain (on-chain transaction) to be confirmed by the rest of the network. Decentralized cryptocurrencies like Ethereum, however, require state replication across all nodes and can not support more than several transactions per second.

Concretely, a blockchain ensures the following properties:

[6] The reference implementation can be found at `https://github.com/amiller/sprites`, Sprites: `0x85DF43619C04d2eFFD7e14AF643aef119E7c8414`, Manager: `0x62E2D8cfE64a28584390B58C4aaF71b29D31F087`.

1. All parties can agree on a consistent log of committed transactions
2. All parties are guaranteed to be able to commit new transactions in a predictable amount of time, Δ.

The time delay, Δ, is meant to capture the worst-case bound on how long it takes to learn about a new transaction, then to publish a transaction in response. We say one unit of time is the maximum time needed to transmit a point-to-point message to any other party.

Modern cryptocurrencies, like Ethereum, also feature smart contracts. A smart contract is an autonomous piece of code that exists at an address in the Ethereum blockchain. It can hold funds like any other address and can act on those funds through its functionality. To execute a piece of code in the contract, a user account must submit a transaction to it specifying the method to be executed. The method's execution may change the state of the contract's balance or persistent storage, and the changes are eventually committed to the blockchain. The main benefits of contracts are that they are essentially autonomous machines that always execute their code correctly. Throughout this paper, we show smart contracts using pseudocode resembling reactive processes that respond to method invocations.

2.2 Blockchain scaling

Proposed scalability improvements fall in roughly two complementary categories. The first, "on-chain scaling," aims to make the blockchain itself run faster [9, 11, 17, 7]. A recurring theme is that the additional performance comes from introducing stronger trust assumptions about the nodes.

The second category of scaling approaches, which includes our work, is to develop "off-chain protocols" that minimize the use of the blockchain itself. Instead, parties transact primarily by exchanging off-chain messages (point-to-point messages), and interact with the blockchain only to settle disputes or withdraw funds.

2.3 Off-chain Payment Channels

There have been many previous payment channel constructions prior to this work. However, for simplicity we present only the approach using signatures over round numbers [15, 18, 2]. We also make the assumption that transactions can depend on a "global" event recorded in the blockchain — and therefore Sprites cannot (we conjecture) be implemented in Bitcoin.

An off-chain payment channel protocol roughly comprises the following three phases:

Channel opening. The channel is initially opened with an on-chain deposit transaction. This reserves a quantity of digital currency and binds it to the smart contract program.

Off-chain payments. To make an off-chain payment, the parties exchange signed messages, reflecting the updated balance. For example, the current state

would be represented as a signed message $(\sigma_A, \sigma_R, i, \$A, \$B)$, where a pair of signatures σ_A and σ_B are valid for the message $(i, \$A, \$B)$, where $\$A$ (resp. $\$B$) is the balance of Alice (resp. Bob) at round number i. Each party locally keeps track of the current balance, corresponding to the most recent signed message.
Dispute handling. The blockchain smart contract serves as a "dispute handler." It is activated when either party suspects a failure, or wishes to close the channel and withdraw the remaining balance. The dispute handler remains active for a fixed time during which either party can submit evidence (e.g., signed messages) of their last-known balance. The dispute handler accepts the evidence with the highest round number and disburses the money accordingly.

The security guarantees, roughly, are the following:
(Liveness): Either party can initiate a withdrawal, and the withdrawal is processed within a predictable amount of time. If both parties are honest, then payments are processed very rapidly (i.e., with only off-chain messages).
(No counterparty risk): The payment channel interface offers Bob a local estimate of his current balance (i.e., how many payments he has received). Alice, of course, knows how much she has sent. The "no counterparty risk" property guarantees that local views are accurate, in the sense that each party can actually withdraw (at least) the amount they expect.

2.4 Linked payments and payment channel networks

Duplex payment channels alone cannot solve the scalability problem; opening each channel requires an on-chain transaction before any payments can be made. To connect every pair of parties in the network by a direct channel would require $O(N^2)$ transactions.

Poon and Dryja [19] developed a method for linking payments across a path of channels where the capacity within each channel is sufficient to facilitate the transfer.

Linked payments are based on the "hashed timelock contract" (HTLC) for conditional payments that relies on a single hash $h = \mathcal{H}(x)$ to synchronize a payment across all channels. We denote an HTLC conditional payment from P_1 to P_2 by the following:

$$P_1 \xrightarrow[h,T]{\$X} P_2$$

which says that a payment of $\$X$ can be claimed by P_2 if the preimage of h is revealed via an on-chain transaction. In the optimistic case, the sender can create and send a new *unconditional* payment with a higher round number. Otherwise, the conditional payment can be canceled after a deadline T. Operationally, opening a conditional payment means signing a message that defines the deadline, the amount of money, and the hash of the secret $h = \mathcal{H}(x)$; and finally sending the signed message to the recipient.

Consider a path of parties, $P_1, ..., P_\ell$, where P_1 is the sender, P_ℓ is the recipient, and the rest are intermediaries. In a linked off-chain payment, Each node P_i opens a conditional payment to P_{i+1}, one after another.

$$P_1 \xrightarrow[h,T_1=T_{\ell-1}+\Theta(\ell\Delta)]{\$X} P_2 \ ... \ P_{\ell-1} \xrightarrow[h,T_{\ell-1}]{\$X} P_\ell \qquad (\lightning)$$

Note that the hash condition h is the same for all channels. However, the deadlines may be different. In fact, Lightning requires that $T_1 = T_\ell + \Theta(\ell\Delta)$ as we explain shortly. The desired security properties of linked payments are the following (in addition to those for basic channels given above):

(Liveness): The entire chain of payments concludes (success or cancellation) within a bounded amount of on-chain cycles. If all parties on the path are honest, then the entire payment should complete successfully using only off-chain messages.

(No counterparty risk): A key desired property is that intermediaries should not be placed at risk of losing funds. During the linked payment protocol, a portion of the channel balance may be "locked" and held in reserve, but it must returned by the conclusion of the protocol.[7] This property poses a challenge that constrains the choice of deadlines $\{T_i\}$ in Lightning. Consider the following scenario from the point of view of party P_i.

$$... \ P_{i-1} \xrightarrow[h,T_i]{\$X} P_i \xrightarrow[h,T_{i+1}]{\$X} P_{i+1} \ ...$$

We need to ensure that if the outgoing conditional payment to P_{i+1} completes, then the incoming payment from P_{i-1} also completes. In the worst case where P_{i+1} attempts to introduce the maximum delay for P_i (which we call the "petty" attacker), the party P_i only learns about x because x is published in the blockchain at the last possible instant, at time T_{i+1}. In order to complete the incoming payment, if P_{i-1} is also petty then P_i must publish x to the blockchain by time T_i. It must therefore be the case that $T_i \geq T_{i+1} + \Delta$, meaning P_i is given an additional grace period of time Δ (the worst-case bound on the time for one on-chain round).

We use the term "collateral cost" to denote the product of the amount of money $\$X$ multiplied by the locktime (i.e., from when the conditional payment is opened to the time it is completed or canceled). Since the payment can be claimed by time $T_\ell + \Theta(\ell\Delta)$ in the worst case, the overall collateral cost is $\Theta(\ell^2 \$X\Delta)$ for each party (see Figure 1 (a)). The worst-case collateral cost may occur because of failures or malicious attacks intended to slow the network. The main goal of our Sprites construction (Section 3) is to reduce this collateral cost.

3 Overview of the Sprites construction

We first give a high-level overview of our construction, focusing on the main improvements versus Lightning [19]: constant locktimes and incremental withdrawals/deposits. We assume as a starting point the duplex payment channel

[7] The intermediary nodes in a path can also be incentivized to participate in the route if the sender allocates an extra fee that will be shared among them.

construction described earlier in Section 2.3 and presented in related works [2, 15, 18]).

3.1 Constant locktime linked payments.

To support linked payments across multiple payment channels, we use a novel variation of the standard "hashed timelock contract" technique [1, 10, 16, 19].

We start by defining a simple smart contract, called the PreimageManager (PM), which simply records assertions of the form "the preimage x of hash $h = \mathcal{H}(x)$ was published on the blockchain before time T_{Expiry}." This can be implemented in Ethereum as a smart contract with two methods, `publish` and `published` (see Figure 5).

Next, we extend the duplex payment channel construction with a conditional payment feature, which can be linked across a path of channels as shown:

$$P_1 \xrightarrow[\mathsf{PM}[h,T_{\mathsf{Expiry}}]]{\$X} P_2 \ ... \ P_{\ell-1} \xrightarrow[\mathsf{PM}[h,T_{\mathsf{Expiry}}]]{\$X} P_\ell \qquad (\star)$$

In the above, the conditional payment of $\$X$ from P_1 to P_2 can be completed by a command from P_1, canceled by a command from P_2, or in case of dispute, will complete if and only if the PM contract receives the value x prior to T_{Expiry}. As with the existing linked payments constructions [15, 18], operationally this means extending the structure of the signed messages (i.e., the off-chain state) to include a hash h, a deadline T_{Expiry}, and an amount $\$X$. To execute the linked payment, each party first opens a conditional payment with the party to their right, each with the same conditional hash. Note that here the deadline T_{Expiry} is also a common value across all channels.

The difference between Sprites and Lightning is how Sprites handles disputes. Instead of locally enforcing the preimage x be revealed on time, in Sprites we delegate this to the global PM contract. In short, each Sprites contract defines a dispute handler that queries PM to check if x was revealed on time, guaranteeing that all channels (if disputed on-chain) will settle in a consistent way (either all completed or all canceled). It then suffices to use a single common expiry time T_{Expiry}, as indicated above ($\star$).

The preimage x is initially known to the recipient; after the final conditional payment to the recipient is opened, the recipient publishes x, and each party completes their outgoing payment. Optimistically, (i.e., if no parties fail), the process finishes after only $\ell + 1$ off-chain rounds. Otherwise, in the worst case, any honest parties that complete their outgoing payment submit x to the PM contract, guaranteeing that their incoming payment will complete. This procedure ensures that each party's collateral is locked for a maximum of $O(\ell + \Delta)$ rounds.

The worst-case delay scenarios for both Lightning and Sprites are illustrated in Figure 2. In the worst-case, the attacker publishes x at the latest possible time. However, the use of a global synchronizing gadget, the PM contract, ensures that

all payments along the path are settled consistently. In contrast, Lightning [19] (and other prior payment channel networks [15, 4, 12, 5]) require the preimage to be submitted to *each* payment channel contract separately, leading to longer locktimes.

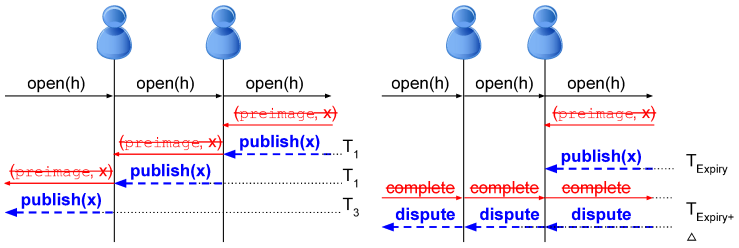

Fig. 2: The worst-case delay scenario, in Lightning (left) and in Sprites (right). The two parties shown are "petty," dropping off-chain messages (~~striken red~~) after the initial open, and sending on-chain transactions (blue) only at the last minute. Disputes in Lightning may cascade, whereas in Sprite they are handled simultaneously.

3.2 Supporting incremental deposits and withdrawals.

A Lightning channel must be closed and re-opened in order for either party to withdraw or deposit currency. Furthermore, all pending conditions must be settled on-chain and no new off-chain transactions can occur for an on-chain round ($O(\Delta)$ time) until a new channel is opened on the blockchain. On the other hand, Sprites permits either party to deposit/withdraw a portion of currency without interrupting the channe.

To support incremental deposits, we extend the off-chain state to include local views, $\mathsf{deposits}_{\{\mathtt{L},\mathtt{R}\}}$, which reflect the total amount of deposits from each party. If one party proposes a view that is too stale (i.e., more than some bound $O(\Delta)$ behind), then the other party initiates an on-chain dispute. Of course, the on-chain dispute handler can read the current on-chain state directly.

To support incremental withdrawals, we implement the following. We extend the off-chain state with an optional withdrawal value wd_i, which can be set whenever either party wishes to make a withdrawal. The on-chain smart contract is then extended with an `update` method that either party can invoke to submit a signed message with a withdrawal value. Rather than close, the smart contract verifies the signatures, disburses the withdrawal, and advances the round number to prevent replay attacks. Further off-chain payments can continue, even while waiting for the blockchain to confirm the withdrawal.

Incremental withdrawals and deposits are also supported in another Ethereum payment network called Raiden [15]. Like Sprites, Raiden allows incremental deposits to be made at any time by any party *without* interrupting the channel. However, unlike Sprites, Raiden does not currently support partial withdrawals and forces a channel to close before any withdrawal is possible.

4 The State Channel Abstraction

In this section, we present the state channel abstraction, which is the key to our modular construction of Sprites payment channels. A state channel generalizes the off-chain payment channel mechanism as described in Section 2.3. The state channel primitive exposes a simple interface: a consistent replicated state machine shared between two or more parties. The state machine evolves according to an arbitrary, application-defined transition function. It proceeds in rounds, during each of which inputs are accepted from every party. This primitive neatly abstracts away the on-chain dispute handling behavior and the use of off-chain signed messages in the optimistic case.

Each time the parties provide input to the state channel, they exchange signed messages on the newly updated state, along with an increasing round number. If at any time a party aborts or responds with invalid data, remaining parties can raise a dispute by submitting the most recent agreed-upon state to the blockchain, along with inputs for the next round. Once activated, the dispute handler proceeds in two phases. First, the dispute handler waits for one on-chain round, during which any party can submit their evidence (i.e., the most recently signed agreed-upon state). The dispute handler checks the signatures on the submitted evidence, and ultimately commits the state with the highest round number. Finally, after committing the previous state, the dispute handler then allows parties to submit new inputs for the next round.

To summarize, the security guarantees of a state channel are:

(Liveness): Each party is able to provide input to each iteration of the state machine, and a corrupt party cannot stall.
(Safety): Each party's local view of the most recent state is finalized and consistent with every other party's view.

A novel feature of our model is a general way to express side effects between the state channel and the blockchain. Besides the inputs provided by parties, the application-specific transition function can also depend on auxiliary input from an external contract C on the blockchain (which, for example, can collect currency deposits submitted by either party). The transition function can also define an auxiliary output for each transition, which is translated to a method invocation on the external smart contract C (e.g., triggering a disbursement of **coins**). This feature generalizes the handling of withdrawals as transfers of on-chain currency.

4.1 Instantiating state channels

We focus on explaining the behavior of the dispute handler smart contract, $\mathsf{Contract}_{\mathsf{State}}$, defined in Figure 3; a detailed description of the local behavior for each party is deferred to the appendix (A.4). At a high level, the off-chain state can be advanced by having parties exchange a signed message of the following form (for the party P_i):

$$\sigma_{r,i} := \mathsf{Sign}_{P_i}(r \| \mathsf{state}_r \| \mathsf{out}_r).$$

where r is the number of the current round, state_r is the result after applying the state transition function to every party's inputs, and out_r is the resulting blockchain output (or $\perp$ if this transition makes no output). In the appendix we describe a leader-based broadcast protocol used to help parties optimistically agree on a vector of inputs. We now explain how $\mathsf{Contract}_{\mathsf{State}}$ handles disputes.

Protocol $\Pi_{\mathsf{State}}(U, P_1, ...P_N)$

Contract $\mathsf{Contract}_{\mathsf{State}}$

Initialize $\mathsf{bestRound} := -1$
Initialize $\mathsf{state} := \emptyset$
Initialize $\mathsf{flag} := \texttt{OK}$
Initialize $\mathsf{deadline} := \perp$
Initialize $\mathsf{applied} := \emptyset$

on **contract input** $\texttt{evidence}(r, \mathsf{state}', \mathsf{out}, \{\sigma_{r,j}\})$:
  discard if $r \leq \mathsf{bestRound}$
  verify all signatures on the message $(r \| \mathsf{state}' \| \mathsf{out})$
  if $\mathsf{flag} == \texttt{DISPUTE}$ then
    $\mathsf{flag} := \texttt{OK}$
    emit $\texttt{EventOffchain}(\mathsf{bestRound} + 1)$
  $\mathsf{bestRound} := r$
  $\mathsf{state} := \mathsf{state}'$
  invoke $C.\texttt{aux\_output}(\mathsf{out})$
  $\mathsf{applied} := \mathsf{applied} \cup \{r\}$

on **contract input** $\texttt{dispute}(r)$ at time T:
  discard if $r \neq \mathsf{bestRound} + 1$
  discard if $\mathsf{flag} \neq \texttt{OK}$
  set $\mathsf{flag} := \texttt{DISPUTE}$
  set $\mathsf{deadline} := T + \Delta$
  emit $\texttt{EventDispute}(r, \mathsf{deadline})$

on **contract input** $\texttt{input}(r, v_{r,j})$ from party P_j:
  if this is the first such activation, store $v_{r,j}$

on **contract input** $\texttt{resolve}(r)$ at time T:
  discard if $r \neq \mathsf{bestRound} + 1$
  discard if $\mathsf{flag} \neq \texttt{PENDING}$
  discard if $T < \mathsf{deadline}$
  apply the update function $\mathsf{state} := U(\mathsf{state}, \{v_{r,j}\}, \mathsf{aux}_{in})$, where the default value is used for any $v_{r,j}$ such that party P_j has not provided input
  set $\mathsf{flag} := \texttt{OK}$
  emit $\texttt{EventOnchain}(r, \mathsf{state})$
  $\mathsf{bestRound} := \mathsf{bestRound} + 1$

Fig. 3: Contract portion of the protocol Π_{State} for implementing a general purpose state channel.

Raising a dispute. Suppose in round r a party fails to receive off-chain signatures from all the other parties for some $(\mathsf{state}_r, \mathsf{out}_r)$ before an $O(1)$ timeout. They then 1) invoke the `evidence` method to provide evidence that round $(r-1)$ has already been agreed upon, and 2) invoke the $\texttt{dispute}(r)$ method, which notifies all the other parties (`EventDispute`).

Resolving disputes off-chain. Once raised, a dispute for round r will be resolved in one of two ways. First, another party may invoke the $\texttt{evidence}(r', ...)$ method to provide evidence that an r or a later round $r' \geq r$ has already been agreed upon off-chain, clearing the dispute (`EventOffchain`). This occurs, for example, if a corrupted node attempts to dispute an earlier already-settled round.

Resolving disputes on-chain. Alternatively, if a party P_j has no more recent evidence than $(r-1)$, they invoke the `input` method on-chain with their input

$v_{r,j}$. After the deadline $T + \Delta$, any party can invoke the `resolve` method to apply the update function to the on-chain inputs (`EventOnchain`).

Avoiding on-chain / off-chain conflicts. We now explain how we avoid a subtle concurrency hazard. Suppose in round r, a party receives the $\texttt{Dispute}(r, T)$ event, and shortly thereafter (say, $T + \epsilon$, for some $\epsilon > 0$), receives a final signature completing the off-chain evidence for round r. It would be incorrect for the party to then invoke $\texttt{evidence}(r, ...)$, since this invocation may not be confirmed until after $T + \Delta + \epsilon$. If a malicious adversary equivocates, providing $\texttt{input}(v'_{r,j})$ on-chain but $v_{r,j}$ off-chain, the off-chain evidence would arrive too late. Instead, upon receiving a $\texttt{Dispute}(r)$ event, if the party does not already have evidence for round r, it pauses the off-chain routine until the dispute is resolved.

Update function U_{Pay}

$U_{\mathsf{Pay}}(\mathsf{state}, (\mathsf{input}_{\mathtt{L}}, \mathsf{input}_{\mathtt{R}}), \mathsf{aux}_{in})$:
- if $\mathsf{state} = \bot$, set $\mathsf{state} := (0, \emptyset, 0, \emptyset)$
- parse state as $(\mathsf{cred}_{\mathtt{L}}, \mathsf{oldarr}_{\mathtt{L}}, \mathsf{cred}_{\mathtt{R}}, \mathsf{oldarr}_{\mathtt{R}})$
- parse $\mathsf{aux}_{\mathsf{in}}$ as $\{\mathsf{deposits}_i\}_{i \in \{\mathtt{L},\mathtt{R}\}}$
- for $i \in \{\mathtt{L}, \mathtt{R}\}$:
  - if $\mathsf{input}_i = \bot$ then $\mathsf{input}_i := (\emptyset, 0)$
  - parse each input_i as $(\mathsf{arr}_i, \mathsf{wd}_i)$
  - $\mathsf{pay}_i := 0, \mathsf{newarr}_i := \emptyset$
  - while $\mathsf{arr}_i \neq \emptyset$
    - pop first element of arr_i into e
    - if $e + \mathsf{pay}_i \leq \mathsf{deposits}_i + \mathsf{cred}_i$:
      - append e to newarr_i
      - pay_i += e
  - if $\mathsf{wd}_i > \mathsf{deposits}_i + \mathsf{cred}_i - \mathsf{pay}_i$: $\mathsf{wd}_i := 0$
- $\mathsf{cred}_{\mathtt{L}}$ += $\mathsf{pay}_{\mathtt{R}} - \mathsf{pay}_{\mathtt{L}} - \mathsf{wd}_{\mathtt{L}}$
- $\mathsf{cred}_{\mathtt{R}}$ += $\mathsf{pay}_{\mathtt{L}} - \mathsf{pay}_{\mathtt{R}} - \mathsf{wd}_{\mathtt{R}}$
- if $\mathsf{wd}_{\mathtt{L}} \neq 0$ or $\mathsf{wd}_{\mathtt{R}} \neq 0$:
  - $\mathsf{aux}_{out} := (\mathsf{wd}_{\mathtt{L}}, \mathsf{wd}_{\mathtt{R}})$
- otherwise $\mathsf{aux}_{out} := \bot$
- $\mathsf{state} := (\mathsf{cred}_{\mathtt{L}}, \mathsf{newarr}_{\mathtt{L}}, \mathsf{cred}_{\mathtt{R}}, \mathsf{newarr}_{\mathtt{R}})$
- return $(\mathsf{aux}_{out}, \mathsf{state})$

Auxiliary smart contract $\mathsf{Contract}_{\mathsf{Pay}}(P_{\mathtt{L}}, P_{\mathtt{R}})$

Initially, $\mathsf{deposits}_{\mathtt{L}} := 0, \mathsf{deposits}_{\mathtt{R}} := 0$

on **contract input** $\texttt{deposit}(\textbf{coins}(\$X))$ from P_i :
- $\mathsf{deposits}_i$ += $\$X$
- $\mathsf{aux}_{in}.\mathsf{send}(\mathsf{deposits}_{\mathtt{L}}, \mathsf{deposits}_{\mathtt{R}})$

on **contract input** $\texttt{output}(\mathsf{aux}_{out})$:
- parse aux_{out} as $(\mathsf{wd}_{\mathtt{L}}, \mathsf{wd}_{\mathtt{R}})$
- for $i \in \{\mathtt{L}, \mathtt{R}\}$ send $\textbf{coins}(\mathsf{wd}_i)$ to P_i

Local protocol Π_{Pay} for party P_i

initialize $\mathsf{pay}_i := 0$, $\mathsf{wd}_i := 0$, $\mathsf{paid}_i = 0$

on **receiving state** $(\mathsf{cred}_{\mathtt{L}}, \mathsf{new}_{\mathtt{L}}, \mathsf{cred}_{\mathtt{R}}, \mathsf{new}_{\mathtt{R}})$ from Π_{State},
- foreach e in new_i: set paid_i += e
- provide $(\mathsf{arr}_i, \mathsf{wd}_i)$ as input to Π_{State}
- $\mathsf{arr}_i := \emptyset$

on **input** $\mathsf{pay}(\$X)$ from $\mathsf{Contract}_{\mathsf{Pay}}$,
- if $\$X \leq \mathsf{Contract}_{\mathsf{Pay}}.\mathsf{deposits}_i + \mathsf{paid}_i - \mathsf{pay}_i - \mathsf{wd}_i$:
  - append $\$X$ to arr_i
  - pay_i += $\$X$

on **input** $\texttt{withdraw}(\$X)$ from $\mathsf{Contract}_{\mathsf{Pay}}$,
- if $\$X \leq \mathsf{Contract}_{\mathsf{Pay}}.\mathsf{deposits}_i + \mathsf{paid}_i - \mathsf{pay}_i - \mathsf{wd}_i$ then wd_i += $\$X$

Fig. 4: Implementation of a duplex payment channel with the Π_{State} primitive.

4.2 Modeling payment channels with state channels

To demonstrate the use of the Π_{state} abstraction, we now construct a duplex payment channel (e.g., as in [2, 18, 15]). In Figure 4, we give a construction that realizes Π_{Pay} given a state channel protocol Π_{State}. Our construction consists of

1) an update function, U_{Pay}, which defines the structure of the state and the inputs provided by the parties, 2) an auxiliary contract $\mathsf{Contract}_{\mathsf{Pay}}$ that handles deposits and withdrawals and 3) local behavior for each party.

The update function U_{Pay} encodes the state with two fields, cred_i and $\mathsf{deposists}_i$, instead of a single "balance" field. This encoding is designed to cope with the fact that blockchain transactions are not synchronized with state updates and may arrive out of order. So when $\mathsf{Contract}_{\mathsf{Pay}}$ receives a deposit of $\mathbf{coins}(x)$, we have it accumulate in a monotonically increasing value, $\mathsf{deposits}_i$, that can safely be passed to `aux_input`. The state then includes cred_i as a balance offset, such that the balance available is $\mathsf{deposits}_i + \mathsf{cred}_i$.

Since the state channel abstraction handles synchronization between the parties, when reasoning about the security of the payment channel we need only to consider the update function. Notice that each party's balance can only be lowered by a pay input provided by them, and the overall sum of balances, withdrawals, and deposits is maintained as an invariant.

As a consequence of our generic state channel, each payment requires two signatures and two rounds of communication, from the sender to the recipient (assuming the sender is the leader, see A.4) and back again. An optimization taken in Lightning and in Raiden is to omit the return trip if receipt of the payment is not necessary. The on-chain dispute resolution requires the same number of transactions as in Lightning: one transaction establishes the deadline (`dispute`, `evidence`, and `input` can be invoked simultaneously) and `resolve` applies the next update on-chain.

5 Linked Payments from State Channels

In this section we complete the Sprites construction, focusing on how we link payments together along a path of payment channels from a sender to receiver. The challenge is to ensure the collateral provided by intermediaries is returned to them within a bounded time.

Our construction for linked payment chains is modular, relying on multiple instances of duplex channels Π_{Pay}. Like Π_{Pay}, the definition for linked payments consists of an update function U_{Linked}, an auxiliary contract, and a local protocol for each party. Figure 5 defines the update function, the auxiliary contract and the preimage management contract, $\mathsf{Contract}_{\mathsf{PM}}$ (a contract accessed through the auxiliary contract). The update function U_{Linked} is an outer layer around the U_{Pay} function (Figure 4), but extends state with a status flag to include support for conditional payments.

To establish a path of linked payments off-chain, the initial sender P_1 first creates a secret x, shares it with the recipient P_ℓ, and creates an outgoing conditional payment to P_2 using $h = \mathcal{H}(x)$. Each subsequent party P_i in turn, upon receiving the incoming conditional payment, establishes an outgoing conditional payment to P_{i+1}. Once the recipient P_ℓ receives the final conditional payment, it multicasts x to every other party.

When a conditional payment is in-flight, all parties on the path must wait for the preimage to be revealed to them by the receiver, P_ℓ, before T_{Crit}; if it arrives on time P_i completes the outgoing payment off-chain. If the outgoing payment doesn't complete before T_{Crit}, but P_i has received the preimage, then P_i sends it to the preimage manager, $\mathsf{Contract}_{\mathsf{PM}}$. By T_{Expiry}, if the preimage was published the payment is completed; otherwise, it is canceled (by all P_i, because publishing the preimage is a global event). Finally, if after T_{Dispute} the payment has failed to complete or cancel, the party raises a dispute and forces the payment be completed or canceled on-chain.

Security Analysis of Linked Payments. Our model begins with parties P_i through P_ℓ that have established $\ell - 1$ payment channels, such that Π^i_{Pay} denotes the payment channel established between parties P_i and P_{i+1}. Given the state channel abstraction, it is easy to check that the desired properties described earlier (Section 2.3) are exhibited by this protocol:

(Liveness) If all parties P_1 through P_ℓ are honest, and if sufficient balance is available in each payment channel, then the chained payment completes successfully after $O(\ell)$ rounds. More specifically, for each channel Π_{Pay}, the outgoing balance $\Pi^i_{\mathsf{Pay}}.\mathsf{cred}_R$ is increased by $\$x$ and each incoming balance $\Pi^i_{\mathsf{Pay}}.\mathsf{cred}_L$ is decreased by $\$x$. If the sender and receiver, P_1 and P_ℓ, are both honest the payment either completes or cancels after $O(\ell + \Delta)$ rounds.

(No counterparty risk) Even if some parties are corrupt, no honest party on the path should lose any money. In the dispute case, the preimage manager, $\mathsf{Contract}_{\mathsf{PM}}$, acts like a global condition. If the preimage manager receives x before time T_{Expiry}, then *every* conditional payment that is disputed will complete. Otherwise they are canceled. Therefore, for an honest party that receives x before $T_{\mathsf{Expiry}} - \Delta$, it is safe to `complete` their outgoing payment. In the worst case then can use the preimage manager and claim their incoming payment.

Implementation and performance analysis. We created a proof-of-concept implementation using Solidity and pyethereum available online[8]. In the typical case, the off-chain communication pattern in Sprites is similar to that of Lightning. We need one round of communication between each adjacent pair of parties to open each conditional payment, and finally one round to complete all the payments.

In the worst-case scenario, each channel that must be resolved via the dispute handler requires one on-chain transaction to initiate the dispute and send the preimage to $\mathsf{Contract}_{\mathsf{PM}}$, and, later, a transaction to complete the dispute and withdraw the balance (Section4.1). Based on our implementation, the dispute process costs up to 137294 gas per disputed channel, or $\approx$ \$0.20 in November 2018.For comparison, in the Lightning Network the typical cost of closing a channel is 0.00002025 BTC ($\approx$ \$0.072)[9].

[8] `https://github.com/amiller/sprites`

[9] Representative Lightning transaction `https://www.blockchain.com/btc/tx/c9e6a9200607871e18fcfdd54dcb0da17ac8eca005101b82c8a807def9885d3e`

Protocol $\Pi_{\mathsf{Linked}}(\$X, T, P_1, ...P_\ell)$

Let $T_{\mathsf{Expiry}} := T + 6\ell + \Delta$.
Let $T_{\mathsf{Crit}} := T_{\mathsf{Expiry}} - \Delta$
Let $T_{\mathsf{Dispute}} := T_{\mathsf{Expiry}} + \Delta + 3$.

Update Function
$U_{\mathsf{Linked},\$X}(\mathsf{state}, \mathsf{in}_{\mathtt{L}}, \mathsf{in}_{\mathtt{R}}, \mathsf{aux}_{in})$

if $\mathsf{state} = \bot$, set $\mathsf{state} := (\mathtt{init}, \bot, (0,0))$
parse state as $(\mathsf{flag}, h, (\mathsf{cred}_{\mathtt{L}}, \mathsf{cred}_{\mathtt{R}}))$
parse $\mathsf{in}_{\mathtt{i}}$ as $(\mathsf{cmd}_i, \mathsf{in}_{\mathtt{i}}^{\mathsf{Pay}})$, for $i \in \{\mathtt{L}, \mathtt{R}\}$
if $\mathsf{cmd}_{\mathtt{L}} = \mathtt{open}(h')$ and $\mathsf{flag} = \mathtt{init}$, then
  set $\mathsf{cred}_{\mathtt{L}}$ $-= \$X$, $\mathsf{flag} := \mathtt{inflight}$, and $h := h'$
else if $\mathsf{cmd}_{\mathtt{L}} = \mathsf{complete}$ and $\mathsf{flag} = \mathtt{inflight}$,
  set $\mathsf{cred}_{\mathtt{R}}$ $+= \$X$, and $\mathsf{flag} := \mathtt{complete}$
else if $\mathsf{cmd}_{\mathtt{R}} = \mathsf{cancel}$ and $\mathsf{flag} = \mathtt{inflight}$,
  set $\mathsf{cred}_{\mathtt{L}}$ $+= \$X$ and $\mathsf{flag} := \mathtt{cancel}$
else if $\mathsf{cmd}_{\mathtt{R}} = \mathtt{dispute}$ or $\mathsf{cmd}_{\mathtt{L}} = \mathtt{dispute}$, and $\mathsf{flag} = \mathtt{inflight}$, and current time $> T_{\mathsf{Expiry}}$, then
  $\mathsf{aux}_{\mathsf{out}} := (\mathtt{dispute}, h, \$X)$ and $\mathsf{flag} = \mathtt{dispute}$
let $\mathsf{state}^{\mathsf{Pay}} := (\mathsf{cred}_{\mathtt{L}}, \mathsf{cred}_{\mathtt{R}})$
$(\mathsf{aux}_{\mathsf{out}}^{\mathsf{Pay}}, \mathsf{state}^{\mathsf{Pay}}) := U_{\mathsf{Pay}}(\mathsf{state}^{\mathsf{Pay}}, \mathsf{in}_{\mathtt{L}}^{\mathsf{Pay}}, \mathsf{in}_{\mathtt{R}}^{\mathsf{Pay}}, \mathsf{aux}_{\mathsf{in}})$
set $\mathsf{state} := (\mathsf{flag}, h, \mathsf{state}^{\mathsf{Pay}})$
return $(\mathsf{state}, (\mathsf{aux}_{\mathsf{out}}, \mathsf{aux}_{\mathsf{out}}^{\mathsf{Pay}}))$

Auxiliary contract $\mathsf{Contract}_{\mathsf{Linked}}$

Copy the auxiliary contract from Figure 5, renaming the `output` handler to $\mathtt{output}^{\mathsf{Pay}}$
on **contract input** $\mathtt{output}(\mathsf{aux}_{\mathsf{out}}^*)$:
  parse $\mathsf{aux}_{\mathsf{out}}$ as $(\mathsf{aux}_{\mathsf{out}}, \mathsf{aux}_{\mathsf{out}}^{\mathsf{Pay}})$
  if $\mathsf{aux}_{\mathsf{out}}^*$ parses as $(\mathtt{dispute}, h, \$X)$ then
    if $\mathsf{PM}.\mathtt{published}(T_{\mathsf{Expiry}}, h)$, then
      $\mathsf{deposits}_{\mathtt{R}}$ $+= \$X$
    else
      $\mathsf{deposits}_{\mathtt{L}}$ $+= \$X$
  $\mathsf{aux}_{\mathsf{in}} := (\mathsf{deposits}_{\mathtt{L}}, \mathsf{deposits}_{\mathtt{R}})$
  invoke $\mathtt{output}^{\mathsf{Pay}}(\mathsf{aux}_{\mathsf{out}}^{\mathsf{Pay}})$

Global Contract $\mathsf{Contract}_{\mathsf{PM}}$

initially $\mathsf{timestamp}[]$ is an empty mapping
on contract input $\mathtt{publish}(x)$ at time T:
  if $\mathcal{H}(x) \notin \mathsf{timestamp}$: then set $\mathsf{timestamp}[\mathcal{H}(x)] := T$
constant function $\mathtt{published}(h, T')$:
  return `True` if $h \in \mathsf{timestamp}$ and $\mathsf{timestamp}[h] \leq T'$
  return `False` otherwise

Fig. 5: Smart contract for protocol Π_{Linked} that implements linked payments with the Π_{State} primitive. Parts of $U_{\mathsf{Linked},\$X}$ that are delegated to the underlying U_{Pay} are colored blue to help readability. See Appendix (Figure 6) for local behavior.

6 Related Works

The first off-chain protocols were Bitcoin payment channels, due to Spilman [23]. These channels, however, only allow for payments to be made in one direction — from Alice to Bob. Subsequent channel constructions by Decker and Wattenhofer [4] as well as Poon and Dryja [19] supported "duplex" payments back-and-forth from either part, however, they require an every growing list of keys to defend against malicious behavior.

Improvements to Payment Channels. Gervais et al. [8] proposed a protocol for rebalancing payment channels entirely off-chain. Dziembowski et al. [5] developed a mechanism for virtual payment channel overlays, enabling two parties with a path to establish a rapid payment channel between them. A limitation of

payment channels is that their security requires honest parties to be online at all times. McCorry et al. [13] discuss how channel participants can hire third parties to arbitrate channel disputes (see Section 2.3). These ideas are all complementary to our work and we think could be combined.

Routing in payment channels. While in our presentation we assume the payment path is given, in reality finding a route is a challenging problem. Sprites can be used with proposed routing protocols [20, 21, 22] which are complimentary. Although the T_{Expire} deadline is defined in terms of the path length, ℓ (see Fig. 5), to avoid revealing path length for privacy, we can pad the deadline to a conservative upper bound. Given that measurements of the Lightning Network [6] today show a diameter of 8, we suppose an upper bound of $\ell = 16$ is conservative. The expiration time is dominated by the block time Δ (1 day, if we follow Lightning and Raiden).

Malavolta et al. [12] identified a potential for deadlock when multiple concurrent payments need to use the same link. They propose a solution, Rayo, that guarantees non-blocking progress. Rayo assumes the existence of global identifiers for payments and a global payment ordering. We conjecture such a global identifier can be implemented on top of Sprites payment channels; for example, it can be derived from the channel address and hash of the proposed state.

Credit networks. Malavolta et al. [14] developed a protocol for privacy-preserving credit networks. The main difference between a payment channel and a credit line is that payment channel balances are fully backed by on-chain deposits, and can be settled without any counterparty risk, where lines of credit seem inherently to expose counterparty risk.

7 Conclusion

Cryptocurrencies face several ongoing challenges: they must be scaled up beyond several transactions per second to accommodate increasing user demand and compete with centralized alternatives. Off-chain payment channel networks are currently a leading proposal to scale blockchain-based cryptocurrencies. However, the current state of the art payment network scaling solutions, like Lightning [19], require collateral to be locked up for a maximum period that scales linearly with the number of hops, $O(\ell\Delta)$. In this paper, we introduced a construction of payment channels and networks, Sprites, that drastically improves upon the current worst-case locktime—reducing it to a constant, $O(\ell + \Delta)$. We also introduce a modular construction for payment channels, building on top of a generalized state channel primitive. State channels abstract away all blockchain interaction, allow arbitrary off-chain protocols (e.g. channels and linked payments) to be more easily defined and analyzed.

Our constant locktime construction relies on a global contract mechanism, which is easily expressed in Ethereum, although it cannot (we conjecture) be emulated in Bitcoin without modification to its scripting system. We therefore pose the following question for future work: what minimal modifications to Bitcoin script would enable constant locktimes?

References

1. Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In *Crypto (2)*, pages 421–439, 2014.
2. Iddo Bentov, Ranjit Kumaresan, and Andrew Miller. Instantaneous decentralized poker. Asiacrypt (to appear) https://arxiv.org/abs/1701.06726, 2017.
3. Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. *OSDI*, 1999.
4. Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*, pages 3–18. Springer, 2015.
5. Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment channels over cryptographic currencies.
6. hashxp. https://hashxp.org/lightning. https://hashxp.org/lightning, September 2018.
7. Emin Gun Sirer Ittay Eyal, Adem Efe Gencer and Robbert van Renesse. Bitcoin-NG: A scalable blockchain protocol. In *NSDI*, 2016.
8. Rami Khalil and Arthur Gervais. Revive: Rebalancing off-blockchain payment networks. ACM CCS (to appear), 2017. http://eprint.iacr.org/2017/823.
9. Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *USENIX Security Symposium*, 2016.
10. Ranjit Kumaresan and Iddo Bentov. How to use bitcoin to incentivize correct computations. In *CCS*, 2014.
11. Loi Luu, Viswesh Narayanan, Kunal Baweja, Chaodong Zheng, Seth Gilbert, and Prateek Saxena. SCP: A computationally-scalable byzantine consensus protocol for blockchains. In *CCS*, 2016.
12. Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks, 2017.
13. Patrick McCorry, Surya Bakshi, Iddo Bentov, Sarah Meiklejohn, and Andrew Miller. Pisa: Arbitration outsourcing for state channels. https://www.cs.cornell.edu/~iddo/pisa.pdf, 2018.
14. Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. Silentwhispers: Enforcing security and privacy in decentralized credit networks. 2016.
15. Raiden Network. http://raiden.network/, 2015.
16. Tier Nolan. Alt chains and atomic transfers. bitcointalk.org, May 2013.
17. Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. Cryptology ePrint Archive, Report 2016/917, 2016. http://eprint.iacr.org/2016/917.
18. Dennis Peterson. Sparky: A lightning network in two pages of solidity. http://www.blunderingcode.com/a-lightning-network-in-two-pages-of-solidity.
19. J. Poon and T. Dryja. The bitcoin lightning network: Scalable off-chain instant payments. 2016. https://lightning.network/lightning-network-paper.pdf.
20. Pavel Prihodko, Slava Zhigulin, Mykola Sahno, Aleksei Ostrovskiy, and Olaoluwa Osuntokun. Flare: An approach to routing in lightning network. Whitepaper. http://bitfury.com/content/5-white-papers-research/whitepaper_flare_an_approach_to_routing_in_lightning_network_7_7_2016.pdf, 2016.
21. Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. Settling payments fast and private: Efficient decentralized routing for path-based transactions. *NDSS*, 2018.

22. Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, Giulia Fanti, and Pramod Viswanath. Routing cryptocurrency with the spider network. *arXiv preprint arXiv:1809.05088*, 2018.
23. Jeremy Spilman. Anti dos for tx replacement. `https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2013-April/002433.html`, 2013.

A Appendix

A.1 Acknowledgements

This work is funded in part by NSF grants CNS-1801321 and CNS-1617676 and a gift from DTR Foundation.

A.2 Further Discussion

Supporting fees Participants who act as intermediaries in a payment path contribute their resources to provide a useful service to the sender and recipient. The intermediaries' collateral is tied up for the duration of the payment, but the sender and recipient would not be able to complete their payment otherwise. Therefore the sender may provide a fee along with the payment, which can be claimed by each intermediary upon completion of the payment. To achieve this, each conditional payment along the path should include a slightly less amount than the last; the difference can be pocketed by the intermediary upon completion. The following example provides a \$1 fee to each intermediary, P_2 and P_3.

$$P_1 \xrightarrow[\mathsf{PM}[h,T_{\mathsf{Expiry}}]]{\$X+2} P_2 \xrightarrow[\mathsf{PM}[h,T_{\mathsf{Expiry}}]]{\$X+1} P_3 \xrightarrow[\mathsf{PM}[h,T_{\mathsf{Expiry}}]]{\$X} P_4$$

A.3 Details of the Linked Payments Construction

In the body of the paper (Section 4) we presented the update function and auxiliary smart contracts (Figure 5) for the state channel protocol Π_{Linked}. In Figure 6 we define the local behavior of the parties.

A.4 Local Protocol for the State Channel Construction

In the body of the paper (Figure 3) we presented the smart contract portion of the state channel protocol. In Figure 7 we define the local behavior of the parties.

Protocol $\Pi_{\mathsf{Linked}}(\$X, T, P_1, ...P_\ell)$

Local protocol for sender, P_1

on **input** `pay` from the environment:
  $x \xleftarrow{\$} \{0,1\}^\lambda$, and $h \leftarrow \mathcal{H}(x)$
  pass $(\texttt{open}, h, \$X, T_{\mathsf{Expiry}})$ as input to Π^1_{State}
  send $(\texttt{preimage}, x)$ to P_ℓ
  if $(\texttt{preimage}, x)$ is received from P_2 before T_{Expiry}, then pass `complete` to Π^1_{State}
at time $T_{\mathsf{Expiry}} + \Delta$, if $\mathsf{PM.published}(T_{\mathsf{Expiry}}, h)$, then
  pass input `complete` to Π^1_{State}
at time T_{Dispute}, then pass input `dispute` to Π^1_{State}

Local protocol for party P_i**, where** $2 \leq i \leq \ell - 1$

on receiving state $(\texttt{inflight}, h, \_)$ from $\Pi^{i-1}_{\mathsf{State}}$
  store h
  provide input $(\texttt{open}, h, \$X, T_{\mathsf{Expiry}})$ to Π^i_{State}
on receiving state $(\texttt{cancel}, \_, \_)$ from Π^i_{State},
  provide input $(\texttt{cancel})$ to $\Pi^{i-1}_{\mathsf{State}}$
on receiving $(\texttt{preimage}, x)$ from P_ℓ before time T_{Crit}, where $\mathcal{H}(x) = h$,
  pass `complete` to Π^i_{State}
  at time T_{Crit}, if state $(\texttt{complete}, \_, \_)$ has not been received from Π^i_{State}, then
    pass contract input $\mathsf{PM.publish}(x)$
at time $T_{\mathsf{Expiry}} + \Delta$,
  if $\mathsf{PM.published}(T_{\mathsf{Expiry}}, h)$, pass `complete` to Π^i_{State}
  otherwise, pass `cancel` to $\Pi^{i-1}_{\mathsf{State}}$
at time T_{Dispute}, pass input `dispute` to $\Pi^{i-1}_{\mathsf{State}}$ and Π^i_{State}

Local protocol for recipient, P_ℓ

on receiving $(\texttt{preimage}, x)$ from P_1, store x and $h := \mathcal{H}(x)$
on receiving state $(\texttt{inflight}, h, \_)$ from $\Pi^{\ell-1}_{\mathsf{State}}$,
  multicast $(\texttt{preimage}, x)$ to each party
  at time T_{Crit}, if state $(\texttt{complete}, \_, \_)$ has not been received from $\Pi^\ell_{\mathsf{State}}$, then
    pass contract input $\mathsf{PM.publish}(x)$
at time T_{Dispute}, pass input `dispute` to $\Pi^{\ell-1}_{\mathsf{State}}$

Fig. 6: Construction for Π_{Linked} with the Π_{State} primitive. (Local portion only. See Figure 5 for the smart contract portion.) Portions of the update function $U_{\mathsf{Linked},\$X}$ that are delegated to the underlying U_{Pay} update function (Figure 5) are colored blue to help readability.

Reaching agreement off-chain The main role of the local portion of the protocol is to reach agreement on which inputs to process next. To facilitate this we have one party, P_1, act as the leader. The leader receives inputs from each party, batches them, and then requests signatures from each party on the entire batch. After receiving all such signatures, the leader sends a `COMMIT` message containing the signatures to each party. This resembles the "fast-path" case of a fault tolerant consensus protocol [3]; However, in our setting, there is no need for a view-change procedure to guarantee liveness when the leader fails; instead the fall-back option is to use the on-chain smart contract.

Protocol $\Pi_{\mathsf{State}}(U, P_1, ...P_N)$

Local protocol for the leader, P_1

Proceed in consecutive virtual rounds numbered r:
Wait to receive messages $\{\texttt{INPUT}(v_{r,j}))\}_j$ from each party.
Let in_r be the current state of aux_{in} field in the the contract.
Multicast $\texttt{BATCH}(r, \mathsf{in}_r, \{v_{r,j}\}_j)$ to each party.
Wait to receive messages $\{(\texttt{SIGN}, \sigma_{r,j})\}_j$ from each party.
Multicast $\texttt{COMMIT}(r, \{\sigma_{r,j}\}_j)$ to each party.

Local protocol for each party P_i (including the leader, L)

$\mathsf{flag} := \texttt{OK} \in \{\texttt{OK}, \texttt{PENDING}\}$; $\mathsf{lastRound} := -1$; $\mathsf{lastCommit} := \bot$

Fast Path (while $\mathsf{flag} == \texttt{OK}$): Proceed in rounds r, with $r := 0$

Wait input $v_{r,i}$ from environment. Send $\texttt{INPUT}(v_{r,i})$ to L.
Wait $\texttt{BATCH}(r, \mathsf{in}'_r, \{v'_{r,j}\}_j)$ from L. Discard if $v'_{r,i} \neq v_{r,i}$ OR in'_r not a *recent* aux_{in}.
$(\mathsf{state}, \mathsf{out}_r) := U(\mathsf{state}, \{v_{r,j}\}_j, \mathsf{in}'_r)$
Send $(\texttt{SIGN}, \sigma_{r,i})$ to P_1, $\sigma_{r,i} := \mathsf{sign}_i(r \| \mathsf{out}_r \| \mathsf{state})$
Wait $\texttt{COMMIT}(r, \{\sigma_{r,j}\}_j)$ from L. Discard if $!(\mathsf{verify}_j(\sigma_{r,j} \| \mathsf{out}_r \| \mathsf{state}))$ for each j.
$\mathsf{lastCommit} := (\mathsf{state}, \mathsf{out}_r, \{\sigma_{r,j}\}_j)$; $\mathsf{lastRound} := r$
If $\mathsf{out}_r \neq \bot$, invoke $\texttt{evidence}(r, \mathsf{lastCommit})$.
If `COMMIT` not received within one time-step, then:
if $\mathsf{lastCommit} \neq \bot$, invoke $\texttt{evidence}(r-1, \mathsf{lastCommit})$ and $\texttt{dispute}(r)$

Handling on-chain events

On $\texttt{EventDispute}(\mathtt{r}, \_)$, if $r \leq \mathsf{lastRound}$, invoke $\texttt{evidence}(\mathsf{lastRound}, \mathsf{lastCommit})$.
Else if $r = \mathsf{lastRound} + 1$, then:
Set $\mathsf{flag} := \texttt{PENDING}$, buffer inputs of "waiting" until returning to fast path.
Send $\texttt{input}(r, v_{r,i})$ to the contract.
Wait to receive $\texttt{EventOffchain}(r)$ or $\texttt{EventOnchain}(r)$ from the contract. Attempt to invoke $\texttt{resolve}(r)$ if Δ elapses, then continue waiting. In either case:
$\mathsf{state} := \mathsf{state}'$
$\mathsf{flag} := \texttt{OK}$
Enter the fast path with $r := r + 1$

Fig. 7: Construction of a general purpose state channel parameterized by transition function U. (Local portion only, for the smart contract see Figure 3.)