Sprites: Payment Channels that Go Faster than Lightning

Andrew Miller *UIUC*
Iddo Bentov *Cornell University*
Ranjit Kumaresan *Microsoft Research*
Patrick McCorry *Newcastle University*

Abstract

It is well known that Bitcoin, Ethereum, and other blockchain-based cryptocurrencies are facing hurdles in scaling to meet user demand. One of the most promising approaches is to form a network of "off-chain payment channels," which are backed by on-chain currency but support rapid, optimistic transactions and use the blockchain only in case of disputes.

We develop a novel construction for payment channels that reduces the worst-case "collateral cost" for off-chain payments. In existing proposals, particularly the Lightning Network, a payment across a path of ℓ channels requires locking up collateral for $O(\ell\Delta)$ time, where Δ is the time to commit a on-chain transaction. Our construction reduces this cost to $O(\ell+\Delta)$. We formalize our construction in the simulation-based security model, and provide an implementation as an Ethereum smart contract. Our construction relies on a general purpose primitive called a "state channel," which is of independent interest.

1 Introduction

Cryptocurrencies such as Bitcoin, Ethereum, and others, derive their security from wide replication, which unfortunately comes at the expense of limited scalability. A leading proposal for improved scaling of cryptocurrencies is to form a network of "off-chain" rapid payment channels, which act like credit lines secured by "on-chain" currency. In this vision of the future, cryptocurrencies will largely be used as collateral, so interaction with the blockchain directly will rarely be needed.

A chief concern for the feasibility of payment networks is whether the "collateral costs" will be prohibitive. In general, collateral cost is the lost opportunity (in dimensions of money $\times$ time) occurring when funds are held in escrow instead of being invested profitably. Currency deposited in a payment channel can earn fees when it is used to facilitate linked payments. However, each in-flight payment along a channel must reserve a portion of that channel's available collateral, preventing its use elsewhere until the payment is settled. In the optimistic case, payments complete quickly, requiring only off-chain point-to-point messages; but if some party fails, the collateral can be tied up for a significant duration, until the balance can be settled on-chain. The more hops in a payment path, the more collateral must

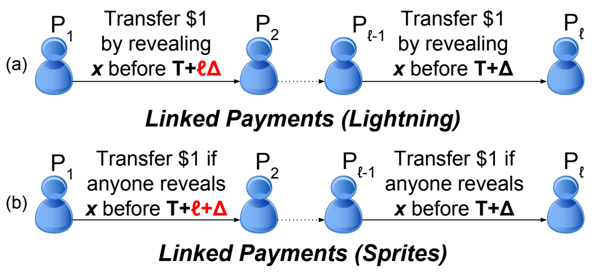

(a)
P1
Transfer $1 by revealing x before T+ℓΔ
P2
Pℓ-1
Transfer $1 by revealing x before T+Δ
Pℓ
Linked Payments (Lightning)
(b)
P1
Transfer $1 if anyone reveals x before T+ℓ+Δ
P2
Pℓ-1
Transfer $1 if anyone reveals x before T+Δ
Pℓ
Linked Payments (Sprites)

Figure 1: Cryptocurrencies like Bitcoin and Ethereum can serve as collateral for a scalable payment network (i.e. a credit network without counterparty risk) [26, 8]. Payment channels (a) let one party rapidly pay another using available collateral, requiring a blockchain transaction only in case of dispute. Payments across multiple channels (b) can be linked using a common condition (such as revealing the preimage of a hash, h). We contribute a novel payment channel (c) improving the worst case delay for ℓ-hop payments from $O(\ell\Delta)$ to $O(\ell+\Delta)$.

be reserved: $O(\ell\$X)$ in total for a $O(X)$ sized payment spanning ℓ channels. Furthermore, due to limitations of the current state-of-the-art payment channels, each link in the path adds an additional worst-case delay. This additional delay is determined by the worst-case confirmation time for an on-chain transaction, which we denote by Δ (i.e., an on-chain transaction may take Δ times longer than an ordinary off-chain message). Thus the worst-case delay is $O(\ell\Delta)$, and so the total collateral cost of a $\$X$ payment over a path of length ℓ is $O(\ell^2\$X\Delta)$.

In this paper we present an improved construction of payment channels, called "Sprites," that reduce the timeout delay from $O(\ell\Delta)$ to $O(\ell+\Delta)$, resulting in a total collateral cost of $O(\ell\$X\Delta)$ for a payment over ℓ hops. Our solution makes use of a feature available today in Ethereum smart contracts, but that cannot (we conjecture) be implemented in Bitcoin — in particular, the ability for a transaction to depend on a "global" event recorded in the blockchain.

Our construction is highly modular; a key contribution of our work is the development of a useful general primitive called a "state channel," which allows two or more parties to maintain an arbitrary off-chain shared process, which can be synchronized on demand (or in case of a dispute) with the blockchain. This abstraction neatly encapsulates the underlying cryptography; by making use of it, our payment channel constructions do not mention digital signatures at all. We formally prove the security of our constructions in the simulation-based framework, and provide an implementation (cf. [19]) in the Solidity smart contract language and pyethereum testbed.

It remains unknown what resulting topology will emerge from cryptocurrency payment channel networks. In the decentralized ideal, users would establish channels with peers in their social network (i.e., forming a scale-free network structure). However, worryingly, high collateral costs associated with long payment paths may create an economic pressure towards a more centralized structure, with most individuals forming channels with only a small number of well connected bank-like hubs. By reducing the collateral cost, our work takes a significant step towards realizing the vision of decentralized credit network.

1.1 Overview of our constructions.

State channels. Our constructions are centered around a general purpose "state channel" primitive. A state channel is established between two (or an arbitrary number of) parties, and represents a consistent shared process, that evolves according to an arbitrary transition function and to which either party can provide input. Each time 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 fails (or responds with invalid data), remaining parties can trigger a "dispute" by submitting the most recent agreed-upon state to the blockchain, along with inputs for the next round. Besides the inputs provided by parties, state transitions can also depend on auxiliary input present on the blockchain itself (e.g., additional currency deposits submitted by either party). State transitions can also include auxiliary outputs with on-chain side effects, e.g. triggering a transfer of on-chain currency.

In our formal construction (Section 4.5) we give a security definition that expresses the desired liveness and consistency properties, and prove that our protocol is correctly designed.

Linked payment channels from state channels. Our bi-directional payment channel (Section 4) is a simple specialization of the state channel functionality, where the state maintains each party's current balance, and deposits/withdrawals are implemented as auxiliary I/O.

To support linked payments, we use a novel variation of the standard "hashed timelock contract" technique [22, 26]. We create a global contract, called the PreimageManager (PM), which records assertions of the form "the preimage x of hash $h = \mathcal{H}(x)$ was published before time T_{Expiry}." We then augment the payment channel construction with a "conditional payment" feature,

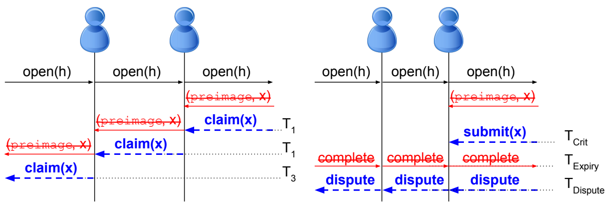


Figure 2: The worst-case delay scenario, in Lightning (left) and in Sprite (right). The two parties shown are "petty," dropping off-chain messages (~~striken red~~) after the initial open, and sending on-chain transactions (blue) only at the last minute. Disputes in Lightning may cascade, whereas in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

where each party in the path reserves conditioned on the presence of a record in this global contract. The following notation indicates a conditional payment of $X from P_1 to P_2, which 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}.

$$P_1 \xrightarrow[\mathsf{PM}[h, T_{\mathsf{Expiry}}]]{\$X} P_2$$

To form a linked payment along a path of channels, each party opens a conditional payment with the party to his right, each with the same hash and expiry.[1]

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

The preimage x is initially shared among the sender and recipient; after the final conditional payment to the recipient is opened, the recipient publishes x, and each party completes its outgoing payment. Optimistically, (i.e., if no parties fail), the process completes after only $\ell + 1$ rounds. Otherwise, in the worst case, any honest parties that completed their outgoing payment submit x to the PM contract, guaranteeing that their incoming payment will complete, and thus conserving their net balance. This procedure ensures that each party's collateral is tied up for a maximum of $O(\ell + \Delta)$ rounds.

Comparison with Lightning. Our construction shows that the Lightning channel design is more complicated inherently necessary. We observe three complexities that arise due to Bitcoin's limited scripting language:

1) Lightning requires a "revocation keys" mechanism each time the "direction" of the channel changes, i.e. each time Alice pays Bob and then Bob pays

[1]The intermediary nodes in a path can be incentivized to participate in the route if the sender allocates an extra fee that will be shared among them. To aid clarity and simplify notation, we omit the description of such fees in the body, but provide more detail in Appendix B.1.

Alice. To defend against malicious behavior, each party must continuously store a copy of every previously revoked keys. Our simpler mechanism based on signatures over increasing round numbers avoids this cost.

2) In case of a dispute, a lightning message can only be claimed on-chain by *one of the two parties revealing a preimage x*. Each link therefore requires an additional timeout round of at least one blockchain round Δ. By making use of a global event, we can avoid this concern.

3) To deposit additional collateral or withdraw a portion of deposited collateral, a Lightning channel must be closed and then reopened, requiring at least two on-chain transactions. If any conditional payments are currently "in-flight," then these must also be settled with on-chain transactions.

With regard to observation 2), consider for example party P_1 that sends \$$X$ to party P_5 via a payment route that uses the intermediaries P_2, P_3, P_4:

$$P_1 \xrightarrow[h,T+3\Delta]{\$X} P_2 \xrightarrow[h,T+2\Delta]{\$X} P_3 \xrightarrow[h,T+\Delta]{\$X} P_4 \xrightarrow[h,T]{\$X} P_5$$

These arrows denote that each $P_{i\geq 2}$ can collect the \$$X$ amount by submitting an on-chain transaction that reveals a preimage x of h before time (or block number) $T+(5-i)\Delta$. The reason for the incremental timeouts $\Delta, 2\Delta, 3\Delta$ is that a malicious party may reveal x just before a timeout. For example, P_4 may claim P_3's outgoing payment by publishing a transaction with x right at time $T+\Delta$; since P_3 must subsequently submit a transaction containing x to the preceding node, P_3 may need an extra grace period (until time $T+2\Delta$) to account for delays in blockchain transactions. Hence, in Lightning, the amount of time for which the collateral may be locked depends on the length of the path. In Sprites, these incremental timeouts are avoided via the global PM contract, which implements a condition that can be consistently observed by all the payment transactions (this is done by updating the state of the contract). The worst-delay dispute scenario for Lightning and for Sprite is illustrated in Figure 2.

In Bitcoin/Lightning, there does not seem to be a good way to have just one Δ timeout. The difficulty arises from the need to ensure that the preimage x will be available on-chain to all the honest parties, so that they will have enough time to create a transaction that depends on x. Unfortunately, the identities of the parties on the path are not known ahead of time. Thus, while a transaction with outputs that refer to their identities can be prepared and signed off-chain (since in the optimistic case we wish to avoid on-chain transactions), this would not be secure against double-spending. Alternatively, one may consider a protocol in which an entity that knows the entire path will relay x to all parties, immediately after the receiver of the payment reveals it. However, this requires a much stronger trust model, since a corrupt entity who does not relay x can cause honest parties to lose their funds. By contrast, our stateful approach is trust-free. Note that cleaning the state (i.e., removing old entries from the PM table) can be supported via a zero-fee transaction the frees storage proportional to the gas that it consumes.

Several proposals, notably the Raiden network [21], make use of Ethereum to implement simplified payment networks [25, 28, 21]. All of these embody observation 1), using signatures and round numbers rather than revoked secrets. Surprisingly, observations 2) and 3) have so far been overlooked; these proposals otherwise mimic Lightning, and fail to make use of Ethereum's generality.

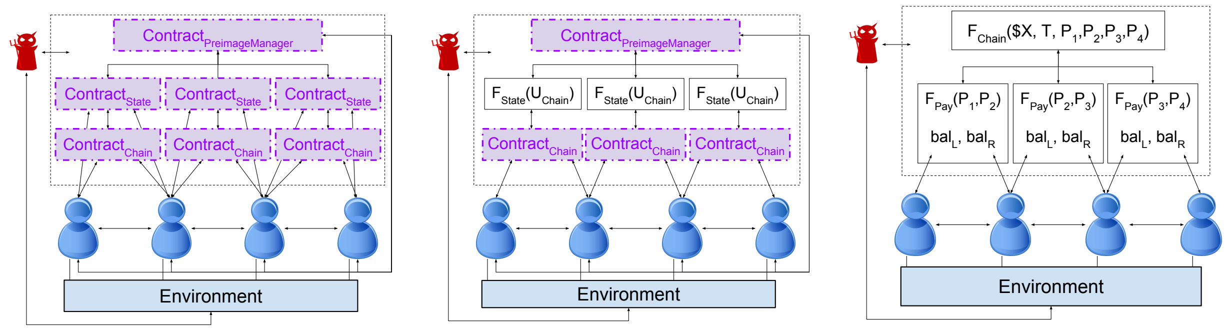
Highlights of our modeling approach. We formalize our constructions and security definitions in a simulation based framework, combining elements from Universal Composability [4] and from the Ethereum smart contract programming model. We describe our target specification as an ideal functionality, logically composed of a payment channel $\mathcal{F}_{\mathsf{Pay}}$ between each party of parties, along with a global "linked payment" functionality $\mathcal{F}_{\mathsf{Linked}}$ that ensures that conditional payments complete or fail in an atomic way.

Since our modular construction relies on an intermediate state channel primitive, $\mathcal{F}_{\mathsf{State}}$, we also model a "hybrid world" that includes both ideal functionalities (which receive input directly from parties) as well as blockchain smart contracts (which implicitly received delayed inputs from parties and enforce rules such as conserved balance of currency). These scenarios are illustrated in Figure 3.

2 Related Work

Unsecured credit networks. Malavolta et al. [17] develop protocols for privacy-preserving, but "unsecured" credit networks, i.e., they represent credit lines that are not "backed" by anything. Links in their credit network represent "unsecured credit," The lines represent no particular guarantee that the credit is backed by anything. Hence, credit lines in these networks model risk of counterparty default.

Silent Whispers does not guarantee payments complete atomically. This is justified by their setting, which is where credit is voluntarily established between nodes (an attacker can reneging on credit they have offered anyway), can be reneged on, without cost.



(a) Real World ("Smart Contracts" and party-to-party communication)

(b) Hybrid World (Smart contracts, plus the generic "state channel" primitive)

(c) Ideal World

Figure 3: Illustration of our main formalism: a construction of payment channels supporting "atomic" chaining (the ideal world, (c)), using a system of Ethereum smart contracts (the real world, (a)). Our construction is modular (b), using a general-purpose "state channel" functionality, $\mathcal{F}_{\mathsf{State}}$, which is of independent interest (Section 4.5).

Their protocol relies on a well-designated (i.e., arguably centralized) set of "landmark nodes," among which a majority must be trusted to be honest.

It should be noted that [17] also addresses the problem of finding a route between the payer and the payee, while we focus on analyzing the properties of our protocol given that the route has already been found.

Anonymous micropayments. Green and Miers [10] present protocols for off-chain micropayments that provide complete anonymity. However, such protocols require the scripting language of cryptocurrency to support verification of blind signatures and non-interactive zero-knowledge (NIZK) proofs, and thus cannot be integrated with Bitcoin (integration with Ethereum is possible, though the script complexity may be overly complex). See [10] for a comparison with the rather heuristic approach to anonymity that the Lightning network takes.

Probabilistic micropayments. An alternative that enables off-chain micropayments via a lottery-based construction is presented by Pass and shelat [23]. However, it requires locking a collateral (larger than the total amount of money that can be paid) to avoid a "front-running" attack. Additionally, [23] presents a different protocol that avoids the need for collateral by relying on a verifiable third party (and can be implemented in the current Bitcoin scripting language).

Chiesa et al. [6] design a lottery-based micropayments system that supports anonymity (via cryptocurrency scripts that verify NIZKs), and also provide an economic analysis for the required collateral.

On-chain scaling. Several works seek to improve the Bitcoin protocol in order to support greater throughput. For instance, Bitcoin-NG [11] is a blockchain protocol that incorporates many microblocks between regular blocks, while Byzcoin [12] and SCP [16] are protocols that use a blockchain to bootstrap committees who would execute a Byzantine agreement protocol (PBFT [5]). However, any public ledger protocol can be greatly helped by off-chain channels, see Section 3.1.

Federated sidechains A related proposal is to run a "sidechain," consisting of an off-chain "consensus protocol" run amongst a set of nodes called "functionaries," which jointly control a balance of on-chain deposits [1, 9]. For example, to withdraw from a sidechain requires signatures from a majority (e.g., 5 out of 7) of the functionaries. This can be viewed as a far more centralized variant of protocols such as Byzcoin [12] and SCP [16].

While our protocols guarantee a consistent state if *any* of the users are honest, if a majority of the functionaries misbehave they can create arbitrary failures.

3 Background and Preliminaries

3.1 Bitcoin and Blockchains

The Bitcoin currency itself is virtual, consisting only of balances stored within the records of the blockchain database. The average cost of a transaction on Bitcoin is 10¢. It takes 10 minutes on average for a Bitcoin block to be found, "confirming" the transaction, but since the mining process is random, this often takes much longer. A natural consequence of the 10-minute blocks and the network propagation delay is that 1-block forks are fairly common (e.g., occur once or twice per day on average), users are typically advised to wait for multiple (e.g., 6) confirmations before considering a transaction finalized. It is well-known that Bitcoin, the most popular cryptocurrency to date (representing a \$15B USD market cap as of February 2017), has severe performance limits. For a transaction to be confirmed takes an hour in expectation, and the overall network is limited to a throughput of around 7 transactions per second [7].

The success of cryptocurrencies has further spurred interest in "blockchain technology," which generally refers to secure databases, shared among multiple distrusting entities. In the case of decentralized cryptocurrencies like Bitcoin, the databases are entirely open to the public, and are powered by the voluntary participation of anonymous "miners." At the other end of the spectrum, "permissioned blockchains" resemble more traditional distributed systems, with a well-defined set of participants and an administrative institution (e.g., a consortium of banks). In both cases, blockchains derive their resiliency through broad replication, which comes at an inherent cost.

Scaling blockchains. Proposed scalability improvements fall in roughly two complementary categories. The first is to make the blockchain itself faster [12, 16, 24, 11]. A recurring theme is that the additional performance comes from introducing strong trust assumptions.

The second category, into which our work falls, is to develop "off-chain" protocols that minimize the use of the blockchain itself. Instead, parties transaction primarily by passing "off-chain" messages (i.e., point-to-point messages from one party to another) amongst themselves, and interact with the blockchain only to settle disputes or to repurpose the funds. Thus, Alice and Bob can make (say) thousands of off-chain transactions between each other, and the public ledger will include only their initial deposit and the transaction that terminated their channel with the final balances. This implies greater scalability, since the nodes who maintain the ledger would never see (and hence do not need to verify) the vast majority of the micropayments. While our techniques are applicable to both settings, they serve to make decentralized systems more competitive with systems that have stronger trust models.

3.2 Blockchain Model

In our development we use the typical idealized model of a Bitcoin-like blockchain [13, 15, 14, 14], as described below. For a detailed survey of cryptocurrency security issues, see Bonneau et al. [3]. For our purposes, a blockchain functions as a shared public database. Any party can write to the blockchain by submitting a "transaction," which propagates throughout the network and is eventually committed into a consistent ordered log. Every party can view all the transactions committed on the blockchain; however, the views are only approximately synchronized. If one party's view comprises the sequence txs_1 and another party's view comprises the sequence txs_2, then it must be that txs_1 is a prefix of txs_2 or vice-versa. Furthermore, if any party's view includes a transaction tx, then every party's view will also include tx after a maximum time bound. To simplify matters, we consider a single time bound, Δ, which bounds the maximum delay "round-trip" time for a blockchain transaction: if some party submits a transaction tx at time T, then every party sees tx confirmed by time $T + \Delta$.

We also make use of "smart contract" programming conventions, inspired by those implemented in Ethereum. Smart contracts give semantic meaning to the transactions submitted to the blockchain; they can be thought of as processes running in the blockchain database that accept input via user-submitted transactions. Smart contracts can be trusted to execute correctly, but do not provide any inherent privacy; the adversary also has the opportunity to reorder and front-run user-submitted inputs.

We make use of several conventions based on features typically found in smart contract programs. Smart contracts have access to a clock (i.e. a "block number") which is approximately synchronized (i.e., to within Δ) of the honest parties. We also assume that the smart contract execution environment provides a built-in notion of **coins**, which can be transferred (conserving total balance) between contracts and parties.

3.3 Simluation Based Security

Our formalism is based on the simulation-based security framework, in particular Universal Composability (UC) [4]. This is a general purpose framework for constructing secure protocols, based on an execution model comprising a system of interactive Turing machines (ITMs). ITMs are defined in a reactive style, by describing how to behave upon receiving a message; the resulting behavior includes modifying a local state, and sending a message to another ITM process.

The UC execution model involves several kinds of processes: an environment, $\mathcal{Z}$, which represents the "external world" and chooses the inputs given to each party and observes the outputs; parties that follow a given protocol Π, and an Byzantine adversary $\mathcal{A}$ that controls corrupted parties. The model also includes functionalities, $\mathcal{F}$, which act like idealized trusted third parties. A functionality serves as the target specification; the "ideal world" contains a functionality that exhibits all the intended properties of the protocol. A functionality in the "real world" is also used to represent network primitives and setup assumptions. A proof in this framework takes the form of a simulator, which translates every attacker $\mathcal{A}$ in the real world into a simulated attacker $\mathcal{S}_{\mathcal{A}}$ in the ideal world, such that the two worlds are indistinguishable to the environment; in other words, the real world is just as good as the ideal world. Thus, we say that protocol Π in the real world, denoted $\mathsf{execReal}(\mathcal{Z},\Pi,\mathcal{A})$, realizes functionality $\mathcal{F}$ in the ideal world, $\mathsf{execIdeal}(\mathcal{Z},\mathcal{F},\mathcal{S}_{\mathcal{A}})$, if the two distributions are indistinguishable.

The simulation based security framework supports modular composition: we can build a protocol that emulates an intermediate functionality $\mathcal{F}_{\mathsf{I}}$ in the real world, and then build a high-level protocol that makes use of $\mathcal{F}_{\mathsf{I}}$ to realize the target functionality $\mathcal{F}$. The composition theorem guarantees that we can make this substitution.

SIDs. In UC, each functionality is associated with a unique string, called the session ID (SID). The SID is essential for the composition theorem, as it ensures that concurrent instances of protocols are kept separate from each other. The practical significance of the session ID is that it is implicitly used as a tag for signatures and hashes to ensure that messages from one protocol instance cannot be replayed in another. To reduce clutter, we elide the handling of SIDs from our presentation.

Smart contracts and functionalities. A new feature in our model is that we define experiments with ideal functionalities as well as "contract" processes, which represent programs running on the blockchain network. This notion is compatible with UC — that is, the multiple functionalities, and the contracts, can be considered as a single combined functionality.

Delayed tasks. Frequently in our ideal functionalities, we use the notation "within $\{\mathrm{R}\}$ rounds: $\{$ *Task* $\}$". This is intended to guarantee that the pseudocode described by *Task* is executed within a bounded time, but the exact time when it is executed is under the control of the adversary. This mechanism is compatible with the traditional UC paradigm; we can imagine implementing a "task queue" mechanism within the functionality. Note that we sometimes specify timeouts in asymptotic notation in order to avoid clutter.

Exceptions in Ideal Functionalities. To simplify our ideal functionality $\mathcal{F}_{\mathsf{Linked}}$, we allow the functionality to raise an exception. Raising an exception immediately sends an `Exception` message to the environment. Since in the real world there is no such mechanism for raising an exception, this would clearly allow the environment to distinguish between the real and ideal worlds. Therefore in our security proof we have the obligation of showing that the simulator we construct never triggers an exception.

4 State Channels and Payment Channels

As a warmup to our full construction, we first present the security definition of a simple bi-directional payment channel, modeled as an ideal functionality $\mathcal{F}_{\mathsf{Pay}}$. We then introduce the state channel primitive $\mathcal{F}_{\mathsf{State}}$, and show how to use it in constructing a payment channel.

4.1 Defining payment channels

A payment channel is established between two parties via a deposit of on-chain currency. Once established with a deposit of on-chain currency, the parties can rapidly pay each other by transferring a portion of this balance using only off-chain messages, resorting to interaction with the blockchain only in case of a dispute or mutual agreement to terminate. Critically, a payment channel should ensure the following (informal) property:

Balance. If either party crashes or deviates from the protocol, then the other party should be able to withdraw their current balance within a bounded time.

We formally model a payment channel with the ideal functionality defined in Figure 4. This functionality is parameterized by the (pseudonyms of) a pair of parties, P_{L} and P_{R}, which are fixed at channel creation time (e.g., via an Ethereum transaction). The functionality keeps track of the local balance of parties, bal_L and bal_R. It also defines "contract input" method, `deposit`, which can be invoked through an on-chain transaction and has the side effect of transferring **coins**. The `pay` method debits the sender's balance immediately, but increments the recipient's balance after a bounded delay. This models the fact that honest parties will immediately mark their collateral as unusable after beginning the payment, but the recipient will only be able to use the payment after the parties reach agreement (or settle a dispute on-chain). Finally, the `withdraw` method triggers a release of **coins**. All of these methods are immediately leaked to the adversary, reflecting the fact that our model does not capture privacy guarantees.

The functionality provides stronger time guarantees when both parties are honest, reflecting that in the optimistic case payments are completed using only off-chain communication; even in the case that some party is corrupt, progress is guaranteed within $O(\Delta)$ rounds by the on-chain dispute process.

4.2 Defining State Channels

State channels interface with a contract C through tapes aux_{in} and aux_{out}. Incoming messages from contracts are

```
**Functionality** $\mathcal{F}_{\mathsf{Pay}}(P_{\mathtt{L}}, P_{\mathtt{R}})$
initially, $\mathsf{bal}_{\mathtt{L}} := 0, \mathsf{bal}_{\mathtt{R}} := 0$
on **contract input** $\mathtt{deposit}(\mathbf{coins}(\$X))$ from $P_i$ :
  within $\Delta$ rounds: $\mathsf{bal}_i \mathrel{+}= \$X$
on **ideal input** $\mathtt{pay}(\$X)$ from $P_i$ :
  discard if $\mathsf{bal}_i < \$X$
  leak $(\mathtt{pay}, P_i, \$X)$ to $\mathcal{A}$
  $\mathsf{bal}_i \mathrel{-}= \$X$
  within $O(1)$ if $P_{\neg i}$ is honest, or else $O(\Delta)$ rounds:
    $\mathsf{bal}_{\neg i} \mathrel{+}= \$X$
    send $(\mathtt{receive}, \$X)$ to $P_{\neg i}$
on **ideal input** $\mathtt{withdraw}(\$X)$ from $P_i$ :
  discard if $\mathsf{bal}_i < \$X$
  leak $(\mathtt{withdraw}, P_i, \$X)$ to $\mathcal{A}$
  $\mathsf{bal}_i \mathrel{-}= \$X$
  within $\Delta$ rounds:
    send $\mathbf{coins}(\$X)$ to $P_i$
```

Figure 4: Functionality model for a bidirectional off-chain payment channel.

delayed by Δ_{in} while outgoing messages are delayed by Δ_{out} to reflect (slow) interaction with an on-chain contract. Off-chain, the channel proceeds in virtual rounds where for each round, inputs from parties are accumulated within a time out Δ_{out}. Following this, $\mathcal{F}_{\mathsf{State}}$ invokes the transition update function U on inputs state (the current state), the inputs $\{v_{r,i}\}$ supplied by the parties, and the external input aux_{in}. The updated state is then sent to all players within a bounded time $\Delta_{\mathsf{OffChain}}$.

Properties Guaranteed by the Functionality The state channel functionality $\mathcal{F}_{\mathsf{State}}$ maintains a consistent sequential view of the state. Second, the input from each party is included in every round, and the state is updated according to correctly-computed steps. Note that the specification provides no inherent input privacy (the inputs are leaked to $\mathcal{A}$), and in fact the adversary can front-run (its inputs in a round can depend on others).

Nonetheless, we describe a generic protocol transformation that implements this functionality for arbitrary states. (This is the general form of the protocol.) Note that $\mathcal{F}_{\mathsf{State}}$ guarantees progress, even if it means making crashed parties "time out" with default values. When all the parties are honest, progress is guaranteed at whatever rate *off-chain* messages can be delivered. Even in the case that some parties are honest, progress is guaranteed at the "on-chain" rate. Our protocol is designed with implementation in Ethereum in mind. The aux_{in} and aux_{out} tapes are intended to be hooked up to other "on-chain"

```
**Functionality** $\mathcal{F}_{\mathsf{State}}(U, C, P_1, \ldots, P_N)$
• Initialize $\mathsf{aux}_{in} := [\bot]$, $\mathsf{ptr} := 0$, $\mathsf{state} := \emptyset$, $\mathsf{buf} := \emptyset$
• on **contract input** $\mathtt{aux\_input}(m)$ from $C$:
    append $m$ to buf, and let $j := |\mathsf{buf}| - 1$
    within $\Delta$: set $\mathsf{ptr} := \max(\mathsf{ptr}, j)$
• proceed sequentially according to virtual rounds $r$, initially $r := 0$
    for each party $P_i$:
      wait to receive input $v_{r,i}$
      if $v_{r,i}$ is not received within $O(1)$ time, set $v_{r,i} := \bot$
      leak $(i, v_{r,i})$ to $\mathcal{A}$
    after receiving all inputs,
      $(\mathsf{state}, o) := U(\mathsf{state}, \{v_{r,i}\}, \mathsf{aux}_{in}[\mathsf{ptr}])$
      send state to each player within time $\Delta$;
      if $o \neq \bot$, within $O(\Delta)$ invoke $C.\mathtt{aux\_output}(o)$
```

Figure 5: Functionality for general purpose state channel

contracts in the eco-system.

4.3 Constructing $\mathcal{F}_{\mathsf{Pay}}$ from $\mathcal{F}_{\mathsf{State}}$

In Figure 6 we give a construction that emulates $\mathcal{F}_{\mathsf{Pay}}$ in the $\mathcal{F}_{\mathsf{State}}$-hybrid world. Our construction consists of an update function, U_{Pay}, which defines the structure of state and inputs provided by parties, an auxiliary contract $\mathsf{Contract}_{\mathsf{Pay}}$ that handles external effects, i.e. deposits and withdrawals, and local behavior for each party.

The protocol Π_{Pay} is somewhat more complicated than the $\mathcal{F}_{\mathsf{Pay}}$ functionality; in particular, while $\mathcal{F}_{\mathsf{Pay}}$ uses a single field representing the available balance of each party, bal_i, the protocol represents this with two fields, cred_i and $\mathsf{deposits}_i$. This encoding is designed to cope with the fact that $\mathcal{F}_{\mathsf{State}}$ only guarantees that auxiliary inputs are loosely synchronized with the state updates. If multiple auxiliary inputs are received within a short time interval (e.g., in the same block), it may be that only the most recent is passed as input to U_{Pay}. Therefore when $\mathsf{Contract}_{\mathsf{Pay}}$ receives a deposit of $\mathbf{coins}(x)$, it accumulates this into a monotonically increasing value, $\mathsf{deposits}_i$, that can safely be passed to aux_{in}. The state then maintains cred_i as a (possibly negative) balance offset, such that P_i's available balance is $\mathsf{deposits}_i + \mathsf{cred}_i$.

In contrast, although the $\mathcal{F}_{\mathsf{State}}$ functionality guarantees that each (non-$\bot$) auxiliary output is eventually processed, though not necessarily in order. Since the $\mathcal{F}_{\mathsf{Pay}}$ functionality makes similar guarantees, it is safe to pass the $\mathsf{wd}_{\{\mathtt{L},\mathtt{R}\}}$ values directly to $C.\mathtt{aux\_output}$.

Since parties' inputs are not validated before being committed, we define U_{Pay} to clamp each party's pay input to within his available balance, and then clamp his wd input to the remaining balance after that.

The local protocol involves translating pay and withdraw invocations into inputs of the form $(\mathsf{pay}, \mathsf{wd})$ passed to $\mathcal{F}_{\mathsf{State}}$. Since $\mathcal{F}_{\mathsf{State}}$ accepts inputs one round by round, but $\mathcal{F}_{\mathsf{Pay}}$ invocations may arrive at any time, the local protocol accumulates pay and wd inputs until the next round begins.

4.4 Simulation-based Security Proof

We now provide a proof sketch that our protocol securely realizes the $\mathcal{F}_{\mathsf{Pay}}$ model. As usual, the proof consists of a simulator $\mathcal{S}$ for the dummy adversary (i.e., the real world adversary that simply follows instructions from the environment). Since the model does not provide any secrecy, and since the $\mathcal{F}_{\mathsf{State}}$-hybrid world hides any cryptography, the simulation is straightforward and deterministic. The simulator runs a local sandboxed execution of Π_{Pay}, which it keeps in perfect correspondence with the state of $\mathcal{F}_{\mathsf{Pay}}$. When the environment asks $\mathcal{A}$ to command corrupted parties to interact with and observe $\mathcal{F}_{\mathsf{State}}$, the simulator $\mathcal{S}$ routes these requests to its sandboxed Π_{Pay}. We can show that the sandboxed execution of Π_{Pay} maintained by $\mathcal{S}$ is identical to Π_{Pay} in the real world.

Theorem 1. *The Π_{Pay} protocol in the $\mathcal{F}_{\mathsf{State}}$-hybrid world realizes the $\mathcal{F}_{\mathsf{Pay}}$ ideal functionality.*

Proof. (Sketch) The ideal world simulator $\mathcal{S}$ for the dummy real world adversary runs a sandboxed execution of Π_{Pay} through which it relays instructions as described below.

Inputs from honest parties. When the simulator $\mathcal{S}$ receives a message of the form $(\mathtt{pay}, P_i, \$X)$ from $\mathcal{F}_{\mathsf{Pay}}$, it provides an input $\mathtt{pay}(\$X)$ to P_i in the sandboxed execution of Π_{Pay}; when $\mathcal{S}$ receives $(\mathtt{withdraw}, P_i, \$X)$, it inputs $\mathtt{withdraw}(\$X)$.

Contract inputs. If a $\mathtt{deposit}(\$X)$ input is delivered to $\mathcal{F}_{\mathsf{Pay}}$, send $\mathtt{deposit}(\$X)$ to the sandboxed $\mathsf{Contract}_{\mathsf{Pay}}$. Note that this creates an $O(\Delta)$-delayed task in $\mathcal{F}_{\mathsf{State}}$ as well as one in $\mathcal{F}_{\mathsf{Pay}}$.

Message delivery in $\mathcal{F}_{\mathsf{State}}$. When the environment asks to execute a delayed task in $\mathcal{F}_{\mathsf{State}}$ (i.e., to advance aux_{in} or to apply a state update), $\mathcal{S}$ routes this request to Π_{Pay}. If the sandboxed $\mathcal{F}_{\mathsf{State}}$ provides output state to a corrupted party, pass state to the environment.

Outputs to honest parties. If honest party P_i in the sandboxed $\mathcal{F}_{\mathsf{State}}$ provides output $(\mathsf{receive}, \$X)$, then deliver the delayed task in $\mathcal{F}_{\mathsf{Pay}}$ that sends the same output to P_i in the ideal world.

$\square$

Update function U_{Pay}

```
$U_{\mathsf{Pay}}(\mathsf{state}, (\mathsf{in}_{\mathtt{L}}, \mathsf{in}_{\mathtt{R}}), \mathsf{aux}_{in})$ :
  if $\mathsf{state} = \bot$, set $\mathsf{state} := (0,0)$
  parse state as $(\mathsf{cred}_{\mathtt{L}}, \mathsf{cred}_{\mathtt{R}})$
  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 $\mathsf{input}_i := (0,0)$
    parse each $\mathsf{input}_i$ as $(\mathsf{pay}_i, \mathsf{wd}_i)$
    $\mathsf{pay}_i := \min(\mathsf{pay}_i, \mathsf{deposits}_i + \mathsf{cred}_i)$
    $\mathsf{wd}_i := \min(\mathsf{wd}_i, \mathsf{deposits}_i + \mathsf{cred}_i - \mathsf{pay}_i)$
  $\mathsf{cred}_{\mathtt{L}} \mathrel{+}= \mathsf{pay}_{\mathtt{R}} - \mathsf{pay}_{\mathtt{L}} - \mathsf{wd}_{\mathtt{L}}$
  $\mathsf{cred}_{\mathtt{R}} \mathrel{+}= \mathsf{pay}_{\mathtt{L}} - \mathsf{pay}_{\mathtt{R}} - \mathsf{wd}_{\mathtt{R}}$
  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 $\mathsf{aux}_{out} := \bot$
  $\mathsf{state} := (\mathsf{cred}_{\mathtt{L}}, \mathsf{cred}_{\mathtt{R}})$
  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** $\mathtt{deposit}(\mathbf{coins}(\$X))$ from $P_i$ :
  $\mathsf{deposits}_i \mathrel{+}= \$X$
  $\mathsf{aux}_{in}.\mathsf{send}(\mathsf{deposits}_{\mathtt{L}}, \mathsf{deposits}_{\mathtt{R}})$
on **contract input** $\mathtt{output}(\mathsf{aux}_{out})$:
  parse $\mathsf{aux}_{out}$ as $(\mathsf{wd}_{\mathtt{L}}, \mathsf{wd}_{\mathtt{R}})$
  for $i \in \{\mathtt{L},\mathtt{R}\}$ send $\mathbf{coins}(\mathsf{wd}_i)$ to $P_i$
```

Local protocol Π_{Pay} for party P_i

```
initialize $\mathsf{pay} := 0$, $\mathsf{wd} := 0$, $\mathsf{cred} := 0$
provide $(0,0)$ as input to $\mathcal{F}_{\mathsf{State}}$
on **receiving state** $(\mathsf{cred}_{\mathtt{L}}, \mathsf{cred}_{\mathtt{R}})$ from $\mathcal{F}_{\mathsf{State}}$,
  if $\mathsf{cred}_i \geq \mathsf{cred}$ then output $(\mathtt{receive}, \mathsf{cred}_i - \mathsf{cred})$
  set $\mathsf{cred} := \mathsf{cred}_i - \mathsf{pay} - \mathsf{wd}$
  provide $(\mathsf{pay}, \mathsf{wd})$ as input to $\mathcal{F}_{\mathsf{State}}$
on **input** $\mathtt{pay}(\$X)$ from the environment,
  if $\$X \leq \mathsf{Contract}_{\mathsf{Pay}}.\mathsf{deposits}_i - \mathsf{pay} - \mathsf{wd}$, $\mathsf{pay} \mathrel{+}= \$X$
on **input** $\mathtt{withdraw}(\$X)$ from the environment,
  if $\$X \leq \mathsf{Contract}_{\mathsf{Pay}}.\mathsf{deposits}_i - \mathsf{pay} - \mathsf{wd}$, $\mathsf{wd} \mathrel{+}= \$X$
```

Figure 6: Implementation of $\mathcal{F}_{\mathsf{Pay}}$ in the $\mathcal{F}_{\mathsf{State}}$-hybrid world (illustrated in Fig. 3(b)).

4.5 Instantiating State Channels

We focus on explaining the behavior of the smart contract, $\mathsf{Contract}_{\mathsf{State}}$, defined in Figure 8 and in the Appendix A a detailed description of the local behavior for each party is provided.

At a very high level, the off-chain updates to the state are synchronized via signatures received from every party. Reaching agreement about which inputs to

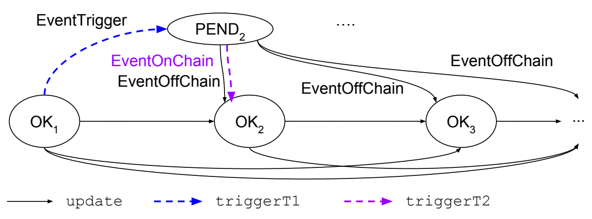


Figure 7: State transitions in $\mathsf{Contract}_{\mathsf{State}}$. The `triggerT1` method transitions from $\mathtt{OK}_r$ to $\mathtt{PENDING}_{r+1}$. From any `OK` or `PENDING` state, an `update` message with signatures from all parties can transfer to `OK` with an equal or later round. The `triggerT2` method applies advances the state by applying inputs provided on-chain.

process next is facilitated by having 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 [5]; 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.

If another party triggers a request for inputs, via the `input` method of the contract, then there are two cases to handle. First, the request for inputs may pertain to an "old" round that has already been surpassed. In this case, honest parties respond with an `update` invocation that effectively cancels the trigger. Otherwise, if the request is for the current round, then the party provides its `input` directly to the contract.

An edge case occurs when a request for on-chain settling occurs, but a `COMMIT` message is received later, thus it may be uncertain whether can be broadcast before the on-chain process is handled. From the time a first $\mathtt{triggerT1}(r,T)$ event is received, a timer begins to determine whether the state proceeds according to off-chain or on-chain inputs by deadline $T+\Delta$. Thus if a party receives a `COMMIT` message with enough time to submit this before the deadline, then they proceed in the "Off-Chain" state. However, if a party does not receive a `COMMIT` message within this time, then they wait to see whether.

Events. Contract events describe transactions that appear on the blockchain. A party "receives" the event once it is buried. This is inspired in party by Ethereum's support for logging events. However, there is a semantic difference namely that events are delivered in Ethereum immediately after a single confirmation, whereas in our usage, an event is only delivered after buried sufficiently, such that we are guaranteed it cannot be reversed.

The most confusing circumstance is if some malicious party P_j provides input v_j to the leader, withholds his signature until the last minute, and invokes the contract with an equivocating input $\mathtt{input}(v'_j)$. In this circumstance, we need to ensure that honest parties proceed with a consistent view. We accomplish this by designing the smart contract to hold the state determined by the on-chain inputs via the `input` invocation in a *pending* state.

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

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

```
Initialize $\mathsf{bestRound} := -1$
Initialize $\mathsf{state} := \emptyset$
Initialize $\mathsf{flag} := \mathtt{OK}$
Initialize $\mathsf{deadline} := \perp$
Initialize $\mathsf{applied} := \emptyset$
on receiving $\mathtt{update}(r, \mathsf{state}', \mathsf{out}, \{\sigma_{r,j}\})$:
  discard if $r \le \mathsf{bestRound}$ and $r \notin \mathsf{applied}$
  verify all signatures on the message $(r \| \mathsf{state}' \| \mathsf{out})$
  if $r = \mathsf{bestRound} + r$ then
    if $\mathsf{flag} == \mathtt{PENDING}$ then
      $\mathsf{flag} := \mathtt{OK}$
      emit $\mathtt{EventOffchain}(\mathsf{bestRound}+1)$
    $\mathsf{bestRound} := r$
    $\mathsf{state} := \mathsf{state}'$
  invoke $C.\mathtt{aux\_output}(\mathsf{out})$
  $\mathsf{applied} := \mathsf{applied} \cup \{r\}$
on receiving $\mathtt{triggerT1}(r)$ at time $T$:
  discard if $r \ne \mathsf{bestRound}+1$
  discard if $\mathsf{flag} \ne \mathtt{OK}$
  set $\mathsf{flag} := \mathtt{PENDING}$
  set $\mathsf{deadline} := T + \Delta$
  emit $\mathtt{EventPending}(r, \mathsf{deadline})$
on receiving $\mathtt{triggerT2}(r)$ at time $T$:
  discard if $r \ne \mathsf{bestRound}+1$
  discard if $\mathsf{flag} \ne \mathtt{PENDING}$
  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 player $P_j$ has not provided input
  set $\mathsf{flag} := \mathtt{OK}$
  emit $\mathtt{EventOnchain}(r, \mathsf{state})$
on receiving $\mathtt{input}(r, v_{r,j})$ from player $P_j$:
  if this is the first such activation, store $v_{r,j}$
```

Figure 8: Contract portion of the protocol Π_{State} for implementing a general purpose state channel, $\mathcal{F}_{\mathsf{State}}$.

Theorem 2. *The* Π_{State} *protocol realizes the* $\mathcal{F}_{\mathsf{State}}$ *functionality.*

The proof can be found in the appendix.

5 Linked Payment Chains

```
**Functionality** $\mathcal{F}_{\mathsf{Linked}}(\$X, T, P_1, ..., P_\ell)$
initially, for each $i \in 1...(\ell-1)$, set $\mathsf{flag}_i := \mathtt{init} \in \{\mathtt{init}, \mathtt{inflight}, \mathtt{complete}, \mathtt{cancel}\}$

**on receiving** $(\mathtt{open}, i)$ from $\mathcal{A}$, if $\mathsf{flag}_i = \mathtt{init}$, then
  if $\mathcal{F}^i_{\mathsf{Pay}}.\mathsf{bal}_{\mathtt{L}} \geq \$X$ then:
      $\mathcal{F}^i_{\mathsf{Pay}}.\mathsf{bal}_{\mathtt{L}} \mathrel{-}= \$X$
      set $\mathsf{flag}_i := \mathtt{inflight}$
  otherwise, set $\mathsf{flag}_i := \mathtt{cancel}$

**on receiving** $(\mathtt{cancel}, i)$ from $\mathcal{A}$, if at least one party is corrupt and $\mathsf{flag}_i \in \{\mathtt{init}, \mathtt{inflight}\}$,
  if $\mathsf{flag}_i = \mathtt{inflight}$ then set $\mathcal{F}^i_{\mathsf{Pay}}.\mathsf{bal}_{\mathtt{L}} \mathrel{+}= \$X$
  set $\mathsf{flag}_i := \mathtt{cancel}$

**on receiving** $(\mathtt{complete}, i)$ from $\mathcal{A}$, if $\mathsf{flag}_i = \mathtt{inflight}$,
  $\mathcal{F}^{i-1}_{\mathsf{Pay}}.\mathsf{bal}_{\mathtt{R}} \mathrel{+}= \$X$
  set $\mathsf{flag}_i := \mathtt{complete}$

**after time** $T + O(\ell + \Delta)$, or after $T + O(\ell)$ if all parties honest, raise an Exception if any of the following assertions fail:
  1. for each $i \in 1...(\ell-1)$, $\mathsf{flag}_i$ must be in a terminal state, i.e., $\mathsf{flag}_i \in \{\mathtt{cancel}, \mathtt{complete}\}$
  2. for each $i \in 1...(\ell-2)$, if $P_i$ is honest, it must not be the case that $(\mathsf{flag}_i, \mathsf{flag}_{i+1}) = (\mathtt{cancel}, \mathtt{complete})$.
  3. if $P_1$ and $P_\ell$ are honest, then $(\mathsf{flag}_1, \mathsf{flag}_{\ell-1}) \in \{(\mathtt{complete}, \mathtt{complete}), (\mathtt{cancel}, \mathtt{cancel})\}$
```

Figure 9: Definition of the chained-payment functionality

The simple payment channels from the previous section are useful when two parties expect to make frequent payments to each other. To support payments in a broader network, however, we wish to support "linked payments" across a path, such that two parties can pay each other indirectly as long as they can find a path of intermediaries with payment channels already established. The challenge is to ensure that the payment occurs atomically, and that the collateral provided by the intermediaries should be returned to them within a bounded time.

In more detail, consider consider a scenario where parties P_1 through P_ℓ have established $\ell-1$ payment channels, such that $\mathcal{F}^i_{\mathsf{Pay}}$ denotes the payment channel established between P_i and P_{i+1}. We desire the following properties:

Optimistic Correctness. 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 of channel $\mathcal{F}^i_{\mathsf{Pay}}$, the outgoing balance $\mathcal{F}^i_{\mathsf{Pay}}.\mathsf{bal}_{\mathtt{R}}$ is increased by \$$x$ and each incoming balance $\mathcal{F}^i_{\mathsf{Pay}}.\mathsf{bal}_{\mathtt{L}}$ is decreased by \$$x$.

Intermediaries do not lose money. Even if some parties are corrupt, then the honest parties on the path, i.e. P_2 through $P_{\ell-1}$ should not lose any money. More specifically, for each party P_i, after a maximum of $O(\ell+\Delta)$ rounds, either the incoming balance ($\mathcal{F}^{i-1}_{\mathsf{Pay}}.\mathsf{bal}_{\mathtt{R}}$) is incremented by \$$X$, or else the outgoing balance ($\mathcal{F}^i_{\mathsf{Pay}}.\mathsf{bal}_{\mathtt{L}}$) is returned to its initial state.

Atomicity. If the sender and receiver, P_1 and P_ℓ, are both honest then the payment either completes or cancels, atomically for both parties.[2] More precisely, after $O(\ell+\Delta)$ rounds, either the payment completes (the outgoing balance of P_1 is decremented by \$$X$ and the incoming balance of P_ℓ is incremented by \$$X$), or else the payment fails, and both parties balances remain unchanged.

5.1 Modeling linked payment chains.

The ideal world, which serves as our formal security definition (illustrated in Figure 9) consists of multiple instances of the duplex channels $\mathcal{F}_{\mathsf{Pay}}$, as well one instance of the payment chain functionality $\mathcal{F}_{\mathsf{Linked}}$. This model is one shot: [3] it describes only a single chained payment, among a fixed path of parties.

The $\mathcal{F}_{\mathsf{Linked}}$ functionality interacts with the individual $\mathcal{F}_{\mathsf{Pay}}$ functionalities directly, in a "white box" way, i.e. by directly manipulating the $\mathsf{bal}_{\{\mathtt{L},\mathtt{R}\}}$ fields. In order to enforce the desired atomicity properties, the functionality also keeps track of a status flag associated with each channel. When the payment is initialized, it reserves a portion of $P_{\mathtt{L}}$'s balance in each $\mathcal{F}_{\mathsf{Pay}}$ instance for a conditional payment, transitioning to the `inflight` status (if a channel on the path has insufficient balance, then then the payment is canceled). From the `inflight` status, the conditional payment must conclude (within bounded time) in one of two ways, either `cancel` in which case the balance is refunded to $P_{\mathtt{L}}$, or `complete` in which case

[2]Note that no guarantees are provided to the sender and receiver if either misbehaves. The payment is voluntary, so the sender could simply choose not to make the payment in the first place. In future work we would like to provide a mechanism for the sender to receive a receipt iff the payment completes.

[3]To generalize the functionality, we would use a generic composition operator to construct an ideal world where many instance of $\mathcal{F}_{\mathsf{Linked}}$ can exist simultaneously, and be brought into existence by parties. To satisfy the composition theorem, we would need to ensure that multiple payment chains are prevented from "interfering" with each other, for example by replaying one message in another. We elide over these issues.

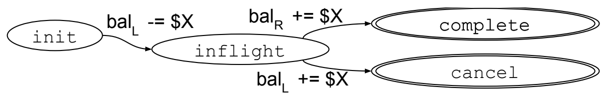


Figure 10: State transitions and side effects in $\mathcal{F}_{\mathsf{Linked}}$ (for each channel $\mathcal{F}^i_{\mathsf{Pay}}$. Within $O(\ell+\Delta)$ rounds, a terminal state is reached.

it is paid to $P_{\mathtt{R}}$. These transitions are summarized in Figure 10.

It is easy to check that the desired properties described earlier are exhibited by the functionality definition of $\mathcal{F}_{\mathsf{Linked}}$. First, notice that the transition $\mathsf{init} \to \mathtt{cancel}$ can only occur if some parties are corrupted, or if the channel balance is insufficient. Furthermore, notice that assertion 2 ensures that honest parties do not lose money. Finally, notice that the individual $\mathcal{F}_{\mathsf{Pay}}$ payment channels continue operating "as normal" even while the chained payment is in progress, i.e. parties can also send (unconditional) payments to each other in the meantime, as well as deposit and withdraw on-chain funds, up to the available amount.

5.2 Implementing Chained Payments

As with our construction for $\mathcal{F}_{\mathsf{Pay}}$, our construction consists of an update function that specializes $\mathcal{F}_{\mathsf{State}}$, as well as auxiliary contracts and local behavior for each party. We focus our discussion on the update function and auxiliary contracts as shown in Figure 11 (see also Figure 13).

The update function $U_{\mathsf{Linked},\$X}$ is an outer layer around the U_{Pay} function (Figure 6), but extends state to include support for a conditional payment, mirroring the status flag in the $\mathcal{F}_{\mathsf{Linked}}$ functionality. The left-hand party for each channel $P_{\mathtt{L}}$, creates a conditional payment by sending an $\mathtt{open}(h)$ instruction to $\mathcal{F}_{\mathsf{State}}$, where h is the hash of a (possibly unknown) secret. Each conditional payment can be concluded in one of three ways: by a `complete` instruction from $P_{\mathtt{L}}$, a `cancel` message from $P_{\mathtt{R}}$, or through a `dispute` case, which can occur only if one of the two parties is corrupt, as we describe shortly.

To establish a chain of linked payments, the initial sender P_1 first creates a secret x, shares 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.

The key challenge is to ensure that if an honest party's outgoing conditional payment completes, then its incoming conditional payment must also complete. In the `dispute` case, whether the conditional payment is canceled or refunded depends on the state of the global preimage manager, $\mathsf{Contract}_{\mathsf{PM}}$, which acts like a global condition: if the preimage manager contract receives x before time T_{Expiry}, then *every* conditional payment that is disputed will complete; otherwise, *every* disputed conditional payment will cancel. Therefore, if an honest party receives x before time $T_{\mathsf{Expiry}} - \Delta$, it is safe to `complete` their outgoing conditional payment, since in the worst case they will be able submit x to $\mathsf{Contract}_{\mathsf{PM}}$ and claim their incoming payment via `dispute`.

In the Appendix we give a security proof that Π_{Linked} realizes the ideal world $(\mathcal{F}_{\mathsf{Linked}}, \mathcal{F}_{\mathsf{Pay}})$. Here we just describe the unintuitive edge cases that the protocol is designed to handle.

First, it is possible that parties receive inconsistent values of h. Since each party creates an outgoing conditional payment with h only after receiving an incoming conditional payment with the same hash h, their balance is guaranteed to be preserved regardless. Second, note that the ideal functionality permits for some conditional payments to `complete` while others `cancel`; however this can only cause a corrupted party to lose money. Finally, we note that in the optimistic case, when all parties are honest and thus all payment conditional payments `complete`, the $\mathsf{Contract}_{\mathsf{PM}}$ is never invoked at all.

6 Disussion and Conclusion

Cryptocurrencies face several ongoing challenges in an uncertain future: they must scale up to accommodate increasing user demand, and they must compete with centralized alternatives. In this work we have provided the first formalized construction of off-chain payment networks, one of the most widely anticipated approaches to scalable payments. Our construction embodies a novel insight that directly improves the worst-case collateral costs compared to Lightning [26], the current state-of-the-art design. Since our improvement is by a factor of $O(\ell)$, where ℓ is the length of a payment path, we especially improve the feasibility of decentralized (rather than star-shaped) payment network structures.

However, our construction relies on a global contract mechanism, which, while easily expressed in Ethereum, cannot (we conjecture) be emulated in Bitcoin without some modification to Bitcoin's scripting system. The reason is not that Ethereum is Turing complete; rather, as we discuss in Section 1.1, the difference appears to involve Bitcoin's "UTXO"-based transaction structure. This structure prevents one transaction from affecting the state of another, unlike Ethereum contracts which can refer to each other in a global namespace (see also [2, Section 2] and [20, Section 4]).

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

```
Let $T_{\mathsf{Expiry}} := T + 6\ell + \Delta$.
Let $T_{\mathsf{Crit}} := T_{\mathsf{Expiry}} - \Delta$ // Last chance to submit $x$
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 state $= \bot$, set 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 $\mathsf{cmd}_{\mathtt{L}} = \mathsf{open}(h')$ and flag $= \mathtt{init}$, then
  set $\mathsf{cred}_{\mathtt{L}}$ $-=$ $\$X$, flag $:= \mathtt{inflight}$, and $h := h'$
otherwise if $\mathsf{cmd}_{\mathtt{L}} = \mathsf{complete}$ and flag $= \mathtt{inflight}$,
  set $\mathsf{cred}_{\mathtt{R}}$ $+=$ $\$X$, and flag $:= \mathtt{complete}$
otherwise if $\mathsf{cmd}_{\mathtt{R}} = \mathsf{cancel}$ and flag $= \mathtt{inflight}$,
  set $\mathsf{cred}_{\mathtt{L}}$ $+=$ $\$X$ and flag $:= \mathtt{cancel}$
otherwise if $\mathsf{cmd}_{\mathtt{R}} = \mathtt{dispute}$ or $\mathsf{cmd}_{\mathtt{L}} = \mathtt{dispute}$, and flag $= \mathtt{inflight}$, and current time $> T_{\mathsf{Expiry}}$, then
  $\mathsf{aux}_{\mathsf{out}} := (\mathtt{dispute}, h, \$X)$ and 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 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 11, 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 timestamp[] is an empty mapping
on contract input $\mathtt{publish}(x)$ at time $T$:
  if $\mathcal{H}(x) \notin$ timestamp: then set timestamp$[\mathcal{H}(x)] := T$
constant function $\mathtt{published}(h, T')$:
  return $\mathtt{True}$ if $h \in$ timestamp and timestamp$[h] \leq T'$
  return $\mathtt{False}$ otherwise
```

Figure 11: Contract portion of the protocol Π_{Linked} for implementing linked payments $(\mathcal{F}_{\mathsf{Linked}}, \mathcal{F}_{\mathsf{Pay}})$ in the $\mathcal{F}_{\mathsf{State}}$-hybrid world. Local behavior for each party is deferred to the appendix.

As we have focused on collateral costs as our key performance objective, our constructions and security definitions do not aim to ensure transaction privacy. Our work is therefore complementary to efforts that focus primarily on privacy. We suspect that our insights can be combined with these; furthermore, we believe our state channel abstraction can serve as a convenient building block for this important future work.

References

[1] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains, 2014.

[2] Iddo Bentov, Ranjit Kumaresan, and Andrew Miller. Instantaneous decentralized poker. `https://arxiv.org/abs/1701.06726`, 2017.

[3] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In S&P, 2015.

[4] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[5] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. *OSDI*, 1999.

[6] Alessandro Chiesa, Matthew Green, Jingcheng Liu, Peihan Miao, Ian Miers, and Pratyush Mishra. Decentralized anonymous micropayments. In *Eurocrypt*, 2017.

[7] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gun Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains. In *Financial Cryptography 3rd Bitcoin Workshop*, 2016.

[8] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *17th SSS*, 2015.

[9] Johnny Dilley, Andrew Poelstra, Jonathan Wilkins, Marta Piekarska, Ben Gorlick, and Mark Friedenbach. Strong federations: An interoperable blockchain solution to centralized third party risks. *CoRR*, abs/1612.05491, 2016.

[10] Matthew D. Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. *IACR Cryptology ePrint Archive*, 2016:701, 2016.

[11] Emin Gun Sirer Ittay Eyal, Adem Efe Gencer and Robbert van Renesse. Bitcoin-NG: A scalable blockchain protocol.

[12] 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.

[13] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy (S&P)*, 2016.

[14] R. Kumaresan, T. Moran, and I. Bentov. How to use bitcoin to play decentralized poker. In *CCS*, 2015.

[15] Ranjit Kumaresan and Iddo Bentov. How to use bitcoin to incentivize correct computations. In *CCS*, 2014.

[16] 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.

[17] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. Silentwhispers: Enforcing security and privacy in decentralized credit networks. *IACR Cryptology ePrint Archive*, 2016:1054, 2016.

[18] Patrick McCorry, Malte Möser, Siamak F Shahandasti, and Feng Hao. Towards bitcoin payment networks. In *Australasian Conference on Information Security and Privacy*, pages 57–76. Springer, 2016.

[19] Andrew Miller. Sprites source code. `https://github.com/amiller/sprites`, 2017.

[20] Andrew Miller and Iddo Bentov. Zero-collateral lotteries in bitcoin and ethereum. IEEE S&B workshop, `https://arxiv.org/abs/1612.05390`, 2017.

[21] Raiden Network. `http://raiden.network/`, 2015.

[22] Tier Nolan. Alt chains and atomic transfers. `bitcointalk.org`, May 2013.

[23] Rafael Pass and Abhi Shelat. Micropayments for decentralized currencies. In *22nd CCS*, 2015.

[24] 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`.

[25] Dennis Peterson. Sparky: A lightning network in two pages of solidity. `http://www.blunderingcode.com/a-lightning-network-in-two-pages-of-solidity`.

[26] J. Poon and T. Dryja. The bitcoin lightning network: Scalable off-chain instant payments. 2016. `https://lightning.network/lightning-network-paper.pdf`.

[27] Jeremy Spilman. Anti dos for tx replacement. `https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2013-April/002433.html`, 2013.

[28] Jehan Tremback and Zack Hess. Universal payment channels. `http://altheamesh.com/documents/universal-payment-channels.pdf`, November 2015.

A Details of State Channel Construction

In the body of the paper (Figure 8) we presented the smart contract portion of the state channel protocol. In Figure 12 we define the local behavior of the parties.

Assertions and sanity checks. Note that that when `EventPending`(r) is received, we can assume that $r \leq \mathsf{bestRound} + 1$ in the view of any honest party. This is because the `EventPending` event can only be triggered for round $\mathsf{bestRound} + 1$ of the contract, and bestRound is only set in the contract after receiving inputs from every honest party.

If a party P_i receives `EventOffchain`(r) while in the $\mathsf{flag} = \texttt{PENDING}$ condition, it can be safely assumed that a `BATCH` message has already been received and state has already been updated to reflect the new state. This is because `EventOffChain` can only be triggered after receiving an update containing signatures from all parties, including P_i.

Lemma 3. *If any party receives* $\mathsf{EventPending}(r, T)$*, then every party will receive from the contract either (1)* $\mathsf{EventOffchain}(r)$ *within time* $T + \Delta$*, or (2)* $\mathsf{EventOnchain}(r)$ *within time* $T + 2\Delta$.

Proof sketch. Since some party received EventPending from the contract, this means that the contract received triggerT1 from one of the parties. Observe that the variable deadline is set to $T + \Delta$. Now, if r corresponds to

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

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

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

on receiving $\mathtt{update}(r, \mathsf{state}', \mathsf{out}, \{\sigma_{r,j}\})$:
  discard if $r \leq \mathsf{bestRound}$ and $r \notin \mathsf{applied}$
  verify all signatures on the message $(r \| \mathsf{state}' \| \mathsf{out})$
  if $r = \mathsf{bestRound} + r$ then
    if $\mathsf{flag} == \mathtt{PENDING}$ then
      $\mathsf{flag} := \mathtt{OK}$
      emit $\mathtt{EventOffchain}(\mathsf{bestRound} + 1)$
    $\mathsf{bestRound} := r$
    $\mathsf{state} := \mathsf{state}'$
  invoke $C.\mathtt{aux\_output}(\mathsf{out})$
  $\mathsf{applied} := \mathsf{applied} \cup \{r\}$

on receiving $\mathtt{triggerT1}(r)$ at time $T$:
  discard if $r \neq \mathsf{bestRound} + 1$
  discard if $\mathsf{flag} \neq \mathtt{OK}$
  set $\mathsf{flag} := \mathtt{PENDING}$
  set $\mathsf{deadline} := T + \Delta$
  emit $\mathtt{EventPending}(r, \mathsf{deadline})$

on receiving $\mathtt{triggerT2}(r)$ at time $T$:
  discard if $r \neq \mathsf{bestRound} + 1$
  discard if $\mathsf{flag} \neq \mathtt{PENDING}$
  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 player $P_j$ has not provided input
  set $\mathsf{flag} := \mathtt{OK}$
  emit $\mathtt{EventOnchain}(r, \mathsf{state})$

on receiving $\mathtt{input}(r, v_{r,j})$ from player $P_j$:
  if this is the first such activation, store $v_{r,j}$
```

Local protocol for the leader, P_1

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

Local protocol for each party P_i (including the leader)

```
$\mathsf{flag} := \mathtt{OK} \in \{\mathtt{OK}, \mathtt{PENDING}\}$
Initialize $\mathsf{lastRound} := -1$
Initialize $\mathsf{lastCommit} := \perp$

Fast Path (while $\mathsf{flag} == \mathtt{OK}$)

Proceed in sequential rounds $r$, beginning with $r := 0$
Wait to receive input $v_{r,i}$ from the environment. Send $\mathtt{INPUT}(v_{r,i})$ to the leader.
Wait to receive a batch of proposed inputs, $\mathtt{BATCH}(r, \mathsf{in}'_r, \{v'_{r,j}\}_j)$ from the leader. Discard this proposal if $P_i$'s own input is omitted, i.e., $v'_{r,i} \neq v_{r,i}$. Discard this proposal if $\mathsf{in}'_r$ is not a *recent* value of $\mathsf{aux}_{in}$ in the contract.
Set $(\mathsf{state}, \mathsf{out}_r) := U(\mathsf{state}, \{v_{r,j}\}_j, \mathsf{in}'_r)$
Send $(\mathtt{SIGN}, \sigma_{r,i})$ to $P_1$, where $\sigma_{r,i} := \mathsf{sign}_i(r \| \mathsf{out}_r \| \mathsf{state})$
Wait to receive $\mathtt{COMMIT}(r, \{\sigma_{r,j}\}_j)$ from the leader. Discard unless $\mathsf{verify}_j(\sigma_{r,j} \| \mathsf{out}_r \| \mathsf{state})$ holds for each $j$. Then:
  $\mathsf{lastCommit} := (\mathsf{state}, \mathsf{out}_r, \{\sigma_{r,j}\}_j)$; $\mathsf{lastRound} := r$
  If $\mathsf{out}_r \neq \perp$, send $\mathtt{update}(r, \mathsf{lastCommit})$ to the contract.
If $\mathtt{COMMIT}$ was not received within one time-step, then:
  if $\mathsf{lastCommit} \neq \perp$, send $\mathtt{update}(r-1, \mathsf{lastCommit})$ and $\mathtt{triggerT1}(r)$ (one after another) to $C$

Handling on-chain events

On receiving $\mathtt{EventPending}(\mathtt{r}, \_)$, if $r \leq \mathsf{lastRound}$, then send $\mathtt{update}(\mathsf{lastRound}, \mathsf{lastCommit})$ to the contract. Otherwise if $r = \mathsf{lastRound} + 1$, then:
  Set $\mathsf{flag} := \mathtt{PENDING}$, and interrupt any "waiting" tasks in the fast path above. Inputs are buffered until returning to the fast path.
  Send $\mathtt{input}(r, v_{r,i})$ to the contract.
  Wait to receive $\mathtt{EventOffchain}(r)$ or $\mathtt{EventOnchain}(r)$ from the contract. In either case:
    $\mathsf{state} := \mathsf{state}'$
    $\mathsf{flag} := \mathtt{OK}$
    Enter the fast path with $r := r + 1$
```

Figure 12: Construction of a general purpose state channel, $\mathcal{F}_{\mathsf{State}}$, parameterized by transition function U. For readability, the left column is duplicated from Fig. 8.

an earlier round (i.e., not the current incomplete round), then an honest party upon receiving EventPending would send an `update` command to the contract with the state (along with signatures) corresponding to the most recent completed round. In this case, it follows that the contract would emit EventOffChain before time $T+\Delta$. On the other hand, suppose r corresponds to the current round. In this case, we have that honest parties would have $r = \mathsf{lastRound}+1$, and they would send their current round input $(r, v_{r,i})$ to the contract (which would be accumulated by C) as their response to EventPending. Then, at time $T+\Delta$, honest parties would send triggerT2 right after $T+\Delta$ which ensures that they get a response from the on-chain contract before time $T+2\Delta$. This concludes the proof sketch.

Lemma 4. *If any honest party receives* $\mathsf{EventPending}(r,T)$ *from the contract, but does not receive* $\mathtt{COMMIT}(r, \_)$ *from the leader before time T, then no party will receive* $\mathtt{COMMIT}(r+1, \_)$ *until receiving* $\mathsf{EventOnchain}(r)$ *or* $\mathsf{EventOffchain}(r)$ *from the contract.*

Proof sketch. Note that for the leader P_1 to generate a valid `COMMIT` message for round $r+1$, it needs signatures from all parties. Suppose some honest party received $\mathsf{EventPending}(r,T)$, then it must hold that $r = \mathsf{bestRound}+1$. This in particular means that the contract received signatures on the state corresponding to the $(r-1)$-th round from all parties. Since P_1 would not be able to forge honest parties' signatures, it follows that the honest parties must have synchronized their state until the $(r-1)$-th round (either on-chain or off-chain). Now suppose, it holds that some honest party P_j did not receive a message `COMMIT` message for round r, then there are two cases to handle. First, if r is not the current round, then in this case honest parties would directly send an `update` command with the most recent completed round along with the latest synchronized state to the contract. This has the effect of making the contract emit EventOffchain for round r. Next if r is indeed the current round, then it follows that honest party would immediately send to the contract (1) an `update` message with the state corresponding to the $(r-1)$-th round, and immediately afterwards (2) a `triggerT1` message for round r. Taken together, these messages have the effect of ensuring that the contract emits EventPending for round r. Following this and since $r = \mathsf{lastRound}+1$, an honest party P_i would respond by sending their inputs $v_{r,i}$ to the contract (irrespective of whether honest P_i received a message of the form $\mathtt{COMMIT}(r, \_)$). Then, the rest of the state is synchronized on-chain (i.e., parties submit their inputs directly to the contract). At time $T+\Delta$, honest parties would send triggerT2 to the contract which would result in the contract emitting EventOnchain for round r. This concludes the proof sketch.

Lemma 5. *If any honest party receives* $\mathsf{EventPending}(r,T)$ *from the contract, and receives* $\mathtt{COMMIT}(r)$ *from the leader before time T, then every party will receive* $\mathsf{EventOffchain}(r)$ *by time* $T+\Delta_{\mathsf{Send}}+\Delta_{\mathsf{Receive}}$.

Proof sketch. By Lemma 3, it follows that we only need to show that $\mathsf{EventOnchain}(r)$ will not be emitted whenever $\mathtt{COMMIT}(r)$ is received before time T. Given that some honest party P_j received $\mathtt{COMMIT}(r)$ from the leader, then this means that all parties must have synchronized their state until round $r-1$. In addition, since P_j also received $\mathsf{EventPending}(r,T)$ from the contract, this implies that some party issued `triggerT1` to the contract. Observe that $r = \mathsf{lastRoundbestRound}+1$ also holds. Now each honest party P_i would respond to the EventPending message with an `update` command that would result in the on-chain contract emitting $\mathsf{EventOffchain}(r)$ within time $T+\Delta$. This concludes the proof.

Theorem 6. *The* Π_{State} *protocol realizes the* $\mathcal{F}_{\mathsf{State}}$ *functionality.*

Proof. (Sketch) The ideal world simulator $\mathcal{S}$ for the dummy real world adversary runs a sandboxed execution of Π_{State} through which it relays instructions as described below.

When the simulator $\mathcal{S}$ receives a message of the form $(i, v_{r,i})$ from $\mathcal{F}_{\mathsf{State}}$ (i.e., leaked honest inputs), it stores this message to use later in the simulation for the current round.

Following this, $\mathcal{S}$ simulates the local protocol for each honest player to the $\mathcal{A}$ using leaked inputs. If the leader is honest, then $\mathcal{S}$ acting as the leader, uses the leaked inputs (from $\mathcal{F}_{\mathsf{State}}$) as inputs received from honest parties. $\mathcal{S}$ waits to receive round inputs from $\mathcal{A}$ for the corrupt parties. If it receives inputs from all corrupt parties, then acting as the leader, $\mathcal{S}$ accumulates these values, and multicasts it to all corrupt parties. Following this, $\mathcal{S}$ waits to receive signatures on the batched inputs. $\mathcal{S}$ generates signatures on behalf of the honest parties (i.e., it uses a simulated public/signing keys for the honest parties). Once all signatures are received, then these signatures (i.e., corresponding to both honest and corrupt parties) are multicasted to the adversary. $\mathcal{S}$ then increments the round number and continues with the simulation. On the other hand, if the leader is corrupt, then $\mathcal{S}$ sends simulated honest messages (according to Π_{State} to $\mathcal{A}$ and receives messages `BATCH` and `COMMIT` from $\mathcal{A}$.

Of course, all this is good only when the corrupt parties follow the off-chain protocol faithfully. Now a corrupt party might not send its inputs or its signature or might directly trigger the contract or might attempt to keep the contract and off-chain executions out of sync. Not supplying inputs is handled in a straightforward manner by just replacing it with $\perp$. Not supplying signatures on batched inputs corresponds to a halt in the off-chain round. This in turn would result in honest parties having to escalate the off-chain round to the on-chain contract. Lemma 3 then shows how the execution gets completed on-chain. Since $\mathcal{S}$ closely mimics the real protocol, the indistinguishability follows from the proof of Lemma 3 in this case. Directly triggering the on-chain contract would result in a state change that notifies the (simulated) honest parties who can then issue an `update` command to the contract. This case is handled by Lemma 5. Finally, $\mathcal{A}$'s attempts to keep the on-chain contract and off-chain executions out of sync are handled by observing that on-chain contract notifications (in particular $\mathsf{EventPending}$) are available to all honest parties who can then recover from inconsistent corrupt inputs by issuing an `update` command to the contract that syncs the state on the on-chain chain with the off-chain state. This case is handled by Lemma 3 and Lemma 5.

Finally, we consider the case when the leader is unable to provide a `COMMIT` message for this round. This could happen if either the leader is corrupt, or the corrupt parties did not submit a signature on the `BATCH` message the contains this round's messages. In either case, Lemma 4 applies and we are guaranteed that the next round honest messages are exchanged only after the current round state is synchronized among the parties.

Contract inputs and outputs (including coins) are handled in the straightforward manner. One thing to note is that the event emitted by the on-chain contract may be delayed. Other than this, simulation proceeds by faithfully applying the parameterized update function to the state and also maintains a variable $\mathsf{bestRound}$ which in addition to variables state and $\mathsf{applied}$, keeps track of the on-chain state (and prevents replay attacks). $\square$

B Details of Linked Payments Construction

In the body of the paper (Figure 11) we presented the update function and auxiliary smart contracts for the state channel protocol Π_{Linked}. In Figure 13 we define the local behavior of the parties.

We state and provide a proof sketch of our main theorem for $\mathcal{F}_{\mathsf{Linked}}$.

Theorem 7. *Protocol Π_{Linked} realizes $\mathcal{F}_{\mathsf{Linked}}$ functionality in the $\mathcal{F}_{\mathsf{State}}$-hybrid world.*

Proof. (Sketch) The ideal world simulator $\mathcal{S}$ for the dummy real world adversary runs a sandboxed execution of Π_{Linked} through which it relays instructions while faithfully simulating honest parties exactly as described in Π_{Linked}. Note that in the simulation, $\mathcal{S}$ would act both as $\mathcal{F}_{\mathsf{State}}$. While $\mathcal{S}$ itself has quite a rich interface via $\mathcal{F}_{\mathsf{Linked}}$, the crux of the proof will be in proving that during the course of this interaction, $\mathcal{F}_{\mathsf{Linked}}$ never raises an `Exception`.

Proposition 8. *For each $i \in 1...(\ell-1)$, flag_i must be in a terminal state, i.e., $\mathsf{flag}_i \in \{\texttt{cancel}, \texttt{complete}\}$ at time $T+O(\ell+\Delta)$ (or at time $T+O(\ell)$ when all parties are honest).*

Proof sketch. First we consider the case when all parties are honest (and each party has sufficient balance). In this case, the sender P_1 starts a fresh round on its state channel (with P_2) where it presents $h \leftarrow \mathcal{H}(x)$ for a randomly chosen x. Note that P_1 also sends x to P_ℓ. Observe that the flag variable on $\mathcal{F}^1_{\mathsf{State}}$ state channel will be set to $\mathsf{inflight}$, following which (honest) party P_2 would start a fresh round on its state channel with P_3 where it simply forwards h that it received from $\mathcal{F}^1_{\mathsf{State}}$. This process repeats until it is P_ℓ's turn where P_ℓ multicasts the preimage x that it received from P_1. Then, each (honest) party P_i simply passes a `complete` command to $\mathcal{F}^i_{\mathsf{State}}$ Note that each of these steps take one time step. Therefore, the whole protocol completes within $\ell+2$ time steps (all off-chain), i.e., by time $T+O(\ell)$. Furthermore, each of the flag variables flag_i would be in a terminal state `complete`.

When not all parties are honest, then it is possible that $\mathcal{F}^i_{\mathsf{State}}$ might receive a `cancel` command from corrupt P_i. In this case, the `cancel` command is propagated all the way back to P_1 by the honest parties. In this case, all the local flag variables in $\mathcal{F}^i_{\mathsf{State}}$ are set to `cancel` and P_1 ends getting its deposit back. Note that the `cancel` event is confirmed on the on-chain auxiliary contract. Since this is settled on-chain, we incur an additional Δ delay. One final scenario is when each $\mathcal{F}^i_{\mathsf{State}}$ is set to $\mathsf{inflight}$ (i.e., in the forward path), and P_ℓ revealed the preimage but a corrupt party does not agree to the update off-chain. Once again, the state channel synchronization process escalates to the on-chain contract where this will be settled, and the flag variables will be set in this case to `complete`. Note that the on-chain escalation will induce an additional Δ delay (but this happens in parallel for each state channel instance) and therefore the proposition holds in this case as well. This concludes the proof of the proposition.

Proposition 9. *For each $i \in 1...(\ell-2)$, if P_i is honest, it must not be the case that $(\mathsf{flag}_i, \mathsf{flag}_{i+1}) = (\texttt{cancel}, \texttt{complete})$ at time $T+O(\ell+\Delta)$ (or at time $T+O(\ell)$ when all parties are honest).*

Proof sketch. We discuss the case when all parties are honest. As described in the proof of Proposition 8, when

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

```
Let $T_{\mathsf{Expiry}} := T + 6\ell + \Delta$.
Let $T_{\mathsf{Crit}} := T_{\mathsf{Expiry}} - \Delta$ // Last chance to submit $x$
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 $\mathsf{state}$ 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 $\mathsf{cmd}_{\mathtt{L}} = \mathsf{open}(h')$ and $\mathsf{flag} = \mathtt{init}$, then
  set $\mathsf{cred}_{\mathtt{L}} \mathrel{-}= \$X$, $\mathsf{flag} := \mathtt{inflight}$, and $h := h'$
otherwise if $\mathsf{cmd}_{\mathtt{L}} = \mathtt{complete}$ and $\mathsf{flag} = \mathtt{inflight}$,
  set $\mathsf{cred}_{\mathtt{R}} \mathrel{+}= \$X$, and $\mathsf{flag} := \mathtt{complete}$
otherwise if $\mathsf{cmd}_{\mathtt{R}} = \mathtt{cancel}$ and $\mathsf{flag} = \mathtt{inflight}$,
  set $\mathsf{cred}_{\mathtt{L}} \mathrel{+}= \$X$ and $\mathsf{flag} := \mathtt{cancel}$
otherwise 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 11, 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
      $\mathsf{deposits}_{\mathtt{R}} \mathrel{+}= \$X$
    else
      $\mathsf{deposits}_{\mathtt{L}} \mathrel{+}= \$X$
    $\mathsf{aux}_{\mathsf{in}} := (\mathsf{deposits}_{\mathtt{L}}, \mathsf{deposits}_{\mathtt{R}})$
  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 $\mathtt{True}$ if $h \in \mathsf{timestamp}$ and $\mathsf{timestamp}[h] \le T'$
  return $\mathtt{False}$ otherwise
```

Local protocol for sender, P_1

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

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

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

Local protocol for recipient, P_ℓ

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

Other messages. Messages involving the $\mathcal{F}_{\mathsf{Pay}}$ interface are routed between the environment and the $\mathcal{F}_{\mathsf{State}}$ functionality according to Π_{Pay} (see Figure 6)

Figure 13: Construction for $\mathcal{F}_{\mathsf{Linked}}$ in the $\mathcal{F}_{\mathsf{State}}$-hybrid world. Portions of the update function $U_{\mathsf{Linked},\$X}$ that are delegated to the underlying U_{Pay} update function (Figure 11) are colored blue to help readability. The left column is duplicated from Fig. 11.

all parties are honest, we have the payment will be settled within time $T + O(\ell)$, and furthermore the flag variables will be such that $\mathsf{flag}_i = \texttt{complete}$ for all i. Next we focus on the interesting case where some parties are corrupt. Then, we need to prove that the flag variables are pairwise consistent. When P_i and P_{i+1} are both honest, this follows from the fact that state channels remain synchronzied no matter how the payment between P_1 and P_ℓ is settled. The interesting case is when say P_i is honest (condition in proposition statement) and P_{i+1} is corrupt. That is, it suffices to prove that if $\mathsf{flag}_i = \texttt{cancel}$, then it must hold that $\mathsf{flag}_{i+1} = \texttt{cancel}$ also holds. Now if P_ℓ was corrupt and P_{i+1} revealed the hash preimage (received from P_ℓ) but the payment settlement was escalated to the preimage manager contract where the preimage was revealed, then it follows from the protocol description that in this case $\mathsf{flag}_i = \texttt{complete}$ (i.e., the proposition precondition does not hold). The remaining cases are relatively straightforward as the only way flag_i is set to $\texttt{cancel}$ is when P_{i+1} supplied $\texttt{cancel}$ to the state channel between P_i and P_{i+1} (and we already handled the case when the preimage manager is involved in changing flag_i to $\texttt{complete}$. This concludes the proof of the proposition.

Proposition 10. *If P_1 and P_ℓ are honest, then* $(\mathsf{flag}_1, \mathsf{flag}_{\ell-1}) \in \{(\texttt{complete}, \texttt{complete}), (\texttt{cancel}, \texttt{cancel})\}$ *at time $T + O(\ell + \Delta)$ (or at time $T + O(\ell)$ when all parties are honest).*

Proof sketch. When all parties are honest, the proposition directly follows from the proof of Propositions 8 and 9. The interesting case is when some parties are corrupt. Note that when both P_1 and P_ℓ are honest, it follows that if P_ℓ received $\mathsf{inflight}$ from $\mathcal{F}^{\ell-1}_{\mathsf{State}}$ then it would multicast the preimage of h to all parties. We analyze two cases depending on whether P_ℓ multicasted the preimage or not. Note that since P_1 and P_ℓ are honest the adversary does not have knowledge (except with negligible probability) of the preimage unless it was revealed by P_ℓ. Suppose P_ℓ revealed the preimage of h. Then, in this case, we will argue that the end state for the flag variables will be $\texttt{complete}$. This is because once the preimage was revealed, all honest parties become aware of it and each honest P_i would issue a $\texttt{complete}$ command to $\mathcal{F}^{i}_{\mathsf{State}}$ and either synchronize the state on-chain (with the help of the preimage manager contract) or off-chain. This combined with the fact that P_1 is honest implies that the end state for the flag variables $\mathsf{flag}_1, \mathsf{flag}_\ell$ would be $\texttt{complete}$. We now turn to the case when P_ℓ never revealed the preimage. Then from the protocol description it follows that the adversary is not aware of the preimage and therefore cannot transition a state channel to $\texttt{complete}$. Therefore in this case, each state channel will result in canceling the $\mathsf{inflight}$ status, and it follows that both P_1 and P_ℓ will end up with flag variables $\mathsf{flag}_1, \mathsf{flag}_\ell$ set to $\texttt{cancel}$. This concludes the proof. $\square$

B.1 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, and the sender and recipient would not be able to complete their payment otherwise. Therefore it is natural for the sender to provide an additional fee along with the payment, which can be claimed by each intermediary upon successful 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.

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

C Code listing of the Ethereum code for State Channel

Our implementation of the state channel contract is given in Fig. 14.

D Survey of Payment Channels

In this section, we provide a short survey of related payment channels Duplex Micropayment Channels, Lightning Channels and Raiden. A comprehensive comparsion for Duplex Micropayment Channels and Lightning Channels is available here [18] and we use its notation for describing all protocols.

Duplex Micropayment Channels A proposal by Decker and Wattenhofer supports bi-directional payments. Its goal is to extend the number of transfers possible compared to Spilman's channels [27] within the same lifetime. Structurally, it relies on a pair of unidirectional channels $C^{A\to B}, C^{B\to A}$ and an invalidation tree with a single active branch that represents the current pair of unidirectional channels. Each node in the tree is a transaction whose absolute lock time is equal or larger than its parent i.e. $(T_{1,99}, T_{2,100}, T_{3,100})$.

Remarkably, lock times have a transient property such that reducing the lock time of a node in the branch will automatically invalidate all children nodes. This invalidation is leveraged if either unidirectional channel exhausts its supply of coins. Both parties co-operate to create a new pair of unidirectional channels with their

```
contract StateChannel {
    address[] public players;
    mapping (address => uint) playermap;
    int bestRound = -1;
    enum Flag { OK, PENDING }
    Flag flag;
    uint deadline;
    mapping ( uint => bytes32[] ) inputs;
    mapping ( uint => bool ) applied;

    bytes32 aux_in;
    bytes32 state;

    event EventPending (uint round, uint deadline);
    event EventOnchain (uint round);
    event EventOffchain (uint round);

    function handleOutputs(bytes32 state) {
        // APPLICATION SPECIFIC REACTION
    }

    function applyUpdate(bytes32 state, bytes32 aux_in, bytes32[] inputs) returns(bytes32) {
        // APPLICATION SPECIFIC UPDATE
    }

    function input(uint r, bytes32 input) onlyplayers {
        uint i = playermap[msg.sender];
        assert(inputs[r][i] == 0);
        inputs[r][i] = input;
    }

    function triggerT1(uint r) onlyplayers {
        assert( r == uint(bestRound + 1) ); // Requires the previous state to be registered
        assert( flag == Flag.OK );
        flag = Flag.PENDING;
        deadline = block.number + 10; // Set the deadline for collecting inputs or updates
        EventPending(r, block.number);
    }

    function triggerT2(uint r) {
        // No one has provided an "update" message in time
        assert( r == uint(bestRound + 1) );
        assert( flag == Flag.PENDING );
        assert( block.number > deadline );

        // Process all inputs received during trigger (a default input is used if it is not received)
        flag = Flag.OK;
        state = applyUpdate(state, aux_in, inputs[r]);
        EventOnchain(r);
        bestRound = int(r);
    }

    function update(Signature[] sigs, int r, bytes32 _state) onlyplayers {
        if (r <= bestRound && applied[r]) return;

        // Check the signature of all parties
        var _h = sha3(r, state);
        for (uint i = 0; i < players.length; i++) {
            verifySignature(players[i], _h, sig);
        }

        // Only update to states with larger round number
        if ( r == bestRound + 1) {
            // Updates for a later round supercede any pending state
            if (status == Status.PENDING) {
                status = Status.OK;
                EventOffchain(uint(bestRound));
            }
            bestRound = r;
            state = _state;
        }
        applied[r] = true;
        handleOutputs(_state);
    }
}
```

Figure 14: Solidity contract for general purpose state channel, corresponding to the pseudocode in Fig. 8.

Channel Establishment:

$A \rightarrow B$: Funding input π_A

$B \rightarrow A$: Funding T^F, Invalidation Tree's first branch $(T_{1,k},...,T_{d,k})$ and a signature $\sigma_{B,i}^{B \rightarrow A}$ for the channel $C_i^{B \rightarrow A}$

$A \rightarrow B$: Signature $\sigma_{A,i}^{A \rightarrow B}$ for the channel $C_i^{A \rightarrow B}$ and a signature for each node in the branch $(\sigma_A^{1,k},...,\sigma_A^{d,k})$

$B \rightarrow A$: Signature for each node in the branch $(\sigma_B^{1,k},...,\sigma_B^{d,k})$ and a signature σ_B^F for the Funding T^F

$A \rightarrow N$: Signed Funding T^F

Send a Payment:

$A \rightarrow B$: Signature $\sigma_A^{A \rightarrow B}$ for channel $C^{A \rightarrow B}$

Send a Conditional Transfer:

$A \rightarrow B$: Signature $\sigma_A^{A \rightarrow B}$ for channel $C^{A \rightarrow B}$ which has an additional conditional transfer (output) based on $H(x)$

Reset Channels:

$A \rightarrow B$: Next branch in the Invalidation Tree $(T_{1,k},...,T_{d,k-1})$ and a signature $\sigma_{A,i+1}^{A \rightarrow B}$ for the channel $C_{i+1}^{A \rightarrow B}$

$B \rightarrow A$ Signature for each node in the branch $(\sigma_B^{1,k},...,\sigma_B^{d,k-1})$ and a signature $\sigma_{B,i+1}^{B \rightarrow A}$ for the channel $C_{i+1}^{B \rightarrow A}$

$A \rightarrow B$: Signature for each node in the branch $(\sigma_A^{1,k},...,\sigma_A^{d,k-1})$

Close channel (Dispute):

$A \rightarrow N$: Each signed node in the current branch $(T_{1,k},...,T_{d,k})$ and the signed channel $C_i^{A \rightarrow B}$

$B \rightarrow N$: The signed channel $C_i^{B \rightarrow A}$

Note: Must wait until the leaf node's expiry time $T_{d,k}$ to settle both channels.

Close channel (Co-operative):

$A \rightarrow C$: Signature σ_A^S for Settlement T^S

$B \rightarrow N$: Signed Settlement T^S

Figure 15: Duplex Micropayment Channels with two parties A, B and the Bitcoin network N.

new respective balance before signing a new subset of nodes with smaller lock times to invalidate the previous active branch. Table 15 and the following will explain each stage:

Channel Establishment: Both parties co-operate to construct an unsigned funding transaction T^F, the first branch of the invalidation tree $(T_{1,k},...,T_{d,k})$, where d is the number of nodes in the tree and k is an absolute lock time that is equal or greater than the previous node's lock time, and the two unidirectional channels $A \rightarrow B, B \rightarrow A$. All transactions are mutually signed in the reverse order, and the funding transaction T^F is sent to the Bitcoin network and stored in the Blockchain.

Send a Payment: The sender creates and signs a new payment transaction T^P that re-distributes the coins in their unidirectional channel. Effectively, this transaction increments the coins sent to the receiver and decrements the sender's share of coins. If either channel exhausts its supply of coins, then both parties must co-operate to reset the balance of both unidirectional channels. To reset, both parties identify a node in the current active branch that has a lock time that is greater than its parent node.

For example, the middle transaction $T_{2,100}$ in the branch $(T_{1,99}, T_{2,100}, T_{3,100})$ is the next available node to have its lock time reduced. Both parties construct a new branch $(T_{1,99}, T'_{2,99}, T'_{3,100})$ and unidirectional channels $A \rightarrow B, B \rightarrow A$ that represents each party's respective balance. Crucially, these transactions are signed in the reverse order such that the unidirectionals channels are signed before $T'_{3,100}$ and $T'_{2,99}$. It is important to note that the first transaction $T_{1,99}$ does not need to be re-signed.

Send a Conditional Transfer: The sender creates a new payment transaction T^P with an additional output that locks coins into the conditional transfer in their unidirectional channel. These coins can be claimed by the receiver if the pre-image x of the hash $H(x)$ is revealed before time t^{expire}. On the other hand, the coins are refunded to the sender if the coins have not been claimed by t^{expire}.

Close Channel (dispute): Raising a dispute requires either party to wait for each node in the invalidation tree's absolute lock time to expire before it can be broadcast and accepted into the Blockchain. Notably, the leaf node's $T_{d,k}$ has two outputs to represent each party's unidirectional channel. If a payment transaction T^P that represents the final balance of the unidirectional channel is not accepted into the Blockchain before t_{refund}, then the channel's locked coins are returned to the sender.

Close Channel (co-operative): Both parties mutually sign a settlement T^S that has no lock time and sends each party their respective balance. This sent transaction is sent to the network and accepted into the Blockchain to close the channel.

Lightning Network A proposal by Poon and Dryja that is composed of two components to support off-chain transactions. The first is Lightning Channels that accepts deposits from two parties to support bi-directional payments without an expiry time. The second is Hashed Time-Locked Transactions (HTLC) that supports atomic

Channel Establishment:	
$A \rightarrow B$:	Revocation hash $H(S_{A,i})$, Funding input π_A
$B \rightarrow A$:	Revocation hash $H(S_{B,i})$, Unsigned Funding T^F, Signature $\sigma^C_{B,i}$ for $A's$ Commitment $T^{C,A}_i$
$A \rightarrow B$:	Signatures $\sigma^C_{A,i}$ for $B's$ Commitment $T^{C,B}_i$ and σ^F_A for Funding T^F
$B \rightarrow N$:	Signed Funding T^F
Send a Payment:	
$A \rightarrow B$:	New revocation hash $H(S_{A,i+1})$
$B \rightarrow A$:	New revocation hash $H(S_{B,i+1})$, Signature $\sigma^C_{B,i+1}$ for $A's$ new Commitment Tx $T^{C,A}_{i+1}$
$A \rightarrow B$:	Pre-image of previous revocation hash $S_{A,i}$, Signature $\sigma^C_{A,i+1}$ for $B's$ new Commitment $T^{C,B}_{i+1}$
$B \rightarrow A$:	Pre-image of previous revocation hash $S_{B,i}$
Send a Conditional Transfer:	
$A \rightarrow B$:	HTLC Output based on $H(x)$ *(Sender's role)*
$A \leftrightarrow B$:	*Repeat Send a Payment with additional HTLC Output*
Close channel (Dispute):	
$A \rightarrow N$:	Latest Commitment T^C_A *(Broadcaster)*
$B \rightarrow N$:	Delivery T^D_B *(Non-broadcaster claims balance)*
$A \rightarrow N$:	Revocable Delivery T^{RD}_A *(Broadcaster claims balance after time t)*
$B \rightarrow N$:	HTLC Delivery T^{HD}_B *(Receiver claims locked coins by revealing x)*
$A \rightarrow N$:	HTLC Refund Delivery T^{HR}_A *(Sender is refunded the locked coins after time t^{expire})*
Close channel (Co-operative):	
$A \rightarrow C$:	Signature σ^S_A for Settlement T^S
$B \rightarrow N$:	Signed Settlement T^S

Figure 16: Lightning Channels with two parties A, B and the Bitcoin network N.

transfers between two or more channels.

Combined, both components support the functionality required to build a payment network on Bitcoin. The coins deposited in each Lightning Channel provides a payment network with its liquidity, while HTLC transactions provides a means for parties that do not share a direct channel to transact with each other.

Before describing Lightning Channels it is important to note that invalidation of previous payments is performed using a penalty approach. There is a single active state that represents each party's balance in the channel and a list of previously revoked states. Cruically, broadcasting a revoked state allows the non-broadcaster to steal all coins in the channel. As such, only the mutually agreed active state that represents both parties current balance should be broadcast to the Blockchain. Table 16 and the following will explain each stage:

Channel Establishment: Both parties exchange new revocation hashes $H(S_{A,i}), H(S_{B,i})$, an agreed relative lock time t and an unsigned Funding Transaction T^F. Next, each party creates, signs and sends the counterparty a Commitment Transaction T^C that represents the channel's current state. Finally, the Funding Transaction can be signed σ^F and accepted into the Blockchain once both parties have received their signed Commitment Transaction $T^{C,A}, T^{C,B}$.

Sending a Payment: Authorising a payment requires both parties to exchange new revocation hashes $H(S_{A,i+1}), H(S_{B,i+1})$ and sign a new pair of Commitment Transactions $T^{C,A}_{i+1}, T^{C,B}_{i+1}$. Next, both parties need to invaldiate the previous pair of Commitment Transactions which simply involves exchanging the previous revocation hashes pre-images $S_{A,i}, S_{B,i}$.

Sending a Conditional Transfer: A conditional payment requires each channel along the route to lock coins that are only released if a secret x of $H(x)$ is revealed. Locking the coins requires the sender to receive a hash $H(x)$ from the previous hop along the route, both parties to exchange new revocation hashes $H(S_{A,i+1}), H(S_{B,i+1})$, and both parties to sign a new pair of Commitment Transactions $T^{C,A}_{i+1}, T^{C,B}_{i+1}$. Similarly to sending a payment, both parties need to exchange the previous revocation hashes pre-image $S_{A,i}, S_{B,i}$ to revoke the previous active state.

Cruically, both Commitment Transactions have an additional output that locks the sender's coins into the transfer. These coins are sent to the receiver if x is revealed, or refunded to the sender if the transfer has expired after time t^{expire}. Notably, this approach supports concurrent conditional transfers as only additional outputs are required to lock the coins.

Close Channel (Dispute): Raising a dispute requires the disputer to broadcast their Commitment Transaction to the Blockchain. The non-broadcaster can claim their coins immediately by signing a Delivery Transaction T^D, whereas the broadcaster must wait a mutually agreed grace period t before claiming their coins using a Revocable Delivery Transaction T^{RD}. If the broadcaster sends a previously revoked Commitment Transaction to

Channel Establishment:	
$A \rightarrow C$:	Counterparty B, settlement time t^{settle}, token α
$A \rightarrow C$:	Deposit d_a *(repeatable)*
$B \rightarrow C$:	Deposit d_b *(optional)*
Send a Payment:	
$A \rightarrow B$:	Signature σ_A^P, total transferred λ_A, nonce $c_{A,i+1}$, locksroot $\Gamma_A{}^{45}$
Where $\sigma_A^{CT} = S_{sk_a}(\alpha, B, \lambda_A, \Gamma_A, c_{A,i+1})$.	
Send a Conditional Transfer:	
$A \rightarrow B$:	Signature σ_A^{CT}, total transferred λ_A, nonce $c_{A,i+1}$ locksroot Γ_A, locked transfer γ_A hash h_A, expiration t_A^{expire}
Where $\sigma_A = S_{sk_B}(\alpha, B, \lambda_A, \Gamma_A, c_{A,i+1})$.	
Close channel:	
$A \rightarrow C$:	Signature σ_B, total transferred λ_B, nonce c_B, locksroot Γ_B *(B's latest transfer)*
$B \rightarrow C$:	Signature σ_A, total transferred λ_A, nonce c_A, locksroot Γ_A *(A's latest transfer)*
$A \rightarrow C$:	Locked transfer γ_B, secret s_B, merkle tree branch θ_B *(unlock transfer, repeatable)*
$B \rightarrow C$:	Locked transfer γ_A, secret s_A, merkle tree branch θ_A *(unlock transfer, repeatable)*
Withdraw:	
$A \rightarrow C$:	Settlement after t^{settle}

Figure 17: Raiden with two parties A, B and the smart contract C.

the Blockchain, then the grace period t provides the non-broadcaster time to issue a Breach Remedy Transaction T^{BR} that steals the broadcaster's share of coins.

Close Channel (Co-operative): Both parties sign a new Settlement Transaction T^S that sends both parties their respective balances. This single transaction is sent to the network and accepted into the Blockchain to close the channel.

Raiden It is inspired by the Lightning Network and is in active development to implement off-chain state channels for Ethereum. Table 17 and the following will explain each stage:

Channel Establishment: A single party creates the channel by submitting the counterparty's Ethereum account B, and a settlement time t^{settle} that provides a time window between closing the channel and settling its final balance. Both parties deposit tokens[6] into the contract if they are satisfied with the settlement time t^{settle}.

Sending a Payment: An incremental counter c is responsible for establishing the order of payments and each unidirectional channel $A \rightarrow B, B \rightarrow A$ has its own counter $c_{A,i}, c_{B,i}$ respectfully. Each payment requires the sender to increment their total transferred λ_A and the counter $c_{A,i+1}$ before signing the payment σ_A^P using their Ethereum account's private key. Notably, the total transferred λ value can be greater than the sender's deposit due to how the final balance is computed during the final settlement which we will present shortly. Also, the lock root Γ is only included and signed if there are pending conditional transfers.

[6]A Raiden-specified asset.

Sending a Conditional Transfer: The sender computes a hash h, the locked transfer amount γ, the conditional transfer's expiration time t^{expire}, the new lockroot Γ and an incremented counter c. This message is signed σ_A^{CT} using the sender's Ethereum account's private key and sent to the receiver. The lock root Γ is the root of a merkle tree for all pending conditional transfers $\gamma_1, ..., \gamma_n$ and must be included in future payments until all conditional transfers $t_1^{expire}, ..., t_n^{expire}$ have expired.

Close Channel: Either party can close the channel by submitting the latest transfer received from the counterparty. This begins the settlement period t^{settle} that provides a grace period for the counterparty to submit the latest transfer received from the closer. Furthermore, it provides time to unlock and claim each pending conditional transfer. This transfer can be claimed by submitting the locked transfer γ, the secret s that is the preimage of the hash h, and a merkle tree branch θ that proves the sender committed to this transfer under the condition that s is revealed before the expiration time t^{expire}. Notably, the current implementation requires a new transaction to claim each conditional transfer.

Settlement: As mentioned previously, the total transferred λ can exceed each sender's deposit in the contract. Determining the final balance requires computing the offset of each total transferred using the following equations:

$$A_{balance} = d_A + (\lambda_B - \lambda_A)$$

$$B_{balance} = d_B + (\lambda_A - \lambda_B)$$

Finally, the party that does not inform the contract to perform the settlement is sent their balance first, and the remaining tokens in the contract is sent to the caller.

Table 1: A comparison on the number of signatures required for each stage.

	Establishment	Payment	Transfer	Reset	Dispute
Duplex	$(d+2)\times 2$	1	1	$(\alpha+1)\times 2$	1×2
Lightning	2×2	1×2	1×2	N/A	3 or $\beta+3$
Raiden	1 or 2	1	1	N/A	$\beta+2$
Sprite	**1 or 2**	$\mathbf{1\times 2}$	$\mathbf{1\times 2}$	**N/A**	**1**

Table 2: Payment Channel Features

	Bi-directional	Storage of previous states	Worst-case delay	Incremental
Lightning [26]	●	$O(N)$	$O(\ell\Delta)$	○
Duplex [8]	◐	$O(1)$	$O(\ell\Delta)$	○
Raiden	●	$O(1)$	$O(\ell\Delta)$	○
Our work	●	$O(1)$	$O(\ell+\Delta)$	●

E Analysis

In this section, we compare Lightning Channels, Duplex Micropayment Channels, Raiden and Sprite. All comparisons are based on the protocol descriptions presented in Appendix D. We focus on the lock times required to atomically route payments across two or more channels and the number of signatures/transactions required for each step in the protocols.

E.1 Lock Times

Lightning, Duplex and Raiden have adopted Hashed Time-Locked Contract (HTLC) for conditional transfers that atomically route coins across two or more channels.

Notably, in a conditional transfer each intermediary party has two roles as the receiver in an incoming channel, and the sender in an outgoing channel. Coins are locked into the conditional transfer for Δ_{delay} blocks which represents the worst-case time that each party must wait before raising a dispute with the Blockchain which is highlighted in Table 2. Cruically, the party must ensure the conditional transfer's expiry time for the incoming channel is greater than the expiry time for the outgoing channel's locked transfer such that $\Delta^{in}_{delay} > \Delta^{out}_{delay}$. This provides a grace period for this party's coins to be claimed in the outgoing channel before claiming their coins in the incoming channel.

Unfortunately, the worst-case delay Δ_{delay} in Duplex must take into account the expiry for both the incoming and outgoing channel's lifetime. This is necessary as disputes cannot be settled in the Blockchain until these channels have expired. The worst-case scenario is if the final receiver's channel has the largest expiry time. In this situation, all hop's along the route's must have a worst-case delay Δ_{delay} greater than the final channel's expiry time.

In Lightning and Raiden the worst-case delay Δ_{delay} only relies on the hop's desired grace period as the channels have no expiry time. Although, the sender that establishes the route must lock their coins into the transfer with a lock time greater than the lock times used by all hops along the route. Ultimately, the coins can be locked for up to $O(\ell\Delta)$ blocks. For example, if the worst-case delay Δ_{delay} is 6 for a route of 10 channels, then the sender's coins are potentially locked for at most 60 blocks.

The introduction of a Preimage Manager in **Sprite improves the worst-case delay Δ_{delay} such that as it remains constant regardless of the route's size**. This is possible as Sprite can enforce the success of all conditional transfers along the route if the pre-image x of the conditional hash $H(x)$ is stored in the appointed Preimage Manager's contract before the worst-case delay Δ_{delay}. To put another way, any hop can raise a dispute by submitting the pre-image with confidence that all hops along the route will atomically accept the conditional transfer. As such, the worst case for the length of time coins are locked into the transfer is $O(\ell+\Delta)$

E.2 Signatures and Transactions

Table 1 highlights the number of signatures required for each stage of Duplex, Lightning, Raiden and Sprite. Next, we provide a comparison on the number of signatures and the number of transactions that are stored in the Blockchain for each stage of the protocols.

Channel Establishment. Lightning and Duplex require both parties to sign the channel's state before cooperatively signing a Funding Transaction that is stored in the Blockchain. In Lighting the state is simply a pair of Commitment Transactions, while in Duplex the state is the first branch of an Invalidation Tree which consists of d nodes and both Unidirectional Channels. On the other hand, both Raiden and Sprite require a single party to sign a transaction to establish the channel with their deposit. The counterparty can sign a second transaction that deposits their coins into the channel.

Payment. All payments and conditional transfers follow the same approach. No additional signatures are required in any scheme to lock coins into a new conditional transfer.

Raiden and Duplex are constructed using a pair of unidirectional channels which requires a single signature from the sender to authorise a payment. Notably, the unidirectional channels in Duplex can exhaust their supply of coins. This requires both parties to co-operatively perform $(\alpha+1)\times 2$ signatures to reset the coins in each channel, where α represents the list of transactions that need to be replaced in Invalidation Tree's current branch.

However, Raiden's unidirectional channels does not require a reset as it only increments the total quantity of transferred coins and an offset is considered during the settlement phase to compute the final balance for both parties.

On the other hand, Lightning and Sprite maintain both party's current balance and sending a payment requires both parties to co-operatively sign a new transaction that updates their balances. Notably, Lightning revokes the previous state after both parties mutually agree a new state. These previous states are invalidated using a penalty approach that requires parties to store all preimages $(x_1,...,x_n)$ and $(s_1,...,s_n)$. This storage requirement is highlighted in Table 2 as $O(N)$, where N is the number of previous states. On the other hand, Sprite increments a nonce which the contract enforces to represent the latest state.

Settlement and Dispute. All protocols can be settled if both parties co-operatively sign a transaction that sends them their final balance.

Duplex and Raiden potentially require the most on-chain transactions in the event of a dispute. In Duplex, the disputer should already have the signatures for all d nodes in the current branch of the Invalidation Tree. Both parties must sign the unidirectional channels that represent their latest payment from the counterparty and send both transactions to the network. In Raiden, each party signs the unidirectional channel that represents the latest payment from the counterparty and again two transactions are stored in the Blockchain[7] However, each conditional transfer that needs to be unlocked requires the receiver to sign an additional transaction β to claim it.

Lightning requires the disputer to sign and send their Commitment Transaction for acceptance into the Blockchain. It is possible for each party to sign a single transaction that claims both their final balance and all unlocked conditional transfers. However, depending on the lock times associated with the conditional transfers, then it is likely that an additional transaction β is required to claim each conditional transfer.

Unlike all others protocols, **Sprite is the only payment channel protocol that does not need to close in order to resolve a dispute for a conditional transfer**. Any party in the route can submit the pre-image x of the conditional hash $H(x)$ to the PreimageManager. This dispute will enforce the conditional transfer confirmation all channels if the preimage is accepted before the expiry time T_{Expiry}. Furthermore, it is possible for both parties to withdraw coins from the channel by co-operatively signing a withdrawal transaction. This can be sent to the Blockchain for the coins to be released without the need to close the channel.

[7]Raiden is in the process of fixing a bug we reported during our analysis of their implementation `https://github.com/raiden-network/raiden/issues/365`.