Pisa: Arbitration Outsourcing for State Channels

Patrick McCorry
IC3 and University College London, UK

Surya Bakshi
IC3 and University of Illinois at Urbana Champaign, USA

Iddo Bentov
IC3 and Cornell University, USA

Sarah Meiklejohn
IC3 and University College London, UK

Andrew Miller
IC3 and University of Illinois at Urbana Champaign, USA

ABSTRACT

State channels are a leading approach for improving the scalability of blockchains and cryptocurrencies. They allow a group of distrustful parties to optimistically execute an application-defined program amongst themselves, while the blockchain serves as a backstop in case of a dispute or abort. This effectively bypasses the congestion, fees and performance constraints of the underlying blockchain in the typical case. However, state channels introduce a new and undesirable assumption that a party must remain on-line and synchronised with the blockchain at all times to defend against *execution fork* attacks. An execution fork can revert a state channel's history, potentially causing financial damage to a party that is innocent except for having crashed. To provide security even to parties that may go off-line for an extended period of time, we present Pisa, a protocol enables such parties to delegate to a third party, called the custodian, to cancel execution forks on their behalf. To evaluate Pisa, we provide a proof-of-concept implementation for a simplified Sprites and we demonstrate that it is cost-efficient to deploy on the Ethereum network.

1 INTRODUCTION

Improving the scalability of cryptocurrencies and blockchains has emerged as an important open problem facing the nascent industry. Although cryptocurrencies have achieved record growth (a total market capitalisation of $300bn as of April 2018), their ability to scale and increase transaction capacity is fundamentally at odds with their approach to security through wide replication [8]; in a typical cryptocurrency, every node processes every transaction. To highlight this tension, Ethereum has relaxed a throughput capacity limit imposed by Bitcoin [23], but as a result some peers lack the resources to verify new transactions in real-time [26].

Payment channels are a leading scalability approach that overcomes this tradeoff [4, 12, 17, 21, 25]. The main idea behind payment channels is optimism: the blockchain serves as a backstop or dispute handler, and in the typical case payments are carried out privately among small groups of parties via off-chain messages. Funds deposited in a payment channel are guaranteed to be secure, even if all other parties in the channel are malicious. This is because the honest party can rely on the blockchain to enforce payments and authorised withdrawals. While this scaling approach was originally proposed for payment applications, a generalization of the technique, called *state channels* [7, 12, 25], promises to bring scalability benefits to other smart contract applications as well, such as auctions and online games [2].

The security guarantees of state channels are ensured by the on-chain dispute handling mechanism. However, this mechanism also introduces a new failure mode, since it assumes each party remains on-line and synchronised with the blockchain at all times. To briefly explain the mechanism, each party in a state channel maintains a local view of the most recent channel state (e.g., account balances in the case of payment channels, and arbitrary application-defined state more generally), along with signatures from every other parties. If at least one other party aborts (or provides invalid data), an honest party must publish the most recent signed state to the blockchain to initiate the dispute process. To handle the case where a malicious party initiates a spurious dispute, other parties are given a fixed time period to submit a *newer* signed state to resolve the dispute. This is effective as long as honest parties remain online and responsive; a node that crashes or goes offline for a long period of time may miss the time window to participate in the dispute process. Worse, a malicious actor may exploit an offline party by reversing (or forking) collectively authorised states in the channel to their benefit (an "execution fork"). The global blockchain can ease the burden on parties by providing a longer grace period during which they can intervene, but this increases the time it takes to progress the program and thus introduces a trade-off between security and performance.

The hazard of execution fork attacks against offline parties is already known in the off-chain scaling community, who have proposed two mitigations thus far. Both existing approaches empower users to appoint a third party to help defend against execution forks on their behalf. The first proposal, called Monitor [10], is inefficient in that it requires the third party to consume $O(N)$ storage, where N is the number of off-chain transactions that have occurred within the state channel. The second proposal, called WatchTower [27], improves upon Monitor's efficiency, but cannot be deployed without consensus rule changes in the Bitcoin network, and cannot directly be adapted to generic state channels in Ethereum. As we explore in Section 3.2, both proposals suffer from the drawback that if the appointed third party fails, there is little recourse for the customer since the protocols do not provide evidence about the appointment.

To overcome the above problems with third-party state channel monitoring, we propose Pisa, which introduces a new third party, called the custodian, which can be applied to generic state channel constructions. In Pisa, all appointments and payments between a party and the custodian are performed using an off-chain payment channel. In return, the party receives a signed receipt from the custodian, which constitutes evidence that custodian agrees to defend the channel. This receipt can later be used to penalise the custodian if they fail to defend against an execution fork on the party's behalf. PISA improves on the efficiency of prior proposals, requiring only $O(1)$ storage from the custodian, just the hash of the program's most recent state. PISA is application-neutral, and can be used with any state channel program, since the custodian only receives a hash of the channel's state rather than the state itself. This also provides a privacy benefit, called state privacy, since the custodian does not learn any details about the off-chain interactions (e.g., payment metadata) except when a dispute is raised via the global blockchain.

To summarise, our contributions are as follows:

- We propose PISA, the first state channel protocol that supports third-party monitoring for arbitrary applications.
- PISA is also the first state channel protocol that provides a customer with publicly verifiable cryptographic evidence in case the third-party fails, which can be used to penalise a faulty third-party.
- We provide a proof of concept implementation of PISA for a simplified Sprites and experimentally demonstrate that it is cost-efficient to deploy on the Ethereum network.

2 BACKGROUND AND RELATED WORK

In this section, we present cryptographic primitives used throughout this paper, an overview of the blockchain, smart contracts and payment channels before discussing related work.

2.1 Cryptographic notation

PISA and more generally state channels rely on cryptographic hash functions and digital signatures. We denote a hash computation as $h \leftarrow \mathsf{H}(\mathsf{msg})$ where H is the cryptographic hash function, msg is the pre-image and h is the resulting hash. For digital signatures, we denote signing as $\sigma_k \leftarrow \mathsf{Sign}(\mathsf{sk}_k, \mathsf{msg})$, where sk_i is the signer's secret key, msg is message to be signed and σ_k is the corresponding signature. Verification is denoted as $0/1 \leftarrow \mathsf{Sig.Verify}(\mathcal{P}, \mathsf{msg}, \sigma_k)$ where $\mathcal{P}$ is the party's corresponding public key and the algorithm returns 1 if the signature is authentic for msg.

2.2 Blockchains, accounts, and smart contracts

All parties independently compute their own pseudonymous identity called an external account, and this is simply the cryptographic hash of a public key. For readability, both external accounts and public keys are interchangeably denoted as $\mathcal{P}$. Once an account is associated with coins on the network, the party can digitally sign a transaction using their corresponding secret key sk. A transaction denotes the sender's account, receiver's account, the number of coins to transfer and a payload. In more expressive platforms like Ethereum, the payload stores bytecode which is used to deploy and instantiate a smart contract (i.e., a program) on the network, or contains instructions for executing a smart contract. The metric for size and computational complexity of this payload is called gas and the signer can set a gas price that they are willing to pay as a transaction fee.

All transactions are published and propagated throughout the peer-to-peer network. Each peer verifies whether the digital signature that authorises the transaction corresponds to an external account with a sufficient balance to cover the transaction's fee. Eventually the transaction should reach a group of peers called miners who collectively participate in a leader election every epoch. The first peer to solve a computationally difficult puzzle is elected as the epoch's leader and atomically creates a new block of transactions. This block is appended to a chain of previous blocks (which results in the name blockchain) and the miner is given a reward alongside the fees from each included transaction. Due to the probabilistic nature of the puzzle, two or more miners may propose a solution (i.e., a competing block) for any given epoch. All competing blocks are distinct forks based on the same parent block. The fork that emerges as the longest (and heaviest) chain is eventually considered the blockchain and only these transactions impact the network's state. The blockchain thus only provides eventual consistency and a transaction cannot be considered final until it is in the longest (and heaviest) chain.

Smart contracts are conceptually a third party that is trusted for correctness and availability but not for privacy [18]. We model a smart contract as a state machine and signed transactions carry commands cmd which execute the state transition from $\mathsf{state}_{\mathsf{i}-1}$ to $\mathsf{state}_{\mathsf{i}}$. The contract's code alongside a transcript of all previous state transitions is recorded in the blockchain and transactions that perform state transitions are deterministically executed by all peers on the network. This deterministic execution implies that the contract cannot store secret values, but the honest execution of its protocol is guaranteed. As mentioned previously, a fee is associated with each state transition performed on the network and the cost of executing a smart contract increases when the blockchain is congested as users compete for the remaining space.

2.3 Payment channels

The concept of a payment channel emerged in Bitcoin to avoid executing all payments on the blockchain. A payment channel allows two parties to deposit coins and continuously re-distribute each party's share of this deposit amongst themselves. This reduces transaction fees paid by the channel's parties and also reduces congestion on the network as computation is performed locally instead of the global network. More generally, both parties are executing state transitions and authorising (i.e. digitally signing) every new state amongst themselves in such a way that only the most recently authorised state should be accepted into the blockchain.

It is crucial that parties can invalidate previous states to ensure the network only accepts the latest state. The simplest approach for state replacement is called replace-by-incentive [9], which emerged to support one-way payment channels. As well, the receiver's incentive is to only publish the state that sends them the most coins. The receiver can either sign the state and publish it to the network, or wait for a new state that increments their balance. This is considered safe as every state requires both parties to authorise it before the state can be accepted into the blockchain. There is also an expiry mechanism that eventually refunds the sender if there is no activity in the channel. However to extend payment channels to support bi-directional payments (i.e. sending coins back and forth), early constructions for state replacement in Bitcoin involved decrementing the channel's expiry time [9, 24]. This is undesirable as every change in payment direction brings the channel's expiry time closer to present time and ultimately restricts the total number of payments.

To overcome issues in early payment channel constructions such as requiring an expiry time and the restricted throughput, Poon and Dryja proposed replace-by-revocation [31] as a state replacement technique. Unlike replace-by-incentive, both parties have a copy of the channel's latest state and the state replacement requires both parties to authorise a new state before revoking the old state. Thus there is a set of revoked states, and only a single valid state. It also introduced the concept of a dispute period where one party can seek assistance from the global blockchain to close the channel based on an authorised state. The counter-party has a fixed time to detect whether this authorised state was previously revoked, and if so, the counter-party can submit evidence to the blockchain that the state was indeed revoked and in return they are sent all coins in channel. Otherwise the dispute period expires and both parties receive their share of the channel's deposit according to the accepted state. As we explore in the Section 3, a payment channel can be generalised for arbitrary protocols involving n parties and this dispute process can be used to enforce the protocol's progression (instead of simply closing the channel).

2.4 Related work

Probabilistic micropayments. To avoid processing all transactions on the blockchain, Pass and shelat [29], and Chiesa et al. [5] proposed using probabilistic payments. Every payment is a lottery and the receiver is only paid upon winning. The former relies on a trusted third party, or tying up more collateral than the largest possible payment. The latter adds support for privacy-preserving payments, and analyses the collateral requirements assuming a rational adversary. Probabilistic payments indeed reduce network congestion, though they increase the variance until the payee receives money, and they also have weaker expressibility (i.e. only supports payments) than a state channel.

Channels based on trusted hardware. Teechain [19] is an off-chain payment channel protocol for Bitcoin that utilizes the Intel SGX technology to produce the settlement transactions via SGX enclaves the run in the parties' computers, rather than by the parties directly. Custodians are in fact unnecessary with Teechain, since an enclave will only agree to produce a signed settlement message (to be sent to the blockchain) that deducts all the amounts that its operator paid to other parties. However, the Teechain protocol is risky even in the case that SGX is completely secure, because a party will lose all of her money if the SGX enclave stops running.[1] The risk can be overcome by allowing rollback (to an enclave backup image) in accord with monotonic hardware counters, but SGX does not have a secure implementation of hardware counters [22, 33]. Teechain proposes an extension where a party duplicates the secret data by using several computers that have SGX enclaves (these computers run continuously and communicate among each other via secure channels in order to backup the secret data). Still, the money will be lost if all of these computers crash at the same time.

Payment networks. Poon and Dryja proposed the concept of a payment (and collateral-based) network for Bitcoin. This allows multiple two-party channels to form a route and synchronise payments across the route using hashed time-locked contracts (HTLC) [24, 31]. Miller et al. proposed how to reduce the worst-case delay of HTLC to constant time for all channels in the route [25]. As well, Khalil et al. proposed REVIVE [17] that relies on this synchronisation technique to allow parties to re-balance their share of coins in a channel without interacting with the blockchain. In terms of privacy, Malavolta et al. proposed how to preserve the route's privacy [21] and, Green and Miers proposed BOLT [14] that allows two channels to transact via a single intermediary channel in a privacy-preserving manner. Instead of synchronising a single payment, Dziembowski et al. proposed Perun [11] that allows two parties to establish a route and conduct multiple payments without interacting with the intermediate channels. As mentioned earlier, Pisa is complementary to payment networks.

3 STATE CHANNELS

A state channel allows a group of mutually distrustful parties to execute an arbitrary application amongst themselves, while bootstrapping trust from the underlying blockchain. The blockchain (and in effect, the smart contract) is consulted only to open the channel, to store the latest authorised state if necessary and to resolve any disputes that occur. Our work builds on state channels to make them more robust in case some parties crash or go offline. Before proceeding to our main protocol, in this section we explain in detail the underlying state channel construction.

We mostly follow the abstraction provided by Miller et al. [25][2] A state channel for n participants $\mathcal{P} = \mathcal{P}_1, ..., \mathcal{P}_n$ is modeled as a state machine that proceeds logically in rounds, and is parameterized by an application-defined space of input commands cmd and a $\mathsf{Transition}$ function. In each round, the state channel receives an input cmd from one of the parties and applies the transition function

$$\mathsf{state}_\mathsf{i} \leftarrow \mathsf{Transition}(\mathsf{state}_{\mathsf{i}-1}, \mathsf{cmd}).$$

The main challenge in constructing a state channel is to ensure that the transition function is applied consistently, even in the case that one or more of the parties is malicious or aborts.

3.1 Generic state channel construction

At a high level, the state channel construction consists of an initialization routine to set up the channel, an off-chain protocol by which parties can collectively authorise new state transitions amongst themselves, and a smart-contract based dispute resolution process in case some party fails.

To initialize a state channel, one party must deploy a smart contract to the network and register all parties $\mathcal{P}$ in the contract to set it up. Notationally, we use `SC` to denote the unique identifier for the state channel contract, and use `SC.setup` (as one example) to denote a function call in this contract, with the function inputs omitted for readability.

Once established, all registered parties participate in an interactive protocol called AuthState to execute and authorise new state transitions. If all parties co-operate, then the contract `SC` is not involved and the state replacement is performed off-chain. To keep track of replaced states, a monotonic counter i is incremented for every state transition and the state associated with the largest i is considered the latest state (i.e. replace by monotonic counter). If

[1] Even due to suspend or hiberation, see https://software.intel.com/en-us/node/708995.

[2] Alternative state channel abstractions include StCon by Bentov et al. [2] and Perun by Dziembowski et al. [11].

one party fails to participate in AuthState, then the state replacement cannot be performed off-chain. Instead, another party in the channel must initiate the dispute process in SC to complete the state transition. To initiate, one party updates the contract with an authorised state (i.e. $\mathsf{state}_{\mathsf{i-1}}$) before triggering the dispute process. This provides a time period for each party to input a command. After this time period, SC selects one command (i.e. cmd is selected by the application's logic), executes it and stores $\mathsf{state}_{\mathsf{i}}$ as the latest authorised state (alongside an incremented i). We now describe the construction in more detail.

Channel flags. A state channel has three flags $[\bot, \mathsf{OK}, \mathsf{DISPUTE}]$ where $\bot$ denotes the channel is not initialised, OK specifies the channel is open and all parties can collectively authorise new states, and DISPUTE signals that one party has triggered a dispute and the state transition from $\mathsf{state}_{\mathsf{i-1}}$ to $\mathsf{state}_{\mathsf{i}}$ must be authorised by the contract.

Channel establishment. One party is responsible for deploying the state channel contract to the blockchain using SC.setup. The channel must be initialised with a list of parties $\mathcal{P} = \mathcal{P}_1, ..., \mathcal{P}_n$ and a timer Δ_{settle} which specifies the minimum length of time for the dispute process. Once the channel establishment is complete, the channel's flag transitions $\bot \rightarrow \mathsf{OK}$ and all parties can begin collectively signing (and authorising) every new state. The first state_1 is dependent on the channel's application.

State replacement. A state channel's integrity relies on all parties collectively invalidating previously authorised states to ensure SC always accepts the latest authorised state. In this construction, all parties collectively participate in an interactive protocol called AuthState (defined in Figure 4) that associates every new $\mathsf{state}_{\mathsf{i}}$ with an incremented counter i. AuthState requires one party $\mathcal{P}_k$ to select a command cmd and locally execute the state transition.[3]

This party separately signs the command and the new $\mathsf{state}_{\mathsf{i}}$ (alongside a newly incremented i). Both signatures $\sigma_k^{\mathsf{cmd}}, \sigma_k^{\mathsf{state}}$ and the values cmd, $\mathsf{state}_{\mathsf{i}}$, i are sent to all other parties. Each party must verify both signed messages and the state transition, according to:

$$0/1 \leftarrow \mathsf{VerifyTransition}(\mathcal{P}_k, \mathsf{cmd}, \sigma_k^{\mathsf{cmd}}, \mathsf{state}_{\mathsf{i-1}}, \mathsf{i}, \sigma_k^{\mathsf{state}}, \mathtt{SC})$$

Once satisfied, an honest party sends all other parties their signature for the new state (and its corresponding i). If an honest party does not receive a signature from all other parties before a local timeout, then the signed cmd (alongside $\mathsf{state}_{\mathsf{i-1}}$) is used to initiate the dispute process and complete the transition. To avoid the cost incurred by the on-chain state transition, all parties must exchange signatures for $\mathsf{state}_{\mathsf{i}}$ before an honest party's local timeout.

Dispute process. Any party within the channel can enforce a state transition on-chain via the dispute process. First, one party updates the contract using SC.setstate with the latest $\mathsf{state}_{\mathsf{i-1}}$, its corresponding counter (e.g. i-1) and a list of signatures $\Sigma_{\mathcal{P}}$. This allows the contract to verify that $\mathsf{state}_{\mathsf{i-1}}$ was indeed authorised by all parties before accepting it. Second, one party initiates the dispute using SC.triggerdispute which transitions the channel's flag from OK to DISPUTE and establishes a deadline $\mathsf{t}_{\mathsf{settle}}$. Third, all parties can input a cmd to be considered for the state transition using SC.input before $\mathtt{SC}.\mathsf{t}_{\mathsf{settle}}$. After the deadline has passed, the final step requires one party to notify the contract using SC.resolve which selects a single command from the list of commands, executes it and performs the state transition to $\mathsf{state}_{\mathsf{i}}$.

There is a danger that one party triggers a dispute based on a previously authorised $\mathsf{state}_{\mathsf{i-1}}$ which has already been replaced off-chain by $\mathsf{state}_{\mathsf{i}}$. If the dispute is successful, then the contract enforces the transition to a forked $\mathsf{state}'_{\mathsf{i}}$ which may be different to $\mathsf{state}_{\mathsf{i}}$. We call this an *execution fork* as $\mathsf{state}'_{\mathsf{i}}$ cannot be replaced by $\mathsf{state}_{\mathsf{i}}$ once it is accepted by the contract. For example, if the application was a payment channel, $\mathsf{state}_{\mathsf{i}}$ may represent sending coins from Alice to Bob, but $\mathsf{state}'_{\mathsf{i}}$ represents a withdrawal that sends Alice her coins. Thus her coins are no longer available to facilitate the payment if $\mathsf{state}_{\mathsf{i}}$ replaced $\mathsf{state}'_{\mathsf{i}}$. Preventing an execution fork requires one party to settle the dispute before $\mathtt{SC}.\mathsf{t}_{\mathsf{settle}}$ by updating the contract with the competing $\mathsf{state}_{\mathsf{i}}$ (alongside i and the corresponding list of signatures) using SC.setstate. If $\mathsf{state}_{\mathsf{i}}$ is accepted, then the contract's flag transitions from DISPUTE to OK and the dispute is settled. This introduces a new assumption that each party must remain on-line and synchronised with the blockchain in order to detect (and settle) execution forks. To get around this, we evaluate solutions proposed in the community for payment channels which allows any party to outsource the responsibility of preventing an execution fork (and settling the dispute process) to a third party called the Monitor.

3.2 Monitor solution

Dryja proposed the concept of a Monitor for replace-by-revocation channels in Bitcoin. This is a third party agent who is appointed by a customer to settle disputes (and prevent execution forks) on their behalf [10]. Briefly, the customer signs a new transaction that rewards both the customer and the Monitor if it is used to settle a dispute. This transaction (and the customer's signature) is encrypted using a secret key that is only revealed if the counter-party publishes a previously invalidated transaction.[4] When a customer appoints the Monitor to watch a channel on their behalf, the customer sends the Monitor this encrypted transaction. Once a dispute is detected in the customer's channel, the Monitor must try to decrypt this encrypted transaction. If the decryption is successful, then the Monitor confirms that the transaction contains the expected payment before settling the dispute.

Only the transaction which settles a dispute is revealed and all intermediary transactions remain are hidden. As well, more than one Monitor can be appointed to watch the channel and the counterparty is unaware of any appointment. However due to the nature of replace-by-revocation channels as presented in Section 2.3, the Monitor is required to store every encrypted transaction received from the customer and this employs a storage requirement of $O(N)$ for the Monitor per customer. To put this into perspective, Dryja claimed that for 10k channels, and 1 million payments per channel, the Monitor will need to store around 1TB of transactions [10].

On the other hand, the Monitor protocol relies on a bonus payment as the Monitor is rewarded only upon successfully settling a

[3] This transition function is available in SC and can be executed locally by the parties, but it is executed on the network only via the dispute process.

[4] In a replace-by-revocation channel, each party has a transaction that only they can broadcast. The unique identifier of the counterparty's transaction is used as the encryption key.

dispute. This reward policy is also under consideration by Raiden [6]. We call this the *double-deposit approach* as the channel must allocate coins for use in the channel and the bonus payment. This introduces an unfair reward policy as only a single Monitor can receive the payment and this results in a race condition as all appointed Monitors must compete to settle the dispute.[5] This is problematic in the context of a state channel as the deposit must be sufficient to support multiple disputes (as opposed to a single dispute that closes the payment channel). There is also no cryptographic evidence if the Monitor aborts and fails to settle a dispute on the customer's behalf. Thus, there is no mechanism for the customer to seek recourse or to publicly prove the Monitor's wrongdoing.

A second proposal by Osuntokun [16, 27] called WatchTower reduces the third party's storage requirement to $O(1)$, but this proposal cannot be deployed without a new consensus rule to update Bitcoin Script.[6] This new rule essentially allows Bitcoin to support the replace-by-monotonic-counter approach presented in Section 3.1 and thus WatchTower is potentially implementable in Ethereum. In terms of payment, it proposes paying the WatchTower via the off-chain payment network for every appointment as opposed to relying on the double-deposit approach. However the customer is not provided with evidence that the WatchTower was appointed to settle disputes on their behalf and thus there is no deterrence mechanism. Finally since WatchTower is designed for payment channels, it only considers settling a single dispute to close the channel as opposed to multiple disputes which is necessary to enforce a smart contract's progression.

4 PISA PROTOCOL

To overcome the above issues, we propose Pisa and a third party called the Custodian. Pisa has three components and it is designed for resolving disputes in the generic state channel construction presented in Section 3.1.

The first component is *publicly verifiable appointments* as the customer has a ratified signed receipt of appointment from the custodian. As well, the custodian has a storage requirement of $O(1)$. The second component is *fair (and real-time) rewards* as the custodian is paid for every new appointment using a one-way payment channel (or alternatively the custodian can be paid using a payment network). The third component is *customer recourse* as the ratified signed receipt can be used to prove the custodian's failed to settle a dispute on the customer's behalf and as a result the custodian's large security deposit is forfeited.

In the rest of this section, we present the goals and requirements before providing an overview of Pisa. Afterwards we highlight the modifications required to the generic state channel construction presented in Section 3.1 and the new custodian contract to facilitate the appointments from customers.

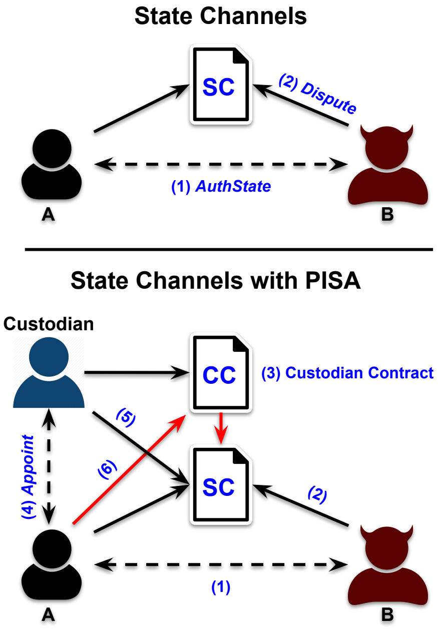

Figure 1: System Overview. In ordinary channels (top), parties communicate off-chain to authorize state transitions (1) through *AuthState* (Figure 4), but can raise a dispute on-chain (2) in case of abort. With Pisa (bottom), the interaction between channel participants, (1) and (2), remains unchanged, but A runs *Appoint* (Figure 3) to delegate to a third-party custodian, (4), who can submit a hashed state on A's behalf if A crashes (5). Pisa further guarantees that if the custodian fails, A obtains a publicly verifiable receipt and can penalize them through an on-chain recourse (6).

4.1 Protocol goals

We present a list of protocol goals for Pisa which focus on the privacy of intermediate states, fairness for the customer and custodian, and practical requirements that ensure Pisa can be deployed.

State privacy. We extend value privacy from [21] such that the custodian does not learn any information about the state it holds unless the state is revealed to the blockchain via the dispute process. In this case the custodian does learn whether they were appointed to settle a dispute for this state.

Fair exchange. An honest customer can review the signed receipt of appointment before deciding whether to pay the custodian in order to ratify it. On the other hand, an honest custodian is always paid once they ratify a signed receipt for the customer.

Non-frameability. A malicious customer or a full collusion of the state channel's parties cannot produce evidence that causes an honest custodian to lose their security deposit.

Recourse as a financial deterrent. A custodian is considered rational and only colludes against the customer if their payout is greater than the loss of their security deposit. An honest customer

[5] In Ethereum, the deposit can be split between all Monitors that respond to settle the dispute, but this increases the number of on-chain transactions and does not provide the Monitor with a fixed reward.

[6] A new opcode called OP_CHECKSIGFROMSTACK that can verify a signed message and parse its message.

should always be able to seek recourse and prove the custodian's wrongdoing.

Efficiency requirements. The custodian stores only information associated with the latest state, meaning its storage requirement is $O(1)$ per customer. The custodian should be paid using the one-way (and off-chain) payment channel.

4.2 Overview of PISA

Figure 1 presents a high-level overview of the system and Figure 2 presents the high-level interaction in PISA between three parties: two arbitrary parties in the state channel, and the custodian (and, implicitly, the contracts via interaction with the network). Initially, the custodian sets up the custodian contract CC, stores a large security deposit and publicly advertises their service. At some point, all parties establish a state channel and begin collectively authorising new states amongst themselves using the AuthState algorithm presented in Figure 4. If a party wishes to go offline for an extended period of time, but wants to ensure that previously invalidated states are not accepted by CC, they may decide to appoint a custodian to watch the state channel on their behalf.

We call this party the customer, who must deposit coins into the custodian's contract to set up a one-way payment channel. Afterwards the customer can outsource the job of monitoring the state channel using the Appoint algorithm in Figure 3. The custodian receives a hash of the state which we denote as $\mathsf{hstate_i}$ (alongside its counter i and signatures from all parties $\Sigma_{\mathcal{P}}$), the state channel contract's identifier SC and a payment for watching the channel. In return the customer is provided a signed receipt of appointment that specifies the monitoring time period.

As mentioned previously, once the customer is offline, all other parties may collude to perform an execution fork. In this case there are two outcomes. First, the custodian may settle a dispute by publishing $\mathsf{hstate_i}$ and the counter i alongside the list of signatures $\Sigma_{\mathcal{P}}$ on behalf of the customer. In this case the correct state hash will be accepted and the smart contract can continue. Otherwise, the custodian may fail to respond during the dispute process. In this case, PISA allows the customer to seek recourse after they come back on-line using both the signed receipt and a record of the dispute in the state channel. If the recourse is successful, then the custodian's contract forfeits the custodian's large deposit.

4.3 Protocol assumptions

We present a list of assumptions for the smart contracts and the threat model.

Smart contracts. We assume a smart contract is a trusted third party that maintains public state. All contracts and payment channels have a unique identifier on the blockchain. The underlying blockchain cannot be compromised and honest parties can always interact with the contracts within a designated grace period.

Threat model. We assume the adversary can control the order of messages, but all messages must be delivered within a designated grace period. The adversary can either corrupt all parties in the channel, $n-1$ parties in the channel and the custodian, the custodian's customer, or just the custodian. However the adversary cannot forge messages from non-corrupted parties.[7] If the custodian is corrupted, then we must assume the adversary is rational and only colludes with other parties in the customer's state channel if the payout is more than the custodian's security deposit. We assume there is an authenticated and secure end-to-end communication channel for parties within the state channel to prevent eavesdropping by a malicious custodian. This also implies there is no collusion between the custodian and a party in the channel, since otherwise it is trivial to eavesdrop on all states.

4.4 State channel modifications

We focus on modifications to the generic state channel construction presented in Section 3.1 and the interaction between parties in the channel to support authorising $\mathsf{hstate_i}$ instead of $\mathsf{state_i}$.

Modifications to the state channel contract. Figure 6 highlights several modifications to the state channel construction. SC.setstate accepts $\mathsf{hstate_i} = H(\mathsf{state_i}||\mathsf{r_i})$ as the latest state if it is authorised by all parties in the channel (instead of $\mathsf{state_i}$). The $\mathsf{state_i}$ (alongside the blinding nonce $\mathsf{r_i}$) is only revealed if one party triggers a dispute and resolves it using SC.resolve. Both modifications allows the custodian to settle disputes by publishing $\mathsf{hstate_i}$ (alongside a list of signatures from all parties in the channel) and this is further explored in Section 4.5. A third modification requires the contract to record the start time $\mathsf{t_{start}}$, settle time $\mathsf{t_{settle}}$ and the new state round stateRound for every successful dispute that performs a state transition using SC.resolve. This evidence is stored for later use by the customer to demonstrate that a custodian failed to settle a dispute on their behalf.

Exchanging collectively authorised state hashes. Figure 4 presents AuthState which is an interactive protocol between all parties to authorise a new state. To initiate a state transition, the initiator $\mathcal{P}_k$ signs a command cmd and separately signs the tuple $(\mathsf{hstate_i}, \mathsf{i}, \mathsf{SC})$. All other parties $\mathcal{P}_2, ..., \mathcal{P}_n$ must verify the state transition upon receiving both signed messages:

```
$\mathsf{VerifyTransition}(\mathcal{P}_k, \mathsf{cmd}, \sigma_k^{\mathsf{cmd}}, \mathsf{r_i}, \mathsf{i}, \mathsf{hstate_i}, \sigma_k^{\mathsf{hstate_i}}, \mathsf{r_{i-1}}, \mathsf{state_{i-1}}, \mathsf{SC})$
if $\mathcal{P}_k \notin \mathsf{SC}.\Sigma_{\mathcal{P}}$ return 0
$\mathsf{state_i} \leftarrow \mathsf{Transition}(\mathsf{state_{i-1}}, \mathsf{cmd})$ return 0
if $\mathsf{hstate_i} \neq \mathsf{H}(\mathsf{state_i}||\mathsf{r_i})$ return 0
set $\mathsf{hstate_{i-1}} := \mathsf{H}(\mathsf{state_{i-1}}||\mathsf{r_{i-1}})$
return $\mathsf{Sig.Verify}(\mathcal{P}_k, (\mathsf{cmd}, \mathsf{hstate_{i-1}}, \mathsf{i}-1, \mathsf{SC}), \sigma_k^{\mathsf{cmd}}) \wedge$
$\mathsf{Sig.Verify}(\mathcal{P}_k, \mathsf{hstate_i}, \mathsf{i}, \mathsf{SC}, \sigma_k^{\mathsf{hstate}})$
```

Briefly, VerifyTransition checks if $\mathsf{state_i}$ is indeed a valid transition from $\mathsf{state_{i-1}}$ using the command cmd and if this transition was authorised by the signer. Then it checks if $\mathsf{state_i}, \mathsf{r_i}$ correspond to $\mathsf{hstate_i}$ and that the signer has indeed authorised $\mathsf{hstate_i}$ (alongside the incremented i). If an honest party does not receive a signature from all other parties before LocalTimeout(), then they can enforce this state transition via the dispute process. To enforce, an honest party updates the contract with the previously authorised $\mathsf{hstate_{i-1}}$ using SC.setstate before initiating a dispute using

[7] This implies the digital signature scheme used in PISA is unforgeable under chosen message attacks (EUF-CMA) which is proven for Schnorr signatures [30]. However for the digital signature algorithm (DSA) such as ECDSA which is used in Ethereum, this is proven using non-standard assumptions [34].

Party K	**Party N**	**Custodian**
	All parties have a secret key sk and have access to SC, CC *via the network.* AuthState() *is defined in Figure 4 and* Appoint() *is defined in Figure 3.*	
	$\mathtt{SC.setup}(\mathcal{P}, \Delta_{\mathsf{settle}})$	
	Multiple states are authorised between $\mathcal{P}_k$,..., $\mathcal{P}_n$	
$\mathsf{AuthState}(\mathcal{P}_k, \mathcal{P}_n, \mathsf{state}_{\mathsf{i}-1})$		
$\mathsf{hstate_i}, \mathsf{i}, \Sigma_{\mathcal{P}}, \mathsf{state_i}, \mathsf{r_i} \leftarrow \mathcal{P}_k$	$\mathsf{hstate_i}, \mathsf{i}, \Sigma_{\mathcal{P}}, \mathsf{state_i}, \mathsf{r_i} \leftarrow \mathcal{P}_n$	
	$\mathsf{Appoint}(\Sigma_{\mathcal{P}}, (\mathsf{hstate_i}, \mathsf{i}), \mathsf{coins})$	
	$\mathsf{receipt}, \mathsf{s_i}, \sigma_C \leftarrow \mathcal{P}_n$	$\mathsf{pay}, \sigma_n \leftarrow C$
	Party N goes off-line	
$\mathtt{SC.setstate}(\mathsf{hstate}_{\mathsf{i}-1}, \mathsf{i}-1, \Sigma_{\mathcal{P}})$ $\sigma_k \leftarrow \mathsf{Sign}(\mathsf{sk}_k, (\mathsf{dispute}, \mathsf{hstate}_{\mathsf{i}-1}, \mathsf{i}-1, \mathtt{SC}))$ $\mathtt{SC.triggerdispute}(\sigma_k)$		
	We present two protocol outcomes based on whether the Custodian is honest or colludes with $\mathcal{P}_k$	
	Outcome 1: *Custodian resolves dispute on behalf of* $\mathcal{P}_n$	
		$\mathtt{SC.setstate}(\mathsf{hstate_i}, \mathsf{i}, \Sigma_{\mathcal{P}})$
	Outcome 2: *Custodian colludes with* $\mathcal{P}_k$ *and afterwards* $\mathcal{P}_n$ *seeks recourse.*	
Selects a cmd $\sigma_k^{\mathsf{cmd}} \leftarrow \mathsf{Sign}(\mathsf{sk}_k, (\mathsf{cmd}, \mathsf{hstate}_{\mathsf{i}-1}, \mathsf{i}-1, \mathtt{SC}))$ $\mathtt{SC.input}(\mathsf{cmd}, \sigma_k^{\mathsf{cmd}})$ $\mathsf{Wait}(\mathtt{SC}.\mathsf{t_{settle}})$ $\mathtt{SC.resolve}(\mathsf{state}_{\mathsf{i}-1}, \mathsf{r}_{\mathsf{i}-1})$		
	Party N returns online $b \leftarrow \mathsf{SeekRecourse}(\mathsf{receipt})$ if $b = 0$, abort $\mathtt{CC.recourse}(\mathsf{receipt.t_{start}}, \mathsf{receipt.t_{expire}}, \mathsf{receipt}.\mathtt{SC}, \mathsf{receipt.i}, \mathsf{receipt.h_i}, \mathsf{s_i}, \mathsf{receipt}.\sigma_C)$	

Figure 2: High level interaction for authorising states and if the customer needs to seek recourse.

SC.triggerdispute. This provides a grace period for all parties to input a command (including the initiator's signed cmd) before $\mathtt{SC}.\mathsf{t_{settle}}$. Once the dispute's deadline has passed, any party can call SC.resolve. This reveals $\mathsf{state_i}$, selects one cmd from the list of inputs (i.e. determined to the application's logic) and performs the state transition on-chain.

4.5 Custodian contract

The custodian contract CC allows a customer to hire the custodian to monitor their state channel. All payments are performed using a one-way (and off-chain) payment channel and the custodian is paid for every new $\mathsf{hstate_i}$ they are appointed to publish on the customer's behalf. Each payment incorporates a fair exchange protocol to ratify a signed receipt of appointment for the customer after the custodian is paid for accepting this task. The signed receipt alongside a list of recorded disputes in SC can later be used in CC to prove the custodian's wrongdoing if the custodian does not settle a dispute and prevent an execution fork. In the following, we provide an overview of the custodian contract which is presented in Figure 7. As before, we denote the contract as CC and interaction with the contract is denoted as a function e.g. CC.setup with the parameters omitted for readability.

Contract flags. There are four flags (⊥, OK, CHEATED, CLOSED). The flag ⊥ specifies that the contract is not initialised. To transition ⊥ → OK, the custodian must invoke SC.setup to set a list of timers and store a large security deposit. Customers can open payment channels while the contract's flag = OK by storing a deposit using SC.deposit. To withdraw the security deposit, the custodian must initiate a closing period SC.stopmonitoring which transitions from OK → CLOSED. It also enforces a time period (e.g. $\Delta_{\mathsf{withdraw}}$) for customers to seek recourse. The custodian withdraws their deposit using SC.withdraw when this time period has expired and if all payment channels are closed. Crucially the contract can transition from any flag to CHEATED if the customer can prove the custodian's wrongdoing using SC.recourse.

Contract establishment. To transition from ⊥ → OK requires the custodian to set the contract's timers $\Delta_{\mathsf{settle}}, \Delta_{\mathsf{withdraw}}$ and store a large security deposit using CC.setup. The first timer Δ_{settle} is used to determine the minimum time period for the custodian to respond and settle the customer's payment channel. The second timer Δ_{withdraw} is used to determine the minimum time period the custodian must wait before their security deposit can be withdrawn. Both timers cannot be retrospectively changed.

Customer's one-way payment channel. The custodian's contract maintains a list of one-way payment channels. Each payment channel has four flags ($\perp$, OK, DISPUTE, CLOSED). Flag $\perp$ implies the one-way payment channel has never existed and CLOSED implies the channel was previously opened. To open the channel and transition from $\perp \rightarrow$ OK or CLOSED $\rightarrow$ OK requires the customer to deposit coins using CC.deposit. For readability, we assume SC represents a new instance of the payment channel and it always has a unique identifier if it is closed and re-opened.

Due to the nature of a one-way payment channel, every new payment signed by the customer increments the number of coins sent to the custodian and only the custodian can mutually sign the final payment to close the channel using CC.setstate. However the customer can signal their desire to close the channel by initiating the dispute using CC.triggerdispute. The custodian must respond using CC.setstate to settle the dispute and redeem their share of the customer's deposit. Otherwise, after the grace period, the customer can return their full deposit using CC.resolve.

Fair exchange of signed receipt. Figure 3 presents our fair exchange protocol which provides the customer with a ratified signed receipt once the payment is accepted by the custodian. To initiate the fair exchange, the customer sends the authorised hstate (alongside all signatures $\Sigma_{\mathcal{P}}$), the channel identifier SC and the expected minimum time period for this appointment Δ_{expire}. We assume that the appointment time period Δ_{expire} chosen by the customer is larger than the payment channel's settlement time period CC.Δ_{settle}. This ensures the custodian must watch the channel for a reasonable period of time if the signed receipt's ratification is enforced on-chain by the customer. As well, the mutually agreed price for each appointment is omitted, but can be fixed or variable. Upon receiving a new appointment request, the custodian must locally verify whether they can settle a dispute on behalf of the customer using:

```
VerifyAppointment($\Sigma_{\mathcal{P}}$, SC, i, hstate, $\Delta_{\text{min}}$)
if i $\leq$ SC.stateRound return 0
if SC.$\Delta_{\text{settle}} \leq \Delta_{\text{min}}$ return 0
if SC.flag $\neq$ OK return 0
return Sig.Verify(SC.$\mathcal{P}$, (hstate, i, SC), $\Sigma_{\mathcal{P}}$)
```

Briefly this checks whether the authorised hstate_i was signed by all parties in SC and the dispute period in the customer's state channel is reasonable (i.e. greater than a minimum bound Δ_{min}). If the custodian is satisfied they can settle a dispute in the channel, then the custodian sends a signed receipt to the customer which includes the start time t_{start}, expiry time t_{expire}, the channel identifier SC and a conditional transfer hash h_i. This receipt is not yet ratified and cannot be used for recourse until the custodian reveals the corresponding pre-image s_i of h_i. The customer must verify the signed receipt can later be used, if necessary, to prove the custodian's wrong doing and it is indeed valid for CC after it is ratified:

```
VerifyReceipt(receipt, SC, CC, i, $\Delta_{\text{fresh}}$, $\Delta_{\text{expire}}$)
if receipt.SC $\neq$ SC return 0
if receipt.CC $\neq$ CC return 0
if receipt.$t_{\text{start}}$ − CurTime() < $\Delta_{\text{fresh}}$ return 0
if receipt.$t_{\text{expire}}$ − receipt.$t_{\text{start}}$ < $\Delta_{\text{expire}}$ return 0
if receipt.i $\neq$ i return 0
return Sig.Verify(C, (receipt.$t_{\text{start}}$, receipt.$t_{\text{expire}}$, SC,
                      CC, i, receipt.$h_i$), receipt.$\sigma_C$)
```

Briefly this checks whether the signed receipt's appointment start time receipt.t_{start} is close to present time Δ_{fresh} and its expiry time receipt.t_{expire} is in the future by at least Δ_{expire}. As well, the customer must check the receipt.i corresponds to the expected the counter for hstate_i and the receipt references both the customer's channel SC and the custodian's channel CC. Once satisfied, the customer initiates a conditional transfer which increases the number of coins sent to the custodian by an agreed price and this transfer is only valid if s_i is revealed. The custodian must locally verify the conditional transfer:

```
VerifyPayment(pay, coins, receipt.$h_i$, CC)
if pay.coins $\neq$ coins return 0
if pay.$h_i \neq$ receipt.$h_i$ return 0
if pay.CC $\neq$ CC return 0
if pay.coins $\leq$ CC.ID[$\mathcal{P}_k$].deposit return 0
return Sig.Verify($\mathcal{P}_k$, (pay.$h_i$, pay.coins, pay.CC), pay.$\sigma_k$)
```

Briefly this checks if the payment is transferring the agreed price, if the conditional transfer hash h_i corresponds to receipt.h_i and if the customer has a sufficient deposit to cover the payment. If satisfied, the custodian sends the customer s_i which ratifies the signed receipt and completes the payment. Otherwise if the customer does not receive s_i before a local timeout, then the customer initiates a dispute using CC.triggerdispute to enforce the receipt's ratification within the time period CC.Δ_{settle}. To claim the payment (and ratify the signed receipt), the custodian must reveal the conditional transfer and the corresponding s_i using CC.setstate. This settles the dispute and returns both parties their share of the deposit. If the dispute expires and s_i is not revealed, then customer's full deposit is returned using CC.resolve and the signed receipt is never ratified.

Seeking Recourse. The customer can check whether the custodian failed to settle a dispute on their behalf:

```
SeekRecourse(receipt)
SC := receipt.SC
for k in SC.disputelist.length
  if SC.disputelist[k].$t_{\text{start}}$ > receipt.$t_{\text{start}}$ $\wedge$
    receipt.$t_{\text{expire}}$ > SC.disputelist[k].$t_{\text{expire}}$ $\wedge$
    receipt.i $\geq$ SC.disputelist[k].stateRound
      return 1
return 0
```

Briefly this checks whether there was a successful dispute while the custodian was expected to monitor the channel. If there is a dispute such that receipt.i $\geq$ SC.stateRound, then an execution fork was performed. The customer can seek recourse by calling CC.recourse with the signed receipt. This requires CC to fetch the list of disputes recorded in SC and verify there is a recorded dispute between receipt.start and receipt.expire. If there is a dispute which satisfies receipt.i $\geq$ SC.stateRound, then the custodian's contract

Customer		**Custodian**
Both parties have a signing key sk, the appointment cost coins and can access SC, CC *via the network.*		
Customer has $\mathsf{hstate_i}, \mathsf{i}, \Delta_{\mathsf{fresh}}, \Delta_{\mathsf{settle}}$ *and the Custodian has* Δ_{min}.		
$\sigma_k \leftarrow \mathsf{Sign}(sk_k, (\mathsf{coins}, \mathsf{CC}))$		
$\mathsf{CC.deposit}(\mathsf{coins}, \sigma_k)$		
	$\xrightarrow{\mathsf{hstate_i}, \mathsf{i}, \mathsf{SC}, \Sigma_{\mathcal{P}}, \Delta_{\mathsf{expire}}}$	
		$b \leftarrow \mathsf{VerifyAppointment}(\Sigma_{\mathcal{P}}, \mathsf{SC}, \mathsf{i}, \mathsf{hstate_i}, \Delta_{\mathsf{min}})$
		if b=0, abort
		$\mathsf{s_i} \overset{r}{\leftarrow} R, \mathsf{h_i} \leftarrow \mathsf{H}(\mathsf{s_i})$
		$\mathsf{t_{start}} \leftarrow \mathsf{CurTime}(), \mathsf{t_{expire}} \leftarrow \mathsf{t_{start}} + \Delta_{\mathsf{expire}}$
		$\sigma_{\mathsf{C}} \leftarrow \mathsf{Sign}(sk_{\mathsf{C}}, (\mathsf{t_{start}}, \mathsf{t_{expire}}, \mathsf{SC}, \mathsf{CC}, \mathsf{i}, \mathsf{h_i})$
		$\mathsf{receipt} \leftarrow (\sigma_{\mathsf{C}}, \mathsf{t_{start}}, \mathsf{t_{expire}}, \mathsf{SC}, \mathsf{CC}, \mathsf{i}, \mathsf{h_i})$
	$\xleftarrow{\mathsf{receipt}}$	
$b \leftarrow \mathsf{VerifyReceipt}(\mathsf{receipt}, \mathsf{SC}, \mathsf{CC}, \mathsf{i}, \Delta_{\mathsf{fresh}}, \Delta_{\mathsf{expire}})$		
if b=0, abort		
$\sigma_k \leftarrow \mathsf{Sign}(sk_k, (\mathsf{receipt.h_i}, \mathsf{coins}, \mathsf{CC}))$		
$\mathsf{pay} \leftarrow (\sigma_k, \mathsf{receipt.h_i}, \mathsf{coins})$		
	$\xrightarrow{\mathsf{pay}}$	
		$b \leftarrow \mathsf{VerifyPayment}(\mathsf{pay}, \mathsf{coins}, \mathsf{receipt.h_i}, \mathsf{CC})$
		if b=0 return 0
We present three protocol outcomes based on depending on the Custodian's response.		
Outcome 1: *Custodian accepts payment within a timely manner*		
$b \leftarrow \mathsf{h_i} = \mathsf{H}(\mathsf{s_i})$	$\xleftarrow{\mathsf{s_i}}$	
if b=0, move to outcome 2 or 3		
Outcome 2: *Custodian does not reveal s_i privately to the customer, but redeems on-chain*		
LocalTimeout()		
$\sigma_k \leftarrow \mathsf{Sign}(sk_k, (\mathsf{close}, \mathsf{CC}))$		
$\mathsf{CC.triggerdispute}(\sigma_k)$		
		$\sigma_{\mathsf{C}} \leftarrow \mathsf{Sign}(sk_{\mathsf{C}}, (\mathsf{h_i}, \mathsf{coins}, \mathsf{CC}))$
		$\mathsf{CC.setstate}(\mathsf{h_i}, \mathsf{coins}, \mathsf{s_i}, \mathsf{pay}.\sigma_k, \sigma_{\mathsf{C}})$
Outcome 3: *Custodian does not reveal s_i privately to the customer and the custodian does not redeem the conditional transfer*		
LocalTimeout()		
$\sigma_k \leftarrow \mathsf{Sign}(sk_k, (\mathsf{close}, \mathsf{CC}))$		
$\mathsf{CC.triggerdispute}(\sigma_k)$		
$\mathsf{Wait}(\mathsf{CC.t_{settle}})$		
$\mathsf{CC.resolve}(\mathcal{P}_k)$		

Figure 3: Appoint: A fair exchange protocol that ensures the Monitor is paid upon validating the customer's receipt.

is marked as cheating CC.flag = CHEATED. This forfeits the custodian's security deposit and allows all customers to immediately close remaining payment channels and withdraw their deposit.

Closing contract. The custodian can signal its desire to stop accepting new customers using CC.stopmonitoring which transitions the flag from OK → CLOSED. The large security deposit can be returned when all payment channels are closed and the grace period $\Delta_{\mathsf{withdraw}}$ has expired. It is crucial the expiry time for a customer's signed receipt is never greater than the withdrawal time such that $\mathsf{t_{withdraw}} > \mathsf{t_{expire}}$ otherwise the custodian's deposit can be returned before the receipt expires.

```
Party K                                                  Party N
    Each party has a signing key sk, a local copy of the previously authorised state
    $\mathsf{hstate}_{i-1}, \mathsf{state}_{i-1}, \mathsf{r}_{i-1}, i-1, \Sigma_{\mathcal{P}}$ and can access SC via the network
Selects a cmd
$\mathsf{state}_i \leftarrow \mathsf{Transition}(\mathsf{state}_{i-1}, \mathsf{cmd})$
$\mathsf{r}_i \xleftarrow{r} R$
$\mathsf{i} \leftarrow \mathsf{i} + 1$
$\mathsf{hstate}_i \leftarrow \mathsf{H}(\mathsf{state}_i || \mathsf{r}_i)$
$\sigma_k^{\mathsf{hstate}} \leftarrow \mathsf{Sign}(\mathsf{sk}_k, (\mathsf{hstate}_i, \mathsf{i}, \mathsf{SC}))$
$\sigma_k^{cmd} \leftarrow \mathsf{Sign}(\mathsf{sk}_k, (\mathsf{cmd}, \mathsf{hstate}_{i-1}, \mathsf{i}-1, \mathsf{SC}))$

$\xrightarrow{(\mathsf{hstate}_i, \mathsf{i}, \sigma_k^{\mathsf{hstate}}), (\mathsf{cmd}, \sigma_k^{\mathsf{cmd}}), \mathsf{state}_i, \mathsf{r}_i}$
                                                         $b \leftarrow \mathsf{VerifyTransition}(\mathcal{P}_k, \mathsf{cmd}, \sigma_k^{\mathsf{cmd}}, \mathsf{r}_i, \mathsf{i}, \mathsf{hstate}_i,$
                                                             $\sigma_k^{\mathsf{hstate}_i}, \mathsf{state}_{i-1}, \mathsf{r}_{i-1}, \mathsf{SC})$
                                                         if b=0, abort

    We present two protocol outcomes based on Party N's response.
----------------------------------------------------------------------
    Outcome 1: $\mathcal{P}_n$ co-operates and authorises the state transition

                                                         $\sigma_n \leftarrow \mathsf{Sign}(sk_n, (\mathsf{hstate}_i, \mathsf{i}, \mathsf{SC}))$
                                                         $\xleftarrow{\sigma_n}$
$b \leftarrow \mathsf{Sig.Verify}(\mathcal{P}_n, (\mathsf{hstate}_i, \mathsf{i}, \mathsf{SC}), \sigma_n)$
if b=0, abort
----------------------------------------------------------------------
    Outcome 2: $\mathcal{P}_n$ aborts and does not authorise the state transition

LocalTimeout()
SC.setstate($\mathsf{hstate}_{i-1}, \mathsf{i}-1, \Sigma_{\mathcal{P}}$)

$\sigma_k \leftarrow \mathsf{Sign}(sk_k, (\mathsf{dispute}, \mathsf{hstate}_{i-1}, \mathsf{i}-1, \mathsf{SC}))$
SC.triggerdispute($\sigma_k$)
SC.input(cmd, $\sigma_k^{cmd}$)

Wait(SC.$\mathsf{t}_{\mathsf{settle}}$)
SC.resolve($\mathsf{state}_{i-1}, \mathsf{r}_{i-1}$)
```

Figure 4: AuthState: Two (out of N) parties authorising a state.

5 SECURITY ANALYSIS

We focus on the protocol's goals outlined in Section 4.1 based on the assumptions presented in Section 4.3.

5.1 State Privacy

Our goal is to protect the privacy of all intermediary states the custodian is appointed to publish if there is a dispute and we consider a malicious custodian who cannot corrupt parties in SC. The custodian receives hstate_i during the interactive Appoint protocol and the pre-image of hstate_i includes a nonce r_i such that $\mathsf{H}(\mathsf{state}_i||\mathsf{r}_i)$. This nonce prevents the custodian simply brute-forcing hstate_i to learn state_i and thus we rely on the pre-image resistance property of a cryptographic hash function. There are two situations when the nonce r_i is revealed. First r_i is revealed during the interactive AuthState protocol, but we previously assumed a malicious custodian cannot eavesdrop on this communication channel. Second r_i is globally revealed (alongside state_i) in SC to complete the dispute process by SC.resolve. As mentioned in the state privacy goal, the custodian learns if a globally revealed state_i corresponds to a hstate_i in which they were appointed to publish.

5.2 Fair Exchange

We consider the case of a malicious customer. After receiving a signed receipt from the custodian, the customer can propose a malformed conditional transfer which is not applicable for CC by including a different hash than supplied by the custodian such that $\mathsf{pay.h}_i \neq \mathsf{receipt.h}_i$ or reference a different contract such that pay.CC $\neq$ CC. As well, the customer can propose a conditional transfer which underpays the custodian, but can be redeemed using CC.setstate. The custodian locally runs VerifyPayment to check the above conditions to ensure the conditional transfer is well-formed, fulfills the payment and can be accepted by CC.setstate. Otherwise the custodian does not reveal s_i and the signed receipt is not ratified.

Step	Command	Cost (gas)	Cost ($)
	Create Contracts		
1	Custodian contract	1,609,613	2.53
2	State contract	1,892,135	2.97
3	Sprites contract	869,280	1.37
	Appoint Custodian		
4	Customer opens payment channel (CC.deposit)	65,504	0.10
5	Customer signals payment channel closure (CC.triggerdispute)	48,763	0.08
6	Customer is refunded (CC.resolve)	37,290	0.06
7	Custodian closes payment channel (CC.setstate)	103,794	0.16
8	Customer seeks recourse (CC.recourse)	51,892	0.08
	State Channel Dispute Process		
9	Party set collectively authorised state (SC.setstate)	90,130	0.14
10	Party initiates dispute process (SC.triggerdispute)	78,667	0.12
11	Party submit command (SC.input)	140,275	0.22
12	Party resolves dispute and state transition is executed on-chain (SC.resolve)	149,494	0.23

Table 1: Cost of Pisa for simplified Sprites. We have approximated the cost in USD ($) using the conversion rate of 1 ether = $785.31 and the gas price of 2 Gwei which reflects the real world costs in May 2018.

Next we consider the case of a malicious custodian. Once the custodian has received pay, the custodian can simply not reveal s_i and wait until receipt.t_{expire} for the signed receipt to expire. To prevent stalling, the customer performs a local time-out before triggering the dispute process in CC and Figure 3 illustrates two outcomes for the fair exchange. The custodian can mutually sign pay and reveal s_i before CC.t_{settle} in order to claim the payment. By claiming the payment in CC, however, the signed receipt is atomically ratified as s_i is publicly revealed. Otherwise the customer is fully refunded via the dispute process and the custodian can no longer accept the payment. It is crucial the receipt's expiry time receipt.t_{expire} is significantly greater than CC.t_{settle} to ensure the receipt can be ratified via the dispute process. As mentioned previously we assumed the customer has chosen a Δ_{expire} which is significantly greater than CC.Δ_{settle}.

Finally we consider whether the custodian can steal the customer's deposit from the one-way payment channel. The nature of a replace-by-incentive and one-way channel, however, is that all payments must be authorised by both the sender and receiver. Therefore the custodian cannot independently authorise a payment which steals the customer's full deposit.

5.3 Non-frameability

We consider if all parties in the SC collude to appoint the custodian and later prevent the custodian settling a dispute using SC.setstate. If successful, this records a dispute in SC and alongside the corresponding signed receipt, the colluding parties can satisfy CC.recourse. The custodian receives hstate$_i$, i, $\Sigma_{\mathcal{P}}$ from the customer which is necessary to settle any future dispute and this is verified using VerifyAppointment. To interfere with the dispute process, the cartel must modify the state of SC such that hstate$_i$, i, $\Sigma_{\mathcal{P}}$ is no longer valid for settling a dispute. The cartel can use SC.setstate to update the contract such that SC.stateRound $\geq$ receipt.i, but this does not record a dispute in the channel and it also implies the custodian is no longer required to settle a dispute. The contract will always accept hstate$_i$ as it is simply a random string of bytes. The only option for the adversary is to register a new party to SC such that the custodian will not have a signature from the newly registered party to settle a future dispute. Parties can only be registered using SC.resolve after the dispute process is complete as this is the only time the state is revealed and a registration command can be processed. The custodian can settle a malicious registration if it results in an execution fork and thus parties cannot register a new party to interfere with $\Sigma_{\mathcal{P}}$.

5.4 Recourse as a financial deterrent

In the fair exchange analysis, it was shown the customer pays for a signed receipt only if it is indeed valid and can be used to seek recourse. However if the customer is off-line, then a malicious custodian can collude with the channel's remaining $n - 1$ parties simply by not settling a dispute and permitting an execution fork in the state channel. The custodian should thus be deterred both from invalidating the customer's signed receipt, and from receiving a payout that is greater than the loss of the security deposit.

The former follows on from the fair-exchange analysis as the customer already has the corresponding s_i and the signed receipt is ratified. To re-iterate, a receipt contains the monotonic counter i, the start and expire time of the appointment t_{start}, t_{expire}, the receipt hash h_i and of course both contracts SC, CC. The timestamp for any execution forks (or disputes) is enforced by SC and thus the signed receipt is always valid between t_{start} and t_{expire}. The only remaining option to invalidate the signed receipt is to record a larger counter SC.stateRound in the contract. However, the dispute process strictly increments the counter by one and thus setting a dispute based on a previously authorised state (i.e. to perform an execution fork) will always have a counter which is less than (or equal to) the receipt. Since the other $n - 1$ parties require the customer's assistance to authorise a new hstate$_i$ with a larger counter than the receipt's counter, the safety of the customer is therefore guaranteed.

The latter deters a dishonest custodian if the outstanding potential payout for cheating is less than the pre-agreed collateral backing. The custodian's potential pay-off for cheating is not easily measurable, since a single customer can appoint the custodian to watch two or more channels, and an appointment is publicly disclosed only if there is a dispute via the global blockchain (or the customer seeks recourse). In Appendix C, we extend Pisa to reserve coins from the security deposit for each customer, parametrized according to a leveraged ratio for discounted service fees. This extension of explicit coin allocation can be used to compensate customers for their loss if the custodian cheats, making the service more attractive relative to burning of the security deposit. However, compensation can also be less secure than burning in several respects. In particular, it may allow the custodian to minimise the loss of their security deposit by compensating a Sybil account.

5.5 Efficiency requirements

We consider if all parties in $\mathtt{SC}$ collude to outsource multiple appointments to the custodian for a single state channel such that the storage requirement is not $O(1)$. The algorithm $\mathsf{VerifyAppointment}$ run by the custodian ensures the counter i is incremented for every new appointment. Furthermore $\mathtt{SC.setstate}$ will settle the dispute as long as the corresponding counter i for $\mathsf{hstate_i}$ is the largest received so far. Thus the custodian only needs to store the $\mathsf{hstate_i}$ (alongside a valid signature for all parties in $\mathtt{SC}$) which is associated with the largest monotonic counter i, although the custodian is required to keep track of the designated time period for each appointment which can potentially require $O(N)$ storage. The custodian's local policy can dictate the number of unique time periods stored and whether each new appointment simply extends the expiry time.

6 PROOF OF CONCEPT IMPLEMENTATION

We present a proof of concept implementation for Pisa based on a simplified version of Sprites [25] to evaluate whether it is gas-efficient to deploy. In the following, we provide a high-level overview of the experiment before evaluating its cost.

Our experiment involves three contracts.[8] Both the state channel contract and the custodian contract are implemented as illustrated in Figure 6 and 7. The third contract is a simplified version of Sprites which stores the full state and it has the `transition` function. This function can only be called by the state channel contract and it is the only function that can modify the state (i.e. execute a state transition). Separating the state channel and the `transition` function into distinct contracts is beneficial as the custodian is only required to audit the state channel contract[9]. In terms of functionality, simplified Sprites only supports withdrawing coins or sending payments.

Table 1 presents the cost of our experiments for Pisa on a private Ethereum network. Steps 1-3 highlight the one-time costs for creating each contract. For the custodian contract, the customer creates a payment channel with the custodian in step 4 and closes the payment channel via the dispute process in steps 5-6. Intermediary (and off-chain) payments between the customer and custodian do not incur any cost. The custodian can close the channel with a mutually authorised payment and both parties are sent their respective share of the deposit in step 7. If the custodian aborts and fails to settle a dispute on the customer's behalf, then step 8 represents the customer seeking recourse and proving the custodian's wrongdoing using a ratified signed receipt. For the state channel contract, step 9 represents updating the contract with the latest authorised state hash. This cost is constant for any party, including when the custodian settles a dispute on the customer's behalf. Steps 10-12 represents a successful dispute as one party triggers the dispute, one party submits a command and one party resolves the dispute. When the dispute is resolved, the `transition` function is invoked in the simplified Sprites contract.

7 DISCUSSION AND FUTURE WORK

Persistent dispute evidence. The Ethereum community are seeking proposals to charge rental fees [35] and expire instantiated contracts (alongside stored data). This is problematic for Pisa as the state channel contract cannot be immediately destroyed if there is a dispute in order to preserve evidence and prove the custodian's wrongdoing. In Appendix A we present another approach using a new contract called the dispute registry. This contract is responsible for storing disputes on behalf of all other contracts and periodically clearing dispute records (e.g. after one week).

Customer crash recovery. It was reported in March 2018 [3] that one party tried to close a replace-by-revocation channel using a previous (and revoked) state, but the counter-party was on-line. Due to the nature of a replace-by-revocation channel, the counter-party was awarded all coins in the channel. At the time, it was implied that this was an attempt to reverse payments and steal the counter-party's coins. However it later emerged that the party had allegedly crashed and lost a copy of the latest state. As a result, their wallet software used the revoked state to close the channel. One desirable feature for a custodian is to support crash recovery by storing an encryption of the state which can later be retrieved and decrypted by the customer. In Appendix B, we propose how to encrypt the state such that it is gas-efficient and compatible with the Ethereum network.

8 CONCLUSION

In this paper, we proposed Pisa to help address a new (and undesirable) security assumption for state channels by allowing a customer to appoint a third party called the custodian to monitor a state channel on their behalf. Pisa is designed to support any application built using a state channel and it is the first protocol that allows the customer to seek recourse if the custodian's fails to settle a dispute on their behalf. To evaluate Pisa, we provided a proof of concept implementation for Sprites to demonstrate that it is cost-efficient to deploy in practice. We hope Pisa provides a new step towards the realisation of state channels as a practical scaling solution for cryptocurrencies.

9 ACKNOWLEDGEMENTS

Patrick McCorry and Sarah Meiklejohn are supported in part by EPSRC grant EP/N028104/1. Andrew Miller is partially supported by gifts from Jump Labs and from CME Group. We would like to thank Lefteris Karapetsas for bringing the problem to our attention and the wider Raiden team for feedback during the development of Pisa. Finally we thank the Ethereum Foundation for funding to support bringing Pisa from research to practice.

[8] Code: https://www.dropbox.com/s/1ls37omfk8bb7vf/custodian.zip?dl=0

[9] In practice, this allows all state channels to have the same bytecode regardless of their application and thus it is straight-forward for the custodian to audit.

REFERENCES

[1] An and Bellare. Does encryption with redundancy provide authenticity? In *EUROCRYPT: Advances in Cryptology: Proceedings of EUROCRYPT*, 2001.

[2] Iddo Bentov, Ranjit Kumaresan, and Andrew Miller. Instantaneous decentralized poker. In *Asiacrypt*, 2017.

[3] bitbug42. Hackers tried to steal funds from a Lightning channel, just to end up losing theirs as the penalty system worked as expected. mar 2018.

[4] Conrad Burchert, Christian Decker, and Roger Wattenhofer. Scalable funding of bitcoin micropayment channel networks. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 361–377. Springer, 2017.

[5] Alessandro Chiesa, Matthew Green, Jingcheng Liu, Peihan Miao, Ian Miers, and Pratyush Mishra. Decentralized anonymous micropayments. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 609–642. Springer, 2017.

[6] Loredana Cirstea. Monitoring service: on-chain rewards proposal, 2018. https://github.com/raiden-network/spec/issues/46.

[7] Jeff Coleman. http://www.jeffcoleman.ca/state-channels/, November 2015.

[8] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, and Emin Gün. On scaling decentralized blockchains. In *Proc. 3rd Workshop on Bitcoin and Blockchain Research*, 2016.

[9] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Stabilization, Safety, and Security of Distributed Systems*, pages 3–18. Springer, 2015.

[10] Thaddeus Dryja. Unlinkable outsourced channel monitoring, 2016. https://youtu.be/Gzg_u9gHc5Q?t=2875.

[11] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment channels over cryptographic currencies. *IACR Cryptology ePrint Archive*, 2017:635, 2017.

[12] Stefan Dziembowski, Sebastian Faust, and Kristina Hostakova. Foundations of state channel networks. *IACR Cryptology ePrint Archive*, 2018:320, 2018.

[13] Oded Goldreich. *Foundations of Cryptography: Basic Tools*, volume 1. Cambridge University Press, 2001.

[14] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 473–489. ACM, 2017.

[15] Garrett Hardin. The tragedy of the commons. In *Science 162*, pages 1243–1248, 1968.

[16] Alyssa Hertig. Bitcoin Lightning Fraud? Laolu Is Building a 'Watchtower' to Fight It. *Coindesk*, February 2018. https://www.coindesk.com/laolu-building-watchtower-fight-bitcoin-lightning-fraud/.

[17] Rami Khalil and Arthur Gervais. Revive: Rebalancing off-blockchain payment networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 439–453. ACM, 2017.

[18] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 839–858. IEEE, 2016.

[19] Joshua Lind, Ittay Eyal, Florian Kelbert, Oded Naor, Peter R. Pietzuch, and Emin Gün Sirer. Teechain: Scalable blockchain payments using trusted execution environments. *CoRR*, abs/1707.05454, 2017.

[20] Loi Luu. Bringing bitcoin to ethereum. April 2018. https://blog.kyber.network/bringing-bitcoin-to-ethereum-7bf29db88b9a.

[21] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 455–471. ACM, 2017.

[22] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback protection for trusted execution, 2017. http://eprint.iacr.org/2017/048.

[23] Patrick McCorry, Ethan Heilman, and Andrew Miller. Atomically trading with roger: Gambling on the success of a hardfork. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 334–353. Springer, 2017.

[24] 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.

[25] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. Sprites: Payment channels that go faster than lightning. *CoRR*, abs/1702.05812, 2017.

[26] Rachel Rose O'Leary. Blockchain bloat: How ethereum is tackling storage issues. January 2018. https://www.coindesk.com/blockchain-bloat-ethereum-clients-tackling-storage-issues/.

[27] Olaoluwa Osuntokun. Hardening Lightning. *BPASE*, February 2018. https://cyber.stanford.edu/sites/default/files/hardening_lightning_updated.pdf.

[28] Rafael Pass and abhi shelat. A course in cryptography, 2007. http://www.cs.cornell.edu/courses/cs4830/2010fa/lecnotes.pdf.

[29] Rafael Pass et al. Micropayments for decentralized currencies. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 207–218. ACM, 2015.

[30] David Pointcheval and Jacques Stern. Security proofs for signature schemes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 387–398. Springer, 1996.

[31] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. *draft version 0.5*, 9:14, 2016.

[32] Tim Ruffing, Aniket Kate, and Dominique Schröder. Liar, liar, coins on fire!: Penalizing equivocation by loss of bitcoins. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 219–230. ACM, 2015.

[33] Raoul Strackx and Frank Piessens. Ariadne: A minimal approach to state continuity. In *25th USENIX Security*, 2016.

[34] Serge Vaudenay. The security of dsa and ecdsa. In *International Workshop on Public Key Cryptography*, pages 309–323. Springer, 2003.

[35] Vitalik Buterin. A simple and principled way to compute rent fees. March 2018. https://ethresear.ch/t/a-simple-and-principled-way-to-compute-rent-fees/1455.

Step	Command	Cost (Gas)	Cost ($)
1	Create DisputeRegistry Contract	1,031,411	
2	Set first dispute	465,606	$0.60
3	Set second dispute	96,796	$0.13
4	Set third dispute	96,796	$0.13
5	Test first dispute record	29,795	$0.04
6	Test second dispute record	30,639	$0.04
7	Test third dispute record	31,483	$0.04
8	Re-creating a record contract	355,583	$0.46

Table 2: Cost of setting and checking disputes in the Dispute Registry. We have approximated the cost in USD ($) using the conversion rate of 1 ether = $785.31 and the gas price of 2 Gwei which reflects the real world costs in May 2018.

A DISPUTE REGISTRY

The latest state of all instantiated contracts (including contracts with little to no activity) must be available in memory for the software to validate future blocks. This directly hinders the network's scalability as it increases the computational resources required to verify blocks in real-time. The community are seeking mechanisms to expire instantiated contracts (and its associated data) in order to reduce this overhead. For example, one proposal is to charge a rental fee for instantiated contracts and automatically delete the contract once its rent expires [35]. This is potentially problematic for Pisa as the state channel contract cannot be immediately destroyed without the authorisation of all parties to ensure the dispute records are available for when the customer returns on-line. To overcome this issue and prevent long-lived contracts, we propose a new contract called the dispute registry denoted as `DR`.

This dispute registry is responsible storing dispute records on behalf of other contracts and a record includes the contract's address `SC`, the start time $\mathsf{t}_{\mathsf{start}}$, settlement time $\mathsf{t}_{\mathsf{settle}}$ and the new monotonic counter stateRound. Other contracts can update the registry with a dispute record using `DR.setdispute`. When the customer seeks recourse, the custodian contract can verify the signed receipt is ratified (and signed by the custodian) before checking for recorded disputes using `DR.checkdispute` to verify the custodian's wrongdoing. All dispute records are organised by the day of submission and any interaction with the dispute registry will attempt to delete expired records (e.g. over 1 week old).

To evaluate whether a dispute registry is gas-efficient, we built a proof of concept implementation and the experiment's result are presented in Table 2. In this experiment, all disputes are associated with a day of the week and the disputes are deleted exactly one week later. Due to limitations in Ethereum for deleting records, the dispute registry must create a new contract for each day of the week. For every new day, the first invocation of the dispute registry will destroy the existing contract (i.e. delete last week's records for this day) and re-create the contract for storing new records which is highlighted in Step 8. Otherwise there is a constant gas cost of 96,796 for inserting a new dispute record. This is reflected in Steps 2-4 where we included three dispute records for a single day. When checking for an execution fork, the dispute registry looks up records for a given day, filters this list to only fetch disputes associated with `SC` and iterates over this list to verify if there was indeed an execution fork. Steps 5-7 highlights that each iteration in the list of disputes requires an additional 850 gas. Finally deleting and re-creating the contract which stores dispute records for a given day (i.e. an occurrence every day) is highlighted in Step 8.

B ENCRYPTING THE STATE

In Pisa we proposed that parties within the channel can authorise the hash of a state hstate. However, there is an advantage for parties to authorise an encryption of the state instead of a commitment. This allows parties to retrieve the state solely from the custodian's data, while a hash commitment cannot (since it is compressing). For example, if $\mathcal{P}_1$ has received all coins in the channel but lost the corresponding authorised state, then $\mathcal{P}_1$ can retrieve the encrypted state from custodian (either directly or via the data that custodian submits to the contract in a dispute), and decrypt it. In particular, such circumstances can arise when $\mathcal{P}_1$ switches between devices while participating in a state channel: if $\mathcal{P}_1$ received the latest authorised state on a mobile phone, and later wishes to continue to participate using a laptop (perhaps the phone became inaccessible), then $\mathcal{P}_1$ can request the encrypted state from the custodian.

To this end, we propose that all parties collectively authorise a secret master seed to deterministically compute private keys for every new state. This allows each party to backup *only* the master seed, and thus be able to reconstruct an encrypted state via only a counter i and the secret master seed. This seed can be determined by electing a single party to propose a seed or requiring the parties to collectively compute a seed using a multi-party protocol based on commit/reveal. No state transitions should occur until all parties have acknowledged the seed and this must be repeated if a party joins the channel at a later time.

Our proposed seeded encryption scheme $\mathcal{E}_{\text{seeded}}$ combines a computationally hiding commitment scheme and a semantically secure encryption scheme. It works as follows:

(1) $sk_i \leftarrow \mathsf{H}(\mathsf{seed}, 0, i), r_i \leftarrow \mathsf{H}(\mathsf{seed}, 1, i)$
(2) $(C_i, d_i) \leftarrow \mathsf{Com}(sk_i, r_i)$
(3) $E_i \leftarrow \mathsf{Enc}_{sk_i}(\mathsf{state}_i)$
(4) $\mathsf{encstate_i} \leftarrow (C_i, E_i, i)$

We assume that H is a pseudorandom function [13, 28], meaning that its outputs are computationally indistinguishable from that of a random oracle. In step (1) we compute the secret key sk_i and the nonce $\mathsf{r_i}$ using the seed. In step (2), the commitment procedure $(c, d) \leftarrow \mathsf{Com}(m, r)$ takes message m and randomness r and outputs commitment c and opening d. Hence, the contract will accept sk_i as valid if and only if the decommitment verification procedure $\mathsf{Decommit}(C_i, (\mathsf{sk}_i, d_i))$ returns 1. In most commitment schemes it is the case that $d = r$, i.e., the opening is the randomness that is fed into the commitment algorithm. In step (3), the encryption procedure $E_i \leftarrow \mathsf{Enc}_{sk_i}(\mathsf{state_i})$ takes the secret key sk_i and $\mathsf{state_i}$, and outputs the encrypted state E_i. Finally in step (4), we denote the final encrypted state as $\mathsf{encstate_i}$ which includes the commitment, encryption and the monotonic counter i.

The following lemma shows that $\mathcal{E}_{\text{seeded}}$ is privacy-preserving and the proof follows from a basic argument: any efficient computation that is done on random input cannot be distinguished from the same computation on pseudorandom input.

Lemma B.1. *If the commitment scheme* Com *is computationally hiding, the symmetric encryption scheme* Enc *is semantically secure, and the hash function is pseudorandom, then the seeded encryption scheme $\mathcal{E}_{seeded}$ is secure.*

Proof sketch. By assumption, Com is computationally hiding and Enc is semantically secure. This implies that, if sk_i and r_i are uniformly random, then no adversary can distinguish the case in which they are given uniformly random X_i and Y_i in place of the honest values C_i and E_i. Now, if there exists some probabilistic polynomial-time (PPT) adversary $\mathcal{A}_1$ that can break the seeded encryption scheme, we show how to use it to construct a PPT adversary $\mathcal{A}_2$ that breaks the pseudorandomness of the hash function. On input u_i and w_i, the goal of $\mathcal{A}_2$ is to decide if they are uniformly random, or if they are of the form $H(\text{seed}, b, i)$ for some i and for $b \in \{0, 1\}$. $\mathcal{A}_2$ honestly computes $(C_i, D_i) \leftarrow \mathsf{Com}(u_i, w_i)$ and $E_i \leftarrow \mathsf{Enc}_{u_i}(\mathsf{state}_i)$ for some arbitrary state. It then invokes $\mathcal{A}_1$ with these as input, and $\mathcal{A}_1$ returns a bit b. It then simulates $\mathcal{A}_1$ on uniformly random inputs X_i and Y_i, and again $\mathcal{A}_1$ returns a bit b'. $\mathcal{A}_2$ then outputs its guess bit as $b \oplus b'$.

It is clear that $\mathcal{A}_2$ is a PPT algorithm, as it must only pick a state, run the commitment and encryption protocols, and simulate the PPT $\mathcal{A}_1$. When the input for $\mathcal{A}_2$ is uniformly random, we know that b and b' differ with only negligible probability, meaning $\mathcal{A}_2$ outputs 0 (as expected). On the other hand, when its input is generated using the hash function, the assumption on $\mathcal{A}_1$ implies $b = 1$ with non-negligible probability. Hence, $\mathcal{A}_2$ breaks pseudorandomness. □

Let us clarify that $\mathcal{E}_{\text{seeded}}$ uses symmetric encryption in order to support a state S_i whose size is arbitrarily large, as otherwise $E_i \leftarrow S_i \oplus sk_i$ is enough. Since $\mathcal{E}_{\text{seeded}}$ is already stateful, a generic realization of the symmetric encryption component for arbitrarily large S_i is $E_i \leftarrow S_i \oplus \mathsf{PRNG}(sk_i)$, where $\mathsf{PRNG}(\cdot)$ is a pseudorandom generator. Multiparty computation protocols may indeed a large off-chain state (e.g., poker [2]), whereas ordinary payment channels can be accomplished with a small state size (cf. Appendix B.1).

The reason for the commitment C_i is that the binding property of the commitment scheme implies that it is infeasible for a malicious party to steal funds by supplying the contract with $sk'_i \neq sk_i$ and D'_i that satisfy $\mathsf{Dec}(C_i, (sk'_i, D'_i)) = 1$. Note, however, that using a hash commitment $C_i = \mathsf{H}(sk_i)$ without the extra r_i value is semantically secure in our setting. This is because the sk_i keys are derived from the incremental inputs (seed, i), which implies that $sk_i \neq sk_{i+1}$ except with negligible probability. In Appendix B.1 we describe an even more efficient scheme.

Ethereum does not natively support symmetric encryption algorithms such as AES. Asymmetric encryption (e.g. ElGamal encryption) can be used, but it restricts the total size of the state that can be encrypted and it is not gas-efficient to decrypt on the Ethereum network. In practice, stateful symmetric encryption can be implemented using the sha256 hash function:

$$E_i \leftarrow S_i \oplus (sk_i || \mathsf{sha256}(sk_i, 1) || \mathsf{sha256}(sk_i, 2) || \cdots).$$

As discussed, $E_i \leftarrow S_i \oplus sk_i$ is enough if the state can be encoded using 256 bits.

The custodian must publish $\mathsf{encstate_i}$, $\Sigma_{\mathcal{P}}$ using SC.setstate to settle a dispute which also implies the contract will store $\mathsf{encstate_i}$. If the dispute process is successful, then one party must reveal sk_i, $\mathsf{r_i}$ in SC.resolve. The contract verifies the commitment to both the secret key sk_i and $\mathsf{r_i}$ before decrypting the state as $\mathsf{state_i} = \mathsf{sha256}(sk_i) \oplus E_i$. Finally the customer is only required to store seed to recreate both sk_i and $\mathsf{r_i}$ to support this decryption. In practice, performing a single decryption requires 32,365 gas (approximately \$0.04 at a conversation rate of 1 ether = \$785.31 and 2 Gwei gas price).[10]

B.1 Seeded Encryption with Reduced Size

For better efficiency, the hash function can be used to implement a variant of stateful symmetric encryption with redundancy. We define the scheme $\mathcal{E}_{\text{eff}}$ as follows.

(1) $sk_i = \mathsf{sha256}(\text{seed}, i)$
(2) $E_i = \mathsf{Enc}_{sk_i}(S_i) \triangleq \mathsf{sha256}(sk_i)\ \mathsf{XOR}\ S_i$
(3) $\mathsf{encstate_i} \leftarrow (E_i, \mathsf{i})$

Subsequently, if the custodian submits the signed $\mathsf{encstate_i}$ to the contract and the parties then submit sk_i, the contract verifies that the $256 - k$ leading bits of $(\mathsf{sha256}(sk_i)\ \mathsf{XOR}\ E_i)$ are zeros.

Thus, if $k = 25$ for example, then an attacker needs to solve PoW of $256 - 25 = 231$ bits in order to find a valid decryption. Note that the use of stateful encryption does not entail additional security risks in our construction (e.g., forgetting the state or allowing a hacker to access the state), since the parties must be able to compute the decrypted state from the index i alone (cf. Section B).

Since the scheme $\mathcal{E}_{\text{eff}}$ is not perfectly binding, the security of $\mathcal{E}_{\text{eff}}$ increases when S_i is comprised of fewer bits, due to two reasons. First, breaking the commitment by solving PoW is more difficult when k is smaller. Second, if S_i is comprised of fewer bits then the total amount of money that can be transferred in the channel is smaller, and hence an attacker that decrypts E_i to $S' \neq S_i$ will be able to steal less money.

Of course, if k is larger then the channel can support larger payments and more precise fractional amounts. For the Sprites channel with $k = 25$ and an implicit constant $V = 10^{12}$, the smallest unit of money that can be transferred in the channel is 10^{12} wei which is $1/10^6$ ETH, and the maximum amount of ETH balance that can be represented in the channel is $2^{25} \cdot 10^{12}/10^{18} \approx 33.5$ ETH (because 1 ETH is 10^{18} wei).

Let us note that the standard security notion of encryption with redundancy [1] precludes an adversary from creating any authentic ciphertext, which is irrelevant in the case of $\mathcal{E}_{\text{eff}}$. This is because the authentication in both $\mathcal{E}_{\text{eff}}$ and $\mathcal{E}_{\text{seeded}}$ is done using digital signatures (from all parties), hence $\mathcal{E}_{\text{eff}}$ and $\mathcal{E}_{\text{seeded}}$ are not susceptible to existential forgery.

C RESERVING SECURITY DEPOSIT

Per Section 5.4, a customer cannot publicly verify the state channels (and their financial value) under the custodian's watch. Thus it is not straightforward to determine the custodian's total payout if they are monitoring multiple channels. The custodian may collude with other parties in the state channels (in fact, custodian itself could be Sybil parties in these state channels) and obtain payouts for not settling disputes in several channels. While the security deposit is indeed destroyed, it is not fair for the customer if they lose-out

[10] Rinkeby: 0x6f4730128d03b73bc41a4a674c54ed6c26772aad98fc12461015306bb5cd649f

because the financial deterrent was not sufficient to prevent the custodian's wrongdoing.

As an alternative to burning the security deposit, we propose reserving coins from the security deposit and compensating each customer if the custodian's wrongdoing can be proven for their state channel. Briefly, when a customer hires a custodian to guard their state channels, the customer and custodian agree upon the total value V of the customer's state channels, and also the minimum length of time Δ_{allocate} this allocation will remain active. Afterwards, the customer publishes this agreement via an on-chain transaction and V of custodian's security deposit will be allocated to the customer. This agreement (and the security deposit allocation) can be negotiated before or after setting up the one-way payment channel. However, when the one-way payment channel is closed, the allocated coins remained locked for the agreed length of time. After the allocation time has expired, the custodian can update CC to re-allocate the coins back to the security deposit. An online customer may agree to unlock earlier in exchange for a small cash-back, thus allowing the custodian to accept more new customers. This ensures the reserved allocation remains effective while there are still pending customer's appointments.

Reserving coins from the security deposit allows the customer to receive the allocated compensation if the custodian's wrongdoing can be proven, and this explicit coin allocation restricts the number of customers that can hire the custodian. The custodian can therefore adjust a price policy for each customer, that sets the fee rate for each appointment depending on the reserved allocation amount and the frequency at which the customer requests appointments.

C.1 Modifications to Custodian Contract

To support the above allocation scheme requires modifications to the CC contract. This involves updating CC to track the customer's allocated coins, how to allocate and how to expire the allocation. Afterwards we discuss how the allocated coins are used to compensate the customer if there is proof of the custodian's wrongdoing.

Tracking allocated coins. We modify CC.ID to track allocated coins for each customer's payment channel, a counter ctr to track the latest allocation received and a timer Δ_{allocate} which is the minimum time coins remain allocated after the payment channel is closed. A new flag is introduced for the customer's payment channel ($\perp$, OK, DISPUTE, ALLOCATED, CLOSED). The payment channel transitions OK $\rightarrow$ ALLOCATED or DISPUTE $\rightarrow$ ALLOCATED if it is closed by either party and it was allocated coins from the security deposit while it was open. After Δ_{allocate} has expired, the custodian can authorise the channel's transition ALLOCATED $\rightarrow$ CLOSED to re-allocate the coins back to the custodian's security deposit.

Allocating coins. Both the customer $\mathcal{P}_k$ and custodian C must authorise allocating a portion of the security deposit by signing:

$$\sigma \leftarrow \text{Sign}(sk, \text{``allocate"}, \text{coins}, \Delta_{\text{allocate}}, \text{ctr}, \text{CC})$$

For readability, CC uniquely identifies the customer's payment channel new instance and ctr ensures all previously signed allocation messages are invalidated when this is accepted by CC.[11]

CC.allocate updates the allocation when the payment channel is open and we define it as follows:

```
function allocate(coins, ctr, $\sigma_k$, $\sigma_C$):
    discard if flag $\neq$ OK
    discard if deposit $<$ coins
    discard if ID[$\mathcal{P}_k$].flag $\neq$ OK
    discard if ID[$\mathcal{P}_k$].ctr $\geq$ ctr
    if Sig.Verify($\mathcal{P}_k$, ("allocate", coins, $\Delta_{\text{allocate}}$, ctr, this), $\sigma_k$)$\wedge$
      Sig.Verify($C$, ("allocate", coins, $\Delta_{\text{allocate}}$, ctr, this), $\sigma_k$)
        set ID[$\mathcal{P}_k$].ctr := ctr
        set deposit := deposit $-$ coins
        set ID[$\mathcal{P}_k$].allocated := ID[$\mathcal{P}_k$].allocated $+$ coins
        set ID[$\mathcal{P}_k$].$\Delta_{\text{allocate}}$ := $\Delta_{\text{allocate}}$
        EventAllocate($\mathcal{P}_k$, coins, $\Delta_{\text{allocate}}$)
```

This allocation is only accepted if the payment channel's flag is flag = OK, the custodian's deposit is sufficient to allocate coins and the counter ctr is the largest received so far. If accepted, CC decrements the security deposit before updating the number of coins allocated for the customer and storing Δ_{allocate}, ctr.

Allocation Expiry. CC.resolve and CC.setstate are responsible for closing the customer's payment channel and distributing each party their share of the deposit. Both functions must be modified to detect if there are allocated coins for the customer (i.e. CC.ID[$\mathcal{P}_k$].allocated $>$ 0) and if so transition the channel's flag OK $\rightarrow$ ALLOCATED and set the allocation's expiry time as $\text{t}_{\text{allocate}} := \text{CurTime()} + \Delta_{\text{allocate}}$. After $\text{t}_{\text{allocate}}$, CC.resolve() can be used by the custodian to transition the customer's payment channel flag ALLOCATED $\rightarrow$ CLOSE and return the coins back to the security deposit. For example, the resolve function is modified as follows:

```
function resolve($\mathcal{P}_k$):
    if flag = CHEATED
      send ID[$\mathcal{P}_k$].deposit to C
      send ID[$\mathcal{P}_k$].allocated to C
      set ID[$\mathcal{P}_k$].deposit = 0, ID[$\mathcal{P}_k$].allocated = 0
    if ID[$\mathcal{P}_k$].flag = ALLOCATED $\wedge$ ID[$\mathcal{P}_k$].$\text{t}_{\text{allocate}}$ $\leq$ CurTime()
      set deposit := deposit + ID[$\mathcal{P}_k$].allocated
      set ID[$\mathcal{P}_k$].allocated = 0
      set ID[$\mathcal{P}_k$].$\Delta_{\text{allocate}}$ = 0
      set ID[$\mathcal{P}_k$].$\text{t}_{\text{allocate}}$ = 0
      set ID[$\mathcal{P}_k$].flag := CLOSED
    if ID[$\mathcal{P}_k$].$\text{t}_{\text{settle}}$ $\leq$ CurTime() $\wedge$ ID[$\mathcal{P}_k$].flag = DISPUTE
      send ID[$\mathcal{P}_k$].deposit coins to C
      set ID[$\mathcal{P}_k$].deposit := 0
      set ID[$\mathcal{P}_k$].$\text{t}_{\text{settle}}$ := 0
      if ID[$\mathcal{P}_k$].allocated = 0
        ID[$\mathcal{P}_k$].flag := CLOSED
      else
        ID[$\mathcal{P}_k$].flag := ALLOCATED
        ID[$\mathcal{P}_k$].$\text{t}_{\text{allocate}}$ = CurTime() + ID[$\mathcal{P}_k$].$\Delta_{\text{allocate}}$
```

If a customer successfully seeks recourse and proves the custodian's wrongdoing, then each customer is compensated their allocated coins.

[11] The custodian should only sign a new insurance message if the most recently signed allocation message was already accepted by CC for this instance of the payment channel.

C.2 Implications of Reserved Coin Allocation

In the following we explore the trade-off between burning the full deposit and enforcing the maximum penalty, or compensating the customer and potentially weakening the custodian's deterrent. Afterwards we discuss a potential tragedy of the commons due to a customer's willingness to accept risk for a discounted service, which is a generic issue for any protocol that relies on a central financial deterrent.

C.2.1 Compensating the customer versus burning. The straightforward approach is to burn (i.e., destroy) the custodian's security deposit if the customer's recourse is successful. This is an obvious deterrent as the custodian always receives the maximum penalty for cheating. To relate the security deposit size to the potential payout of an attack by a malicious custodian, the burning scheme can also allocate a portion of the security deposit for each customer. The only purpose of coin allocation in this case is to restrict the payout that the custodian may obtain (in an attack on the customers) from becoming greater than the value in the customers' state channels. However, this would be unattractive to customers as they are not compensated for their loss, in particular when an honest customer specifies the full maximal amount V for the reserved allocation and thus has to pay a premium for the maximum deterrent.

Alternatively, we proposed that each customer can be compensated from a pre-arranged allocation of the custodian's security deposit. Customers that specify the true amount V for the reserved allocation entail no risk, as their signed receipt guarantees full compensation in case they are damaged. This approach requires the custodian to lock the entire value of the customers' state channels as collateral, hence the fee for each appointment may be excessively high. A rational customer may thus request a smaller allocation $V' < V$ as the value of their state channels (and pay lower fees), based on their level of confidence in the custodian. This increases the systemic risk of an attack by a malicious custodian (see also Appendix C.2.4).

Another deficiency of the compensation scheme is that a malicious customer may attempt to steal money from the custodian by mounting a DoS attack on the custodian server(s), in order to obtain compensation after custodian fails to respond to a dispute. This attack becomes more likely in the case that custodian agreed to an appointment with a short waiting Δ_{min} time.

C.2.2 Compensation with leverage. In the previous section, it was assumed if the custodian's wrongdoing is proven, the coins reserved for the non-compromised customers remain locked. Ideally, the coins reserved for the undamaged customers can be used to fully compensate other customers, which can also permit the custodian's security deposit to be smaller.

Hence, to scale the service to a large number of customers, we extend the custodian contract CC to allow $1:\ell$ leverage, meaning that an honest customer's reserved portion of the security deposit of the custodian can be ℓ times smaller than the value of the customer's state channels. For simplicity, let us assume that $\ell \geq 1$ is an immutable parameter of CC (generalising to a variable leverage per customer is straightforward).

Specifically, a customer requests V as the *insurance amount*, and V/ℓ portion of the custodian's security deposit is reserved to the customer. For $\ell > 1$, the security deposit is divided equally among any customers who have submitted a signed receipt to prove the custodian failed to respond to a dispute on their behalf. This requires all customers to provide evidence of the custodian's wrongdoing within a fixed time period after the first signed receipt is submitted. The timer $\Delta_{\mathsf{withdraw}}$ can be re-used for this purpose, since all signed receipts should expire before this timeout (cf. Section 4.5). Thus, all damaged customers are compensated after the $\Delta_{\mathsf{withdraw}}$ time period. Depending on how many customers sought recourse, a damaged customer with insurance amount V is guaranteed to receive between V/ℓ and V compensation.

The $1:\ell$ leveraged ratio lets the customers accept a higher risk in return for discounted service fees. When ℓ is larger, the financial deterrent against an attack by the custodian is weakened. Moreover, in Appendix C.2.3 we discuss the malicious custodian's prospects of using a Sybil to receive the remains of the security deposit via compensation: when a Sybil is used, the financial deterrent is lower bounded by the coins reserved for every non-Sybil customer, and the deterrent is effective only if the custodian's payout after compromising a threshold of state channels is expected to be smaller than the coins reserved to non-Sybil customers.

C.2.3 Compromising a threshold of state channels. For a given leverage ℓ, we evaluate the custodian's payout depending on the probability of successfully compromising state channels from all customers concurrently. Let us consider a simplified model in which

(1) Each customer specified V as the state channels' value.
(2) A successful attack on each customer is an independent event that occurs with probability p.
(3) A successful attack on a customer yields custodian the entire V profit.

Let us assume k customers among which $s \leq k$ are Sybil customers. Let $t \leq k - s$ denote the number of failed attacks. Thus, a payout of $V \cdot (k - s - t) - \frac{kV}{\ell}\frac{k-s-t}{k-t}$ will occur with $B(k - s - t; k - s, p)$ probability, i.e., the binomial density $\binom{k-s}{k-s-t}p^{k-s-t}(1-p)^t$. Note that in practice it is enough for the custodian to use one Sybil with sV allocation instead of s Sybils with V allocation each.

It can thus be seen that it is profitable (in expectation) for custodian to employ Sybil customers when p is small enough, or when the leverage ℓ is small enough. When p or ℓ are high, the malicious custodian will avoid Sybils and instead try to attack all of the k customers – in this case the deterrent of compensation is equal to the deterrent of burning.

To take concrete numbers, suppose for example $k = 5$, $V/\ell = 20$, which imply the following payout vectors:

- $s = 0: \ (100\ell - 100, 80\ell - 100, 60\ell - 100, 40\ell - 100, 20\ell - 100)$
- $s = 1: \ (80\ell - 80, 60\ell - 75, 40\ell - 66.6, 20\ell - 50)$
- $s = 2: \ (60\ell - 60, 40\ell - 50, 20\ell - 33.3)$
- $s = 3: \ (40\ell - 40, 20\ell - 25)$
- $s = 4: \ (20\ell - 20)$

Here, with $\ell = 10$ and $p = 1/2$ we get that custodian's expected payout is 403.12 with zero Sybils, 338.75 with one Sybil, 261.25 with two Sybils, 177.5 with three Sybils, and 90 with four Sybils. The highest probability payout is also monotonic: assuming at least one successful attack, the median payout is 500 with zero Sybils, 350 with two Sybils, and 180 with four Sybils.

Figure 5 demonstrates the optimal Sybil amount s for other values of ℓ and p, given $V/\ell = 20$ and $k = 5$ customers. The y-axis shows the custodian's expected payout, and the bars range from $s = 0$ to $s = 4$ Sybils. Note that using a Sybil would be more profitable than locking a smaller security deposit, as it allows the custodian to recover a portion of the security deposit. E.g., $\ell = 10, p = 1/10$ with $k = 4$ honest customers give 52.5 expected payout, as opposed to 61.9 payout with one Sybil among $k = 5$ customers (shown in Figure 5).

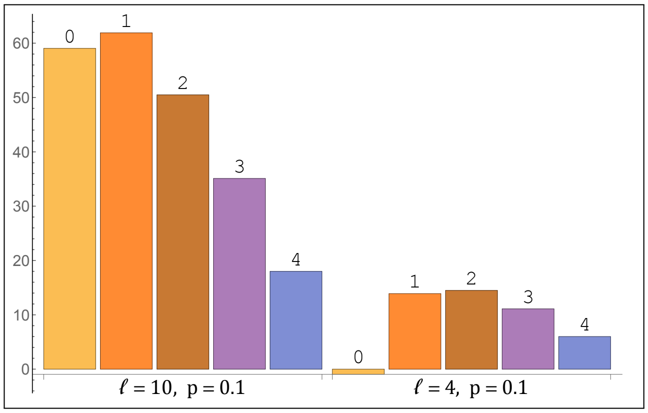

Figure 5: Expected payout with Sybil customers.

The first assumption in our model is for ease of notation: the discretization (i.e., uniform customers) conveys the main idea regarding the conditions under which $s = 0$ is optimal. The second assumption simplifies the model by treating the attacks on the customers as uniform independent events. The third assumption of the simplified model is in fact pessimistic, as it is unlikely that a malicious custodian will always be able to steal the entire value in the state channels. For instance, in a 2-party payment channel the custodian can use an old state where the balance of payments is in favor of the dishonest party, but there might not be an old state where the balance of payments is such that the dishonest party received all the money.

We also note that the parameter p decreases when a customer employs several different custodians, since any honest custodian that responds to the dispute with the latest state will render the attack on the customer unsuccessful.

C.2.4 Further implications. The collateral ratio of the security deposit against the value of the customer's state depends on the risk customers are collectively willing to accept. Let us discuss some additional implications of collateral backing. There is a potential tragedy of the commons [15] as it is collectively in the best for all customers to prevent the custodian's payout becoming greater than the security deposit, but individually customers may deviate and increase the risk for a discounted service, freeloading on the insured amount of the other customers. It is potentially possible to mitigate this risk by including extra rules in the one-way payment contract, that may punish the customer if the value in the outstanding state channels (i.e., the concurrent state channels for which the customer requested appointments) is larger than the value that the customer specified in the initial on-chain hiring transaction, but we leave this aspect for the full implementation. Second, there is a potential denial of service attack as the customer can specify a large value and afterwards not engage with the custodian (i.e., not pay for ongoing services). Both implications appear to be wider (and unexplored) issues for protocols that rely on using a single deposit as a deterrent to prevent a third party's misbehaviour. There are protocols emerging that also propose adopting this central financial deterrent approach and may have the same implications which includes asynchronous payment channels [32], Kyber network [20], and decentralised anonymous micro-payments [5].

Modified generic state channel contract

```
flag := $\perp$
stateRound := 0
state, hstate := $\varepsilon$
$\mathcal{P}$, disputelist, cmdlist := $\emptyset$
$t_{settle}$, $t_{start}$, $\Delta_{settle}$ := 0

function setup($\mathcal{P}$, $\Delta_{settle}$):
    discard if flag $\neq \perp$
    set $\Delta_{settle}$ := $\Delta_{settle}$, $\mathcal{P}$ := $\mathcal{P}$, flag := OK
    EventSetup($\mathcal{P}$, $\Delta_{settle}$)
function setstate(hstate$_i$, i, $\Sigma_{\mathcal{P}}$):
    discard if i $\leq$ stateRound
    if Sig.Verify($\mathcal{P}$, (hstate$_i$, i, this), $\Sigma_{\mathcal{P}}$)
      set stateRound := i
      set hstate := hstate$_i$
      set cmdlist := $\emptyset$; flag := OK
      EventEvidence(stateRound, hstate$_i$)
function triggerdispute($\sigma_k$):
    discard if flag $\neq$ OK
    discard if $\mathcal{P}_k \notin \mathcal{P}$
    if Sig.Verify($\mathcal{P}_k$, (dispute, hstate, stateRound, this), $\sigma_k$)
      set flag := DISPUTE
      set $t_{start}$ := CurTime()
      set $t_{settle}$ := $t_{start}$ + $\Delta_{settle}$
      EventDispute($t_{settle}$)
function input(cmd, $\sigma_k$):
    discard if flag $\neq$ DISPUTE
    discard if $\mathcal{P}_k \notin \mathcal{P}$
    if Sig.Verify($\mathcal{P}_k$, (cmd, hstate$_i$, stateRound, this), $\sigma_k$)
      set cmdlist[$\mathcal{P}_k$] := cmd
      EventInput(cmd, $\mathcal{P}_k$)
function resolve(state$_i$, r$_i$):
    discard if CurTime() $\leq t_{settle}$
    discard if hstate $\neq H$(state$_i$, r$_i$)
    if flag = DISPUTE
      set state := transition(state$_i$, cmdlist)
      set $\mathcal{P}$ := state.$\mathcal{P}$
      set cmdlist := $\emptyset$; flag := OK
      set stateRound := stateRound + 1
      add (stateRound, $t_{start}$, $t_{settle}$) to disputelist
      EventResolve(stateRound)
function transition(state, cmdlist) internal:
    // Implement application logic.
    // Only executable by the contract on the network.
    // Locally executed by parties to verify a transition.
```

Figure 6: Modifications to the state channel construction from Sprites

Custodian contract

```
flag := $\perp$
IDC := $\emptyset$
$\Delta_{settle}$, $\Delta_{withdraw}$, $t_{withdraw}$ := 0
deposit, profit := 0

function setup(coins, $\sigma_C$, $\Delta_{withdraw}$, $\Delta_{settle}$,):
    discard if flag $\neq \perp$
    if Sig.Verify(C, coins, $\sigma_C$)
      set C := C, deposit := coins
      set $\Delta_{withdraw}$ := $\Delta_{withdraw}$, $\Delta_{settle}$ := $\Delta_{settle}$, flag := OK
    EventSetup()
function deposit(coins, $\sigma_k$):
    discard if flag $\neq$ OK or flag $\neq \perp$
    discard if ID[$\mathcal{P}_k$].flag = DISPUTE
    if Sig.Verify($\mathcal{P}_k$, (coins, this), $\sigma_k$)
      if ID[$\mathcal{P}_k$].flag = CLOSED
        set ID[$\mathcal{P}_k$].flag := OK
      set ID[$\mathcal{P}_k$].deposit += coins
    EventDeposit($\mathcal{P}_k$, coins)
function setstate(h$_i$, coins, s$_i$, $\sigma_k$, $\sigma_C$):
    discard if flag = CHEATED
    discard if coins > ID[$\mathcal{P}_k$].deposit
    discard if h$_i$ $\neq$ H(s$_i$).
    if Sig.Verify(C, (h$_i$, coins, , this$^a$), $\sigma_C$)
    $\wedge$ Sig.Verify($\mathcal{P}_k$, (h$_i$, coins, this), $\sigma_k$)
      set profit += coins
      send ID[$\mathcal{P}_k$].deposit $-$ coins to $\mathcal{P}_k$
      set ID[$\mathcal{P}_k$].flag := CLOSED
      set ID[$\mathcal{P}_k$].deposit := 0
      set ID[$\mathcal{P}_k$].$t_{settle}$ := 0
      EventEvidence($\mathcal{P}_k$)
function triggerdispute($\sigma_k$):
    if ID[$\mathcal{P}_k$].flag = OK $\wedge$
      Sig.Verify($\mathcal{P}_k$, (close, this), $\sigma_k$)
      set ID[$\mathcal{P}_k$].flag := DISPUTE
      set ID[$\mathcal{P}_k$].$t_{settle}$ := CurTime() + $\Delta_{settle}$
      EventDispute($\mathcal{P}_k$, ID[$\mathcal{P}_k$].$t_{settle}$)
function resolve($\mathcal{P}_k$):
    if flag = CHEATED $\vee$
    (ID[$\mathcal{P}_k$].$t_{settle}$ $\leq$ CurTime() $\wedge$ ID[$\mathcal{P}_k$].flag = DISPUTE)
      send ID[$\mathcal{P}_k$].deposit coins to C
      set ID[$\mathcal{P}_k$].flag := CLOSED
      set ID[$\mathcal{P}_k$].deposit := 0
      set ID[$\mathcal{P}_k$].$t_{settle}$ := 0
      EventResolve($\mathcal{P}_k$)
```

[a] For readability, we assume this is a unique identifier for the contract and a new instance of the payment channel (e.g. to prevent replay-attacks of previously signed messages, a counter may be incremented for every new deposit or when the channel is settled)

```
Custodian contract (continued)
function stopmonitoring($\sigma_C$):
    discard if flag = CHEATED
    if Sig.Verify(C, (stop, this), $\sigma_C$)
        set flag := CLOSED
        set $t_{withdraw}$ := CurTime() + $\Delta_{withdraw}$
        EventClose($t_{withdraw}$)
function withdraw($\sigma_C$):
    discard if flag = CHEATED
    if Sig.Verify(C, (withdraw, this), $\sigma_C$)
        if ID.length = 0 $\wedge$ CurTime() > $t_{withdraw}$ $\wedge$ flag = CLOSED
            set profit += deposit, deposit := 0
        send profit to C
        set profit := 0
        EventWithdraw()
function recourse($t_{start}$, $t_{expire}$, SC, i, $h_i$, $s_i$, $\sigma_C$):
    discard if flag = CHEATED
    discard if $h_i \neq H(s_i)$
    set chan := lookup(SC)
    if Sig.Verify(C, ($t_{start}$, $t_{expire}$, SC, this, i, $h_i$), $\sigma_C$)
        for $k$ in chan.disputelist.length
            if chan.disputelist[k].$t_{start}$ > $t_{start}$
            $\wedge$ $t_{expire}$ > chan.disputelist[k].$t_{expire}$
            $\wedge$ i $\geq$ chan.disputelist[k].stateRound
                set flag := CHEATED
                EventForfeit()
```

Figure 7: Custodian contract