PRIVACY PRESERVING OUTSOURCING OF STATE CHANNEL ARBITRATION IN MICRORAIDEN

BY

SURYA BAKSHI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Assistant Professor Andrew Miller

ABSTRACT

Decentralized blockchain-based cryptocurrencies like Ethereum and Bitcoin offer a new way to hold and transact money. However, storage requirements for every node and difficulty in transaction confirmation make them difficult to match traditional payment processors. A proposed solution to scalability is the use of payment channels that allow mutually distrustful parties to create and authorize payments between one another off-line. Not only does this allow payments to be processed quickly, but it also reduces transaction volume in the underlying blockchain.

An unsolved problem with off-chain payment channels is that participants in the channel must be on-line and alert to channel events *all the time*. If any participant in the channel goes off-line for any reason (power outage, process crashes, cost too high) that party stands to lose money if other parties attempt to reverse payments. In an ideal world, a solution would involve a third party that can process payments on behalf of each party, but it requires trust establishment.

In this work, we present a protocol that solves the monitoring problem called Pisa. Pisa allows channel participants to appoint distrusted third parties to watch their channel and handle finalization on their behalf without revealing any linkable state information. We also propose a fair exchange protocol that ensures that payment for appointment of a third party guarantees a penalty if the third party cheats. Further, we implement Pisa on top of an existing Ethereum payment channel framework, μ-Raiden, and we demonstrate its additional overhead to channel operation.

To Satoshi, for always being Craig Wright.

ACKNOWLEDGMENTS

I'd like to acknowledge my advisor, Andrew Miller *TBUH*, for helping me explore the field of cryptocurrencies and discover the kind of research that I really enjoy doing. I would also like to acknowledge a close friend, Philip Daian *DBUH*, for initially sparking an interest in cryptocurrencies, introducing me to my current advisor and for playing an integral part in my academic interest and growth. Of course, none of this would be possible without the 1109 group, including but not limited to: Philip Daian, Bharagava Manja, Will Schellhorn, Kai Demler, Matt Skrzypczyk. Without their continuous support and fostering of a fast-paced learning environment throughout my education, I might never have even considered pursuing graduate study.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Decentralized cryptocurrencies like Bitcoin and Ethereum have risen to prominence in recent years due to the promise of their underlying technology: the blockchain. At a high level, the blockchain is a consensus protocol where nodes in the network compete to finalize and order transactions in a consistent and tamper-resistant public ledger [1]. The protocol rewards users for taking part in the computationally intensive leader election with newly minted coins and transaction fees. All transactions that are finalized by the leader are replicated and verified by *every* node in the network.

Although this form of network-wide replication provides stronger security guarantees for finalized transactions, it is a significant bottleneck for transaction throughput. With rising transaction volumes, the requirements to store and verify computation for every transaction may shrink the number of nodes actually capable of performing the task. To prevent requirements exceeding the available resources in the network, this trade-off has resulted in cryptocurrencies with low throughput compared to modern payment processors. For example, Visa, at peak capacity, can handle up to 47,000 transaction per second [2] (real usage hovers around 4,000 tps, even at peak usage). When compared to Bitcoin's 7 transactions per second [3, 4] it is clear that decentralized blockchains are still very far away from handling modern transaction requirements. Many solutions have been proposed which radically redesign the blockchain consensus mechanisms [5, 6, 7].

This trade-off has also been a point of contention in the cryptocurrency community [3, 8]. Some proposed solutions from one side want to increase the block size allowing more transactions to be finalized per block, but they sacrifice the requirements of nodes to store and verify larger blocks. Bitcoin has stagnated, in this respect, with few layer 1 changes[1] aimed at increasing transaction throughput. Ethereum, however, has a higher network through-

[1]Layer 1 protocols deal directly with the functioning of the blockchain.

put, but at the cost of the inability of some peers to verify transactions in real-time. For example, fast sync modes were introduced for those nodes unable to process all transactions in real-time. This has led many to turn to layer 2 solutions.[2]

This thesis focuses on the scaling approach of off-chain payment channels in which mutually distrusting parties can authorize payments among themselves by locally executing the state transitions of a payment smart contract. In a payment channel, the blockchain is no longer used to validate and authorize payments, but, instead, it is used only to open and close payment channels [9, 10]. They are very promising scalability solutions as they can easily be implemented on top of existing blockchains like Ethereum and Bitcoin, and there are actually companies putting considerable effort behind building them [11, 9, 12]. One such company, Raiden, is trying to launch an Ethereum payment channel network and recently launched μ-Raiden [13], a lite version of Raiden that implements one-way payment channels for paywalled content. The drawback of payment channels, though, is that channel participants are required to always be on-line and alert to counter-parties trying to finalize their channels on an expired state. When off-line, finalization of an older state is tantamount to an irreversible theft of money.

A proposed solution to this problem is for participants to appoint a third party to intervene on their behalf when a channel is being finalized. As of today, this idea has been discussed in its application to Raiden and Bitcoin's Lightning Network [9] (LN), however, only LN has created proposals for a monitoring protocol. The proposal in LN requires the third party to store the hash of every previous state so that the channel can settle on the most recent state and expired states can be revoked if published. This approach has a few drawbacks: the design of LN requires the monitor's storage per channel to scale linearly with the number of payments, the monitor is only compensated if the final state is disputed, and inactivity by the monitor is not penalized. Ethereum's main payment channel proposal, Raiden, currently does not have any solution to the monitoring problem.

This work presents a new protocol for third-party monitoring of payment channels in Ethereum called Pisa. The protocol allows channel participants to appoint one or more monitors every time a new state is authorized. Pisa

[2]Layer 2 solutions do not modify the blockchain architecture but only interact with it.

also ensures that monitors do not stay on-line and use storage just to watch a channel whose final state is never disputed by paying them in a one-way payment channel every time a new state is outsourced. Furthermore, only the hash of the most recent authorized state is provided to the monitor to preserve the privacy of the channel and ensure a constant storage requirement per channel. Finally, monitors and their customers exchange payments for receipts that can be used to penalize the monitor for not correctly responding to channel finalization.

The rest of the thesis is organized as follows:

- Chapter 2 introduces the key concepts critical in understanding the blockchain's consensus mechanism and the cost of on-chain transactions. It also introduces payment channels at a more detailed level, setting up the rest of the thesis' discussion of Pisa.
- Chapter 3 describes the monitoring protocol at a high level in the context of generalized payments channels. In this chapter the specifics of the monitor's contract are described along with the protocol between the monitor and customer that facilitates fair exchange and the penalties for incorrect behavior.
- Chapter 4 describes an implementation of Pisa in μ-Raiden, a uni-directional payment channel framework currently deployed in Ethereum. The additional costs involved in running Pisa are presented and compared with μ-Raiden.

CHAPTER 2

BACKGROUND

This chapter explains the necessary background information needed to understand the remainder of this work. Section 2.1 begins by explaining the blockchain, the mechanisms that provide security and the concept of pseudonymous accounts. It goes on to explain how transactions are created, validated and update the state of the blockchain. Finally, Section 2.2 describes generalized payment channels and their impact on scalability in cryptocurrencies.

2.1 The Blockchain

A blockchain is a distributed network that maintains a globally consistent log of transactions (Figure 2.1). The transactions are created by accounts on the network and modify the blockchain's global state. Peers in the network compete to win rounds of a leader election protocol and finalize transactions by adding them to the current blockchain. In the scope of cryptocurrencies, the leaders that are elected are called miners. In each round, miners compete to complete a heavy computational task for election as leader of that round. Once elected, the miner finalizes transactions by collating them into a block and appending the block onto the current blockchain. The new block references the previous block and incorporates its hash into the computation work that was performed. Not only does a large computation act as a deterrent to creating blocks that violate the rules of the network, but it makes it difficult to revert transactions in old blocks as they are linked to every subsequent block–requiring all subsequent blocks to be changed. All nodes in the network then validate the block and append it to their local copy of the blockchain. Although expensive, replication of the blockchain on all nodes ensures that a malicious majority is needed in order to subvert the rules on consensus. The changes made by the new transactions are applied and the

states of the relevant accounts are updated, hence updating the global state of the blockchain. The remainder of this section will focus on the blockchain as it is implemented in Ethereum.

Accounts. There are two types of accounts in Ethereum: externally owned accounts and contract accounts. Externally owned accounts are controlled by ECDSA private keys. Contract accounts are controlled by the contract code that is associated with them. Externally owned accounts do not have associated code, and can send messages to any other account by creating and signing a transaction. Contract accounts, on the other hand, will only execute their code when triggered to do so by a incoming message. The triggered code is then allowed to read and write from its persistent storage, create new messages or even new contracts (but never transactions).

Transactions. Transactions are signed data packets that store a message that is sent from an external account to any other account on the network. A transactions contains:

1. The recipient of the message.
2. A signature by the sender.
3. The amount of Ether being transferred.
4. Optional, arbitrary data.
5. A `GASLIMIT` value that limits the maximum number of computational steps allowed by execution of the transaction.
6. A `GASPRICE` value that specifies the amount of Ether to be payed per computational unit used during execution.

The first few fields enable currency transfers and are common to all cryptocurrencies. The data field is optional but serves a special purpose in Ethereum. When a transaction sends a message to an external account, the `data` field does not serve a default purpose; it can be any arbitrary data the sender wishes to include. When a transaction sends a message to a contract account, the data field is read by the contract and used to determine what code to execute. Specifically, the `data` field is parsed to determine the function being called and its input arguments.

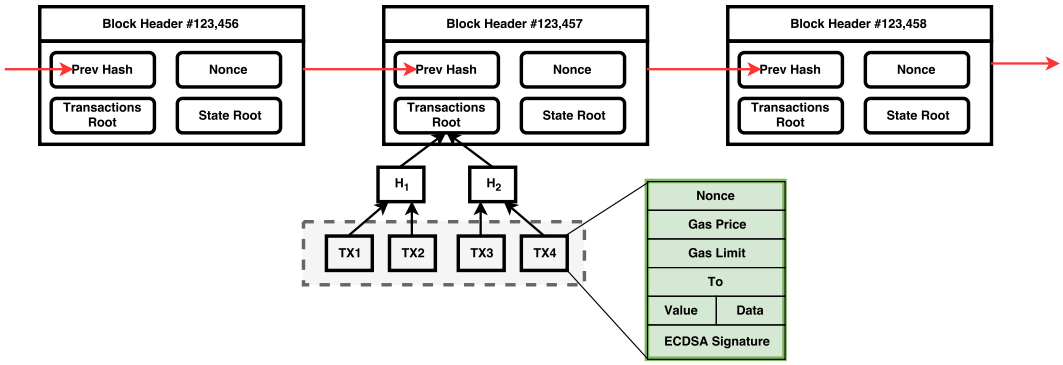

Figure 2.1: Representation of a blockchain showing the transactions in a block and the contents of the transaction.

The last two fields have to do with a new feature in Ethereum called gas. At a high level gas is used to limit the execution of smart contract code when it is triggered and its state changes are finalized. Limiting the units of gas allowed in a particular transaction's execution ensures that nodes verifying them have a hard upper bound on the amount of computation they need to do. The gas price is used to compensate miners for the work expended in executing a contract and including it in the next block. Blocks also enforce a limit on the total amount of computation allowed in all transactions within them.

Although gas is an effective method for ensuring nodes are not consumed by large computations, it also makes block space a valuable and costly resource. As a consequence, block space is very limited and transactions must fight to be included in a block as soon as possible. Transactions that pay a higher gas price to miners will obviously be preferred and confirmed more quickly. This poses a big problem for scalability as it can drive transaction fees up as more users, and more transactions, enter the network. Similarly, scalability faces further price increases with recent proposals to commoditize resources on the Ethereum network like transaction relaying.

Clearly, with larger adoption and more transactions, Ethereum fees may increase to a point where small value transfers may not be reasonable anymore. This is especially true for the purchase of cheap physical goods made in person, where small value transfers need to be confirmed quickly. Therefore, off-chain payment solutions solve this problem by allowing arbitrarily small payment to be made to another user without requiring frequent interaction with the blockchain.

```
1 event ExampleLog(string _arbitrary_data);
2
3 function foo() {
4     ExampleLog("This is event will be logged.");
5 }
```

Figure 2.2: When the function `foo` is executed an `ExampleLog` event is created. The event can relay arbitrary information about the execution to any party watching this contract for events.

The final feature of Ethereum that is relevant to this work is events. In Ethereum, a smart contract can generate logs be emitting events that are recorded in the blockchain. An event can be invoked at any time during contract execution and include any arbitrary data. An example of an event is shown in Figure 2.2. When an event is created by a contract, it is recorded in the same block as the transaction that generated it and can be easily queried for with Ethereum clients. A common practice in developing applications with a smart contract back-end is to use events external inputs like a user depositing money. For example a payment channel can use events to indicate when users deposit funds and being an off-line channel.

2.2 Payment Channels

The protocol described in this work relies on a construction of payment channels based on the generalized state channels described by Miller et al. [14]. The construction allows an arbitrary number of parties, $p_1, ..., p_n$ to authorize and send payments between themselves without the need to publish transactions to the blockchain. The only blockchain interaction required is to create channels and settle their final state.

A payment channel is defined by its participants, their current balances, a state flag, and a monotonically increasing state counter. A state flag is always one of three values: $\{\perp, \texttt{OK}, \texttt{DISPUTE}\}$. In order, they represent an uninitialized channel, a channel operating off-chain, and a channel handling a settlement of its final state on-chain. *Despite potential confusion, the settlement stage is referred to by the* `DISPUTE` *flag because any participant can dispute the proposed final state during this time. The timers and variables associated with this period all have the "settle" subscript to indicated that the*

channel is settled after this stage.

2.2.1 Channel Creation

To initialize a channel, the contract creator must specify the amount of time, δ_{settle}, after a dispute is raised, that the participants have to challenge the latest proposed final state. Participants indicate agreement with the parameter by giving the contract some initial amount of Ether and a signature. Once all participants have provided a signature and an Ether deposit, the channel's state flag moves from $\perp$ to `OK`, the state counter initializes to 0, and all participants begin handling payments off-chain. u The initial deposits are recorded on-chain and comprise the first state of the channel. All subsequent off-chain state updates must build on top of it. In the rest of this section, the "state" of the channel is defined as the most recent authorizes set of participants' balances.

2.2.2 Payments

After initialization of the channel, the initial state of the channel is just the deposit of each of the participants. From there, the integrity of the channel relies on the participants collectively signing new states and revoking the old ones, creating arbitrary payments between them. The state counter is ascribed to every new state to ensure a global agreement on the chronology of all previous states from S_1 to an arbitrary S_i.

A payment from one participant to another can be initiated by any party. Figure 2.3 demonstrates a participant, Mike, proposing a payment from himself to Bob. To propose a new state Mike must first define what the updated balances of the new state will look like. Then to propose the new state S'_{i+1} he creates a signature:

$$\sigma_M = \mathsf{Sign}_{\text{Mike}}(S'_{i'}, i') \tag{2.1}$$

and sends it to Alice, Sarah and Bob. Each of them verifies that Mike has a sufficient balance to pay Bob 1 coin, checks the validity of his signature and asserts that $i' = i + 1$. If the proposed payment is accepted, they agree to it by creating a similar signature with their keys and broadcasting it to everyone else. The proposal is committed as the new state when every party

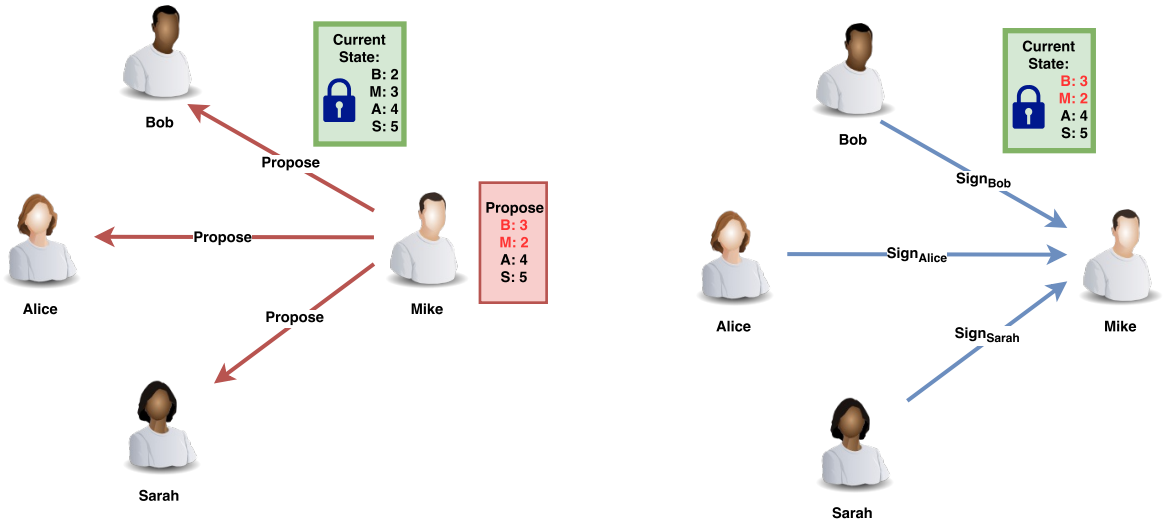

Figure 2.3: Mike proposes a new state to the other participants indicating that he wants to pay Bob 1 coin. The participants may choose to reject it, but they approve it by signing off on it. Once all participants sign it, it becomes the new state of the channel.

receives and verifies the signatures from everyone else. The flexibility of this generalized payment channel also allows Mike to propose an update where by he debits Bob's accounts and pays himself. One can imagine a case where Mike is able to automate payments from Bob's account in return for some good or service.

When a participant initiates a new state, a local timeout is initialized to wait for responses from the rest of the channel. If no new state authorization is seen by the initiator within the local timeout, the protocol resorts to settling on-chain so that his money is no longer locked in a channel where his payments are not being authorized.

2.2.3 Settlement

Any party can attempt to stop the payment channel and settle its final balance on the blockchain. The first step in finalizing a payment channel is calling the **setstate** (Figure 2.4) contract function with the last channel state, the new state counter, and the channel participants' signatures. In the example in Figure 2.3, if Alice wants to finalize the channel she must submit:

$$S_i, i, \sigma_{\text{Alice}}, \sigma_{\text{Sarah}}, \sigma_{\text{Mike}}, \sigma_{\text{Bob}} \tag{2.2}$$

```
function setstate((state′, i), σ_P):
    discard if i ≤ stateRound
    if Sig.Verify(P, (state′, i′, this.ID), Σ_P)
      set stateRound := i′
      set state := state′
      EventEvidence(stateRound, state)
```

Figure 2.4: Pseudo-code for the **setstate** function in a payment channel. Only more recent states with a larger counter than the current stateRound will be accepted. The state and counter value must also be signed by all channel participants. The signature also include the address of the payment channel contract, this.ID, for replay protection.

where $\sigma_k = \mathsf{Sign}_k(S_i, i)$. This allows the contract to verify that S_i was authorized by all parties before accepting it. The second step is for Alice to call **triggerdispute** (Figure 2.5) which initiates a settlement period.

When the settlement period is initiated, the channel's state flag transitions from `OK` to `DISPUTE` and it establishes a deadline $t_{\text{settle}} := \textsc{CurTime}() + \delta_{\text{settle}}$. Each party is notified by the contract of the triggered settlement period and the computed t_{settle} deadline before which any participant can dispute the final state of the channel. If the state S_i submitted by the settlement initiator is not the latest authorized state in the channel, then finalization of that state will mean that somebody will not get the coins they have previously been paid. Once finalized, the state of the channel can no longer be changed, so participants need to be active in detecting and responding to settlement attempts on expired states.

In order to dispute the proposed final state, any participant may provide the contract with a different state $S_{i'}, i'$ and the corresponding signatures from all participants. The contract will assert that any new state being submitted has a counter such that $i' = i + 1$. This ensures that the contract will only accept states newer than the ones already committed by others. If the contract can validate that the new state, $S_{i'}$, was authorized by all participants, it then becomes the new temporary final state. Disputes can be raised indefinitely until the t_{settle} has passed at which point the latest state to be submitted is considered final. Once the settlement period has ended and the final state of the channel is determined, any participant may call **resolve** (Figure 2.6) to claim their final balance of coins.

```
function triggerdispute(state_i, r_i, σ_k):
    discard if flag ≠ OK
    discard if state ≠ ε
      set flag := DISPUTE
      set t_start := CurTime()
      set t_settle := t_start + Δ_settle
      EventDispute(t_settle)
```

Figure 2.5: Once a state is set, this function begins the process of closing the channel. All participants are notified of the dispute and can respond with their newest state by calling **setstate**.

```
function resolve():
    discard if CurTime() ≤ t_settle
    if flag = DISPUTE
      set flag := OK
      send state balances to 𝒫
      EventResolve(stateRound)
```

Figure 2.6: Resolve function called after the dispute period ends. Sends each participant his or her final balance encoded in state.

CHAPTER 3

PROTOCOL

Off-chain payment channels presented in Section 2.2 greatly reduce the cost of paying individuals using cryptocurrencies by removing the need to finalize every payment on the blockchain. Not only does this reduce the fees paid per payment, but it also reduces transaction bloat in the underlying blockchain network. This means the network can support more users, making many small-valued payments to each other without substantial increase in block space demand. There is a downside to this approach, however, that has lead to a lack of adoption or deployment of such systems in the wild. Payment channels require participants to always be on-line and active in the channel. This means being alert and responding to channel closes/disputes when other participants publish old authorized states. A user, though, can go off-line for extended periods of time for a variety of reasons: power outages, computer failures, connectivity issues, etc. Therefore, a party that is at risk of going off-line stands to lose money by using a payment channel. At least with on-chain transactions, there is no need to stay on-line once they are created as the rules of the network will ensure they are correctly executed and published.

Example. Assume Alice and Bob open a payment channel with each other where either of them can craft payments to each other. The channel starts with a deposit of $10 from each of them. Over the course of multiple rounds, both propose state updates which pay the other party some amount of money. When Bob's process crashes and is no longer active in the channel, Alice is also unable to advance the state further–effectively locking her money in the channel indefinitely. Therefore, Alice will attempt to close the channel and reclaim the remainder of her funds. If she notices that Bob is not on-line, she can submit an older state to the chain and reverse some of her payments. If Bob does not come back on-line before the settlement deadline has passed, Alice has successfully settled the channel in a state where she has taken some of Bob's money.

In an ideal world, a solution to this problem would be an intermediary that processes all off-line payment on their behalf. This third party would always be on-line and watch the state to ensure it closes correctly. This section outlines a protocol called Pisa, that allows channel participants to appoint third parties to monitor channels on their behalf to relax their liveness requirements.

3.1 Overview

Pisa enables participants in the payment channel to appoint one or more monitors to watch their channel for disputes. Participants interact with the monitor's contract which implements a one-way channel with a large deposit, c_M, and additional functionality to watch other channels. A customer communicates with the monitor and passes along the hash of the current state and a payment for every new appointment.

Any participant in the channel can appoint a monitor to respond with evidence in the case of settlement. The appointment of a monitor, however, is for a single state only. If the monitor is appointed for the current state, S_i, it acts correctly only if it submits evidence of the state S_i. Therefore, participants who want continuous monitoring of their channel must appoint a monitor for every new authorized state. To appoint a monitor, a customer provides the monitor with the hash of the current state, $H(S_i, r_i)$,[1] the round number i, signatures, $\sigma_{\mathcal{P}}$ such that

$$\sigma_i = \mathsf{Sign}(H(S_i, r_i), i) \tag{3.1}$$

and payment for watching the channel. In return, the monitor gives the customer a signed receipt for the appointment specifying the round being watched and the period of the appointment. The receipt serves as proof that the customer can provide in the future to penalize the monitor for not acting correctly during channel settlement.

When a channel close is requested, the monitor can act in one of two ways. The monitor can respond with the latest hashed state that it was appointed for along with the correct signatures. If it does responds with the latest

[1] r_i is a random blinding factor that makes discovering pre-images more difficult.

appointed state, no other party will be able to finalize an older state. This ensures the channel closes *at least* in the customer's latest outsourced state. The monitor can choose to collude with the other participants in the channel by not submitting the correct state during a dispute. An older state may favor the other participants of the channel, and a bribe can be paid to the monitor to act maliciously. However, a customer can come back on-line and submit the monitor's signed receipt as proof of appointment and claim its large deposit.

The amount of the monitor's deposit is crucial to the security of the protocol. Customers should only appoint monitors whose large deposit is large enough such that the other participants could not pay the monitor enough from an older state to make it profitable for the monitor to cheat.

3.2 Goals

This section presents the goals of Pisa such that the privacy of every intermediate channel state S_i is not revealed to the monitor and that fairness is preserved for both customer and monitor.

3.2.1 State Privacy

In Pisa the monitor should not learn anything about the actual state of the channel at any time except finalization. The monitor receives only the hash of the current authorized state, $H(S_i, r_i)$. Although the payments themselves are hidden, the necessity of revealing the state's round number does not hide when the payments were made.

3.2.2 Fairness

When a monitor accepts an appointment for a particular state in a channel, it is always paid for it by the customer. The monitor also provides a receipt to the customer indicating the terms of the appointment and its duration. However, Pisa must ensure that neither party can cheat in the exchange of payment for receipt. This requires Pisa to accomplish two goals:

- **Fair Exchange** In the generalized case, a fair exchange is a protocol where two mutually distributing parties can exchange digital goods in a way that guarantees: both parties receive the other good or neither of them do. When a customer and a monitor are exchanging a receipt for a payment, Pisa must ensure that the two goods are exchanged fairly. This ensures that the monitor cannot accept payments without providing evidence of the appointment, and that the customer cannot obtain a valid receipt without paying for it.
- **Non-frameability** When the monitor has acted correctly, a customer should not be able to provide a receipt to frame the monitor for wrongdoing.

3.2.3 Monitor Cost

An ideal protocol for third-party monitoring minimizes the storage requirements for the monitor. Therefore, a goal of Pisa is to limit the monitor's resource requirement to be $O(1)$. Pisa should also ensure that the monitor is rewarded for storage updates for every new appointment. To further reduce the operating cost, all payments between the customer and monitor are handled by an off-chain uni-directional payment channel; this ensures a continuous revenue stream for the monitor.

3.3 Assumptions

This section outlines the assumption made by the protocol on the capabilities and behavior of the channel participants, the customer and the monitor.

3.3.1 Channel Participants

The participants of this channel are assumed to act rationally. Each participant's decision strategy will choose the action that gives that party the greatest final balance in the payment channel. In a payment channel, each party will always attempt to finalize the channel on a state that gives that party the highest payoff, even if it is not the latest one. In the simpler case of uni-directional payment channels the receiver of the payments will necessarily publish the latest agreed upon state as it will always pay that participant

the most. Naturally, the client will always prefer to finalize on an earlier state.

The channel participants are assumed not to collude with the monitor in order to reveal the pre-image of any state. If participants reveal the state, the monitor can eavesdrop on the channel with ease. However, the participants may collude with the monitor to settle their channel on an older state. If the other participants can profit from paying the monitor more than the monitor's large deposit, they will always attempt to do so. Finally, it is assumed that any party that is off-line for more than the length of a dispute settlement period will have appointed at least one monitor to respond to disputes.

3.3.2 Customer and Monitor

Any channel participant that appoints a monitor to watch a channel is called a *customer* of that monitor. Customers are also rational and will try to maximize their payoff in their protocol with the monitor. This means that, if possible, the customer will always try to reverse the payments to the monitor or craft evidence against the monitor if feasible.

The monitor is assumed to be a rational and curious actor. Rationality implies monitors will watch a channel appointed to them only if they have been appropriately paid to store state information and will correctly respond to closures if they cannot profit from cheating. For example, if the channel participants offer bribes greater than the monitor's contract deposit then the monitor will choose to collude with them to the close channel in an incorrect state. Curiosity implies that the monitor will attempt to extract the state information from information provided by the customer if computationally feasible.

3.3.3 Contracts

Pisa does not take into account edge cases on the blockchain where the outcome of contract execution may be reverted or altered due to hard forks, transaction reorganization, or any other blockchain failures. Therefore, contracts are assumed to be honest third parties with immutable code that al-

ways executes correctly. For the monitor contract, specifically, it is assumed that the contract can look up disputes in the channel that it is watching and extract the relevant information.

3.4 Payment Channel Modifications

This section focuses on the changes made to the generalized payment channel contract in Section 2.2. The new payment channel supports participants authorizing the hash of state, $H(S_i, r_i)$, instead of just S_i.

The first modification to the payment channel construction supports reporting the authorized state hash of the current state through the **setstate** function (Figure 3.1). To preserve the state's privacy on-chain, each participant exchanges signatures for $H(S_i, r_i)$ where r_i is a blinding nonce. Without the nonce, it would be computationally feasible for the monitor to brute force the pre-image of the state since the starting balances of each party are known at channel creation time. The round number, in this case, is not hidden so that the contract is able to order the hashed states it sees and only accept more recent rounds. Furthermore, the state, S_i, and the nonce, r_i, are only revealed to the monitor when the channel is being finalized and the final state must be revealed for the contract to pay out to its participants.

```
function setstate((hstate, i), σ_P):
    Discard if flag≠ DISPUTE
    discard if i ≤ stateRound
    if Sig.Verify(P, (hstate, i), σ_P)
      set stateRound := i
      set hstate := hstate
      EventEvidence(stateRound, hstate)
```

Figure 3.1: Modified **setstate** function that accepts the blinded hash of the current state instead of the state itself.

3.5 Monitor Contract

This section provides an overview of the monitor contract which allows customers to appoint the monitor for a specific state. It implements a one-way

Monitor contract

flag := $\perp$
ID, monitor := $\emptyset$
$\Delta_{\text{settle}}, \Delta_{\text{withdraw}}, \text{t}_{\text{withdraw}} := 0$
deposit, profit := 0

function `setup`(M, deposit, σ_{M}, Δ_{withdraw}, Δ_{settle},):

  if Sig.Verify(M, deposit, σ_{M})
    set M := M, deposit := deposit
    set $\Delta_{\text{withdraw}} := \Delta_{\text{withdraw}}$, $\Delta_{\text{settle}} := \Delta_{\text{settle}}$, flag := OK
  `EventSetup`()

function `deposit`($\mathcal{P}_k$, $\text{c}_{\text{deposit}}$, Sig):

  discard if flag $\neq$ OK
  discard if ID[].flag = DISPUTE
  if Sig.Verify(, c_{deposit}, $\sigma_{\mathcal{P}_k}$)
    if ID[].flag = CLOSED
      set ID[].flag := OK
    set ID[].deposit += c_{deposit}
  `EventDeposit`(, c_{deposit})

Figure 3.2: This first part of the monitor contract is where the contract parameters are set up by the monitor. It also specified a **deposit** to which a customer can deposit coins to be used in the payment channel. A customer must deposit coins before the monitor can be appointed to watch any channel.

payment channel for the customer to pay the monitor per appointment, and it also allows the customer to forfeit the monitor's deposit if it acts incorrectly. In the following, each part of the monitor contract is accompanied by a snippet of pseudo-code.

The monitor's contract has four state flags: {$\perp$, `OKAY`, `CHEATED`, `CLOSED`}. The contract state starts in state $\perp$ to indicate that it is uninitialized. Once the monitor has submitted a large deposit, c_M, to the contract and sets $\Delta_{\text{settle}}, \Delta_{\text{withdraw}}$ using the **setup** function (Figure 3.2), the contract transitions from $\perp \rightarrow$ `OK`. Customers can now open payment channels with the monitor by depositing money through the **deposit** function.

Monitor's Payment Channel. The monitor contract stores a list of all

of the payment channels that customers have opened, indexed by unique identifiers. A customer's payment channel can be in one of four flag states: $\perp$, `OK`, `DISPUTE`, `CLOSED`. In order to transition a channel from $\perp \rightarrow$ `OK`, or from `CLOSED` $\rightarrow$ `OK` the customer must deposit money into the channel via the **deposit** function (Figure 3.2). As this is a one-way payment channel, the monitor can choose to unilaterally close any channel it has open at any time. The contract does not allow a settlement period in this case as the monitor will always submit the most recent authorized state. The customer can request a payment channel be closed as well, but will always have to wait until the monitor is given enough time to dispute the final state of the channel.

Fair Exchange. Once the channel is open, the customer can pay the monitor for every new hashed state, $H(S_i, r_i)$, it wants the monitor to publish on its behalf. In return for the payment and the hashed state, the monitor creates and signs a receipt as proof of appointment. The receipt and payment are exchanged using the fair exchange protocol illustrated in Figure 3.3.

To initiate the fair exchange, the customer sends the current state hash, the current round number, the amount of the payment (c) and an expire time (t_{expire}). The customer must also sign all of these parameters as well as the channel participants' signatures on the state hash. The monitor can check the state hash that is sent along with the signatures and round number. If satisfied with the payment and the validity of the signatures, the monitor issues a receipt to the customer containing the state hash, round number, the length of the appointment ($\texttt{CurTime}() + \Delta_{\mathsf{settle}}$), and a conditional transfer hash h_i. In order for the receipt to be ratified, the monitor must reveal the pre-image of h_i, s_i as it needs to be provided for the customer to request recourse in the future. The customer creates a signed conditional transfer that pays the monitor if the pre-image of h_i is revealed. The monitor can choose to reveal the pre-image to the customer off-line once it receives the conditional payment. If the monitor does not reveal the pre-image to the customer, the customer raises a dispute through the **triggerdispute** function (Figure 3.4), moving the contract flag from `OK` to `DISPUTE`. To claim the conditional payment, the monitor can reveal the pre-image by forcing an on-chain update to the channel by called **setstate**. Calling this function completes the conditional payment and sets a dispute settlement period, but it also reveals the pre-image of h_i–validating the customer's receipt. Of

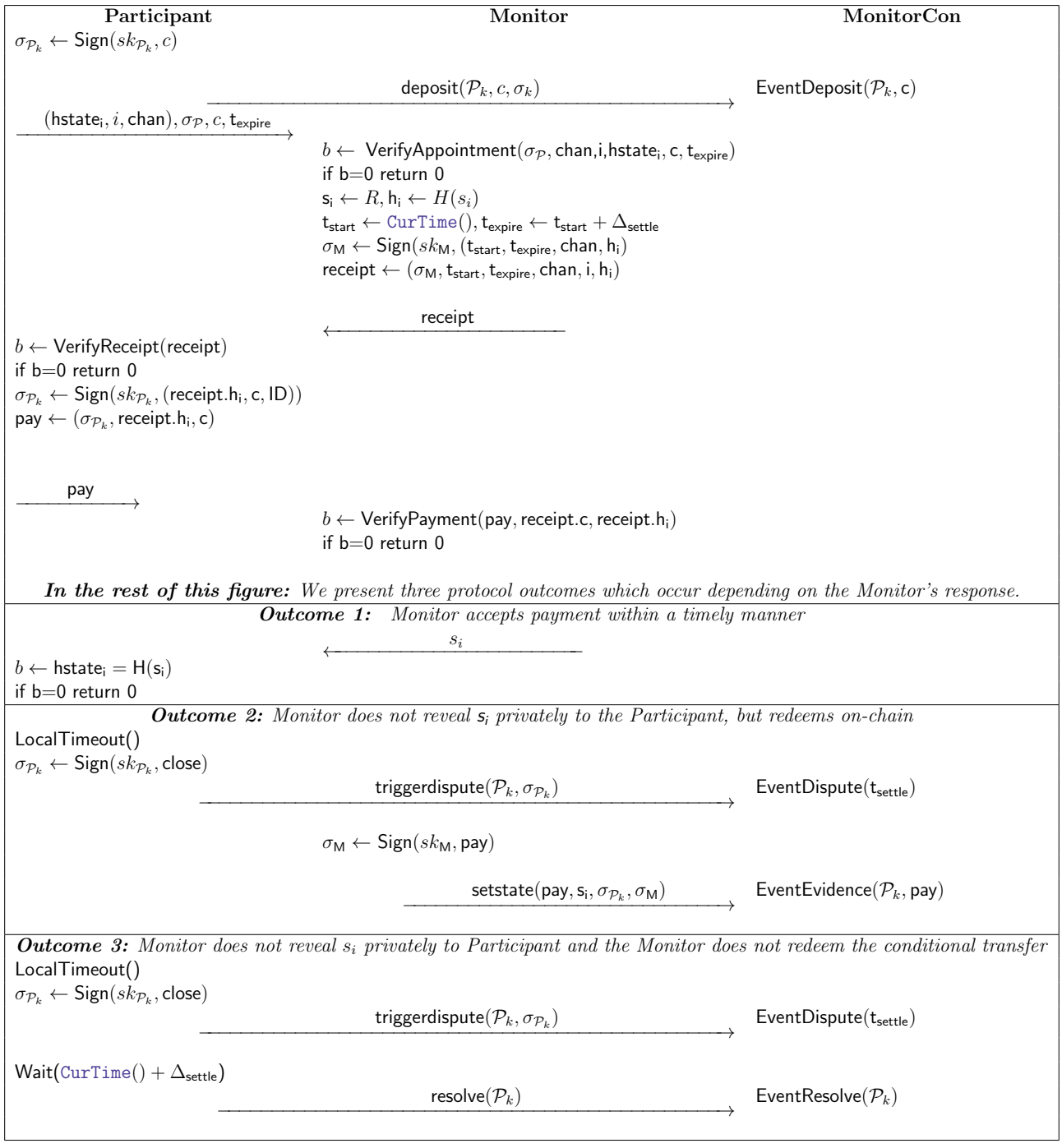

Figure 3.3: A fair exchange protocol that ensures the monitor is paid upon validating the customer's receipt. If the monitor redeems the conditional payment in any way, the pre-image is always revealed and the receipt always becomes valid.

Monitor contract (continued)

function `setstate`(state, s_i, Sig.σ_M):

discard if flag = CHEATED
discard if state.k $\neq$ ID[$\mathcal{P}_k$].k
discard if state.payout $>$ ID[$\mathcal{P}_k$].deposit
if state.conditional_transfer = TRUE
discard if state.hash $\neq$ H(s_i).
if Sig.Verify(monitor, state, σ_{monitor})
$\wedge$ Sig.Verify(, state, σ_{sender})
set profit += state.payout
send ID[$\mathcal{P}_k$].deposit $-$ state.payout **coins to** sender
set ID[] := $[0, 0, \text{CLOSED}, \text{ID[sender].k} + 1]$
`EventEvidence`(, state_i)

function `triggerdispute`(, $\sigma_{\mathcal{P}_k}$):

if ID[].flag = OK $\wedge$ Sig.Verify(sender, close, σ_{sender})
set ID[].flag := DISPUTE
set ID[].t_{settle} := `CurTime`() + Δ_{settle}
`EventDispute`(, ID[sender].t_{settle})

function `resolve`($\mathcal{P}_k$):

if flag = CHEATED $\vee$
(ID[$\mathcal{P}_k$].$t_{\text{settle}} \leq$ `CurTime`() $\wedge$ ID[$\mathcal{P}_k$].flag = DISPUTE)
send ID[].deposit **coins to** C
set ID[] := $[0, 0, \text{CLOSED}, \text{ID}[\mathcal{P}_k].\text{k} + 1]$
`EventResolve`()

Figure 3.4: This part of the monitor contract implements the payment channel functionality like the generalized payment channels in Section 2.2. The **setstate** function here differs slightly to force the monitor to reveal the pre-image from a conditional transfer. **resolve** allows the customer to forfeit the monitor's deposit.

course, the monitor can also choose not to claim the payment and let the settlement period expire. After expiration the customer can call **resolve** to retrieve his deposit and close the channel.

The fair exchange protocol specified shows that for all the cases where the monitor accepts the conditional payment from the customer, the receipt that was provided becomes valid. There is no way for the monitor to be paid for a new appointment without creating a valid receipt. Similarly, once the conditional payment is signed and sent to the monitor, the customer has already obtained a signed receipt and cannot reverse the payment if it is redeemed by the monitor. Hence, the exchange is only valid if both parties get what they want: a receipt for a payment. Otherwise, neither is valid.

Recourse. If the monitor gets paid during the fair exchange, the pre-image of h_i is always revealed and the receipt validated. When a customer's channel triggers a dispute within the period of the appointment, the monitor is obligated to respond with the correct hashed state. If the monitor does not respond, the customer can submit the signed receipt along with the pre-image of the conditional payment to the **recourse** (Figure 3.5) function to forfeit the monitor's deposit. When **recourse** is called, the monitor's contract checks the closing state and closing time of the outsourced channel against the receipt to determine whether the monitor acted correctly. If not, the monitor's deposit is forfeited and the contract state transitions to `CHEATED`. At this point the customer can call **resolve** to claim its remaining deposit and close the channel.

Closing the Channel. Finally, the monitor can also stop monitoring and prevent new customers from starting new channels. If the contract is in the `OK` state, the monitor can call **stopmonitoring** (Figure 3.5) which transitions the contract from `OK` $\rightarrow$ `CLOSED`. A withdrawal period is also set ($t_{\text{withdraw}} = \texttt{CurTime}() + \Delta_{\text{withdraw}}$), after which the monitor can reclaim the contract deposit, c_M. Despite being closed, however, the monitor is still obligated to respect outstanding customers' valid receipts. An important note on this protocol is that the appointment period, t_{expire}, should always be less than t_{withdraw}. Otherwise, the monitor can reclaim the deposit, c_M, before a customer's receipt expires. The monitor would no longer have to respect the receipt as there is no deposit to forfeit. Prudence is required on the part of the customer to ensure that the monitor is not paid for a receipt that will allow this to happen.

Monitor contract (continued)

```
function stopmonitoring(σ_M):

    if Sig.Verify(M, stop, σ_M)
      set flag := CLOSED
      set t_withdraw := CurTime() + Δ_withdraw
      EventClose(t_withdraw)

function withdraw(σ_M):

    discard if flag = CHEATED
    if Sig.Verify(M, withdraw, σ_M)
      if ID.length = 0 ∧ CurTime() > t_withdraw ∧ flag = CLOSED
        set profit += deposit, deposit := 0
      send profit to M
      set profit := 0
      EventWithdraw()

function recourse(receipt, s_i, σ_M):

    discard if flag = CHEATED
    discard if receipt.h_i ≠ H(s_i)
    set chan := lookup(receipt.chan)
    if Sig.Verify(M, receipt, σ_M)
      for k in chan.disputes.length
        if chan.disputes[k].start > signedreceipt.start
        ∧ signedreceipt.expire > chan.disputes[k].end
        ∧ signedreceipt.i ≥ chan.disputes[k].stateRound
          set flag := CHEATED
          EventForfeit()
```

Figure 3.5: This final part of the monitor contract, allows the customer to take action against a monitor by submitting the receipt received during fair exchange. The **stopmonitoring** allows the monitor to unilaterally close the channel because it will always close on the latest state.

CHAPTER 4

IMPLEMENTATION

In this work, we implemented Pisa as an add-on to an existing payment channel system called μ-Raiden [13]. Although there exist other payment channel systems such as the Lightning Network [9] or Raiden [11], μ-Raiden remains the only one that has been deployed on the Ethereum main network at the time this work began. This chapter starts by providing background on how μ-Raiden works out-of-the box. Next, the modifications made to the existing channel management contract and the new monitor contract implementation are described (the actual smart contract code can be found in Figure A.1). The final section describes the overhead incurred by the added fault tolerance provided by Pisa.

4.1 Background on μ-Raiden

This section describes how μ-Raiden implements a payment channel and the different steps involved in local state authorization and on-chain settlement. The next section builds off this background to describe the modifications required to implement Pisa.

μ-Raiden implements an off-chain payment channel framework that facilitates frequent micropayments of ERC20 [15] tokens between clients and service providers. Some of the use cases for this framework include pay-per-use services where automatic microtransactions act as a replacement for obtrusive advertising and subscription walls [13]. For this reason, it only supports uni-directional payment channels where many clients open channels with one server.

Basic Deployment. A server wanting to accept micropayments first needs to create and deploy a special ERC20 token contract through which channel deposits are made. The server also deploys a contract to manage its payment

channels called `RaidenMicroTransferChannels`. This contract stores a list of the server's open channels and notifies it when on-chain channel events occur. The server continuously listens for contract events or http connections to begin accepting payments and serving content.

4.1.1 Channel Creation

Before requesting paywalled content, the customer must first obtain the underlying token that the channel manager contract uses. In the default setting, the server requires a client to spend at least 100 finney[1] to mint 50 tokens for itself. A client requesting an "expensive"[2] resource creates a `GET` request to the server for the endpoint it wants. If a channel with the server does not already exist, it notifies the client to initialize a channel by transferring tokens to the contract's address. This triggers the channel contracts token fallback procedure which then calls `createChannel`. Both the server and client wait until a confirmed `ChannelCreated` event is detected before proceeding with off-chain payments.

When the channel is created, the contract creates a unique identifying key for the channel:

```
1 // Create unique identifier from sender, receiver and current
      block number
2 bytes32 key = getKey(_sender_address, _receiver_address,
    open_block_number);
```

The contract indexes the channels data with `key` and stores the channel creation block and the initial deposit. When the `ChannelCreated` event is detected by the server, it creates a local copy of the channel in its database and responds to the client's `GET` request indicating that it is ready to accept an off-chain payment.

4.1.2 Making a Payment

To pay the server for a resource the client creates an update to the current state called a balance message proof. The balance message proof encodes the details of the channel such as the sender address, receiver address and open

[1] 1 finney = 0.001 Ether.

[2] Expensive means that the resource requires a micropayment to access it.

```
bytes32 balance_message_hash = keccak256(
    keccak256(
        'string message_id',
        'address receiver',
        'uint32 block_created',
        'uint192 balance',
        'address contract'
    ),
    keccak256(
        'Sender balance proof signature',
        _receiver_address,
        _open_block_number,
        _balance,
        _address(this)
    )
);
```

Figure 4.1: Solidity code used to re-create the `keccak256` hash of the balance message with the receiver address, the creation block number, the newest balance of the channel and the address of the channel manager contract.

block number. Additionally, for every new state the message also encodes the new balance of the channel. The balance message proof is created by the client with the balance field incremented by the value of the payment. The client signs the message and sends it to the server for verification. The precise encoding of the balance message is shown in Figure 4.1.

The implementation deviates from the generalized protocol described in Chapter 3 omitted the state counter. This is because the payment channels in μ-Raiden are uni-directional. Therefore, the balance field in the state is a monotonically increasing value, and is treated as a proxy for the state counter. For example, when settled on-chain, the contract can always distinguish the chronology of two states by looking at their balances. Another consequence of this kind of channel is that the server will always close the channel with the latest balance message that pays it the most. So, the contract allows the server to unilaterally close a payment channel at any time with the latest state.

If the server accepts the balance message, it responds to the client with the resource that it requested. If the message is not valid for any reason, the server will notify the client to try again. For example, the server can notify the client of lacking token funds to back the payment and to resolve

it before attempting another payment. The channel can proceed in this way indefinitely until either party attempts to close the channel and settle the final balance on-chain.

4.1.3 Settling On-Chain

At some point in the life of a μ-Raiden payment channel, one of the parties will attempt to close the channel and withdraw its own funds. If the server requests the channel be closed, the contract fast tracks the request and immediately closes the channel as mentioned in Section 4.1.2. When a client finishes its session with the server, it can do one of two things: 1. it can leave the existing channel open for reuse at a later time, 2. it can attempt to close the channel and claim its funds. In the former case, the client simply disconnects from the server, and the server maintains the channels in its database. In the latter case, the client submits a close request through the `uncooperativeClose` function. The function allows the client to specify any desired balance to close on even if it is not the correct final balance. Hence, it can do so without submitting a signed balance message. The contract then creates and stores a `closing_request` for the channel in question and creates a settlement period in which the server can respond with a newer state: $t_{\text{settle}} = \texttt{CurTime}() + \Delta_{\text{settle}}$. Finally, the contract emits a `ChannelCloseRequested` event which identifies the channel, the settlement timeout and the requested closing balance.

A requested closing balance less than the actual latest balance will prompt the server to submit proof of a larger balance. As mentioned before, the server can unilaterally settle channels. Therefore, the server calls `cooperativeClose` (the function is show in Figure 4.2) with the latest balance message signed by the client along with a closing signature. The closing signature is a signed message from the server that indicates an intent to close the channel. If the balance message is signed by the client and the closing message is signed by the receiver, the contract accepts them and settles the channel. The contract then deletes the channel and `closing_request` and tranfers the appropriate amount of tokens to each party's account.

```
1     function cooperativeClose(
2         address _receiver_address,
3         uint32 _open_block_number,
4         uint192 _balance,
5         bytes _balance_msg_sig,
6         bytes _closing_sig)
7         external
8     {
9         // Derive sender address from signed balance proof
10        address sender = extractBalanceProofSignature(
11            _receiver_address,
12            _open_block_number,
13            _balance,
14            _balance_msg_sig
15        );
16
17        // Derive receiver address from closing signature
18        address receiver = extractClosingSignature(
19            sender,
20            _open_block_number,
21            _balance,
22            _closing_sig
23        );
24        require(receiver == _receiver_address);
25
26        // Both signatures have been verified and the channel
              can be settled.
27        settleChannel(sender, receiver, _open_block_number,
              _balance);
28    }
```

Figure 4.2: Sample code of the cooperative close mechanism in μ-Raiden. The `closing_request` and balance message are checked for correct signatures before the channel is settled, i.e. correct balances are transferred to each party and the channel is deleted.

4.2 Modifications

The main goal of the Pisa is to provide fault tolerance for a specific payment channel failure mode without incurring substantial overhead cost Pisa deployment. Hence, the modifications made to μ-Raiden are minimal, and the on-chain costs incurred by Pisa are compared to the loss incurred by failure.

This section outlines the changes and new features added to μ-Raiden in order to implement the protocol described in Chapter 3. The implementation reflect the capabilities of the monitor to interact with the server's contract and submit evidence for the channels that it is watching. It also implements a new time period in which the monitors of a channel can submit their evidence. Finally, the monitor's contract is created to facilitate a uni-directional channel between customers and the monitor, and allow customers to request recourse if the monitor misbehaves.

For the remainder of this section, the protocol will refer to the monitor's smart contract code (Figure A.1) and other snippets from the code.

Changes to μ-Raiden Contract. The first change made to the channel contract is the addition of a new timer for monitor submission. When a channel close is requested instead of a single t_{settle} deadline, two deadlines are created:

$$t_{\text{monitor}} = \texttt{CurTime}() + \Delta_{\text{monitor}} \tag{4.1}$$

$$t_{\text{settle}} = t_{\text{monitor}} + \Delta_{\text{settle}} \tag{4.2}$$

The first deadline is only relevant to the server in that it has to wait until the deadline has passed to proceed with closing the channel. The contract also provides a mechanism for the monitor to submit evidence when a channel close (Figure 4.3) is requested and for his contract to request channel closing information. In all three of these modifications there is no overhead cost incurred by a server deploying Pisa in both normal operation and in the case of failure as they are only relevant to the monitor.

Monitor Contract The monitor contract closely resembles the pseudo-code algorithm described in Section 3.5. A copy of the actual smart contract is shown in Listing A.1 in Appendix A, and will be referred to often in the rest of the chapter.

```
function monitorEvidence(
    address receiver, uint32 open_block_number,
    bytes32 balance_msg_hash, bytes balance_msg_sig)
    external view
{
    // 's' is correct iff it produces a valid channel key
    address s = ECVerify.ecverify(balance_msg_hash,
        balance_msg_sig);
    bytes32 key = getKey(s, receiver, open_block_number);

    // Make sure this channel and its closing_request exist
    require(channels[key].open_block_number > 0);
    require(closing_requests[key].settle_block_number > 0);

    // Make sure the monitor responds when he is allowed to
    require(closing_requests[key].monitor_block_number >
        block.number);

    // Set the monitor's submitted evidence
    closing_requests[key].evidence = balance_msg_hash;
    closing_requests[key].evidence_sig = balance_msg_sig;

    emit MonitorInterference(s, open_block_number,
        balance_msg_sig);
}
```

Figure 4.3: The `monitorEvidence` function that records the monitor's submitted state hash.

4.3 Modifications to μ-Raiden Off-Chain

Client-Server Communication. The client-server communication remains largely unchanged as Pisa is mainly concerned with the customer-monitor communication. The first step in implementing the monitoring protocol is establishing a seed for the blinding terms used in the state hashes. However, for the purposes of this implementation, a secret sharing scheme was not implemented, but there exist protocols that facilitate public randomness generation between a group of people [16]. The client and server in this case are assumed to have already agreed upon a seed beforehand that will be used for the duration of the channel. Since the purpose of the implementation is to demonstrate overhead cost, the actual randomness used in the implementation is irrelevent as long as it does not affect the on-chain cost.

Server-Monitor Communication. Unlike communication between the server and client, the server-monitor communication is handled by Python's multiprocessing module. This module implements a high-level API for handling sockets. The monitor deploys a listener that constantly listens for new socket connections from a customer. The server, on the other hand, only attempts to connect to the monitor once. From then on, only the monitor is triggered by server messages and never vice versa.

4.4 Customer and Monitor Setup

When the server starts up, it first establishes a connection with the monitor. The monitor sends its contract's address and its own Ethereum address so the potential customer can verify its parameters. Figure 4.4 shows the customer's communication with the monitor and its contract before the connection is finalized. This step requires prudence on the part of the customer to assure that the settlement and withdrawal timers are set such that the monitor can not claim the deposit until all disputes are settled.

The remainder of the implementation exactly follows the protocol proposed in Chapter 3. Therefore, the remainder of this chapter compares the gas and storage costs of Pisa during channel faults and during misbehavior by the monitor.

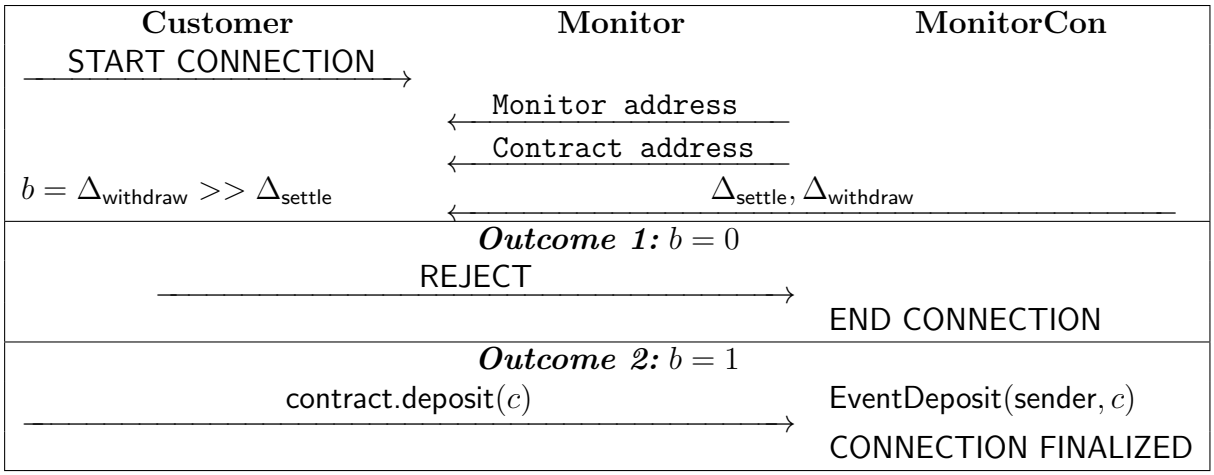

Figure 4.4: The initial communication between the monitor and potential customer. The customer rejects this monitor contract if its parameters are not suitable.

4.5 Cost of Pisa

It is important that the cost of implementing Pisa on top of the uni-directional channel framework does not dissuade participants from using it. Not only will it impact deployment of Pisa, but transaction costs might influence parties in the protocol to deviate from the specification. It is important to measure the on-chain costs in different scenarios and ensure the failure conditions do not significantly impact the participants. Unless otherwise specified, the costs represented by Tables 4.1, 4.2 and 4.3 are in units of Ether. When a dollar amount is presented, it is based on the exchange rate of $402.35 per Ether as reported by EtherScan on Tuesday April 3, 2018 [17]. All operations described below including contract delpoyment take place on the Rinkeby test network.

Contract Deployment. The first cost incurred by the server is the deployment of its contracts augmented with the modifications described in Section 4.2. The `RaidenMicroTransferChannels` contract was augmented with functionality for the monitor to submit hashed states, and for the monitor's contract to extract closing information for recourse. The exact cost of deploying the contract is shown in Table 4.1. This cost is incurred once as the server will deploy and operate only one copy of this contract. For this reason, the small increase in deployment cost is not very important as cost per-channel with many clients will quickly dominate it.

The monitor's cost in deploying his contract in Table 4.1 is a little more than one-third the cost of the server's contract even though they both support uni-directional channels. This large difference in the cost can be explained by two facts. First, the server's contract allows for senders to create one-way channels to *any* address. This means that the same contract can be used by many different server operators. Therefore, in order to access a particular channel the contract needs to index the list of channels with the hash of the sender, receiver and open block number of the channel. Furthermore, the ability to have multiple channels open from the same sender increases the complexity of balance and closing proofs that the contract must verify.

Client-Server In μ-Raiden, the client-server protocol is modified the least. Described in Section 4.2, it is only modified to support blinded nonces in the balance message proofs. The balance message encoding in Figure 4.1 now includes the blinded nonce as part of the message hash. This additional, but marginal, cost is incurred by the server when it closes a channel by submitting the client's balance proof. The additional work in including the blinded nonce is reflected in the row corresponding to `cooperativeClose` in Table 4.2. However, the nonce accounts for only a small part of the difference in cost. The other comes into play during channel settlement.

Monitor Equivocates The fair exchange of the receipt for the customer's payment is critical to ensuring the fairness property described in Section 3.2. In the case where none of the parties in the protocol equivocate, the fair exchange of the monitor's receipt for the customer's conditional payment only incurs payment for appointment. All additional steps required by Pisa are performed off-line for every new state that is outsourced.

The customer, however, incurs additional overhead if the monitor equivocates and does not reveal the pre-image during the exchange. If the monitor tries to close the channel by not revealing the pre-image, the customer must create an on-chain transaction, calling `triggerDispute`, attempting to close the channel with the monitor. However, despite the small cost in triggering a dispute (Table 4.3 function `triggerDispute`), failure in fair exchange can still complete a new appointment if the monitor responds to the dispute by calling `setstate` and redeeming the latest conditional payment on-chain. This reveals the pre-image and makes the latest signed receipt valid. If the monitor does not redeems the payment, the customer resolves the channel once its settlement period had passed. The customer may choose not to re-

solve the channel as the remaining funds are less than the cost of retrieving them. Therefore, at worst, the customer can expect to lose the cost of one transaction in closing the channel with the monitor. Examining the gas cost in Table 4.3, even in the case of monitor equivocation the gas cost of channel resolution is very low while monitor appointment and subsequent recourse is still guaranteed. Furthermore, the added cost of closing a payment channel is not a deterrent to its use, as evidenced by actual payment channel adoption and usage.

Client Settlement Up to this point, the overhead cost to the customer has been discussed in the context of isolated protocol failures in Pisa. This section describes the overhead in the failure condition that Pisa is designed to protect against: an off-line party being cheated by the counter-party. When the server goes off-line for any period of time, it is assumed that at least one monitor was assigned to watch the channel on the server's behalf. This means that prior to going off-line, the server engaged in one or more fair exchanges to outsource the latest state of the channel to a monitor. In the worst case, as described in the previous paragraph, the monitor equivocates and the fair exchange is completed on-chain–yielding minimal overhead to the server. Once off-line, the customer suffers no additional on-chain penalties in closing the channel as the monitor disputes incorrect channel closes on the server's behalf, and the client calls `uncooperativeClose` with any desired balance. The contract emits a `ChannelCloseRequested` event with the appropriate deadline and closing balance information for monitors and channel participants. The monitor responds by submitting the latest hashed state as evidence to the channel such that the channel closes on the latest hashed state given by the customer. After the state closes and the customer comes back on-line, the customer only needs to settle his payment channel to reclaim his funds and send the remainder to the client. If the monitor cheats and does not adhere to the signed receipt, then the customer can call `recourse` in the monitor's contract. Recourse is a particularly expensive computation as it invokes another contract and verifies a signature, however, the larger cost (Table 4.3 row "recourse") is compensated for by the large deposit the customer claims from the monitor's contract.

It is evident from the discussion of the overhead cost of Pisa that in the best case where no parties equivocate, the customer's only added cost comes from creating a channel with the monitor and paying for the appointments.

Table 4.1: A comparison of the cost involved in deploying the contracts for μ-Raiden and for Pisa. The price represented here is 1 Ether = \$402.36 [17]. There is only a marginal increase in the cost of deployment.

	μ-Raiden	Pisa	
Contract	Server	Server	Monitor
CustomToken	0.00326	0.00326	-
RaidenMicroTransferChannels	0.00770	0.00890	-
MonitorContract	-	-	0.00468
Ether	0.01096	0.01216	0.00468
Cost	\$4.4099	\$4.8927	\$1.8830

Table 4.2: The gas costs involved in Pisa when no parties deviate from the protocol. The monitor reveals the pre-image on-line so no interaction with the monitor contract is required.

		μ-Raiden		Pisa		
Phase		Server	Client	Server	Monitor	Client
Channel Creation	`deposit`	-	-	0.00135	-	-
	`createChannel`	-	0.00178	-	-	0.00178
Fair Exchange	`triggerDispute`	-	-	*Pre-image is*		
	`setstate`	-	-	*revealed off-line.*		
Uncoop Close	`uncooperativeClose`	-	0.00107	-	-	0.00120
	`cooperativeClose`	0.00112	-	0.00123	-	-
Monitor Evidence	`monitorEvidence`	-	-	-	0.00355	-
	Total	0.00112	0.00285	0.00258	0.00355	0.00289

This overhead is advantageous as the customer is safeguarding a large channel balance from the client attempting the cheat. The amount that is paid is left to an agreement between the customer and monitor where the customer will only agree on a price that minimized his expected loss from channel disputes. In the worse case, the fair exchange between customer and monitor fails, and the customer must settle his channel with the monitor through an on-chain transaction. This added overhead, however, still guarantees that the customer's channel will at least close on a state that was paid to outsource. Even upon failure of the monitor, *all* of the customer's described overhead costs are compensated for by the monitor's forfeited deposit.

Table 4.3: The first subtotal represents the case where the monitor does *not* reveal the pre-image in the conditional payment off-line, but does attempt to redeem the payment on-chain. This makes the signed receipt valid for recourse. The second subtotal replaces the fair exchange in the first one with the case where the monitor does not reveal the pre-image, and does not redeem the payment on-chain. The final row shows the cost to the customer of requesting recourse from the monitor's contract. The recourse costs are not calculated into a sub total.

Phase		Pisa		
		Server	Monitor	Client
Channel Creation	`deposit`	0.00135	-	-
	`createChannel`	-	-	0.00178
Fair Exchange	`triggerDipsute`	0.00105	-	-
	`setstate`	-	0.00131	-
Uncoop Close	`uncooperativeClose`	-	-	0.00120
	`cooperativeClose`	0.00123	-	-
Monitor Evidence	`monitorEvidence`	-	0.00355	-
	Sub Total	0.00363	0.00486	0.00289
Fair Exchange No Payment	`triggerDispute`	0.00105	-	-
	`resolve`	0.00083	-	-
	Sub Total	0.00446	0.00355	0.00289
Monitor Cheats	`recourse`	0.00131	-	-
	`resolve`	0.00083	-	-

CHAPTER 5

CONCLUSION AND DISCUSSION

Decentralized cryptocurrencies currently face large hurdles in scaling their transaction throughput to match modern payment processors. Payment channels are a promising solution that off-loads the work of creating, authorizing and validating payments between individuals to a local protocol that is executed off-chain by channel participants. Raiden is an experimental proposal for payment channels and payment channel networks on Ethereum, however, it does not have any solution to the liveness requirement of channel participants. The construction of Pisa presented in this thesis removes the requirement by allowing users to appoint third parties to monitor payment channels on their behalf while preserving the privacy of every intermediate state and ensuring a constant storage requirement. Third parties store the hash of the most recent outsourced state and respond to channel closes requests with the most recent state on behalf of their customers. Futhermore, under the assumption of rational and curious actors, the protocol also introduces a novel fair exchange protocol where customers receive signed receipts of appointment which guarantee compensation in case of a cheating monitor.

The practicality of the protocol is also demonstrated by our implementation of Pisa in μ-Raiden, a framework for unidirectional payment channels. The implementation demonstrates the minimal changes that are required to deploy Pisa and the little added on-chain cost to the customer. In the worst-case scenario where a monitor equivocates during the outsouring protocol, the added cost to the customer is only two on-chain transactions that handle closing of a payment channel. Even under equivocation, however, the fair exchange protocol guarantees that monitor's are still obligated to respond to according to any previously exchanged receipt and that the customer is guarantees compensation by the monitor's large deposit. Pisa solves a fundamental problem with current payment channels in ethereum and its construction and implementation in this thesis demonstrates its strong fairness

guarantees and low operating cost.

APPENDIX A

MONITOR'S SMART CONTRACT

Listing A.1: This is the monitor's smart contract written on Solidity. It manages all payment channels to customers and can check for customer recourse.

```
contract StateGuardian {
    event CustomerDeposit(
        address indexed _sender_address,
        uint indexed _sender_deposit
    );
    event Dispute(
        address indexed _customer_address,
        uint32 indexed _open_block_number,
        address indexed _sender_address,
        uint32 _channel_settle
    );
    event Resolve(
        address indexed _sender_address
    );
    event Evidence(
        address indexed _customer_address,
        address indexed _sender_address,
        uint32 indexed _pre_image,
        uint32 _open_block_number
    );
    event Close(
        uint32 _t_withdraw
    );
    event Withdraw();
    event RecourseResult(
        bytes32 indexed _evidence,
        bytes32 _receipt_hash,
        bool indexed _cheated
    );

```

```
31      enum Flags { OK, DISPUTE, CLOSED, CHEATED }
32
33      struct channel {
34          uint deposit;
35          uint32 t_settle;
36          Flags flag;
37          uint payout;
38          RaidenMicroTransferChannels caddr;
39      }
40
41      Flags flag;
42      uint public profit;
43      uint32 public delta_settle;
44      uint32 public delta_withdraw;
45      uint32 t_withdraw;
46      uint public num_customers;
47      uint public guardian_deposit;
48      address public monitor = msg.sender;
49
50      mapping (address => channel) public ID;
51
52      function setup(uint32 _delta_withdraw, uint32
            _delta_settle)
53          payable external
54      {
55          require(msg.sender == monitor);
56          guardian_deposit = msg.value;
57          delta_settle = _delta_settle;
58          delta_withdraw = _delta_withdraw;
59          t_withdraw = 0;
60          flag = Flags.OK;
61      }
62      function deposit(address caddr)
63          payable external
64      {
65          require(msg.value > 0);
66          require(flag == Flags.OK);
67          require(ID[msg.sender].flag != Flags.DISPUTE);
68
69          if (ID[msg.sender].flag == Flags.CLOSED) {
70              num_customers += 1;
71              ID[msg.sender].flag = Flags.OK;
72          }
```

```
73          ID[msg.sender].deposit += msg.value;
74          ID[msg.sender].caddr = RaidenMicroTransferChannels(
                caddr);
75          num_customers += 1;
76          CustomerDeposit(msg.sender, msg.value);
77      }
78      function extract_state_signature(
79          address _sender, uint32 _open_block_number,
80          uint32 _payout, bool _cond_transfer,
81          bytes32 _hash, bytes _signature)
82          internal view
83          returns (address)
84      {
85          bytes32 message_hash = keccak256(
86              keccak256(
87                  'address sender',
88                  'uint32 open block number',
89                  'uint32 payout',
90                  'bool cond_transfer',
91                  'bytes32 hash'
92              ),
93              keccak256(
94                  _sender,
95                  _open_block_number,
96                  _payout,
97                  _cond_transfer,
98                  _hash
99              )
100         );
101         address signer = ECVerify.ecverify(message_hash,
                _signature);
102         return signer;
103     }
104     function setstate(
105         address _sender, uint32 _open_block_number,
106         uint32 _payout, bool _cond_transfer,
107         bytes32 _hash, uint32 _pre_image,
108         bytes _customer_sig, address _customer)
109         view external
110     {
111         require(msg.sender == monitor);
112         require(flag != Flags.CHEATED);
113         require(_payout != ID[_customer].payout);
```

```
114         require(_payout <= ID[_customer].deposit);
115         require(keccak256(_pre_image) == _hash);
116 
117         address customer_signer = extract_state_signature(
118             _sender,
119             _open_block_number,
120             _payout,
121             _cond_transfer,
122             _hash,
123             _customer_sig
124         );
125         require(customer_signer == _customer);
126         profit += _payout;
127         _customer.transfer(ID[_customer].deposit - _payout);
128 
129         ID[_customer].deposit = 0;
130         ID[_customer].t_settle = 0;
131         ID[_customer].flag = Flags.CLOSED;
132         ID[_customer].payout = 0;
133         num_customers -= 1;
134 
135         Evidence(_customer, _sender, _pre_image,
                _open_block_number);
136     }
137     function triggerdispute(address _sender, uint32
            _open_block_number)
138         external
139     {
140         if (ID[msg.sender].flag == Flags.OK) {
141             ID[msg.sender].flag = Flags.DISPUTE;
142             ID[msg.sender].t_settle = uint32(block.number +
                    delta_settle);
143             Dispute(msg.sender, _open_block_number, _sender,
                    ID[msg.sender].t_settle);
144         }
145     }
146     function resolve()
147         external
148     {
149         if (flag == Flags.CHEATED ||
150            (ID[msg.sender].t_settle < block.number &&
151             ID[msg.sender].flag == Flags.DISPUTE))
152         {
```

```
153             msg.sender.transfer(ID[msg.sender].deposit);
154             ID[msg.sender].t_settle = 0;
155             ID[msg.sender].deposit = 0;
156             ID[msg.sender].flag = Flags.CLOSED;
157             ID[msg.sender].payout = 0;
158             num_customers -= 1;
159             Resolve(msg.sender);
160         }
161     }
162
163     function stopmonitoring()
164         external
165     {
166         require(msg.sender == monitor);
167         flag = Flags.CLOSED;
168         t_withdraw = uint32(block.number + delta_withdraw);
169         Close(t_withdraw);
170     }
171     function withdraw()
172         external
173     {
174         require(flag != Flags.CHEATED);
175         require(msg.sender == monitor);
176
177         if (num_customers == 0 &&
178             block.number > t_withdraw &&
179             flag == Flags.CLOSED)
180         {
181             profit += guardian_deposit;
182             guardian_deposit = 0;
183             msg.sender.transfer(profit);
184             profit = 0;
185
186             Withdraw();
187         }
188     }
189     function extractreceiptsignature(
190         address receiver, address sender,
191         uint32 open_block_number, uint32 t_start,
192         uint32 t_expire, bytes32 image,
193         bytes32 balance_message_hash, bytes monitor_sig)
194         returns(address)
195     {
```

```
196         bytes32 receipt_hash = keccak256(
197             keccak256(
198                 'address customer',
199                 'address sender',
200                 'uint32 block created',
201                 'uint32 start time',
202                 'uint32 expire time',
203                 'bytes32 image',
204                 'bytes32 evidence'
205             ),
206             keccak256(
207                 msg.sender,
208                 sender,
209                 open_block_number,
210                 t_start,
211                 t_expire,
212                 image,
213                 balance_message_hash
214             )
215         );
216 
217         address signer = ECVerify.ecverify(receipt_hash,
                monitor_sig);
218         return signer;
219     }
220     function recourse(
221         address sender, uint32 open_block_number,
222         bytes32 image, uint32 t_start,
223         uint32 t_expire, bytes32 balance_message_hash,
224         bytes monitor_sig, uint32 pre_image)
225         external
226     {
227         require(flag != Flags.CHEATED);
228         require(keccak256(pre_image) == image);
229 
230         address signer = extractreceiptsignature(
231             msg.sender,
232             sender,
233             open_block_number,
234             t_start,
235             t_expire,
236             image,
237             balance_message_hash,
```

```
238             monitor_sig
239         );
240         require(signer == monitor);
241         uint32 block_number;
242         bytes32 evidence;
243 
244         (block_number, evidence) = ID[msg.sender].caddr.
                getClosingInfo(sender, msg.sender,
                open_block_number);
245 
246         if (
247             open_block_number <= t_start &&
248             block_number < t_expire &&
249             evidence != balance_message_hash)
250         {
251             flag = Flags.CHEATED;
252             RecourseResult(evidence, balance_message_hash,
                    true);
253         } else {
254             RecourseResult(evidence, balance_message_hash,
                    false);
255         }
256     }
257 }
```

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] Visa, "Visa stress test." [Online]. Available: https://www.visa.com/blogarchives/us/2013/10/10/stress-test-prepares-visanet-for-the-most-wonderful-time-of-the-year/index.html

[3] "Block size limit controversy," 2018. [Online]. Available: https://en.bitcoin.it/wiki/Block_size_limit_controversy

[4] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer et al., "On scaling decentralized blockchains," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 106–125.

[5] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing Bitcoin security and performance with strong consistency via collective signing," in *USENIX Security Symposium*, 2016.

[6] L. Luu, V. Narayanan, K. Baweja, C. Zheng, S. Gilbert, and P. Saxena, "SCP: A computationally-scalable Byzantine consensus protocol for blockchains," in *CCS*, 2016.

[7] R. Pass and E. Shi, "Hybrid consensus: Efficient consensus in the permissionless model," Cryptology ePrint Archive, Report 2016/917, 2016, http://eprint.iacr.org/2016/917.

[8] G. Caffyn, "What is the Bitcoin block size debate and why does it matter?" Aug. 2015. [Online]. Available: https://www.coindesk.com/what-is-the-bitcoin-block-size-debate-and-why-does-it-matter/

[9] J. Poon and T. Dryja, "The Bitcoin lightning network:scalable off-chain instant payments." [Online]. Available: https://lightning.network/lightning-network-paper.pdf

[10] C. Decker and R. Wattenhofer, "A fast and scalable payment network with Bitcoin duplex micropayment channels," in *Symposium on Self-Stabilizing Systems*. Springer, 2015, pp. 3–18.

[11] "The raiden network." [Online]. Available: https://raiden.network/

[12] "Truebit: A scalable verification solution for blockchains." [Online]. Available: https://truebit.io/

[13] "Micro-raiden: A payment channel framework for fast and free off-chain erc20 token transfers." [Online]. Available: https://raiden.network/micro.html

[14] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry, "Sprites: Payment channels that go faster than lightning," *CoRR*, vol. abs/1702.05812, 2017. [Online]. Available: http://arxiv.org/abs/1702.05812

[15] E. Foundation, "Erc20 token standard," https://theethereum.wiki/w/index.php/ERC20_Token_Standard.

[16] I. Cascudo and B. David, "Scrape: Scalable randomness attested by public entities," in *International Conference on Applied Cryptography and Network Security*. Springer, 2017, pp. 537–556.

[17] "Etherscan." [Online]. Available: https://etherscan.io