zkChannels Private Payments Protocol[DRAFT]

Bolt Labs, Inc.

2021-08-29

Contents

Chapter 1. Summary 5

1.1. Overview of zkChannels 6

Chapter 2. Cryptographic Building Blocks and Notation 9

2.1. Preliminaries 10

2.2. Pointcheval Sanders signatures 14

Chapter 3. Off-Network and On-Network Subprotocols 19

3.1. Assumptions and Notation 19

3.2. Sessions 19

3.3. Off-network Channel Protocol Π_{zkAbacus} 20

3.4. On-network Escrow and Disbursement Protocol $\Pi_{\mathsf{zkEscrowAgent}}$ 26

Chapter 4. zkChannels 37

4.1. Preliminaries 37

4.2. System set up 37

4.3. Channel establishment 37

4.4. Channel payments 39

4.5. Channel closing 40

4.6. Discussion 41

4.7. Implementation overview 42

Bibliography 43

Appendix. 45

A. Proof Strategy Overview 45

B. Draft Π_{zkAbacus} ideal functionality 46

C. Draft $\Pi_{\mathsf{zkEscrowAgent}}$ ideal functionality 57

CHAPTER 1

Summary

In this chapter, we provide a high-level overview of the zkChannels protocol and provide an outline of the rest of the document.

1.1. Overview of zkChannels

zkChannels is a flexible protocol that enables anonymous and scalable payments. zkChannels integrates with a wide variety of payment networks, including account-based and even unspent transaction output-based (UTXO) cryptocurrencies, provided these networks meet relatively basic requirements. The zkChannels protocol is inspired by the Lightning Network (LN) and features low-cost, private, off-network transactions via a peer-to-peer network of bidirectional payment channels using cryptographic privacy-preserving techniques. That is, the goal of zkChannels is to provide stronger privacy guarantees than those afforded by current solutions. To do so, we rely on non-interactive zero-knowledge proofs of knowledge (NI-ZKPoK), blind signatures, and commitments. With zkChannels, off-network transactions are inherently unlinkable, and efficient.

A *payment channel* allows two parties to escrow funds in an on-network account and then make payments to each other off chain. An on-network arbitration mechanism allows participants to then close on their most recent balances. Typically, payment channel protocols are *symmetric* with respect to the participants. The zkChannels protocol, however, is *asymmetric*. zkChannels allows a *customer* (C) and a *merchant* (M) to open a bidirectional payment channel with respect to a payment network capable of arbitration, which we refer to as the *arbiter* (J)[1]. As is standard for payment channels, both parties may close the channel at any time and an honest party is guaranteed to be paid at least the balance they are owed. Our protocol, however, achieves privacy at the cost of asymmetry: the customer initiates all payments, each of which is anonymous and cannot be linked to any other payments. That is, the merchant is at most pseudonymous and remains identifiable across all channels; the customer is at most pseudonymous during channel establishment and closure, but has the ability to make payments anonymously *as long as they have an open channel with sufficient balance*. That is, the customer's anonymity set for a payment is the set of all customers with whom the given merchant has a channel open.

Privacy-preserving payment channels are composed of two parts, an off-network mechanism to allow payments and a set of on-network procedures that allow for escrow and arbitration during closure. To achieve a protocol that works for virtually any payment network, we design an off-network payment channel protocol that integrates with a separately-defined arbitration protocol.

In zkChannels, all payments must be initiated by the customer, but both positive and negative payment values are supported. The payment channel consists of a sequence of *states*: the customer always has a special *closing authorization signature* for the current state when they initiate a payment with a current *payment tag*; they then receive a closing authorization signature for the new state, invalidate the old state by providing the merchant a *revocation secret*, and finally receive a payment tag for the new state. From the merchant's perspective, each successful payment results in a revocation secret that allows the merchant to track whether a given state has been spent, but the merchant learns nothing else about the payment except the amount. The merchant is, however, confident that payments are only successful if they are initiated on a valid, unspent state of sufficient balance. The zkChannels protocol itself is agnostic as to the purpose of the payments, but does provide the integration points for request and provision of services, and in particular, we identify at which point of our payment subprotocol a party may reasonably consider a payment *complete*.

At a high level, the arbiter allows funds to be kept in *accounts*. Accounts have *encumbrances* that specify how funds may be disbursed. Accounts may be created, modified, or closed through *transactions*. In the case of a cryptocurrency network, transactions that are *accepted* and/or *confirmed* are public; participants are responsible for monitoring the ledger for relevant transactions. Theoretically, an arbiter without a public ledger may be used, but requires direct communication between the arbiter and channel participants and is not the focus of this paper. The arbiter J is

[1] In practice, we use a cryptocurrency network as the arbiter J

responsible for receiving, validating, and processing transactions. If the arbiter is a payment network such as a cryptocurrency, all zkChannels transactions must be well-formed according to the given payment network's rules; the instantiation must give an exact specification.

From the customer's perspective, they can always initiate channel closure, even if the merchant aborts during the payment process; the only caveat is that a merchant may accept the payment amount without issuing a new payment tag or providing the agreed-upon service, but this is always a risk in customer-merchant relations.

The customer and/or merchant funds the channel with an escrow transaction on the payment network, in such a way that each party is convinced they will be able to close the channel with the correct balances via party-specific closing transactions. That is, the merchant receives a channel-specific closing transaction that spends from the escrow transaction and is valid as long as the channel remains open, and the customer can always form a closing transaction that spends from either the escrow or the merchant closing transaction, as appropriate.

Each payment made on a channel results in an updated closing authorization signature for the customer, which they can use to close down the channel with the most recent balances. If the customer double spends, the merchant can use the revocation secret to craft a dispute transaction that spends from the customer closing transaction. As punishment, if the dispute transaction is deemed valid by the network, the merchant receives the entire channel balance.

A consequence of our dispute mechanism is that both parties must be online or designate a watchtower service to track transactions related to their open channels. A unilateral close by either party triggers a dispute period: if the merchant closes a channel, the customer must respond by posting their own closing transaction with the correct channel balances before the dispute period is over, or all the money in the channel goes to the merchant; similarly, if a customer tries to close with an outdated state, the merchant has to use the corresponding revocation secret before the dispute period is over.

We briefly describe the protocol in four phases:

(1) **System setup.** Each merchant using zkChannels should generate a long-lived keypair for use with all channels.

(2) **Channel initialization and establishment.** The customer generates an asymmetric keypair for use with a single channel, for which the public key is called the *channel public key*. Each channel is associated with a unique channel identifier, which we denote by cid. The parties negotiate initial channel balances and negotiate any needed auxiliary data for on-network transactions. The customer then forms an initial *state* for the channel. This state consists of the channel identifer, the initial customer and merchant balances, and a state-specific value called the *revocation lock*. To establish the channel:
   (a) The merchant provides the customer with a *closing authorization signature* that allows the customer to close down the channel on the initial state balances, provided the funds are escrowed on network in the expected manner.
   (b) The customer provides the merchant with any authorization necessary for the merchant to initiate channel closure on network.
   (c) The parties, each satisfied that they can close down the channel on appropriate balances, fund the channel.
   (d) Once the channel funding has been confirmed on network, the merchant *activates* the channel by issuing the customer a *payment tag*, which the customer will be able to use to make a payment from the initial state. The customer then uses a special *unlinking* protocol to ensure privacy of all payments made on the channel. At this point, the channel is considered established by both parties.

(3) **Channel payments.** Channel payments are initiated by the customer from a current state s and proceed as follows:
   (a) The customer forms and commits to a new state s' and proves that this state is formed from the current state s using the payment amount ϵ. The customer also proves they have

a valid, unspent payment tag for s. The merchant knows ϵ and can reject the payment if the amount is unacceptable.

(b) If the merchant is satisfied, the merchant blindly issues a closing authorization signature on s'. This closing authorization signature allows the customer to initiate channel closure for the balances indicated in s'.

(c) If the customer is satisfied the closing authorization signature is valid, the customer sends a revocation secret on s to the merchant. The revocation secret is linked to the revocation lock for s and allows the merchant to prove double spends and claim the entire channel balance as punishment.

(d) The merchant validates the revocation lock/secret pair and, if satisfied, issues a payment tag on s'. This tag allows the customer to spend from s' in the future and completes the payment protocol.

If the merchant does not provide the requested service for a given payment, the customer must dispute this matter outside of the zkChannels protocol. That is, the customer must close the channel on the most recent state, even if that state reflects a payment made for services the merchant did not render.

(4) **Channel closure.** There are three options for channel closure:

(a) The customer and merchant can collaborate off-network to create a mutual closing transaction. This requires fewer on-network transactions and is therefore cheaper.

(b) The customer can unilaterally initiate channel closure by using their current closing authorization signature to create an on-network customer closing transaction. This transaction:

(i) spends from the escrow transaction,

(ii) allows the merchant balance that is indicated in the associated closing authorization message to be redeemed by the merchant immediately;

(iii) allows the indicated customer balance to be redeemable by the customer after a timeout or by the merchant immediately if a double spend is shown.

That is, the timeout allows the merchant to craft a dispute transaction that proves a double spend, and if the merchant successfully shows a double spend, the effect is that the entire channel balance is paid out to the merchant.

(c) The merchant can unilaterally initiate channel closure by sending their signed merchant closing transaction to the payment network. This transaction pays the entire channel balance to the merchant after a timeout. The timeout allows the customer to dispute the balance allocation: to do so, the customer uses their current closing authorization signature to create a new closing transaction and posts this to the network.

CHAPTER 2

Cryptographic Building Blocks and Notation

2.1. Preliminaries

In this chapter, we describe and establish notation for the core cryptographic primitives useful for understanding and implementing the zkChannels protocol.

For disambiguation among protocols, we reference an algorithm Alg from a scheme Π as $\Pi.\mathsf{Alg}$, unless the underlying scheme is clear from context. We indicate smampling an element s from a set S uniformly at random as $s \xleftarrow{\$} S$. A tuple of ℓ elements of a set S is in the set S^ℓ.

We use multiplicative notation for group operations, so a scalar multiplication of $g \in G$ by a is denoted g^a. A group G has identity element 1_G, and the set of non-identity elements is denoted $G^* = G \backslash \{1_G\}$.

The ring of integers mod q is denoted $\mathbb{Z}_q$.

2.1.1. Commitment Schemes.

Definition 1 (Commitment scheme). A *commitment scheme* Π_{Com} with security parameter λ is a tuple of algorithms $(\mathsf{Setup}, \mathsf{Commit}, \mathsf{Decommit})$, where the following hold:

- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$. The algorithm Setup take a security parameter 1^λ and generates public parameters pp for Π_{Com}, namely a finite message space $\mathcal{X}$, a finite randomness space $\mathcal{R}$, and a finite commitment space $\mathcal{C}$.
- $\mathsf{Commit}(\mathsf{pp}, m, r) \to \mathsf{com}(m; r)$. This algorithm defines a function that takes as input pp, a message $m \in \mathcal{X}$, and an optional randomness parameter $r \in \mathcal{R}$, and outputs a commitment $\mathsf{com}(m; r) \in \mathcal{C}$.
- $\mathsf{Decommit}(\mathsf{pp}, \mathsf{com}, m, r) \to \{\mathsf{false}, \mathsf{true}\}$. On input pp and a tuple (com, m, r), $\mathsf{Decommit}$ outputs either true or false. For all $m \in \mathcal{X}$ and $r \in \mathcal{R}$, this algorithm must satisfy $\mathsf{Decommit}(\mathsf{pp}, \mathsf{Commit}(\mathsf{pp}, m, r), m, r) = \mathsf{true}$.

If $\mathsf{Decommit}(\mathsf{pp}, \mathsf{com}, m, r) = \mathsf{true}$, then we say that the pair (m, r) is an *opening* of the commitment com.

Commitment schemes should satisfy the following properties:

Definition 2 (Hiding). A commitment scheme Π is *information-theoretically hiding* if for all $x \in \mathcal{X}$ and $\mathsf{com} \in \mathcal{C}$, we have $\Pr[x \mid \mathsf{com}] = \Pr[x]$.

Definition 3 (Binding). A commitment scheme Π is *computationally binding* if for all p.p.t. adversaries $\mathcal{A}$, there exists a negligible function negl, such that

$$\Pr[\Pi.\mathsf{Commit}(\mathsf{pp}, x, r) = \Pi.\mathsf{Commit}(\mathsf{pp}, x', r') \wedge x \neq x' | \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda), x, x', r, r' \leftarrow \mathcal{A}(\mathsf{pp})] < \mathsf{negl}(\lambda).$$

We may relax the definition of hiding to the computational setting if desired. Similarly, we may require more stringent security for the binding property and define information-theoretically binding commitments. In practice, a scheme that is computationally both hiding and binding is often sufficient. However, for our purposes, we prefer an information-theoretically hiding commitment scheme.

We remark that commitment schemes may be information-theoretically hiding or information-theoretically binding, but not both. If no randomness parameter r is supplied, the resulting commitment is not information-theoretically hiding. In such a case, we may refer to the commitment of a message $\mathbf{m}$ as $\mathsf{com}(\mathbf{m}; \perp)$.

If the inputs $\mathbf{m}$ and r are clear from context, we sometimes denote the output of Commit as com. or simply $\mathsf{com}(\mathbf{m})$.

Example 2.1 (Pedersen Commitments). We define Pedersen commitments [**Ped92**] over a vector of messages from $\mathbb{Z}_q$, where G is a cyclic group of prime order q. Here the order q is implicitly dependent on the security parameter λ; these parameters must ensure the computational hardness of the discrete logarithm problem in G.

We have the following algorithms:

- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$. The algorithm Setup take a security parameter λ and generates public parameters $\mathsf{pp} = (q, G, h, g_1, g_2, \ldots, g_\ell)$, where q is a prime, G is a cyclic group of order q, and $h, g_1, g_2, \ldots, g_\ell$ are non-identity elements chosen uniformly at random from G (ensuring both that h and g_i for $1 \leq i \leq \ell$ are generators and that discrete logarithm relationships among the chosen generators are unknown). We sometimes use the notation $\mathbf{g} = (g_1, g_2, \ldots, g_\ell)$.
- $\mathsf{Commit}(\mathsf{pp}, \mathbf{m}; r) \to \mathsf{com}(\mathbf{m}; r)$. This algorithm defines a function that takes as input pp, a message $\mathbf{m} = (m_1, m_2, \ldots, m_\ell) \in \mathbb{Z}_q^\ell$, and an optional randomness parameter $r \in \mathbb{Z}_q$, which should be chosen uniformly at random. Commit then outputs a commitment $\mathsf{com}(\mathbf{m}; r) = h^r \prod_{i=1}^{\ell} {g_i}^{m_i}$, which we sometimes denote as com if $\mathbf{m}$ and r are clear from context.
- $\mathsf{Decommit}(\mathsf{pp}, \mathsf{com}, \mathbf{m}, r) \to \{\mathsf{false}, \mathsf{true}\}$. On input pp and a tuple $(\mathsf{com}, \mathbf{m}, r)$, $\mathsf{Decommit}$ checks if $\mathsf{com} \overset{?}{=} h^r \prod_{i=1}^{\ell} {g_i}^{m_i}$, and outputs true if com is a valid commitment to the message $\mathbf{m}$ or false otherwise.

Pedersen commitments have the useful property that they are *homomorphic*: for $\mathbf{m}, \mathbf{m}' \in \mathbb{Z}_q^\ell$ and randomness parameters $r, r' \in \mathbb{Z}_q$, we have $\mathsf{Commit}(\mathbf{m}; r) \cdot \mathsf{Commit}(\mathbf{m}'; r') = \mathsf{Commit}(\mathbf{m} + \mathbf{m}'; r + r')$.

2.1.2. Zero Knowledge Schemes. A *Zero-Knowledge (ZK) proof* is a method that enables a prover to convince a verifier that a given statement is true without revealing any additional secret information. Informally, a *Zero-Knowledge Proof of Knowledge (ZKPoK)* is a special kind of proof where the statement is only that the prover possesses some secret information. The ZKPoK is typically interactive but can made non-interactive. We briefly review the main concepts here.

Zero-Knowledge proofs were first introduced by Goldwasser, Micali, and Rackoff [**GMR85**].

Definition 4 (ZK proof). A *zero-knowledge proof system* for a language L is a pair (P, V), where P is the prover and V the verifier, satisfying

(1) *Completeness*: For all $x \in L$, a verifier V accepts after interacting with the prover P.
(2) *Soundness*: For all $x \notin L$, and for all provers P^*, a verifier V rejects after interacting with P^* with probability at least $\frac{1}{2}$.
(3) *Perfect zero-knowledge*: For all verifiers V^*, there exists a simulator S^* that is a randomized polynomial time algorithm such that for all $x \in L$,

$$\{\mathsf{transcript}((P, V^*)(x))\} = \{S^*(x)\}.$$

A slightly different definition is the *proof of knowledge*:

Definition 5 (ZKPoK). Let x be a statement in the language L and define a relation $R = \{(x, w) : x \in L, w \in W(x)\}$, where $W(x)$ is the set of possible witnesses for x. A *zero-knowledge proof of knowledge (ZKPok)* is a pair (P, V), where P is the prover and V is the verifier, satisfying

(1) *Completeness*: For all $x \in L$ and w a valid witness for x, a verifier $V(x)$ accepts after interacting with the prover $P(x, w)$.
(2) *Validity*: There exists a knowledge extractor E that can extract the witness given access to a possibly malicious prover P^* with probability as high as the probability that P^* convinces an honest verifier. This guarantees that a prover that doesn't know the witness cannot convince an honest verifier.

Definition 6 (ZKAoK). A *zero-knowledge argument of knowledge (ZKAoK)* is a limitation of Definitions 4 and 5 to only allow polynomial-time malicious provers P^* (in the previous definitions, this prover could have been unbounded).

For concreteness, we briefly review the Schnorr protocol [**Sch91**].

Example 2.2 (Schnorr Protocol). Let G be a cyclic group of prime order q, and choose non-identity elements $h, g_1, g_2, ..., g_\ell \in G$ uniformly at random. Assume we have a Pedersen commitment

$A := \mathsf{com}(\mu_1, \ldots, \mu_\ell; \rho) = h^\rho \prod_{i=1}^{\ell} g_i^{\mu_i}$. Standard notation for a proof of knowledge of the opening of such a commitment is

$$\mathsf{PK}\{(\mu_1, \ldots, \mu_\ell, \rho) : A = h^\rho \prod_{i=1}^{\ell} g_i^{\mu_i}\}.$$

When we wish to emphasize the opening secrets $\mu_1, \ldots, \mu_\ell, \rho$ (*i.e.*, because we wish to prove some further properties about these secrets), we instead write

$$\mathsf{PK}\{(\mu_1, \ldots, \mu_\ell, \rho) : A = \mathsf{com}(\mu_1, \ldots, \mu_\ell; \rho)\}.$$

We give the basic honest-verifier zero knowledge Schnorr protocol for a prover P to prove knowledge of the opening of the commitment A to a verifier V:

(1) Commitment phase: The prover P picks $s, t_1, \ldots, t_\ell \in \mathbb{Z}_q$ uniformly at random, calculates $T = h^s \prod_{i=1}^{\ell} g_i^{t_i}$, and sends T to V.
(2) Challenge phase: The verifier V picks $c \in \mathbb{Z}_q$ uniformly at random and sends c to P.
(3) Response phase: The prover P sets $z_0 := c\rho + s$ and $z_i := c\mu_i + t_i$ for $1 \leq i \leq \ell$, then sends $\{z_i\}_{0 \leq i \leq \ell}$ to V.
(4) The verifier V accepts if $h^{z_0} \prod_{i=1}^{\ell} g_i^{z_i} = TA^c$.

More generally, we can call this type of zero-knowledge proof of knowledge a Sigma (Σ-) protocol. Similar to the Schnorr proof these more general protocols consist of three rounds: a commitment phase (1), a challenge phase (2), and a response phase (3).

We can easily make the Schnorr protocol a non-interactive zero knowledge proof of knowledge by applying Fiat-Shamir; that is, we choose a cryptographic hash function H and replace the verifier's input by $c := H(T)$. The Fiat-Shamir transform is a standard tool that transforms an interactive Σ-protocol into a non-interactive one [**FS87**]. The basic idea is to use a cryptographic hash function to hash the commitment from the first phase and then use this hash output as the verifier's challenge. By modeling the hash function as a random oracle, we can prove the soundness and zero-knowledge property of the non-interactive protocol in the Random Oracle Model. We remark that this is a weak form for Fiat-Shamir: if the adversary has the freedom to choose the statement A, this proof system is no longer sound [**BPW12**]. In the strong form of Fiat-Shamir, we include both the statement and the commitment as inputs to the hash function, i.e., $c := H(A, T)$; this is necessary in certain applications.

2.1.3. Signatures.

Definition 7 (Signature scheme). A signature scheme Π_{Sig} consists of the following algorithms:

– $\mathsf{Setup}(1^\lambda)$, which takes as input a security parameter λ and generates the public parameters pp for the scheme, which implicitly define a finite message space $\mathcal{X}$, a finite keypair space $\mathcal{K}$, and a finite signature space $\mathcal{Y}$.

– $\mathsf{KeyGen}(\mathsf{pp})$ is a probabilistic algorithm that takes the public parameters pp as input and outputs a keypair $(pk, sk) \in \mathcal{K}$, where pk is a *public key* and sk is a corresponding *secret key*.

– $\mathsf{Sign}(\mathsf{pp}, sk, m)$ is a (possibly probabilistic) algorithm that takes as input a secret key $sk \in \mathcal{K}$ and message $m \in \mathcal{X}$ and outputs a signature $\sigma \in \mathcal{Y}$.

– $\mathsf{Verify}(\mathsf{pp}, pk, m, \sigma)$ is a deterministic algorithm that takes as input a public key pk, a message m, and a signature σ and outputs either true or false. For every possible (pk, sk) output by KeyGen and every valid message m, this algorithm must satisfy

$$\mathsf{Verify}(\mathsf{pp}, pk, m, \mathsf{Sign}(sk, m)) = \mathsf{true}.$$

For ease of notation, we sometimes omit pp in the description of inputs to Sign and Verify. To initialize the system, run $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$. A signer P generates keys $(pk, sk) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$. To sign a message $m \in \mathcal{X}$, the signer P computes $\sigma \leftarrow (\mathsf{Sign}(\mathsf{pp}, sk, m)$. Anyone may verify P's signature σ on a message m by running $\mathsf{Verify}(\mathsf{pp}, pk, m, \sigma)$.

Definition 8 (Blind signatures). A blind signature scheme Π_{BlindSig} consists of the following algorithms:

- $\mathsf{Setup}(1^\lambda)$, which takes as input a security parameter λ and generates the public parameters pp for the scheme, which implicitly define a finite message space $\mathcal{X}$, a blinded message space $\mathcal{X}'$, a finite keypair space $\mathcal{K}$, a finite signature space $\mathcal{Y}$, and a blinded signature space $\mathcal{Y}'$, and a randomness space $\mathcal{R}$.
- $\mathsf{KeyGen}(\mathsf{pp})$ is a probabilistic algorithm that takes the public parameters pp as input and outputs a keypair $(pk, sk) \in \mathcal{K}$, where pk is a *public key* and sk is a corresponding *secret key*.
- $\mathsf{Blind}(\mathsf{pp}, m, t)$ takes as input a message $m \in \mathcal{X}$ and blinding factor $t \in \mathcal{R}$ and outputs a blinded message $m' \in \mathcal{X}'$.
- $\mathsf{Sign}(\mathsf{pp}, sk, m')$ is a (possibly probabilistic) algorithm that takes as input a secret key $sk \in \mathcal{K}$ and a blinded message $m' \in \mathcal{X}'$ and outputs a blinded signature $\sigma' \in \mathcal{Y}'$.
- $\mathsf{Unblind}(\mathsf{pp}, \sigma', t)$ takes as input a blinded signature $\sigma' \in \mathcal{Y}'$ and a blinding factor $t \in \mathcal{R}$ and outputs an unblinded signature $\sigma \in \mathcal{Y}$.
- $\mathsf{Verify}(\mathsf{pp}, pk, m, \sigma)$ is a deterministic algorithm takes as input a public key pk, a message m, and a signature σ and outputs either true or false.

For ease of notation, we sometimes omit pp in the description of inputs. To initialize the system, run $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$. A signer S generates keys $(pk, sk) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$. A party P can request a blind signature on a message $m \in \mathcal{X}$ by choosing $t \xleftarrow{\$} \mathcal{R}$ and computing $m' \leftarrow \mathsf{Blind}(\mathsf{pp}, m, t)$ and sending m' to the signer S. To sign the blinded message $m' \in \mathcal{X}'$, the signer S computes $\sigma' \leftarrow (\mathsf{Sign}(\mathsf{pp}, sk, m')$ and sends σ' to the requesting party P. To obtain an unblinded signature, party P computes $\sigma \leftarrow \mathsf{Unblind}(\mathsf{pp}, \sigma', t)$. Anyone may verify S's signature σ on a message m by running $\mathsf{Verify}(\mathsf{pp}, pk, m, \sigma)$.

A blind signature scheme Π_{BlindSig} should satisfy the standard properties of a signature scheme and the following properties:

(1) *Correctness*: For every possible (pk, sk) output by KeyGen and every valid message m, Π_{BlindSig} must satisfy

$$\mathsf{Verify}(\mathsf{pp}, pk, m, \mathsf{Unblind}(\mathsf{pp}, \mathsf{Sign}(\mathsf{pp}, sk, \mathsf{Blind}(\mathsf{pp}, m, t)), t)) = \mathsf{true}.$$

(2) *Blindness*: For all $m \in \mathcal{X}$, $m' \in \mathcal{X}'$, we have $\Pr[m \mid m'] = \Pr[m]$.

We sometimes also desire blind signatures schemes to satisfy *randomizability*. In this case, given a signature $\sigma \in \mathcal{Y}$ on a message m, we may produce a valid signature $\sigma' \neq \sigma$ on the same message m, such that $\Pr[\sigma \mid \sigma'] = \Pr[\sigma]$.

2.1.4. Revocation lock schemes.

Definition 9 (Revocation Lock Scheme). We define a *revocation lock scheme*, Π_{Rlock}, as a tuple $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Verify})$, where

- $\mathsf{Setup}(1^\lambda)$ takes a security parameter 1^λ as input and outputs the public parameters pp for the scheme, which implicitly define a finite *revocation lock space*, $\mathcal{RL}$, and a finite *revocation secret space*, $\mathcal{RS}$.
- $\mathsf{KeyGen}(\mathsf{pp})$ is a probabilistic algorithm that takes public parameters pp as input and outputs a *revocation lock pair* $(rl, rs) \in \mathcal{RL} \times \mathcal{RS}$, where rl is a *revocation lock* and rs is a *revocation secret*.

– $\mathsf{Verify}(rl, rs)$ is a deterministic algorithm that takes a revocation lock $rl \in \mathcal{RL}$ and revocation secret $rs \in \mathcal{RS}$ as input and outputs a boolean $b \in \{\mathsf{false}, \mathsf{true}\}$. For every λ and every pair $(rl, rs) \leftarrow \mathsf{KeyGen}(1^\lambda)$, we require $\mathsf{Verify}(rl, rs) = \mathsf{true}$.

Security: For all p.p.t. adversaries $\mathcal{A}$, there exists a negligible function negl such that

$$\Pr[\mathsf{Verify}(rl, rs) = \mathsf{true} \mid (rl, \cdot) \leftarrow \mathsf{Gen}(1^\lambda), rs \leftarrow \mathcal{A}(rl)] < \mathsf{negl}(\lambda).$$

Revocation locks are a formalization of the *hash lock* construction, commonly used in blockchain applications. This implementation instantiates revocation locks as standard hash locks, relying on the collision resistance of some standard cryptographic hash function H. We use H to create revocation locks by defining $\Pi_{\mathsf{Rlock}}.\mathsf{KeyGen}(1^\lambda) = (\mathsf{H}(r), r)$. Similarly, $\Pi_{\mathsf{Rlock}}.\mathsf{Verify}(rl, rs)$ recomputes the hash on rs and compares it to the given rl.

2.2. Pointcheval Sanders signatures

zkChannels uses a *randomizable signature scheme with efficient zero knowledge protocols*, which we denote by Π_{ZKSig}, which features the following additional properties:

(1) an efficient blind signature protocol for a user to obtain a signature on the messages in a commitment without the signer learning anything about the underlying messages;
(2) an efficient protocol for proving knowledge of a signature; and
(3) signatures should be *randomizable*, *i.e.*, given a signature σ on a message m, it should be possibly to construct a new signature σ' for m that is information-theoretically unlinkable to σ in the absence of m.

Examples of ZKSig schemes are bilinear CL signatures, due to Camenisch and Lysyanskaya (CL) [**CL04**] and Pointcheval Sanders (PS) signatures [**PS16, PS18**]. In both of these schemes, the commitment scheme used is a Pedersen commitment with generators determined by the signer's public key. To avoid confusion with general Pedersen commitments (in which generators should be chosen such that no party knows discrete logarithm relationships among them) we will refer to this type of commitment as a Π_{ZKSig}-commitment. Since PS signatures are more efficient and take full advantage of the type 3 pairing setting, we instantiate zkChannels using PS rather than CL signatures. We use the original version of PS signatures [**PS16**] because we require all three of the above properties, whereas the modified approach [**PS18**] sacrifices either full randomizability or the use of efficient protocols for proving knowledge of a signature.[1]These signatures require the following computational assumption:

Definition 10 (PS Assumption). Let G_1, G_2, and G_T be cyclic groups of order $q$that admit a type 3 pairing $e\colon G_1 \times G_2 \to G_T$, and set generators $g \xleftarrow{\$} G_1^*$ and $\tilde{g} \xleftarrow{\$} G_2^*$. For $x, y \in \mathbb{Z}_q^*$ and an element $m \in \mathbb{Z}_q$, define a *PS pair on the tuple* (x, y, m) as a pair (h, h^{x+my}), where $h \in G_1^*$. For randomly generated $x, y \xleftarrow{\$} \mathbb{Z}_q^*$, define an oracle $\mathcal{O}^{(x,y)}$ that takes as input an element $m \in \mathbb{Z}_q$, chooses $h \xleftarrow{\$} G_1^*$, and outputs the PS pair (h, h^{x+ym}). Given $(g, g^y, \tilde{g}, \tilde{g}^x, \tilde{g}^y)$ and unlimited access to the oracle $\mathcal{O}^{(x,y)}$, no adversary can efficiently generate a PS pair on a tuple (x, y, m) for any m not previously input to $\mathcal{O}^{(x,y)}$.

We discuss the efficient protocols and our modifications in more detail in Section 12, but we present the basic signature scheme (without efficient protocols) here first.

[1]That is, the modified approach [**PS18**] requires the signer choose an additional, random element from the message space to be included as part of the signature. A consequence of this is that the signature is no longer randomizable. Randomizability may be recovered by instead setting this additional element deterministically using a hash of the message $\mathbf{m}$ for an appropriate hash function H. However, if both blind signatures and proofs of knowledge of a signature need to be supported, the corresponding PoKs must prove that the pair $(\mathbf{m}, \mathsf{H}(\mathbf{m}))$ is well-formed. This renders the "efficient protocols" inefficient.

Definition 11 (Multi-message PS signatures (Π_{multiPS})). The basic multi-message signature scheme, which we denote by Π_{multiPS}, is as follows.

- $\mathsf{Setup}(1^\lambda)$: On security parameter λ, this algorithm outputs $\mathsf{pp} \leftarrow (q, G_1, G_2, G_T, e)$, where G_1, G_2, G_T are cyclic groups of order q that admit a type 3 pairing $e\colon G_1 \times G_2 \to G_T$.
- $\mathsf{KeyGen}(\mathsf{pp})$: This algorithm selects $\tilde{g} \xleftarrow{\$} G_2^*$, and $(x, y_1, \ldots, y_\ell) \xleftarrow{\$} (\mathbb{Z}_q^*)^{\ell+1}$, and then computes $(\tilde{X}, \tilde{Y}_1, \ldots, \tilde{Y}_\ell) \leftarrow (\tilde{g}^x, \tilde{g}^{y_1}, \ldots, \tilde{g}^{y_\ell})$, and sets $sk = (x, y_1, \ldots, y_\ell)$ and $pk = (\tilde{g}, \tilde{X}, \tilde{Y}_1, \ldots, \tilde{Y}_\ell)$. Note that the public key consists of elements of G_2 and the secret key consists of elements of $\mathbb{Z}_q^*$.
- $\mathsf{Sign}(sk, m_1, \ldots, m_\ell)$: For a message tuple $(m_1, \ldots, m_\ell) \in \mathbb{Z}_q^\ell$, this algorithm selects an element $h \xleftarrow{\$} G_1^*$ and outputs a signature $\sigma \leftarrow (h, h^{x+\sum_{i=1}^{\ell} y_i m_i})$. Note that the message tuple consists of elements in $\mathbb{Z}_q$ and the signature consists of a pair of elements in G_1.
- $\mathsf{Verify}(pk, (m_1, \ldots, m_\ell), \sigma)$: This algorithm parses σ as (σ_1, σ_2) and checks that $\sigma_1 \neq 1_{G_1}$ and $e(\sigma_1, \tilde{X} \cdot \prod_{i=1}^{\ell} \tilde{Y}_i^{m_i}) = e(\sigma_2, \tilde{g})$. It outputs true is both checks pass and false otherwise.

PS signatures are *randomizable*. That is, a party can create a signature σ' that is a *perfectly unlinkable* to σ by choosing $r \xleftarrow{\$} \mathbb{Z}_q$ and setting $\sigma' = (\sigma_1^r, \sigma_2^r)$. The new signature σ' will verify on the same underlying message.

We may achieve blind signatures using Pedersen-like commitments:

Definition 12 (Blind PS signatures (Π_{PS})). We present the PS scheme, which we denote by Π_{PS}, that allows blind signatures with efficient protocols here.

- $\mathsf{Setup}(1^\lambda)$: On security parameter λ, this algorithm outputs $\mathsf{pp} \leftarrow (q, G_1, G_2, G_T, e)$, where G_1, G_2, G_T are cyclic groups of order q that admit a type 3 pairing e.
- $\mathsf{KeyGen}(\mathsf{pp})$: This algorithm selects $g \xleftarrow{\$} G_1^*$, $\tilde{g} \xleftarrow{\$} G_2^*$, and $sk_{\mathsf{multiPS}} = (x, y_1, \ldots, y_\ell) \xleftarrow{\$} (\mathbb{Z}_q^*)^{\ell+1}$. The algorithm then computes $(X, Y_1, \ldots, Y_\ell) \leftarrow (g^x, g^{y_1}, \ldots, g^{y_\ell})$ and $(\tilde{X}, \tilde{Y}_1, \ldots, \tilde{Y}_\ell) \leftarrow (\tilde{g}^x, \tilde{g}^{y_1}, \ldots, \tilde{g}^{y_\ell})$, and sets $sk = X$ and $pk = (g, Y_1, \ldots, Y_\ell, \tilde{g}, \tilde{X}, \tilde{Y}_1, \ldots, \tilde{Y}_\ell)$. The public key pk may be thought of as consisting of a basic multi-message PS public key, namely $pk_{\mathsf{multi}} = (\tilde{g}, \tilde{X}, \tilde{Y}_1, \ldots, \tilde{Y}_\ell)$, and auxiliary information, namely $pk_{\mathsf{aug}} = (g, Y_1, \ldots, Y_\ell)$.
- $\mathsf{BlindSign}(m_1, m_2 \ldots, m_\ell)$: This protocol allows a party to obtain a signature on a message tuple $(m_1, \ldots, m_\ell) \in \mathbb{Z}_q^\ell$ without revealing any information about the message tuple to the signer. The party receives a blinded signature σ', from which they may extract a regular (unblinded) signature σ.
  (1) On input a message tuple $(m_1, m_2, \ldots, m_\ell)$, the party selects $t \xleftarrow{\$} \mathbb{Z}_q$ and computes the commitment $C \leftarrow g^t \prod_{i=1}^{\ell} Y_i^{m_i}$. The party then sends C to the signer, together with a proof of knowledge of the opening of C (which may be interactive). Note that C is a Pedersen commitment defined over the generators in pk_{aug}.
  (2) If the signer accepts the proof of knowledge, they select $u \xleftarrow{\$} \mathbb{Z}_q$, compute $\sigma' \leftarrow (g^u, (XC)^u)$, and send σ' to the requesting party.
  (3) The requesting party outputs (σ', t).
- $\mathsf{Unblind}(\sigma', t)$: This algorithm takes as input a blinded signature σ' and blinding factor t, parses the signature σ' as (σ_1', σ_2'), and computes $\sigma \leftarrow (\sigma_1', \sigma_2'/{\sigma_1'}^t)$.
- $\mathsf{Verify}(pk, (m_1, m_2, \ldots, m_\ell), \sigma)$: This algorithm takes as input an unblinded signature, σ, parses σ as (σ_1, σ_2), and checks that $\sigma_1 \neq 1_{G_1}$ and $e(\sigma_1, \tilde{X} \cdot \prod_{i=1}^{\ell} \tilde{Y}_i^{m_i}) = e(\sigma_2, \tilde{g})$. It outputs true is both checks pass and false otherwise. That is, this algorithm computes $\Pi_{\mathsf{multiPS}}.\mathsf{Verify}(pk_{\mathsf{multi}}, (m_1, m_2, \ldots, m_\ell), \sigma)$ and outputs the result.

To initialize the system, run $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$. A signer S generates keys $(pk, sk) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$. A party P can request a blind signature on a message $(m_1, m_2, \ldots, m_\ell)$ by running the interactive protocol $\mathsf{BlindSign}(m_1, m_2, \ldots, m_\ell)$ with signer S and obtaining a pair (σ', t). Party P then unblinds σ' by computing $\sigma \leftarrow \mathsf{Unblind}(\mathsf{pp}, \sigma', t)$. Anyone may verify S's signature σ on a message $(m_1, m_2, \ldots, m_\ell)$ by running $\mathsf{Verify}(\mathsf{pp}, pk, (m_1, m_2, \ldots, m_\ell), \sigma)$.

Blind PS signatures also support the following procedures, which are useful building blocks for efficient protocols:

– $\mathsf{RandomizeSig}(\sigma)$: This probabilistic algorithm takes as input a signature $\sigma = (\sigma_1, \sigma_2) \in \mathcal{Y}$, chooses $r \overset{\$}{\leftarrow} \mathbb{Z}_q^*$ and outputs (σ_1^r, σ_2^r).

– $\mathsf{Sign}(sk_{\mathsf{multiPS}}, m_1, \ldots, m_\ell) = \Pi_{\mathsf{multiPS}}.\mathsf{Sign}(sk_{\mathsf{multiPS}}, m_1, \ldots, m_\ell)$ for $sk_{\mathsf{multiPS}} = (x, y_1, \ldots, y_\ell)$.

– $\mathsf{BlindSig}(\sigma, t)$: This algorithm takes as input a signature σ and a blinding factor t and outputs a blinded signature $\sigma' = (\sigma_1, \sigma_2 \cdot \sigma_1^t)$.

2.2.1. Proving knowledge of a signature and variants. Let $pk = (\tilde{g}, \tilde{X}, \tilde{Y}_1, \ldots, \tilde{Y}_\ell)$ be a public key for the basic multi-message signature scheme, and let $\sigma = (\sigma_1, \sigma_2)$ be a valid signature on $(m_1, \ldots, m_\ell)$ under pk. The user may prove knowledge of a signature for the verifier by doing the following:

(1) Blind and randomize σ by selecting $r \overset{\$}{\leftarrow} \mathbb{Z}_q^*$ and $t \overset{\$}{\leftarrow} \mathbb{Z}_q$, and computing $\sigma' = (\sigma_1', \sigma_2') = (\sigma_1^r, (\sigma_2 \cdot \sigma_1^t)^r)$.

(2) Send σ' together with a zero-knowledge proof of knowledge π to the verifier, where

$$\pi = \mathsf{PK}\{(m_1, \ldots, m_\ell, t) : e(\sigma_1', \tilde{X}) \cdot \prod_{i=1}^{\ell} e(\sigma_1', \tilde{Y}_i)^{m_i} \cdot e(\sigma_1', \tilde{g})^t = e(\sigma_2', \tilde{g})\}.$$

Note that this proof is a standard proof of knowledge of an opening of the commitment $A = e(\sigma_2', \tilde{g})/e(\sigma_1', \tilde{X}) \in G_T$ for generators $h_0 = e(\sigma_1', \tilde{g})$ and $\{h_i = e(\sigma_1', \tilde{Y}_i)\}_{\{1 \le i \le \ell\}}$. That is, π proves knowledge of an opening $(m_1, \ldots, m_\ell; t)$ such that $A = h_0^t \prod_{i=1}^{j} h_i^{m_i}$.

(3) The verifier should should check that $\sigma_1' \neq 1_{G_1}$ and that π verifies.

We can modify the above protocol to prove additional properties about the underlying signed message $(m_1, \ldots, m_\ell)$. For example, assume we have additional generators $h_1, \ldots, h_{\ell+1} \in G$ chosen uniformly at random, where G is a group of prime order q. We can show that a fresh commitment M (which has not previously been seen by the signer) commits to the same underlying signed message. That is, we can choose $t' \in \mathbb{Z}_q$ uniformly at random, form the Pedersen commitment $M = h_{\ell+1}^{t'} \prod_{i=1}^{\ell} h_i^{m_i}$, and modify the above protocol as follows:

$$\pi = \mathsf{PK}\{(m_1, \ldots, m_\ell, t, t') : e(\sigma_1', \tilde{X}) \cdot \prod_{i=1}^{\ell} e(\sigma_1', \tilde{Y}_i)^{m_i} \cdot e(\sigma_1', \tilde{g})^t = e(\sigma_2', \tilde{g})$$
$$\wedge\, M = h_{\ell+1}^{t'} \prod_{i=1}^{\ell} h_i^{m_i}\}.$$

In the above, we prove knowledge of a valid signature on the tuple of messages $(m_1, \ldots, m_\ell)$ underlying the commitment M. This can be accomplished using the generic linear protocol. We can include additional constraints on the underlying message elements, such as range proofs, or even partially reveal the underlying message. We can also form a commitment M' and prove that M' has underlying secrets with more complicated relationships to the underlying messages, e.g., a linear relationship.

When we wish to give a high-level description of an application of this type of zero knowledge proof for a protocol Π_{ZKSig} (e.g., we wish to avoid including the public key and pairing details), we will sometimes use the following notation for the proof of knowledge π, where $(m_1, \ldots, m_\ell)$ is the underlying message, t is the blinding factor for the signature σ', and t' is the blinding factor for the commitment M:

$$\pi = \mathsf{PK}\{(m_1, \ldots, m_\ell, t, t') : \mathsf{BlindVerify}(pk, (m_1, \ldots, m_\ell, t), \sigma') = \mathsf{true} \\ \wedge M = \mathsf{com}(m_1, \ldots, m_\ell; t')\}.$$

We emphasize that π alone is insufficient for the verifier to be convinced. The verifier needs to perform any additional signature checks required by the underlying scheme Π_{ZKSig}. In the case of PS signatures, the verifier should check that $\sigma'_1 \neq 1_{G_1}$.

2.2.2. Range proofs. In this section, we give an overview of the range proof technique from Camenisch, Chaabouni, and shelat [**CCs08**]. The original paper makes use of Boneh-Boyen signatures [**BB04**]; we use (single-message) Pointcheval-Sanders signatures [**PS16**], described in Section 11 instead. Fix public parameters $\mathsf{pp} \leftarrow (q, G_1, G_2, G_T, e)$ as in Section 11. We will also make use of the (single-message) Pedersen commitment scheme Π_{Com} over G_2 with fixed generators $g, h \in G_2$.

Let A be a commitment to the value μ; that is, $A = \mathsf{com}(\mu; \rho) = g^\mu h^\rho$ for some ρ chosen uniformly at random from $\mathbb{Z}_q$. We are interested in proving that $\mu \in [a, b]$, where $a, b \in \mathbb{N}$, using a zero knowledge proof of knowledge. This corresponds to the following statement:

$$\mathsf{PK}\{(\mu, \rho) : A \leftarrow \mathsf{Commit}(\mu; \rho) \wedge \mu \in [a, b]\}.$$

The range proof technique given by Camenisch, Chaabouni, and shelat [**CCs08**] builds on their set membership protocol, so we provide a brief sketch here. In order to prove that a given, committed element μ belongs to a set Φ, the prover first needs the verifier to send a signature of each element of Φ. The prover can then blind the signature corresponding to μ and use this blind signature to prove in zero knowledge that they possess a signature on the committed element.

The basic range technique treats intervals of the form $[0, u^\ell)$. In order to show $\mu \in [0, u^\ell)$, we first write μ in u-ary notation and then commit to the u-ary representation of μ and prove (in zero knowledge) that each of these digits are members of the set $[0, u-1]$, as per the set membership technique above. If we want to show that the value μ lies within an arbitrary interval $[a, b]$ instead, we choose u, ℓ such that $u^{\ell-1} < b < u^\ell$ and then prove both $\mu - b + u^\ell \in [0, u^\ell)$ and $\mu - a \in [0, u^\ell)$.

The scheme is as follows:

Common Input:

- PS single-message basic signature public parameters $\mathsf{pp} \leftarrow (q, G_1, G_2, G_T, e)$[2];
- Pedersen commitment generators $g, h \in G_2$;[3]
- Range parameters $u, \ell \in \mathbb{N}$;
- Pedersen commitment A.

Prover Input: $\mu, \rho \in \mathbb{Z}_q$ such that $A = g^\mu h^\rho$, where $\mu \in [0, u^\ell)$ and $\rho \in \mathbb{Z}_q$ is chosen uniformly at random.

Protocol:

(1) First, the verifier chooses a PS signing keypair (sk, pk). That is, the verifier selects $x, y \xleftarrow{\$} \mathbb{Z}_q$ and $\tilde{g} \xleftarrow{\$} G_2^*$, computes $(\tilde{X}, \tilde{Y}) \leftarrow (\tilde{g}^x, \tilde{g}^y)$, and then sets $sk = (x, y)$ and $pk = (\tilde{g}, \tilde{X}, \tilde{Y})$.

(2) Next, the verifier signs each integer in $[0, u-1]$. That is, for each integer $i \in [0, u-1]$, the verifier selects $h_i \xleftarrow{\$} G_1^*$ and computes $\sigma_i = (\sigma_{i,1}, \sigma_{i,2}) = (h_i, h_i^{x+y\cdot i})$. Finally, the verifier sends pk together with the signatures $\{\sigma_i\}_{0 \le i \le u-1}$ to the prover.

(3) The prover first verifies that pk is a well-formed public key and that the received signatures $\{\sigma_i\}_{0 \le i \le u-1}$ are valid under pk. That is, $\tilde{g}$, $\tilde{X}$, and $\tilde{Y}$ should all be non-identity elements in G_2.

[2]The prover/verifier should be confident that these parameters are valid, *i.e.*, q is prime, the groups G_1, G_2, and G_T are of prime order q and admit a type 3 pairing e.

[3]These generators should be generated such that no discrete logarithm relationships are known.

(4) The prover writes μ in base u. That is, they determine integer coefficients $\mu_j \in [0, u-1]$ for $0 \le j \le \ell - 1$ such that $\mu = \sum_{j=0}^{\ell-1} \mu_j u^j$.

(5) The prover and verifier perform the following proof of knowledge:

$$\mathsf{PK}\{(\rho, \{\mu_j, \sigma_{\mu_j}\}_{0\le j\le \ell-1}) : A = h^\rho \prod_{j=0}^{\ell-1} (g^{u^j})^{\mu_j} \wedge \{\Pi_{\mathsf{PS}}.\mathsf{Verify}(pk, \mu_j, \sigma_{\mu_j}) = \mathsf{true}\}_{0\le j\le \ell-1}\}.$$

This proof contains a proof of knowledge of the opening μ of the commitment A written in base u with coefficients μ_j, as well as a proof of knowledge of a signature on each of the coefficients μ_j.

CHAPTER 3

Off-Network and On-Network Subprotocols

zkChannels is a composition of an off-network channel protocol, which we denote by Π_{zkAbacus}, and an on-network arbitration protocol, which we denote by $\Pi_{\mathsf{zkEscrowAgent}}$. In this chapter, we present each subprotocol and then define the composition in Chapter 4. For each, we fix a merchant M and a customer C, and then define how to either establish an off-network channel, or open and close an on-network escrow account, respectively.

3.1. Assumptions and Notation

We assume the customer has a way of obtaining and validating a merchant's zkChannels public keys.

We write interactive procedures using the notation $\mathsf{Alg}(in, (in)_C, (in)_M)$, where in denotes shared inputs, *i.e.*, inputs common to both parties, $(in)_C$ denotes customer-specific inputs, and $(in)_M$ denotes merchant-specific inputs. Party-specific inputs are assumed to be secret by default. We use "customer outputs" to denote information the customer receives as a result of the protocol interaction and similarly, "merchant outputs" to denote information the merchant receives as a result of the protocol interaction. Protocol outputs are party-specific and are not shared knowledge between the customer and merchant. We use the notation $(out)_P \leftarrow \mathsf{Alg}()$ to denote that party P receives output out from the algorithm Alg.

3.2. Sessions

We assume the customer and merchant interact over a communication channel, called a *session*, for interactive protocols. Sessions satisfy the following properties:

- Each session satisfies confidentiality and integrity.
- Merchants are identifiable (perhaps pseudonymously) and authenticated by the customer across all sessions, but the merchant identity/pseudonym is not explicitly revealed to third parties.
- Customer identities/pseudonyms are not explicitly revealed to third parties. The protocols Π_{zkAbacus} and $\Pi_{\mathsf{zkEscrowAgent}}$ (and the composed protocol $\Pi_{\mathsf{zkChannels}}$ in Chapter 4) are agnostic as to whether the merchant knows a customer's real-world identity. No customer-specific information that might be linked back to either $\Pi_{\mathsf{zkAbacus}}.\mathsf{Initialize}$ or $\Pi_{\mathsf{zkAbacus}}.\mathsf{Activate}$ is revealed during a $\Pi_{\mathsf{zkAbacus}}.\mathsf{Pay}$ session, and no information that might be linked back to a $\Pi_{\mathsf{zkAbacus}}.\mathsf{Pay}$ session is revealed during $\Pi_{\mathsf{zkAbacus}}.\mathsf{Close}$, so long as the customer closes honestly.

We assume the customer and the merchant interact with the arbiter over a communication channel that satisfies confidentiality. The protocol $\Pi_{\mathsf{zkEscrowAgent}}$ is agnostic as to whether the arbiter knows participants' real-world identities or merely pseudonyms. In practice, the merchant uses the same public keys for zkChannel escrow accounts, so the merchant's escrow accounts on the arbiter ledger are linkable.

3.3. Off-network Channel Protocol Π_{zkAbacus}

The protocol Π_{zkAbacus} is an interactive off-network private payments channel protocol. This is a two-party protocol that provides privacy and correctness of channel state updates and allows the merchant to verify that a customer-provided state is indeed the most recent state for the given channel.

3.3.1. Building Blocks. We use the following building blocks to construct Π_{zkAbacus}:

- A signing primitive Π_{ZKSig} that is a zero-knowledge signature scheme admitting efficient protocols and randomizability of signatures. Further, Π_{ZKSig} must be defined for tuples of messages. Let $\mathcal{X}$ be the underlying message space, $\mathcal{K}$ be the key space, $\mathcal{Y}$ be the signature space, and $\mathcal{R}$ be the randomness space for this primitive. Our use of Π_{ZKSig} has the additional requirement that $\mathcal{X}$ is a group, which we write additively. This signature scheme defines:
  - A commitment scheme Π_{com} with message space $\mathcal{X}$, commitment space $\mathcal{Y}$, and randomness space $\mathcal{R}$.
  - A signature protocol with the standard signature methods, for message space $\mathcal{X}$, signature space $\mathcal{Y}$, and randomness space $\mathcal{R}$.
  - A blind signature protocol with the standard blind signature methods, for message space $\mathcal{X}$, blinded message space $\mathcal{Y}$, blinded signature space $\mathcal{Y}$, and randomness space $\mathcal{R}$.
  - A method to randomize signatures, $\mathsf{RandomizeSig}\colon \mathcal{Y} \to \mathcal{Y}$.
  - A NI-ZKPoK protocol for proving the conjunction of the following types of statements with respect to elements from $\mathcal{X}$:
    - Proving knowledge of openings and partial openings of commitments.
    - Proving knowledge of a signature.
    - Proving equality and linear relationships of committed values.
    - Proving that a committed value lies in a given range.
- A cryptographic, collision-resistant hash function $\mathsf{H}\colon \{0,1\}^* \to \mathcal{X}$.
- A revocation lock scheme Π_{Rlock} defined for revocation lock space $\mathcal{X}$.
- We treat monetary values as integers, *i.e.*, we have an allowed integer range $(-a, a)$ for positive $a \in \mathbb{Z}$. We need a method to encode these allowed monetary values in our message space $\mathcal{X}$ and a method to decode elements in $\mathcal{X}$ as monetary values; these methods should preserve addition in $\mathbb{Z}$ and and the group operation in $\mathcal{X}$. We encode values via a homomorphism $\mu\colon (-a,a) \to \mathcal{X}$ and decode values via a homomorphism $\mu'\colon \mathcal{X} \to \mathbb{Z}$. For readability, we do not surface the usage of μ and μ' in the following high-level description. The implementation should specify how to handle monetary values.[1]

3.3.2. Procedures. The protocol Π_{zkAbacus} uses the following procedures:

- $\mathsf{Setup}(1^\lambda)$: This algorithm takes as input a security parameter 1^λ and outputs pp, which consists of the public parameters for Π_{ZKSig}, H, and Π_{Rlock}.
- $\mathsf{Init}(\mathsf{pp})$: This merchant-run algorithm takes as input public parameters pp and initializes the following databases to $\emptyset$:
  - An *activation database*, which we denote by S_0. This database contains pairs of the form $(x,y) \in (\mathcal{X},\mathcal{Y})$. This can be queried with a value x to retrieve the (unique) entry (x,y), if such an entry exists.[2]
  - A *nonce database*, which we denote by S_1. This database contains elements of the form $n \in \mathcal{X}$, where semantically n is a nonce.

[1] In our implementation, we set allowed monetary values to be $(-2^{63}, 2^{63})$ and use Pointcheval Sanders signatures on BLS12-381, so here $\mathcal{X}$ is the field $\mathbb{Z}_q$ for the pairing groups G_1, G_2, and G_T of prime order q.

[2] Semantically x acts as a channel identifier and y is a verified commitment to the initial state for the channel with identifier x. The channel identifier must be unique.

- A *revocation lock database*, which we denote by S_2. This database contains pairs of the form $(x, y) \in (\mathcal{RL}, \mathcal{RS})$, where $\mathcal{RL} = \mathcal{X}$ and $\mathcal{RS}$ are the revocation lock and revocation secret space defined by Π_{Rlock}, respectively.

The algorithm then runs $(pk, sk) \leftarrow \Pi_{\mathsf{ZKSig}}.\mathsf{KeyGen}$, and outputs the tuple $(S_0, S_1, S_2, (pk, sk))$ (to the merchant).

– $\mathsf{Initialize}(\mathsf{pp}, pk, cid, B_0^C, B_0^M, (S_0, sk)_M)$: This interactive procedure expects, as shared input, a tuple $(\mathsf{pp}, pk, cid, B_0^C, B_0^M)$, where $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$ are public parameters, the key pk is a merchant public key for Π_{ZKSig} and the values $cid, B_0^C, B_0^M \in \mathcal{X}$ are initial channel parameters. Specifically, cid is a unique *channel identifier*, B_0^C is the customer's initial channel balance, and B_0^M is the merchant's initial channel balance.[3] The private merchant inputs are an activation database S_0 and a secret key sk corresponding to pk. There are no private customer inputs. In the following, the element $close \in \mathcal{X}$ is a fixed flag defined by the implementation.

The protocol proceeds as follows:

(1) The customer does the following:
  (a) Chooses a *nonce*, $n_0 \xleftarrow{\$} \mathcal{X} \backslash \{close\}$, and generates a *revocation keypair*, $(rl_0, rs_0) \leftarrow \Pi_{\mathsf{Rlock}}.\mathsf{KeyGen}()$.
  (b) Chooses blinding factor $\tau \xleftarrow{\$} \mathcal{R}$ and forms $A'' = \mathsf{com}_{\mathsf{ZKSig}}(s_0; \tau)$, where $s_0 = (cid, n_0, rl_0, B_0^C, B_0^M)$ is the *initial state*.[4]
  (c) Chooses a blinding factor $\overline{\tau} \xleftarrow{\$} \mathcal{R}$ and forms $A' = \mathsf{com}_{\mathsf{ZKSig}}(\bar{s}_0; \overline{\tau})$, where $\bar{s}_0 = (cid, close, rl_0, B_0^C, B_0^M)$ is the customer's *initial closing state*.[5]
  (d) Generates a proof

$$\begin{aligned}\pi = \mathsf{PK}\{(n_0, rl_0, \tau, \overline{\tau}) : A'' &= \mathsf{com}_{\mathsf{ZKSig}}(cid, n_0, rl_0, B_0^C, B_0^M; \tau) \\ \wedge\, A' &= \mathsf{com}_{\mathsf{ZKSig}}(cid, close, rl_0, B_0^C, B_0^M; \overline{\tau})\}.\end{aligned}$$

  (e) Send A'', A', and π to the merchant.

(2) The merchant does the following:
  (a) Checks that $(cid, \cdot) \notin S_0$; and if this check fails, aborts and output $\perp$.
  (b) Checks that $A'', A', \in \mathcal{Y}$ and that π verifies with respect to A', A'', cid, $close$, B_0^C, and B_0^M; and if this check fails, aborts and outputs $\perp$.
  (c) Otherwise, computes and sends

$$\widehat{\sigma_0} = \Pi_{\mathsf{ZKSig}}.\mathsf{BlindSign}(sk, A')$$

  to the customer.
  (d) Sets $S_0' = S_0 \cup (cid, A'')$ and outputs S_0'.

(3) The customer checks that $\widehat{\sigma_0} \in \mathcal{Y}$, computes

$$\sigma_0 = \Pi_{\mathsf{ZKSig}}.\mathsf{Unblind}(\widehat{\sigma_0}, \overline{\tau}),$$

and checks that

$$\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk, \bar{s}_0, \sigma_0) = \mathsf{true}.$$

If this check fails, they abort and output $(\perp, \perp, \perp, \perp, \perp)$. Otherwise, they output $(\bar{s}_0, s_0, rs_0, \tau, \sigma_0)$.

See also Figure 3.1.

[3]The values of these inputs are agreed on outside of the Π_{zkAbacus} protocol.

[4]A customer *state* is used for tracking current channel balances and, in combination with a merchant signature on the state, making payments.

[5]Each customer state is associated with a *closing state*. Closing states are used for closing channels, in combination with a merchant signature on the closing state. Domain separation between states and closing states is enforced using the special close flag *close*.

- $\mathsf{Activate}(\mathsf{pp}, pk, cid, (s_0, \tau)_C, (S_0, sk)_M)$. This procedure expects a pair (pk, cid) as shared input, where $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$ are public parameters and the key pk is a merchant public key for Π_{ZKSig}. The private merchant inputs are an activation database S_0 and a secret key sk, where sk is the secret key corresponding to pk. The private customer inputs are a state s_0 and a corresponding blinding factor τ. The protocol proceeds as follows:
  (1) The merchant checks for an entry $(cid, A'') \in S_0$. If no such entry exists, they abort and output $\perp$. Otherwise, they compute and send the following to the customer: a *payment tag*, namely $\widetilde{pt}_0 = \Pi_{\mathsf{ZKSig}}.\mathsf{BlindSign}(sk, A'')$. The merchant then outputs a success bit, which we denote by *success*.
  (2) The customer checks that $\widehat{pt}_0 \in \mathcal{Y}$, computes
  $$pt_0 = \Pi_{\mathsf{ZKSig}}.\mathsf{Unblind}(\widehat{pt}_0, \tau),$$
  and checks that
  $$\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk, \bar{s}_0, \sigma_0) = \mathsf{true}.$$
  If this check fails, they abort and output $\perp$. Otherwise, they output pt_0.
- $\mathsf{Pay}(\mathsf{pp}, pk, \epsilon, (s_i, pt_i)_C, (S_1, S_2, sk)_M)$. This interactive protocol expects public parameters $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$, a merchant public key, pk, for Π_{ZKSig}, and a payment amount $\epsilon \in \mathbb{Z}$ as shared input. This protocol allows a customer to make a payment to the merchant on a *state* $s_i = (cid, n_i, rl_i, B_i^C, B_i^M)$ using payment tag pt_i. The merchant inputs are two databases, S_1 and S_2, and a secret key sk corresponding to pk.[6]

  The customer output from this protocol is a tuple containing:
  (1) A new state $s_{i+1} = (cid, n_{i+1}, rl_{i+1}, B_i^C - \epsilon, B_i^M + \epsilon)$ and associated *closing state* $\bar{s}_{i+1} = (cid, close, rl_{i+1}, B_i^C - \epsilon, B_i^M + \epsilon)$.
  (2) An associated payment tag, which we denote by pt_{i+1}, that satisfies
  $$\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk, s_{i+1}, pt_{i+1}) = \mathsf{true};$$
  (3) A *closing authorization signature*, which we denote by σ_{i+1}, that satisfies
  $$\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk, \bar{s}_{i+1}, \sigma_{i+1}) = \mathsf{true};$$
  (4) A payment status indicator $\textit{pay-status}_C$.[7]

  The merchant output from this protocol is a tuple $(S_1', S_2', \textit{pay-status}_M)$, where $S_1' = S_1 \cup \{n_i\}$ and $S_2' = S_2 \cup \{(rl_i, rs_i)\}$ are updated databases, and $\textit{pay-status}_M$ is a payment status indicator.[8]

  We present the details of Pay in Figure 3.2.
- $\mathsf{Close}(\mathsf{pp}, pk, cid, (\bar{s}, \sigma)_C, (S_2)_M)$. This interactive procedure expects public parameters $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$, a merchant public key pk from Π_{ZKSig}, and a channel identifier cid as shared input. Private customer inputs are a closing state $\bar{s} = (cid, close, rl, B^C, B^M)$ and a Π_{ZKSig}-signature σ. Private merchant input is a revocation lock database S_2. The protocol proceeds as follows:
  (1) The customer computes a randomized signature, namely
  $$\sigma' = \Pi_{\mathsf{ZKSig}}.\mathsf{RandomizeSig}(\sigma),$$
  and sends the signature σ', together with $\bar{s}$, to the merchant.

[6] The databases S_1 and S_2 kept by the merchant contain *state revocation information*. The database S_1 contains nonces and the database S_2 contains revocation pairs, *i.e.*, revocation locks together with their corresponding revocation secrets. We stress that in practice, the merchant does not need to track that a given nonce and revocation pair are contained in the same state.

[7] The counter $\textit{pay-status}_C$ is used to specify customer closing logic.

[8] In practice, the counter $\textit{pay-status}_M$ may be used by the merchant to make decisions with respect to the context of the given payment. In particular, success or failure of the payment from the merchant's perspective may depend in a non-trivial way on the value of $\textit{pay-status}_M$.

(2) If $\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk, \bar{s}, \sigma') = \mathsf{true}$ and $(rl, \cdot) \notin S_2$, the merchant adds $(rl, \cdot)$ to S_2 and outputs $(cid, rl, \perp, \mathsf{true})$. Otherwise, the merchant outputs $(\perp, \perp, \perp, \perp)$ if the signature check fails or else $(cid, rl, rs, \mathsf{false})$, if the signature check passes but $(rl, rs) \in S_2$.

3.3.3. Protocol Description. To initialize system parameters, run $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$. A merchant M then runs $(S_0, S_1, S_2, (pk, sk)) \leftarrow \mathsf{Init}(\mathsf{pp})$. To establish a channel together, a customer and the merchant M agree (outside of the protocol) on initial channel parameters consisting of a unique channel identifier cid and initial customer and merchant balances, which we denote by B_0^C and B_0^M, respectively.[9]

The parties each initialize a *status* for the channel by setting $status = \mathsf{uninitialized}$. They then run $\mathsf{Initialize}(\mathsf{pp}, pk, cid, B_0^C, B_0^M, (S_0, sk)_M)$. If this procedure runs successfully they each set $status = \mathsf{initialized}$.

The parties then run $\mathsf{Activate}(\mathsf{pp}, pk, cid, (s_0, \tau)_C, (S_0, sk)_M)$. If this procedure completes successfully, they each set $status = \mathsf{ready}$. These two sessions, $\mathsf{Initialize}$ and $\mathsf{Activate}$, must be linked; that is, the merchant *must* be assured that they are communicating with the same customer in both sessions and that the entry (cid, A'') matches that from the linked $\mathsf{Initialize}$ session.

If the $\mathsf{Activate}$ session is successful, the customer holds a payment tag, namely pt_0, on an initial state $s_0 = (cid, n_0, rl_0, B_0^C, B_0^M)$, and a closing authorization tag, namely σ_0', on the corresponding closing state $\bar{s}_0 = (cid, close, rl_0, B_0^C, B_0^M)$.

The customer can then make a sequence of payments $(p_i)_{i \geq 0}$ to the merchant by initiating the interactive protocol $\mathsf{Pay}(\mathsf{pp}, pk, \epsilon_i, (s_i, pt_i)_C, (S_1, S_2, sk)_M$ (see Figure 3.2) for each payment p_i, so long as the channel status is ready. For a given payment p_i, the customer extracts the resulting $pay\text{-}status_C$ from the output of Pay and does the following:

(1) If $pay\text{-}status_C = \mathsf{revocation\text{-}incomplete}$ or $pay\text{-}status_C = \mathsf{revocation\text{-}complete}$, the customer sets $status = \mathsf{frozen}$.
(2) If $pay\text{-}status_C = \mathsf{state\text{-}updated}$, then the customer may continue to make additional payments on the channel.

When the customer wishes to close the channel, they initiate $\mathsf{Close}(\mathsf{pp}, pk, cid, (\bar{s}, \sigma)_C, (S_2)_M)$ on the most recent closing state $\bar{s}$ and corresponding closing authorization signature σ. The customer should then mark the channel as closed. Similarly, if the merchant knows how to contact the customer, they can request the customer participate in a Close session. The merchant should mark the channel as closed if their output is $(cid, rl, \perp, \mathsf{true})$ and they may flag the channel (and corresponding customer, if appropriate) as corrupted if their output is $(cid, rl, rs, \mathsf{false})$. Outputs of $(\perp, \perp, \perp, \perp)$ are ignored, unless the merchant knows the identity of the customer and the Close session is authenticated for both parties, in which case the merchant may flag the customer as corrupt.

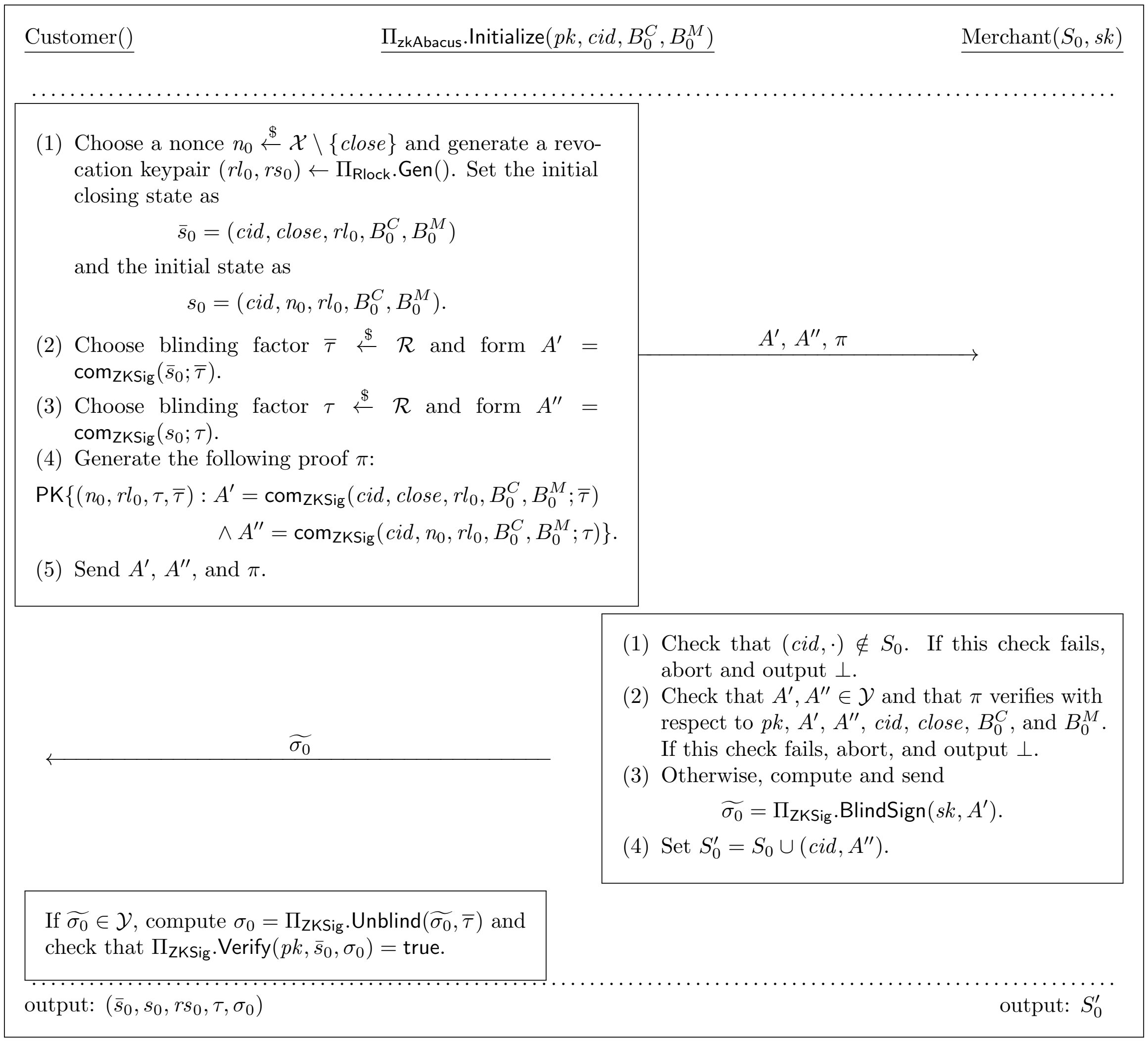
3.3.4. Discussion. The protocol Π_{zkAbacus} is the component of zkChannels that allows channel state to be kept and updated privately by the customer. As such, this protocol is primarily a *bookkeeping protocol* that provides correctness and privacy of state updates, and allows the two parties to cooperatively close the channel. The protocol as sketched is a two-party protocol that does not directly handle adversarial closes, but we do provide the merchant with some recourse in the event the customer sends an outdated state to the merchant in $\mathsf{Close}()$:

(1) The merchant knows which channel the customer wishes to close.
(2) The merchant knows whether or not the provided state is outdated.
(3) If the state is outdated, the merchant knows the corresponding revocation pair.
(4) Revocation pairs are verifiable by any party, not just protocol participants.

This construction provides the foundation on which we may extend this protocol to define zkChannels, which includes arbitration for the disbursement of escrowed funds backing the Π_{zkAbacus} channel.

However, Π_{zkAbacus} channels themselves cannot be closed by the merchant and require honest participation by the customer to close. If the customer sends an outdated state to close, the channel

[9]All channel parameters should be elements of $\mathcal{X}$.



Customer() $\Pi_{\mathsf{zkAbacus}}.\mathsf{Initialize}(pk, cid, B_0^C, B_0^M)$ Merchant(S_0, sk)

(1) Choose a nonce $n_0 \xleftarrow{\$} \mathcal{X} \setminus \{close\}$ and generate a revocation keypair $(rl_0, rs_0) \leftarrow \Pi_{\mathsf{Rlock}}.\mathsf{Gen}()$. Set the initial closing state as

$$\bar{s}_0 = (cid, close, rl_0, B_0^C, B_0^M)$$

and the initial state as

$$s_0 = (cid, n_0, rl_0, B_0^C, B_0^M).$$

(2) Choose blinding factor $\overline{\tau} \xleftarrow{\$} \mathcal{R}$ and form $A' = \mathsf{com}_{\mathsf{ZKSig}}(\bar{s}_0; \overline{\tau})$.

(3) Choose blinding factor $\tau \xleftarrow{\$} \mathcal{R}$ and form $A'' = \mathsf{com}_{\mathsf{ZKSig}}(s_0; \tau)$.

(4) Generate the following proof π:

$$\mathsf{PK}\{(n_0, rl_0, \tau, \overline{\tau}) : A' = \mathsf{com}_{\mathsf{ZKSig}}(cid, close, rl_0, B_0^C, B_0^M; \overline{\tau}) \wedge A'' = \mathsf{com}_{\mathsf{ZKSig}}(cid, n_0, rl_0, B_0^C, B_0^M; \tau)\}.$$

(5) Send A', A'', and π.

$\xrightarrow{A',\ A'',\ \pi}$

(1) Check that $(cid, \cdot) \notin S_0$. If this check fails, abort and output $\bot$.

(2) Check that $A', A'' \in \mathcal{Y}$ and that π verifies with respect to pk, A', A'', cid, $close$, B_0^C, and B_0^M. If this check fails, abort, and output $\bot$.

(3) Otherwise, compute and send

$$\widetilde{\sigma_0} = \Pi_{\mathsf{ZKSig}}.\mathsf{BlindSign}(sk, A').$$

(4) Set $S_0' = S_0 \cup (cid, A'')$.

$\xleftarrow{\widetilde{\sigma_0}}$

If $\widetilde{\sigma_0} \in \mathcal{Y}$, compute $\sigma_0 = \Pi_{\mathsf{ZKSig}}.\mathsf{Unblind}(\widetilde{\sigma_0}, \overline{\tau})$ and check that $\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk, \bar{s}_0, \sigma_0) = \mathsf{true}$.

output: $(\bar{s}_0, s_0, rs_0, \tau, \sigma_0)$ output: S_0'

Figure 3.1. $\Pi_{\mathsf{zkAbacus}}.\mathsf{Initialize}$

remains open. This is because the merchant does not have enough information to determine (or prove) the current channel state in the event of a malicious customer close, *i.e.*, we do not provide a mechanism for *state reconciliation*. We leave extensions of Π_{zkAbacus} that allow for state reconciliation as future work.

3.3.5. Implementation details. The instantiation of the scheme Π_{ZKSig} impacts the structure of the zero knowledge proofs. If we use Pointcheval Sanders (Π_{PS}), we are working over the group $\mathbb{Z}_q$; that is, the values committed to in a state as well as payment amounts will be interpreted as values in $\mathbb{Z}_q$, so we must be careful how we encode positive and negative values and show (in zero knowledge) that values are in the expected range when making payments.

In particular, during system setup, $\Pi_{\mathsf{zkAbacus}}.\mathsf{Setup}(1^\lambda)$ runs $\Pi_{\mathsf{PS}}.\mathsf{Setup}(1^\lambda)$, which outputs $\mathsf{pp}_{\mathsf{PS}} = (q, G_1, G_2, G_T, e)$, where G_1, G_2, and G_T are cyclic groups of order q that admit the type 3 pairing e.[10] These parameters $\mathsf{pp}_{\mathsf{PS}}$ are then included in the public parameters pp for Π_{zkAbacus}.

[10]As a notational reminder, we use $G_1^* = G_1 \backslash \{1_{G_1}\}$ and $G_2^* = G_2 \backslash \{1_{G_2}\}$ to denote the set of non-identity elements of G_1 and G_2, respectively.

Customer(s_i, pt_i) $\Pi_{\mathsf{zkAbacus}}.\mathsf{Pay}(\epsilon, pk)$ Merchant(S_1, S_2, sk)

state: $s_i = (cid, n_i, rl_i, B_i^C, B_i^M)$

Customer:

(1) Initialize *pay-status*$_C$ to $\mathsf{uninitialized}$. Generate a revocation keypair $(rl_{i+1}, rs_{i+1}) \leftarrow \Pi_{\mathsf{Rlock}}.\mathsf{Gen}()$ and a nonce $n_{i+1} \xleftarrow{\$} \mathcal{X}$. Set the new state as

$$s_{i+1} = (cid, n_{i+1}, rl_{i+1}, B_i^C - \epsilon, B_i^M + \epsilon)$$

and the new closing state as

$$\bar{s}_{i+1} = (cid, close, rl_{i+1}, B_i^C - \epsilon, B_i^M + \epsilon).$$

(2) Prepare the following commitments:
  (a) Choose commitment randomness ρ_i and form $A = \mathsf{com}(rl_i; \rho_i)$.
  (b) Choose blinding factor $\overline{\tau}_{i+1}$, and form $A' = \mathsf{com}_{\mathsf{ZKSig}}(\bar{s}_{i+1}; \overline{\tau}_{i+1})$.
  (c) Choose blinding factor τ_{i+1}, and form $A'' = \mathsf{com}_{\mathsf{ZKSig}}(s_{i+1}; \tau_{i+1})$.

(3) Generate the following proof π:

$$\begin{aligned}\mathsf{PK}\{(cid, B_i^C, B_i^M, rl_i, n_{i+1}, rl_{i+1}, \tau_{i+1}, \overline{\tau}_{i+1}, \rho_i, pt_i) :\\ \Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk, (cid, n_i, rl_i, B_i^C, B_i^M), pt_i) = \mathsf{true}\\ \wedge A = \mathsf{com}(rl_i; \rho_i)\\ \wedge A' = \mathsf{com}_{\mathsf{ZKSig}}(cid, close, rl_{i+1}, B_i^C - \epsilon, B_i^M + \epsilon; \overline{\tau}_{i+1})\\ \wedge A'' = \mathsf{com}_{\mathsf{ZKSig}}(cid, n_{i+1}, rl_{i+1}, B_i^C - \epsilon, B_i^M + \epsilon; \tau_{i+1})\\ \wedge 0 \le (B_i^C - \epsilon) \wedge 0 \le (B_i^M + \epsilon)\}.\end{aligned}$$

(4) Send commitments, n_i, and π. Set *pay-status*$_C$ = $\mathsf{revocation\text{-}incomplete}$.

$$\xrightarrow{n_i,\ \pi,\ A,\ A',\ A''}$$

Merchant:

(1) Initialize *pay-status*$_M$ to $\mathsf{uninitialized}$. Then:
  (a) Check $n_i \in \mathcal{X}$ and $n_i \notin S_1$; and
  (b) Check that π verifies with respect to n_i, A, A', A'', ϵ, and pk.

If either check fails, abort, and output $(S_1, S_2, \textit{pay-status}_M)$.

(2) Otherwise, compute and send

$$\widehat{\sigma_{i+1}} = \Pi_{\mathsf{ZKSig}}.\mathsf{BlindSign}(sk, A').$$

Set $S_1' = S_1 \cup \{n_i\}$, set *pay-status*$_M$ = $\mathsf{revocation\text{-}incomplete}$, and continue.

$$\xleftarrow{\widehat{\sigma_{i+1}}}$$

Customer:

(1) If $\widehat{\sigma_{i+1}} \in \mathcal{Y}$, compute $\sigma_{i+1} = \Pi_{\mathsf{ZKSig}}.\mathsf{Unblind}(\widehat{\sigma_{i+1}}, \overline{\tau}_{i+1})$ and check that $\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk, \bar{s}_{i+1}, \sigma_{i+1}) = \mathsf{true}$. If not, abort and output $(\perp, \perp, \perp, \textit{pay-status}_C)$.

(2) Send rl_i, rs_i, and ρ_i, and set *pay-status*$_C$ = $\mathsf{revocation\text{-}complete}$.

$$\xrightarrow{rl_i,\ rs_i,\ \rho_i}$$

Merchant:

(1) Verify the following:
  (a) $(rl_i, \cdot) \notin S_2$;
  (b) $\Pi_{\mathsf{com}}.\mathsf{Decommit}(A, rl_i, \rho_i) = \mathsf{true}$; and
  (c) $rl_i = \mathsf{H}(rs_i)$.

(2) If not all checks verify, abort and output $(S_1', S_2, \textit{pay-status}_M)$.

(3) Otherwise, compute and send

$$\widehat{pt_{i+1}} = \Pi_{\mathsf{ZKSig}}.\mathsf{BlindSign}(sk, A'').$$

Set $S_2' = S_2 \cup \{(rl_i, rs_i)\}$, set *pay-status*$_M$ = $\mathsf{revocation\text{-}complete}$, and output $(S_1', S_2', \textit{pay-status}_M)$.

$$\xleftarrow{\widehat{pt_{i+1}}}$$

Customer:

If $\widehat{pt_{i+1}} \in \mathcal{Y}$, compute $pt_{i+1} = \Pi_{\mathsf{ZKSig}}.\mathsf{Unblind}(\widehat{pt_{i+1}}, \tau_{i+1})$ and check that $\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk, s_{i+1}, pt_{i+1}) = \mathsf{true}$. If this check passes, set *pay-status*$_C$ = $\mathsf{state\text{-}updated}$. Otherwise, abort and output $(s_{i+1}, \sigma_{i+1}, \perp, \textit{pay-status}_C)$.

output: $(s_{i+1}, \sigma_{i+1}, pt_{i+1}, \textit{pay-status}_C)$

output: $(S_1', S_2', \textit{pay-status}_M)$

Figure 3.2. $\Pi_{\mathsf{zkAbacus}}.\mathsf{Pay}$

The merchant M then runs $\Pi_{\mathsf{zkAbacus}}.\mathsf{Init}(\mathsf{pp})$. As part of this algorithm, a PS-public key pair, $(pk, sk) \xleftarrow{\$} \Pi_{\mathsf{PS}}.\mathsf{KeyGen}(\mathsf{pp}_{\mathsf{PS}})$ is generated, where

$$pk = (g, Y_1, \dots, Y_5, \widetilde{g}, \widetilde{X}, \widetilde{Y}_1, \dots, \widetilde{Y}_5),$$

and

$$sk = (x, y_1, \dots, y_5, X).$$

Here $g \in G_1^*$ and $\widetilde{g} \xleftarrow{\$} G_2^*$ are generators, the tuple $(x, y_1, \dots, y_5) \xleftarrow{\$} \mathbb{Z}_q^6$ forms the secret key for signing unblinded messages, and $X \leftarrow g^x$ forms the secret key for signing blinded messages.[11] The corresponding public key is then set using $(Y_1, \dots, Y_5) \leftarrow (g^{y_1}, \dots, g^{y_5})$ and using $(\widetilde{X}, \widetilde{Y}_1, \dots, \widetilde{Y}_5) \leftarrow (\widetilde{g}^x, \widetilde{g}^{y_1}, \dots, \widetilde{g}^{y_5})$.

The proof π in $\Pi_{\mathsf{zkAbacus}}.\mathsf{Pay}$ (see Figure 3.2) then becomes:

$$\begin{aligned}
\mathsf{PK}\{&(cid, B_i^C, B_i^M, rl_i, n_{i+1}, rl_{i+1}, \tau_{i+1}, \overline{\tau}_{i+1}, \rho_i, pt_i) : \\
&\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk, (cid, n_i, rl_i, B_i^C, B_i^M, \tau_i), pt_i) = \mathsf{true} \\
&\wedge C = \mathsf{com}(rl_i; \rho_i) \\
&\wedge C' = \mathsf{com}_{\mathsf{ZKSig}}(cid, n_{i+1}, rl_{i+1}, B_i^C - \epsilon, B_i^M + \epsilon; \tau_{i+1}) \\
&\wedge C'' = \mathsf{com}_{\mathsf{ZKSig}}(cid, close, rl_{i+1}, B_i^C - \epsilon, B_i^M + \epsilon; \overline{\tau}_{i+1}) \\
&\wedge (B_i^C - \epsilon), (B_i^M + \epsilon) \in [0, \mathsf{val}_{\mathsf{max}})\},
\end{aligned}$$

where $\mathsf{val}_{\mathsf{max}}$ is related to Π_{ZKSig} and the allowed integer range for monetary values. For Pointcheval Sanders signatures, the message space is the group $\mathbb{Z}_q$. To encode a monetary value expressed as an integer $x \in (-a, a)$ for an appropriate, positive $a \in \mathbb{Z}$, we map x to its congruence class modulo q.

Setting $a = \frac{q-1}{2}$ is a natural choice from a theoretical perspective: $\mathbb{Z}_q$ values in the range $[0, \frac{q-1}{2})$ are interpreted as positive and values in the range $[\frac{q-1}{2}, q-1]$ as negative. This choice of a is certainly sufficiently large for encoding balances, but impractical. Instead, we keep the above encoding and interpretation of positive and negative, but set $a = 2^{63}$. We then set $\mathsf{val}_{\mathsf{max}}$, the upper bound for valid balances, as $\mathsf{val}_{\mathsf{max}} = a$. In our implementation, we use the technique from Camenisch, Chaabouni, and shelat [**CCs08**] with PS signatures for the range proofs, so we write $\mathsf{val}_{\mathsf{max}} = u^\ell$, where $u = 128$ and $\ell = 9$ in our current implementation. Details about the range proof technique we use are in Chapter 2, Section 2.2.2. We remark that our range proofs require the use of single-message PS signatures on all possible range values with a merchant-chosen keypair. This can be done each time we run $\Pi_{\mathsf{zkAbacus}}.\mathsf{Pay}$, but this is not efficient. Instead, we can have the merchant choose and publish the range proof parameters and signatures at system setup; all customers then make use of the same list of signatures for the range proofs used in $\Pi_{\mathsf{zkAbacus}}.\mathsf{Pay}$ sessions. The merchant should not use this keypair for any other purpose.

We refer the reader to Chapter 2, Section 12 for a more detailed description of the verification step for pt_i in the above zero knowledge proof of knowledge.

3.4. On-network Escrow and Disbursement Protocol $\Pi_{\mathsf{zkEscrowAgent}}$

The protocol $\Pi_{\mathsf{zkEscrowAgent}}$ specifies how a customer and a merchant can request an arbiter, or payment network, to act as escrow agent for zkChannel funds, and defines a set of algorithms the arbiter should run to achieve the desired escrow and disbursement properties. In practice, the arbiter algorithms are realized in the context of an account-based cryptocurrency via a *smart contract* specification, and parties request the arbiter run algorithms via *transactions* that create and call into the smart contract.

[11]We include both pieces as part of sk for convenience.

3.4.1. Building blocks and notation. In this section, we introduce an abstraction of the arbiter, or payment network, that will be responsible for escrowing and distributing channel funds. In practice, we expect the arbiter to be a cryptocurrency, although other types of payment networks, such as trusted third parties, may certainly be used.

An *arbiter* (J) is a funds-controlling entity that allows for movement of funds according to pre-specified authorization rules. Additionally, the arbiter keeps *time*. Specific funds (and their associated authorization rules) are associated with *accounts*: an account consists of a balance, `bal`, and a set of encumbrances, encumb, which specify how movement of funds may be authorized. The arbiter associates each account with a unique identifier. The arbiter's job is to enforce account encumbrances and move funds between accounts as authorized.

We use the notation $J[P]$ to denote the set of accounts for which a party P holds the necessary credentials to move funds; sometimes we say an account with identifier *acct-id* that satisfies $\textit{acct-id} \in J[P]$ is *owned* by P. As shorthand, we sometimes write $\textit{acct-id}_P$ to emphasize account ownership. We assume the arbiter keeps an ordered, public ledger of accepted requests.[12]

The arbiter should implement the following cryptographic primitives:

- A signature scheme Π_{Sig}.
- A revocation lock scheme Π_{Rlock}.
- The algorithm $\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}$ for a ZK-signature scheme Π_{ZKSig}.

If we wish to emphasize the cryptographic primitives implemented by the arbiter, we may write $J^{(\Pi_{\mathsf{Sig}}, \Pi_{\mathsf{Rlock}}, \Pi_{\mathsf{ZKSig}}.\mathsf{Verify})}$.

3.4.2. Arbiter algorithms. The on-network arbitration protocol $\Pi_{\mathsf{zkEscrowAgent}}$ relies on the following procedures run by the arbiter J:

- $\mathsf{Register}(\textit{acct-id}_C, \textit{acct-id}_M, pk_C, pk_M, pk'_M, cid, B_0^C, B_0^M)$. This algorithm takes the following elements as inputs, which we call *channel registration parameters*:
  1. Account information $\textit{acct-id}_C \in J[C]$. This input is the customer's *funding and disbursement account identifier*.
  2. Account information $\textit{acct-id}_M \in J[M]$. This input is the merchant's *funding and disbursement account identifier*.
  3. Customer public key $pk_C \in \Pi_{\mathsf{Sig}}.\mathsf{KeyGen}()$.
  4. Merchant public key $pk_M \in \Pi_{\mathsf{Sig}}.\mathsf{KeyGen}()$.
  5. Merchant closing authorization key $pk'_M \in \Pi_{\mathsf{ZKSig}}.\mathsf{KeyGen}()$.
  6. Channel identifier cid.
  7. Customer balance B_0^C.
  8. Merchant balance B_0^M.

  If the above inputs are well-formed, this algorithm computes a *contract identifier*, which we denote by $J[cid]$, and records the $\mathsf{Register}(\textit{acct-id}_C, \textit{acct-id}_M, pk_C, pk_M, pk'_M, cid, B_0^C, B_0^M)$ call on its ledger, indexed by the contract identifier $J[cid]$. Otherwise the algorithm aborts.
- $\mathsf{Fund}(J[cid], role, \sigma)$. This algorithm takes as input a contract identifier, a flag $role \in \{C, M\}$, and a signature $\sigma \in \Pi_{\mathsf{Sig}}.\mathsf{Sign}()$. The algorithm retrieves the registration parameters associated to $J[cid]$, namely $(\textit{acct-id}_C, \textit{acct-id}_M, pk_C, pk_M, pk'_M, cid, B_0^C, B_0^M)$, and checks that the signature σ is a valid Π_{Sig}-signature that authorizes a transfer of amount B_0^{role} from $\textit{acct-id}_{role}$ to the contract $J[cid]$, and that $\textit{acct-id}_{role}$ contains at least B_0^{role}. If this check passes, the algorithm records the $\mathsf{Fund}(J[cid], role, \sigma)$ call on its (ordered) ledger, indexed by the contract identifier $J[cid]$, and updates the balance in $J[cid]$ to include B_0^{role} additional funds and subtracts B_0^{role} funds from the balance in $\textit{acct-id}_{role}$. Otherwise, the algorithm aborts and outputs $\bot$.

[12]If a non-cryptocurrency arbiter is used, this may be replaced by direct messaging between the arbiter and the parties regarding account transactions.

– $\mathsf{Disburse}(J[cid], B^C, B^M, (rl, \sigma_{\mathsf{ZKSig}}, \sigma^C_{\mathsf{Sig}}), (rs, \sigma^M_{\mathsf{Sig}}))$ is an algorithm that takes as input a contract identifer $J[cid]$, a customer balance B^C, a merchant balance B^M, and one of the following types:

(A) a pair of signatures $\sigma^C_{\mathsf{Sig}}, \sigma^M_{\mathsf{Sig}} \in \Pi_{\mathsf{Sig}}.\mathsf{Sign}()$;

(B) a tuple containing a revocation lock rl, a signature $\sigma_{\mathsf{ZKSig}} \in \Pi_{\mathsf{ZKSig}}.\mathsf{Sign}()$, and signature $\sigma^C_{\mathsf{Sig}} \in \Pi_{\mathsf{Sig}}.\mathsf{Sign}()$ and an optional pair $(rs, \sigma^M_{\mathsf{Sig}})$ containing a revocation secret, rs, and a signature, $\sigma^M_{\mathsf{Sig}} \in \Pi_{\mathsf{Sig}}.\mathsf{Sign}()$.

The algorithm does the following retrieves the data associated to the contract identifier, namely pk_C, $acct\text{-}id_C$, pk_M, $acct\text{-}id_M$, pk'_M, cid and the total balance B. Set $\bar{s} = (cid, close, rl, B^C, B^M)$.

If input is of type (A), the algorithm proceeds as follows:

(A1) Checks that $B \geq B^C + B^M$,

$$\Pi_{\mathsf{Sig}}.\mathsf{Verify}(pk_C, (cid, B^C, B^M), \sigma^C_{\mathsf{Sig}}) = \mathsf{true}$$

and

$$\Pi_{\mathsf{Sig}}.\mathsf{Verify}(pk_M, (cid, B^C, B^M), \sigma^M_{\mathsf{Sig}}) = \mathsf{true}.$$

If not, aborts and outputs $\perp$.

(A2) Records the

$$\mathsf{Disburse}(J[cid], B^C, B^M, (\cdot, \cdot, \sigma^C_{\mathsf{Sig}}), (\cdot, \sigma^M_{\mathsf{Sig}}))$$

call on its (ordered) ledger, indexed by the contract identifier $J[cid]$. Adds B^C funds from $J[cid]$ to $acct\text{-}id_C$ and B^M funds from $J[cid]$ to $acct\text{-}id_M$.

If input is of type (B), the algorithm proceeds as follows:

(B1) Checks that $B \geq B^C + B^M$,

$$\Pi_{\mathsf{Sig}}.\mathsf{Verify}(pk_C, (\bar{s}, \sigma_{\mathsf{ZKSig}}), \sigma^C_{\mathsf{Sig}}) = \mathsf{true},$$

and

$$\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk'_M, \bar{s}, \sigma_{\mathsf{ZKSig}}) = \mathsf{true}.$$

If not, aborts and outputs $\perp$.

(B2) If no pair $(rs, \sigma^M_{\mathsf{Sig}})$ is included as input, records the

$$\mathsf{Disburse}(J[cid], B^C, B^M, (rl, \sigma_{\mathsf{ZKSig}}, \sigma^C_{\mathsf{Sig}}), (\cdot, \cdot))$$

call on its (ordered) ledger, indexed by the contract identifier $J[cid]$, and adds B^C funds from $J[cid]$ to $acct\text{-}id_C$ after a time delay of δ, and B^M funds from $J[cid]$ to $acct\text{-}id_M$ immediately.

(B3) If a pair $(rs, \sigma^M_{\mathsf{Sig}})$ is specified, checks that

(a) $\Pi_{\mathsf{Sig}}.\mathsf{Verify}(pk_M, (J[cid], \bar{s}, rs), \sigma^M_{\mathsf{Sig}}) = \mathsf{true}$; and

(b) $\Pi_{\mathsf{Rlock}}.\mathsf{Verify}(rl, rs) = \mathsf{true}$.

If either check fails, records the

$$\mathsf{Disburse}(J[cid], B^C, B^M, (rl, \sigma_{\mathsf{ZKSig}}, \sigma^C_{\mathsf{Sig}}), (\cdot, \cdot))$$

call on its (ordered) ledger, indexed by the contract identifier $J[cid]$, adds B^C funds from $J[cid]$ to $acct\text{-}id_C$ after a time delay of δ and B^M funds from $J[cid]$ to $acct\text{-}id_M$ immediately. Otherwise, records the

$$\mathsf{Disburse}(J[cid], B^C, B^M, (rl, \sigma_{\mathsf{ZKSig}}, \sigma^C_{\mathsf{Sig}}), (rs, \sigma^M_{\mathsf{Sig}}))$$

call on its (ordered) ledger, indexed by the contract identifier $J[cid]$, subtracts $B^C + B^M$ funds from $J[cid]$ and adds $B^C + B^M$ funds to $acct\text{-}id_M$.

– $\mathsf{Expiry}(J[\mathit{cid}], \sigma^M_{\mathsf{Sig}})$. This algorithm takes as input a contract identifier $J[\mathit{cid}]$ and a signature $\sigma^M_{\mathsf{Sig}} \in \Pi_{\mathsf{Sig}}.\mathsf{Sign}()$. The algorithm retrieves the data associated to the contract identifier, namely pk_C, $\mathit{acct\text{-}id}_C$, pk_M, $\mathit{acct\text{-}id}_M$, pk'_M, cid, and total balance B, and checks that $B \geq B^C + B^M$ and $\Pi_{\mathsf{Sig}}.\mathsf{Verify}(\mathit{pk}_M, (\mathit{cid}, B^C, B^M), \sigma^M_{\mathsf{Sig}}) = \mathsf{true}$. If not, aborts and outputs $\perp$. If yes, records the $\mathsf{Expiry}(J[\mathit{cid}], \sigma^M_{\mathsf{Sig}})$ call on its (ordered ledger), and after a time delay of δ', subtracts $B^C + B^M$ funds from $J[\mathit{cid}]$ and adds $B^C + B^M$ funds to $\mathit{acct\text{-}id}_M$

3.4.3. Protocol. The merchant is assumed to hold a revocation pair database S_2.

To establish an escrow account, the customer and merchant agree on account parameters $(\mathit{acct\text{-}id}_C, \mathit{acct\text{-}id}_M, \mathit{pk}_C, \mathit{pk}_M, \mathit{pk}'_M, \mathit{cid}, B^C_0, B^M_0)$. The merchant provides the customer a signature σ_{ZKSig} on a tuple $\bar{s}_0 = (\mathit{cid}, \mathit{close}, \mathit{rl}_0, B^C_0, B^M_0)$, where $(\mathit{rl}_0, \cdot) \in \Pi_{\mathsf{Rlock}}.\mathsf{KeyGen}$. That is, $\Pi_{\mathsf{Sig}}.\mathsf{Verify}(\mathit{pk}'_M, \bar{s}, \sigma_{\mathsf{ZKSig}}) = \mathsf{true}$.[13]

The customer then requests the arbiter run

$$J[\mathit{cid}] \leftarrow \Pi_{\mathsf{zkEscrowAgent}}.\mathsf{Register}(\mathit{acct\text{-}id}_C, \mathit{acct\text{-}id}_M, \mathit{pk}_C, \mathit{pk}_M, \mathit{pk}'_M, \mathit{cid}, B^C_0, B^M_0)$$

and $\Pi_{\mathsf{zkEscrowAgent}}.\mathsf{Fund}(J[\mathit{cid}], C, \sigma_C)$, where σ_C is a valid Π_{Sig}-signature that authorizes a transfer of amount B^C_0 from $\mathit{acct\text{-}id}_C$ to the contract $J[\mathit{cid}]$. If successful, they send $J[\mathit{cid}]$ to merchant. If either request fails, they abort and output $\perp$.

The merchant checks that the ledger for the account $J[\mathit{cid}]$ contains the expected calls. If $B^M_0 > 0$, the merchant requests the arbiter run $\Pi_{\mathsf{zkEscrowAgent}}.\mathsf{Fund}(J[\mathit{cid}], M, \sigma_M)$, where σ_M is a valid Π_{Sig}-signature that authorizes a transfer of amount B^M_0 from $\mathit{acct\text{-}id}_M$ to the contract $J[\mathit{cid}]$. If successful, they send *success* to customer. If the ledger check or the funding request fails, they abort and output $\perp$. The customer then checks that the arbiter has accepted $\Pi_{\mathsf{zkEscrowAgent}}.\mathsf{Fund}(J[\mathit{cid}], M, \sigma_M)$; if this check fails, the customer should proceed to initiate a unilateral close, as specified below.

Parties can close the escrow account either collaboratively or unilaterally. As a consequence, both parties must watch for entries in the arbiter's ledger relevant to the the contract identifier.

To close the account collaboratively, the customer and merchant agree on closing balances and generate the signatures σ^C_{Sig} and σ^M_{Sig} over the message (cid, B^C, B^M), respectively, and request the arbiter run $\mathsf{Disburse}(J[\mathit{cid}], B^C, B^M, (\cdot, \cdot, \sigma^C_{\mathsf{Sig}}), (\cdot, \sigma^M_{\mathsf{Sig}}))$.

To close the channel unilaterally, the merchant generates the appropriate signature σ^M_{Sig} and requests the arbiter run $\mathsf{Expiry}(J[\mathit{cid}], \sigma^M_{\mathsf{Sig}})$.

To close the account unilaterally, either initially or in response to arbiter acceptance of $\mathsf{Expiry}(J[\mathit{cid}], \sigma^M_{\mathsf{Sig}})$, the customer generates the appropriate signature σ^C_{Sig} and requests the arbiter run $\mathsf{Disburse}(J[\mathit{cid}], B^C, B^M, (\mathit{rl}, \sigma_{\mathsf{ZKSig}}, \sigma^C_{\mathsf{Sig}}), (\cdot, \cdot))$. A merchant who holds a revocation secret rs, where rs satisfies $\Pi_{\mathsf{Rlock}}.\mathsf{Verify}(\mathit{rl}, \mathit{rs}) = \mathsf{true}$, then claims the entire channel balance by requesting that the arbiter run $\mathsf{Disburse}(J[\mathit{cid}], B^C, B^M, (\mathit{rl}, \sigma_{\mathsf{ZKSig}}, \sigma^C_{\mathsf{Sig}}), (\mathit{rs}, \sigma^M_{\mathsf{Sig}}))$ in response to the customer's unilateral close.

3.4.4. Discussion. The protocol $\Pi_{\mathsf{zkEscrowAgent}}$ is the component of zkChannels that backs a Π_{zkAbacus} channel with an escrow account and arbitration functionality. As such, this protocol is a three-party *escrow agent protocol* that achieves the following:

(1) Two parties, a customer and a merchant, may form an escrow account, the disbursement of which is controlled by the third party, namely the arbiter.
(2) Disbursement handles the following close scenarios:
    (a) *Cooperative close.* If the two parties cooperate to agree on a closing state, the arbiter ensures that the escrow account funds are disbursed according to this allocation.
    (b) *Customer-initiated close.* If the customer provides a state that has been previously approved by the merchant, the arbiter allows both parties to eventually claim the

[13]This may be thought of as *bootstrapping* the escrow account to ensure that a customer may always close, even in the absence of cooperation from the merchant.

indicated portion of the escrow account. However, the merchant has an opportunity to dispute in the event the customer provides an outdated state. If the merchant disputes, the arbiter decides if the merchant's dispute is correct, and if so, awards the entire escrow account balance to the merchant.

(c) *Merchant-initiated close.* The merchant can force disbursement of escrowed funds, in which case the arbiter gives the customer an opportunity to reveal the latest state and close on that allocation instead. If the customer fails to provide a state, however, the arbiter awards all money in the escrow account to the merchant. Similarly, if the customer provides an outdated state, the merchant has an opportunity to dispute, as detailed above.

In practice, it is crucial that the arbiter provides enough time for parties to respond to non-cooperative closes.

3.4.5. Realization of $\Pi_{\mathsf{zkEscrowAgent}}$ on an account-based cryptocurrency. In an account-based network such as Tezos or Ethereum there is an explicit notion of an account that maps well to our notion of accounts above. To understand how to realize the $\Pi_{\mathsf{zkEscrowAgent}}$ algorithms, however, we first need to establish some basic mechanisms for how the network constructs and modifies accounts, as well as the types of encumbrances that may be placed on accounts.

Changes to accounts are achieved by sending the network *transactions*. The network's job is to check the legitimacy of and process *transactions*, which are messages that disburse or update the encumbrances of funds in a specified account (or accounts). The network checks legitimacy of received transactions and records accepted transactions in the public ledger, and then enforces the relevant encumbrances on the specified accounts. Transactions can split, combine, create, modify, or remove accounts according to pre-specified rules. Illegitimate transactions are ignored.

More formally, a transaction is a pair (κ, T), where κ is a tuple of *instructions* specifying one or more *accounts* and the new encumbrances, and T is an *authorization sequence*, which contains evidence showing the requirements of any pre-existing instructions have been met. An arbiter defines the expected format and syntax of transaction instructions, as well as a language $\mathcal{L}$. An *encumbrance* is defined as a logical formula $\phi \in \mathcal{L}$, paired with a partial assignment of the free variables in ϕ. The arbiter defines the semantics of $\mathcal{L}$.

A transaction that is a syntactically correct according to the formal rules and language $\mathcal{L}$ of the arbiter is *well-formed*. A well-formed transaction (κ, T) acting on one or more accounts is *legitimate* if the valuation of the specified accounts' encumbrances, with assignments drawn from T and possibly contextual information, such as current time, is `true`. The arbiter will update accounts according to received legitimate transactions.

The language $\mathcal{L}$ specifying how funds may moved is typically quite expressive, particularly in the case that *smart contracts* are supported. In particular, smart contracts allow for a complete specification of the allowed states for an account as a finite state machine and support near-arbitrary transitions. Tezos, for example, supports smart contracts in the Michelson programming language (which is Turing Complete), and Ethereum supports a variety of programming languages. For zkChannels, we can specify the needed encumbrances using a smart contract.

In the following, we do not attempt to write a complete ideal functionality that captures the formal language of an account-based arbiter. Instead, we detail the necessary procedures for a customer and merchant to open, fund, and disburse funds from an escrow account using transactions sent to the arbiter. Each party must continuously monitor the arbiter's ledger for updates to the escrow account; transactions should not be considered *accepted* until they have been on the ledger for some minimum length of time.[14] In the following, we use the notation `tx`(*params*) to denote the

[14]To borrow terminology from account-based arbiters in which the ledger is realized as a sequence of blocks, this typically means that the parties establish an expected *confirmation depth* and do not consider a transaction final until that depth has been reached.

instructions of a transaction of type `tx`, and $\texttt{tx}(\textit{params}; T)$ to denote a *full transaction*, replete with authorization sequence T.

To open and fund an escrow account, the customer and merchant must agree on a smart contract and establish this contract with the arbiter by sending a transaction

$$\texttt{contract}(\textit{acct-id}_C, \textit{acct-id}_M, pk_C, pk_M, pk'_M, cid, B_0^C, B_0^M).$$

This establishes an escrow account as a smart contract. They must then each fund their portion of the smart contract by sending a transaction $\texttt{escrow}(J[cid], \textit{role}, \sigma))$ for $\textit{role} \in \{C, M\}$ as appropriate. The smart contract and funding transactions processing are in Figure 3.3.

To close an escrow account, the parties send a sequence of transactions to the arbiter to realize $\mathsf{Disburse}()$ and $\mathsf{Expiry}()$. These transactions are in Figure 3.4. The procedure the customer uses to close the escrow account is in Figure 3.5 and the procedure the merchant uses to to close the escrow account is in Figure 3.6. In both figures, we use message-based processing as shorthand, including writing "from the arbiter J" as though the arbiter sends messages to the participants, but in practice, this means that the message in question has been detected on the arbiter's ledger. We stress that the correctness of the protocol for each party relies on their adherence to these procedures. A party who fails to monitor the arbiter ledger or otherwise deviates from this specification risks losing their escrow account funds.

The following details the checks an arbiter enforces on the transactions received during escrow account establishment. In processing a transaction, the arbiter also enforces balance checks, *i.e.*, if the transaction specifies an allocation (B) such that B exceeds the amount of money in the specified funding account, the arbiter rejects the transaction.

```
on contract($acct\text{-}id_C, acct\text{-}id_M, pk_C, pk_M, pk'_M, cid, B_0^C, B_0^M$) : // $\Pi_{\mathsf{zkEscrowAgent}}.\mathsf{Register}()$
  Expect $B_0^C$ funds from $acct\text{-}id_C$.
  Expect $B_0^M$ funds from $acct\text{-}id_M$.
  Set $status = \mathsf{initialized}$ and $timeout = \perp$.
  Set encumbrances:
    if ($status = \mathsf{initialized}$){Allow refund transaction.}
    if ($status = \mathsf{contract\text{-}funded}$){
      Allow mutual-close($J[cid], B^C, B^M; \sigma_C, \sigma_M$).
      Allow expiry($J[cid]; \sigma_M$).
      Allow close($J[cid], rl, B^C, B^M; \sigma_C, \sigma_{\mathsf{ZKSig}}$).
    }
    if ($status = \mathsf{pending\text{-}close}$){
      if ($type = \mathsf{um}$){
        Allow close($J[cid], rl, B^C, B^M; \sigma_C, \sigma_{\mathsf{ZKSig}}$).
        if ($\mathsf{CurrentTime} > J[cid].timeout$){
          Allow claim($J[cid], M; \sigma_M$).
        }
      } else {
        Allow dispute($J[cid], rl; rs, \sigma_M$).
        if ($\mathsf{CurrentTime} > J[cid].timeout$){Allow claim($J[cid], C; \sigma_C$)}
      }
    }
  Set contract identifier $J[cid]$.
  Record contract in ledger.

on escrow($J[cid], role, \sigma$)) : // $\Pi_{\mathsf{zkEscrowAgent}}.\mathsf{Fund}()$
  Add $B^{role}$ funds from $acct\text{-}id_{role}$.
  if ($B^C + B^M$ funds added){Set $status = \mathsf{contract\text{-}funded}$.}
```

Figure 3.3. Opening an escrow account on account-based cryptocurrency J

The following details the checks the arbiter enforces on the transactions received. In processing a transaction, the arbiter also enforces balance checks, *i.e.*, if the transaction specifies an allocation (B^C, B^M) such that $B^C + B^M$ exceeds the total amount of money in the escrow account, the arbiter rejects the transaction.

```
on close($J[cid], rl, B^C, B^M; \sigma_C, \sigma_{\mathsf{ZKSig}}$) :
  Set $\bar{s} = (cid, close, rl, B^C, B^M)$.
  if ($\Pi_{\mathsf{Sig}}.\mathsf{Verify}(pk_C, \mathtt{close}(J[cid], rl, B^C, B^M), \sigma_C) \neq \mathsf{true}$) return
  if ($\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk'_M, \bar{s}, \sigma_{\mathsf{ZKSig}}) \neq \mathsf{true}$) return
  Move $B^M$ funds from $J[cid]$ to $acct\text{-}id_M$.
  Post to ledger, set $J[cid].timeout = \mathsf{CurrentTime} + \delta$, and set $type = \mathsf{uc}$.

on dispute($cid, rl; rs, \sigma_M$) :
  if (($\Pi_{\mathsf{Rlock}}.\mathsf{Verify}(rl, rs) = \mathsf{true}$) and ($\Pi_{\mathsf{Sig}}.\mathsf{Verify}(pk_M, \mathtt{dispute}(cid, rl), \sigma_M) = \mathsf{true}$)){
    Post to ledger and move remaining funds from $J[cid]$ to $acct\text{-}id_M$.
  } else return

on expiry($J[cid]; \sigma_M$) :
  if ($\Pi_{\mathsf{Sig}}.\mathsf{Verify}(pk_M, \mathtt{expiry}(J[cid]), \sigma_M) \neq \mathsf{true}$) return
  Post to ledger, set $J[cid].timeout = \mathsf{CurrentTime} + \delta'$, and set $type = \mathsf{um}$.

on mutual-close($J[cid], B^C, B^M; \sigma_C, \sigma_M$) :
  if ($\Pi_{\mathsf{Sig}}.\mathsf{Verify}(pk_C, \mathtt{mutual\text{-}close}(J[cid], B^C, B^M), \sigma_C) \neq \mathsf{true}$) return
  if ($\Pi_{\mathsf{Sig}}.\mathsf{Verify}(pk_M, (J[cid], \mathsf{mutual\text{-}close}, cid, B^C, B^M), \sigma_M) \neq \mathsf{true}$) return
  Post to ledger and move $B^C$ funds from $J[cid]$ to $acct\text{-}id_C$ and $B^M$ funds from $J[cid]$ to $acct\text{-}id_M$

on claim($J[cid], role; \sigma$) :
  if $\Pi_{\mathsf{Sig}}.\mathsf{Verify}(pk_{role}, \mathtt{claim}(J[cid], role), \sigma) \neq \mathsf{true}$ return
  if ($role = C$ and $type \neq \mathsf{uc}$) or ($role = M$ and $type \neq \mathsf{um}$) return
  Post to ledger and move remaining funds to $acct\text{-}id_{role}$.
```

Figure 3.4. $\Pi_{\mathsf{zkEscrowAgent}}$ realization of $\mathsf{Disburse}()$ and $\mathsf{Expiry}()$ as transactions sent to an account-based cryptocurrency J.

```
Initiate $\Pi_{\mathsf{zkEscrowAgent}}$ unilateral close:

  Send $(\mathsf{close}, cid, \mathsf{uc})$ to self.
Initiate $\Pi_{\mathsf{zkEscrowAgent}}$ mutual close:

  Send $(\mathsf{close}, cid, \mathsf{mutual})$ to self.
Message processing:

on $(\mathsf{close}, cid, type)$ from self : // Send customer closing transaction to $J$.
  Retrieve most recent closing state $\bar{s} = (cid, close, rl, B^C, B^M)$ from memory.
  Retrieve closing authorization signature $\sigma_{\mathsf{ZKSig}}$ from memory.
  Retrieve contract identifier $J[cid]$ from memory.
  if $(type = \mathsf{mutual})\{$
    Send $(\mathsf{mutual\text{-}close}, J[cid], \bar{s}, \sigma_{\mathsf{ZKSig}})$ to $M$.
  $\}$
  else $\{$// $type \in \{\mathsf{uc}, \mathsf{um}\}$
    Construct $\mathtt{close}(J[cid], rl, B^C, B^M)$.
    Compute $\sigma_C = \Pi_{\mathsf{Sig}}.\mathsf{Sign}(sk_C, \mathtt{close}(J[cid], rl, B^C, B^M))$.
    Send $\mathtt{close}(J[cid], rl, B^C, B^M; \sigma_C, \sigma_{\mathsf{ZKSig}})$ to $J$.
  $\}$

on $\mathtt{expiry}(J[cid])$ from $J$ : // Respond to unilateral merchant close.
  Send $(\mathsf{close}, cid, \mathsf{um})$ to self.

on $(\mathtt{close}(J[cid], rl, B^C, B^M), time)$ from $J$ : // Finish processing close.
  Construct $\mathtt{claim}(J[cid], C)$.
  Compute $\sigma_C = \Pi_{\mathsf{Sig}}.\mathsf{Sign}(sk_C, \mathtt{claim}(J[cid], C))$.
  At time $time + \delta'$, send $\mathtt{claim}(J[cid], C; \sigma_C)$ to $J$.

on $(\mathsf{mutual\text{-}close}, \sigma_M)$ from $M$ :
  Retrieve pending mutual close data from memory.
  if $\Pi_{\mathsf{Sig}}.\mathsf{Verify}(pk_M, (J[cid], \mathsf{mutual\text{-}close}, cid, B^C, B^M), \sigma_M) \neq 1$ return
  Construct $\mathtt{mutual\text{-}close}(J[cid], B^C, B^M)$.
  Compute $\sigma_C = \Pi_{\mathsf{Sig}}.\mathsf{Sign}(sk_C, \mathtt{mutual\text{-}close}(J[cid], B^C, B^M))$.
  Send $\mathtt{mutual\text{-}close}(J[cid], B^C, B^M; \sigma_C, \sigma_M))$ to $J$.
```

Figure 3.5. $\Pi_{\mathsf{zkEscrowAgent}}$ customer close procedure on account-based cryptocurrency J.

```
Initiate $\Pi_{\mathsf{zkEscrowAgent}}$ unilateral close:

  Send $(\mathsf{close}, cid, \mathsf{um})$ to self.
Message processing:

on $(\mathsf{close}, cid)$ from self : // Send expiry transaction to $J$.
  Retrieve contract identifier $J[cid]$ from memory.
  Construct $\texttt{claim}(J[cid], M)$.
  Compute $\sigma_M = \Pi_{\mathsf{Sig}}.\mathsf{Sign}(sk_M, \texttt{expiry}(J[cid]))$.
  Send $\texttt{expiry}(J[cid]; \sigma_M)$ to $J$.

on $(\texttt{expiry}(J[cid]), time)$ from $J$ : // Claim escrow account funds after timeout.
  Construct $\texttt{claim}(J[cid], M)$.
  Compute $\sigma_M = \Pi_{\mathsf{Sig}}.\mathsf{Sign}(sk_M, \texttt{claim}(J[cid], M))$.
  At time $time + \delta$, send $\texttt{claim}(J[cid], M; \sigma_M)$ to $J$.

on $(\texttt{close}(J[cid], rl, B^C, B^M), time)$ from $J$ : // Check and process customer close transaction.
  if $(rl, \cdot) \in S_2$ {
    Retrieve entry $(rl, rs)$ from $S_2$.
    Construct $\texttt{dispute}(J[cid], rl)$.
    Compute $\sigma_M = \Pi_{\mathsf{Sig}}.\mathsf{Sign}(sk_M, \texttt{dispute}(J[cid], rl))$.
    Before time $time + \delta'$, send $\texttt{dispute}(J[cid], rl; rs, \sigma_M)$ to $J$.
  } else {Add $(rl, \cdot)$ to $S_2$.}

on $(\mathsf{mutual\text{-}close}, J[cid], \bar{s}, \sigma_{\mathsf{ZKSig}})$ from $C_i$ :
  Retrieve contract identifier $J[cid]$ from memory.
  if $\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk'_M, \bar{s}, \sigma_{\mathsf{ZKSig}}) \neq 1$ return
  Extract revocation lock $rl$ from $\bar{s}$.
  if $(rl, \cdot) \in S_2$ return
  Compute $\sigma_M = \Pi_{\mathsf{Sig}}.\mathsf{Sign}(sk_M, (J[cid], \mathsf{mutual\text{-}close}, cid, B^C, B^M))$.
  Send $(\mathsf{mutual\text{-}close}, \sigma_M)$ to $C_i$.
```

Figure 3.6. $\Pi_{\mathsf{zkEscrowAgent}}$ merchant close procedure on account-based cryptocurrency J.

CHAPTER 4

zkChannels

We define zkChannels as a composition of Π_{zkAbacus} and $\Pi_{\mathsf{zkEscrowAgent}}$. Composition of channel and escrow account setup is done sequentially, but channel payments and channel closure must handle concurrency of the merchant revocation lock database. We assume the merchant revocation lock database implements atomic operations.

4.1. Preliminaries

The customer and merchant interact with each other and the arbiter using sessions as defined in Chapter 3.2. In the composed protocol $\Pi_{\mathsf{zkChannels}}$, the payment protocol depends on $\Pi_{\mathsf{zkAbacus}}.\mathsf{Pay}$ and reveals no information that is linkable to any part of $\Pi_{\mathsf{zkEscrowAgent}}$; as a consequence, nothing that is sent to the arbiter will be linkable to a particular $\Pi_{\mathsf{zkChannels}}.\mathsf{Pay}$ session. In practice, a customer may use an anonymizing networking tool such as Tor or Nym to provide stronger anonymity for all $\Pi_{\mathsf{zkChannels}}$ sessions. This is to evade traffic analysis by third parties during all sessions and by the merchant during payments. If desired, communication with the arbiter may also rely on an anonymizing networking tool.

4.2. System set up

To initialize system parameters, run $\mathsf{pp} \leftarrow \Pi_{\mathsf{zkAbacus}}.\mathsf{Setup}(1^{\lambda})$.

The merchant M runs $pk_M \leftarrow \Pi_{\mathsf{Sig}}.\mathsf{KeyGen}()$, chooses a funding and disbursement account $acct\text{-}id_M$, and runs $(S_0, S_1, S_2, (pk'_M, sk'_M)) \leftarrow \Pi_{\mathsf{zkAbacus}}.\mathsf{Init}(\mathsf{pp})$. The activation database S_0 and nonce database S_1 are internal to Π_{zkAbacus}, but the revocation lock database S_2 is shared between Π_{zkAbacus} and $\Pi_{\mathsf{zkEscrowAgent}}$.

4.3. Channel establishment

An overview of channel establishment appears in Figure 4.1.

The interactive protocol $\Pi_{\mathsf{zkChannels}}.\mathsf{Establish}$ takes as shared input the merchant public key information pk_M and pk'_M (together with associated account $acct\text{-}id_M$). Customer inputs consist of account information with the desired arbiter, $J[C]$, initial customer balance B^C and initial merchant balance B^M. Merchant inputs consist of their secret keys for $\Pi_{\mathsf{zkEscrowAgent}}$ and Π_{zkAbacus} and their activation database S_0 for Π_{zkAbacus}. Both parties contribute randomness to the *channel identifier*; this channel identifier must be unique, which is achieved (except with negligible probability) by instantiating the identifier as the output of a cryptographic collision-resistant hash function.

(1) The customer prepares to run Π_{zkAbacus} and $\Pi_{\mathsf{zkEscrowAgent}}$. That is, they initialize channel status, which we denote by *status*, to $\perp$. They run $pk_C \leftarrow \Pi_{\mathsf{Sig}}.\mathsf{KeyGen}()$ and choose a funding and disbursement account $acct\text{-}id_C$ (over which they have control), and choose a random contribution to the channel identifier, namely $cid_C \in \{0,1\}^{\lambda}$. They send pk_C, $acct\text{-}id_C$, cid_C, B^C_0, and B^M_0 to the merchant.

(2) The merchant prepares to run Π_{zkAbacus} and $\Pi_{\mathsf{zkEscrowAgent}}$. That is, they initialize the channel status, which we denote by *status*, to $\perp$. They check that each element of the customer's message is well-formed and abort and output $\perp$ if not. They decide whether or not to accept the channel request. If yes, they choose at random a contribution to the channel identifier, namely $cid_M \in \{0,1\}^{\lambda}$, and send $(\mathsf{accept}, acct\text{-}id_M, cid_M)$ to the

Table 1. Channel status values

initialized	Set by C after they have received valid closing authorization signature for initial state. Set by M once they have sent closing authorization signature for initial state. For both parties, this is immediately after the $\Pi_{\mathsf{zkAbacus}}.\mathsf{Initialize}()$ call completes successfully.
contract-funded	Set by each party as soon as they have verified that the contract is funded.
ready	Set by C after they have received pt_0; set by M once they have sent pt_0
frozen	Set by C after failed payment.
pending-close	Set by each party after either sending a $\Pi_{\mathsf{zkEscrowAgent}}.\mathsf{Disburse}()$ or $\Pi_{\mathsf{zkEscrowAgent}}.\mathsf{Expiry}()$ call to the arbiter or reading such a call in the arbiter's ledger.
closed	Set by C and M when channel balances have been disbursed by the arbiter.

customer. If no, they send reject message, abort, and output $\perp$. They set channel identifier $cid = \mathsf{H}(cid_C, cid_M, pk_C, pk_M, pk'_M)$.

(3) The customer checks that $cid_M \in \{0,1\}^\lambda$. If yes, they set channel identifier $cid = \mathsf{H}(cid_C, cid_M, pk_C, pk_M, pk'_M)$ and continue. Otherwise, they abort and output $\perp$.

(4) The customer and merchant proceed with Π_{zkAbacus}. That is, they collaboratively run

$$\Pi_{\mathsf{zkAbacus}}.\mathsf{Initialize}(pk'_M, cid, B_0^C, B_0^M, (S_0, sk'_M)_M);$$

this involves a single round of communication. See section 3.3.2 for details. At a high level, the customer forms and proves correctness of their initial state $s_0 = (cid, n_0, rl_0, B_0^C, B_0^M)$ and initial closing state $\bar{s}_0 = (cid, close, rl_0, B_0^C, B_0^M)$ in zero knowledge and receives a blind signature on $\bar{s}_0$, namely σ_0, their initial closing authorization signature, from the merchant. If successful, both parties set the channel status to initialized.

(5) The customer proceeds with $\Pi_{\mathsf{zkEscrowAgent}}$. That is, they request that the arbiter run

$$J[cid] \leftarrow \Pi_{\mathsf{zkEscrowAgent}}.\mathsf{Register}(acct\text{-}id_C, acct\text{-}id_M, pk_C, pk_M, pk'_M, cid, B_0^C, B_0^M).$$

If successful, they request that the arbiter run $\Pi_{\mathsf{zkEscrowAgent}}.\mathsf{Fund}(J[cid], C, \sigma_C)$, where σ_C is a valid Π_{Sig}-signature that authorizes a transfer of amount B_0^C from $acct\text{-}id_C$ to the contract. If successful, they send $J[cid]$ to merchant. If either of the procedures from $\Pi_{\mathsf{zkEscrowAgent}}$ fails, they abort and output $\perp$.

(6) The merchant proceeds with $\Pi_{\mathsf{zkEscrowAgent}}$. That is, they check that the ledger for the account $J[cid]$ contains the expected calls. If $B_0^M > 0$, the merchant requests that the arbiter run $\Pi_{\mathsf{zkEscrowAgent}}.\mathsf{Fund}(J[cid], M, \sigma_M)$, where σ_M is a valid Π_{Sig}-signature that authorizes a transfer of amount B_0^M from $acct\text{-}id_M$ to the contract. If successful, they send *success* to customer. If the ledger check or the funding request fails, they abort and output $\perp$.

(7) If dual-funded, the customer checks that $\Pi_{\mathsf{zkEscrowAgent}}.\mathsf{Fund}(J[cid], M, \sigma_M)$ has been accepted by the arbiter; if this check fails, the customer should proceed to initiate a unilateral close, and then output $\perp$.

(8) The customer and merchant run $\Pi_{\mathsf{zkAbacus}}.\mathsf{Activate}(\mathsf{pp}, pk'_M, cid, (s_0, \tau)_C, (S_0, sk)_M)$. The customer receives and stores a payment tag pt_0 and sets $status =$ ready, if successful, and otherwise initiates a unilateral close in $\Pi_{\mathsf{zkEscrowAgent}}$, and then outputs $\perp$. Similarly, the merchant sets $status =$ ready if successful, and otherwise initiates a unilateral close in $\Pi_{\mathsf{zkEscrowAgent}}$, and then outputs $\perp$.

$\Pi_{\mathsf{zkChannels}}.\mathsf{Establish}(\mathsf{pp}, pk_M, \mathit{acct\text{-}id}_M, pk'_M, (J[C], B_0^C, B_0^M)_C, (sk_M, sk'_M, S_0)_M)$

Initialize:

(1) The merchant-specific parameters for channel and escrow accounts for Π_{zkAbacus} and $\Pi_{\mathsf{zkEscrowAgent}}$ are fixed at merchant setup.

(2) Remaining channel and escrow account parameters for Π_{zkAbacus} and $\Pi_{\mathsf{zkEscrowAgent}}$ are picked as follows:

(a) The customer chooses initial balances B_0^C, and B_0^M.

(b) The customer chooses Π_{zkAbacus} customer parameters pk_C and $\mathit{acct\text{-}id}_C$.

(c) Both the customer and merchant contribute to the generation of the channel identifier cid (used in both Π_{zkAbacus} and $\Pi_{\mathsf{zkEscrowAgent}}$) by picking cid_C and cid_M, respectively. They then set $\mathit{cid} = \mathsf{H}(\mathit{cid}_C, \mathit{cid}_M, pk_C, pk_M, pk'_M)$.

(3) The merchant either accepts or rejects the channel request.

(4) They run $((\bar{s}_0, s_0, rs_0, \tau_0, \sigma_0)_C, (S'_0)_M) \leftarrow \Pi_{\mathsf{zkAbacus}}.\mathsf{Initialize}(pk'_M, \mathit{cid}, B_0^C, B_0^M, (S_0, sk'_M)_M)$.

(5) They then set channel status to $\mathsf{initialized}$.

Open: The customer and merchant follow $\Pi_{\mathsf{zkEscrowAgent}}$ to open an escrow account with identifier *contract-ID* bound to the Π_{zkAbacus} channel cid. They then set channel status to $\mathsf{contract\text{-}funded}$.

Activate: The customer and merchant complete channel setup in Π_{zkAbacus} for channel cid by running $((s_0, pt_0)_C, (\mathit{success})_M) \leftarrow \Pi_{\mathsf{zkAbacus}}.\mathsf{Activate}(pk'_M, \mathit{cid}, (s_0, \tau_0)_C, (S'_0, sk'_M)_M)$. They then set channel status to ready.

Figure 4.1. Overview of zkChannels channel establishment protocol

4.4. Channel payments

Say a customer C wishes to purchase a good or service, a description of which we denote by x, from the merchant M using their previously established zkChannel with identifier cid. Assume the cost is ϵ.

(1) To make a payment on channel cid, the customer C establishes a session with the merchant M and sends a *payment request message* containing the tuple (x, ϵ) to M.

(2) The merchant checks (x, ϵ) and decides whether to accept or reject the payment. If the former, they send $\mathsf{accept\text{-}payment}$; if the latter, they send $\mathsf{reject\text{-}payment}$ and abort.

(3) Upon receipt of $\mathsf{reject\text{-}payment}$, the customer C aborts the session. Upon receipt of $\mathsf{accept\text{-}payment}$, the customer and merchant run

$$\Pi_{\mathsf{zkAbacus}}.\mathsf{Pay}((pk'_M, \epsilon, (s_i, pt_i)_C, (S_1, S_2, sk'_M)_M)),$$

where s_i and pt_i are the most recent channel state and associated payment tag for channel cid, respectively.

(4) Whatever the purpose of payment, the indicator $\mathit{pay\text{-}status}_M$ can be used by the merchant to make decisions about whether a payment is complete and/or a service should be provided. That is, the merchant uses $\mathit{pay\text{-}status}_M$ and the context of the payment x to determine further action, such as the provision of a requested good or service. More precisely, the merchant extracts the resulting $\mathit{pay\text{-}status}_M$ from $\Pi_{\mathsf{zkAbacus}}.\mathsf{Pay}$ output and behaves as follows:

(a) If $\mathit{pay\text{-}status}_M = \perp$, the payment has unequivocably failed. No good or service should be provided.

(b) If $\mathit{pay\text{-}status}_M = \mathsf{revocation\text{-}incomplete}$, then if $\epsilon < 0$, the merchant should consider the payment complete. Otherwise, the merchant should consider the payment incomplete and no good or service should be provided.

   (c) If $\mathit{pay\text{-}status}_M = \textsf{revocation-complete}$, then the merchant should consider the payment complete.

(5) The customer extracts the resulting $\mathit{pay\text{-}status}_C$ from $\Pi_{\textsf{zkAbacus}}.\textsf{Pay}$ output and does the following:
   (a) If $\mathit{pay\text{-}status}_C = \textsf{revocation-incomplete}$, the customer sets $\mathit{status} = \textsf{frozen}$, initiates unilateral closure on state $\bar{s}_i$ as specified in Section 4.5.[1]
   (b) If $\mathit{pay\text{-}status}_C = \textsf{revocation-complete}$, the customer sets $\mathit{status} = \textsf{frozen}$, initiates unilateral closure on state $\bar{s}_{i+1}$ as specified in Section 4.5.
   (c) If $\mathit{pay\text{-}status}_C = \textsf{state-updated}$, then the customer should expect to receive the requested good or service, and may continue to make additional payments on the channel.

4.5. Channel closing

Both parties watch the arbiter's ledger for calls related to the channel escrow account, as in $\Pi_{\textsf{zkEscrowAgent}}$. Three types of closing are supported, inherited from $\Pi_{\textsf{zkAbacus}}$ and $\Pi_{\textsf{zkEscrowAgent}}$:

(1) To close the account collaboratively:
   (a) The customer initiates

$$\Pi_{\textsf{zkAbacus}}.\textsf{Close}(\textsf{pp}, pk, \mathit{cid}, (\bar{s}, \sigma)_C, (S_2)_M)$$

   on the most recent closing state $\bar{s} = (\mathit{cid}, \mathit{close}, \mathit{rl}, B^C, B^M)$ and corresponding closing authorization signature σ.[2]
   (b) If $\Pi_{\textsf{zkAbacus}}.\textsf{Close}()$ is successful, the merchant outputs $(\mathit{cid}, \mathit{rl}, \bot, \textsf{true})$, and then generates the appropriate signature $\sigma^M_{\textsf{Sig}}$ authorizing disbursement from the escrow account for the requested balances. The merchant then adds $(\mathit{rl}, \cdot)$ to their revocation-pair database S_2 and sends $\sigma^M_{\textsf{Sig}}$ to the customer. If unsuccessful, the merchant output is $(\mathit{cid}, \mathit{rl}, \mathit{rs}, \textsf{false})$, in which case they should abort the collaborative close and output $\bot$.
   (c) The customer checks that $\Pi_{\textsf{Sig}}.\textsf{Verify}(pk_M, (\mathit{cid}, \textsf{mutual-close}, B^C, B^M), \sigma^M_{\textsf{Sig}}) = \textsf{true}$. If yes, they generate the appropriate signature $\sigma^C_{\textsf{Sig}}$ authorizing disbursement from the escrow account for the requested balances and request the arbiter run

$$\textsf{Disburse}(J[\mathit{cid}], B^C, B^M, (\cdot, \cdot, \sigma^C_{\textsf{Sig}}), (\cdot, \sigma^M_{\textsf{Sig}})).$$

   If no, they abort and output $\bot$.[3]
   (d) Once the $\textsf{Disburse}(J[\mathit{cid}], B^C, B^M, (\cdot, \cdot, \sigma^C_{\textsf{Sig}}), (\cdot, \sigma^M_{\textsf{Sig}}))$ call is accepted by the arbiter (*i.e.*, is recorded on its ledger) the customer and merchant set $\mathit{status} = \textsf{closed}$.

(2) To close the channel unilaterally, the merchant sets $\mathit{status} = \textsf{pending-close}$, generates the appropriate signature $\sigma^M_{\textsf{Sig}}$, and requests the arbiter run $\textsf{Expiry}(J[\mathit{cid}], \sigma^M_{\textsf{Sig}})$. After arbiter acceptance of the call and completion of the funds transfer after time δ', the customer and merchant each set $\mathit{status} = \textsf{closed}$.

(3) To close the account unilaterally, either initially or in response to a run of $\textsf{Expiry}(J[\mathit{cid}], \sigma^M_{\textsf{Sig}})$, the customer sets $\mathit{status} = \textsf{pending-close}$, generates the appropriate signature $\sigma^C_{\textsf{Sig}}$ and

[1] If the customer has a valid closing signature on the new state, but has not yet sent the revocation information for the previous state, we could differentiate customer behavior based on whether the value of the payment is positive or negative, and allow a customer to close on either the old or the new state according to what is most advantageous monetarily. However, in practice, this situation arises if the customer process crashes without losing the new closing signature but before atomically preparing and sending the revocation information, and this window of possibility is on the order of milliseconds. A bug that causes a crash at this point would be problematic if a refund has been issued.

[2] If the merchant knows how to contact the customer, they can request the customer participate in a collaborative close.

[3] In this case, the customer should initiate a unilateral customer close.

requests the arbiter run

$$\mathsf{Disburse}(J[cid], B^C, B^M, (rl, \sigma_{\mathsf{ZKSig}}, \sigma^C_{\mathsf{Sig}}), (\cdot, \cdot)),$$

where σ_{ZKSig} is the closing authorization signature for the closing state $(cid, close, rl, B^C, B^M)$.

After arbiter acceptance of the call, the merchant sets $status = \mathsf{pending\text{-}close}$ and checks to see if there exists a revocation pair $(rl, rs) \in S_2$. If yes, before time δ passes, the merchant claims the entire channel balance from the escrow account by requesting that the arbiter run

$$\mathsf{Disburse}(J[cid], B^C, B^M, (rl, \sigma_{\mathsf{ZKSig}}, \sigma^C_{\mathsf{Sig}}), (rs, \sigma^M_{\mathsf{Sig}})),$$

in response to the customer's unilateral close. If no, the merchant adds $(rl, \cdot)$ to their revocation pair database S_2.[4]

After completion of the funds transfer after time δ, the customer and merchant each set $status = \mathsf{closed}$.

4.6. Discussion

The main focus of zkChannels is the achievement of customer payment history privacy. This choice involves certain tradeoffs. While we always guarantee an unlinkable method to close and that no honest party loses their share of the escrowed funds, we do not ensure (1) the usability of the channel for future payments; (2) the merchant actually provides the requested service; or, most interestingly, (3) payments occur only on channels with an open on-network escrow account. We argue that the lack of these properties is relatively benign in the context of a traditional customer-merchant relationship.

From the perspective of a customer, the most natural use cases for a point-to-point private payment channel involve a merchant who is *semi-trusted*. That is, the merchant either already has an established reputation or the customer is willing to build a trusted relationship with an unknown merchant over the course of a sequence of payments. The customer does not risk their entire channel balance in engaging with the merchant; they risk only a single payment amount, a risk profile that maps well to current practice. In other words, we do not attempt to replace all mechanisms of trust in society with zkChannels.

The perspective of the merchant with respect to the lack of property (3) above is a more interesting question. Say the customer makes a series of payments on the underlying Π_{zkAbacus} channel corresponding to intermediate states $s_0, s_1, \ldots, s_m$, and then closes the channel on s_i for some $i < m$. Our use of on-network punishment in $\Pi_{\mathsf{zkEscrowAgent}}$ actually amounts to a *soft punishment*: the customer is still able to make a valid, correct off-network payment using the most recent state s_m, since they do not reveal the associated nonce or revocation lock on closing the escrow account. This amounts to the conversion of the underlying Π_{zkAbacus} payment channel from one backed by on-network funds to one backed by a merchant gift card. The issue is that the transactions the customer may use to close the escrow account in $\Pi_{\mathsf{zkEscrowAgent}}$ do not include enough information for the merchant (or anyone else) to infer anything about the closing state *except whether or not it is outdated*. That is, we do not provide a reconciliation mechanism for the merchant to derive the most recent state and prevent further payments. However, we stress that, in this situation, the customer cannot make payments for which they have not already transferred the corresponding funds to the merchant. We leave the extension of our protocol to provide reconciliation mechanisms (both off- and on-chain) as future work.

Another caveat of zkChannels is the need to handle concurrent payments (realized using Π_{zkAbacus}) and payments concurrent with escrow account closes (realized using $\Pi_{\mathsf{zkEscrowAgent}}$). It is crucial

[4]If concurrent operations are allowed, the merchant database must be carefully constructed to avoid the situation in which concurrent calls to the database allow an on-network escrow account close and a payment to happen on the same state simultaneously. Alternatively, the protocol for closing the escrow account may be modified to detect this situation with high probability.

that the merchant procedures for processing payments and closing escrow accounts ensures that a merchant detect this situation and respond appropriately. In the abstract, we can require the merchant handle operations sequentially, *i.e.*, we can require the revocation database be read- and write-locked. However, in practice, this solution is not scalable, and so must be handled carefully by the implementation of the merchant database.

4.7. Implementation overview

We defer to the zkChannels specification [**Inc21**] for implementation details. In particular, we detail how to realize an account-based arbiter using Tezos for $\Pi_{\mathsf{zkEscrowAgent}}$; this involves defining and implementing a notification service to automatically monitor escrow accounts. We also sketch our approach to handle concurrent channel payment and closing activity, which involves implementing a merchant database with atomic operations.

Bibliography

[BB04] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In Christian Cachin and Jan L. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, pages 56–73, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[BPW12] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How Not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Xiaoyun Wang, and Kazue Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, volume 7658, pages 626–643. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[CCs08] Jan Camenisch, Rafik Chaabouni, and abhi shelat. Efficient protocols for set membership and range proofs. In Josef Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008*, pages 234–252, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[CL04] Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In Matt Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, pages 56–72, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[GMR85] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 291–304, New York, NY, USA, 1985. ACM.

[Inc21] Bolt Labs Inc. zkChannels: A specification of the blockchain interactions. `https://github.com/boltlabs-inc/zkchannels-spec`, 2021.

[Ped92] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 129–140, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[PS16] David Pointcheval and Olivier Sanders. Short Randomizable Signatures. In Kazue Sako, editor, *Topics in Cryptology - CT-RSA 2016*, volume 9610, pages 111–126. Springer International Publishing, Cham, 2016.

[PS18] David Pointcheval and Olivier Sanders. Reassessing security of randomizable signatures. In Nigel P.Editor Smart, editor, *Topics in Cryptology – CT-RSA 2018*, volume 10808, page 319–338. Springer International Publishing, 2018.

[Sch91] C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, Jan 1991.

A. Proof Strategy Overview

Note to the reader: This section is a rough sketch. In particular, the ideal functionality is not finalized.

We now give an overview of the strategy we will follow to prove the security of our construction.

We note that our protocol is extraordinarily complex. This means that analyzing the entire protocol at once produces a proof that it too long and complicated to meaningfully consume. While this proof might be *technically* correct, it does little to increase the reader's confidence that the construction is correct. As such, we will attempt to break down the proof into more manageable subcomponents, and then argue about the composition of the subcomponents. Namely, we give definitions for the off-network channel functionality $\mathcal{F}_{\mathsf{zkAbacus}}$ and the on-network arbitration functionality $\mathcal{F}_{\mathsf{zkEscrowAgent}}$, and then argue that their composition results in the desired functionality $\mathcal{F}_{\mathsf{zkChannels}}$.

All of our proofs will be simulation-based security proofs that allow corruption of an arbitrary subset of the customer and the merchant. This entails showing that our protocols realize some *ideal functionality*. This ideal functionality is trusted to perform prescribed actions and can maintain tamper-proof state across interactions. By showing that the protocol realizes this functionality, we show that the protocol provides the same security and privacy guarantees as the ideal functionality.

A.1. Off-Network Channels: zkAbacus. The zkAbacus functionality manages changes to the state held by the customer and the merchant. Note that in the context of the off-network channel, the state is not inherently meaningful; we relegate interpreting the meaning of the off-network state to the on-network arbitration functionality discussed next.

The zkAbacus functionality provides interfaces for: (1) initializing and activating the virtual channel on some mutually agreed upon (but arbitrary) balance, (2) allowing the customer to make payments over the virtual channel, which update the balances in the channel, and (3) closing down the channel. When closing down a channel, the ideal functionality determines if the closure is *honest* or *dishonest*, sending this determination to an arbiter J. Again, as noted above, the meaning of this determination is left to the on-network arbitration functionality.

The protocol Π_{zkAbacus} has an initialize phase, a pay phase, and a close phase. During the initialize phase, the customer and merchant exchange non-anonymous information required to show that the first balances are being set up correctly. At the end of this phase, the customer can anonymously initiate a run of the pay protocol, which updates the balances by some value ϵ. We note that there are several abort conditions during the pay protocol that result in the customer having different options during the close phase. Finally, the close phase is very simple: the custom makes a claim about the closing balances for the channel. If these balances do not represent the most recent balances in the channel, the merchant will be able to detect this and show that the closure is dishonest.

We argue that the protocol realizes the ideal functionality using a standard simulation-based proof. In this argument, we define a simulator that runs the real protocol with the corrupt players while interacting only with the ideal functionality. Because the ideal functionality effectively hides information from the simulator, the simulator must be able to run the real protocol without any knowledge of the honest player's private state and without knowing any more about their actions than is provided by the ideal functionality.

A.2. On-Network Arbitration: zkEscrowAgent. The zkEscrowAgent functionality manages the resources that are put into the virtual channel. Specifically, in the case of a payment channel, the functionality manages the creation of an escrow account that hold funds from the customer and merchant, and dispersing those funds as appropriate when the channel is closed down. Note that the zkEscrowAgent functionality need not see any of the payment interactions between the customer and the merchant. Instead, it only operates on information explicitly provided to it.

The zkEscrowAgent functionality provides interfaces for: (1) the creation of an escrow account, (2) the funding of that account, and (3) dispersal of the funds in that account. The funding of the account comes from funds that are given to the players during set-up. This represents the initial holdings of players coming into this protocol. When funds are dispersed, they are allocated back into the control of the player as appropriate.

The protocol $\Pi_{\mathsf{zkEscrowAgent}}$ is defined in the presence of a trusted network functionality. This network is capable of creating new accounts and enforcing rules on how the funds in the account are distributed. The players specify the rules for the new account and then move their funds into the account.

A.3. Composition: zkChannels. The composing the two functionalities with a simple protocol realizes the complete $\mathcal{F}_{\mathsf{zkChannels}}$ functionality. This functionality captures the end-to-end goals of our project. The players start by creating and funding a new account with $\mathcal{F}_{\mathsf{zkEscrowAgent}}$. Next, they initialize a virtual channel with the corresponding balances from the new account using $\mathcal{F}_{\mathsf{zkAbacus}}$. Then, there is an arbitrary length of time during which the customer may initialize new payments using $\mathcal{F}_{\mathsf{zkAbacus}}$. When the customer closes down the virtual channel, this provides the necessary information to $\mathcal{F}_{\mathsf{zkEscrowAgent}}$ to determine how funds should be distributed.

We note that there do not appear to be any significant obstacles to composing these two protocols. The main barrier to parallel composition is rewinding, as that can desynchronize different elements of the protocol. However, any rewinding that happens during simulation is highly localized. Specifically, we only use rewinding to extract from zero-knowledge proofs that have been flattened by Fiat-Shamir.

The zkChannels functionality can be seen as the combination of zkEscrowAgent and zkAbacus. In handles both the escrow and allocation of funds, as well as payments on that channel. Because it has visibility both into the actually currency accounts and the payments that are made on the channel, it can be use to correctly enforce closure behavior based on all of the payments that were made.

B. Draft Π_{zkAbacus} ideal functionality

In the following section, we describe the ideal functionality. Figures C.2-**??** describe channel opening, payments from the customer and merchant perspective, respectively, and channel closing.

B.1. Channel Opening. We begin by describing the ideal functionality's interfaces for opening a new channel. There are two such interfaces: initialize-channel is called by a customer wishing to create a new channel. activate-channel is called by the merchant when they wish to accept a channel and allow the customer to make payments on that channel.

```
on $(\mathsf{initialize\text{-}channel}, cid, B_C^0, B_M^0)$ from $C_i$ for some $i \in \{1, \ldots, \ell\}$ :
  if $(\exists \mathbb{C}[cid])$ return
  // Set $(\mathsf{status}, C, \epsilon, pid, \mathsf{cust\text{-}strat}, \mathsf{states\text{-}list}, \mathsf{aborted\text{-}pay\text{-}close})$.
  Set $\mathbb{C}[cid] = (\mathsf{initialized}, C_i, \bot, \bot, \bot, [(\bot, B_C^0, B_M^0)], \mathsf{false})$
  Send $(\mathsf{initialized}, cid, C_i, B_C^0, B_M^0)$ to $M$

on $(\mathsf{activate\text{-}channel}, cid)$ from $M$ :
  // Channel must be initialized.
  if $(\nexists \mathbb{C}[cid]$ or $\mathbb{C}[cid].\mathsf{status} \neq \mathsf{initialized})$ return
  Set $\mathbb{C}[cid].\mathsf{status} = \mathsf{open}$
  Send $(\mathsf{activated}, cid)$ to $M$
```

B.2. Channel Payments. We continue by describing the ideal functionality's interfaces for making payments. There are two such interfaces: pay-offered is called by the customer when they wish to make a payment. The interface accepts a $\mathsf{strat} \in \{\mathsf{honest}, \mathsf{dishonest}\}$ parameter that specifies if the customer wishes to freeze the payment channel. dishonest can only be used by a corrupted customer. If the merchant is corrupt, they use the pay-instr interface to specify if they wish to freeze the channel. The interface accepts a $\mathsf{instr} \in \{\mathsf{reject}, \mathsf{freeze\text{-}after\text{-}pay}, \mathsf{honest\text{-}accept}\}$ parameter to specify if the payments should be rejected, the channel should be frozen, or the payment should be accepted (respectively).

```
// A customer makes a pay offer with a prespecified strategy $\mathsf{strat} \in \{\mathsf{honest}, \mathsf{dishonest}\}$.
on $(\mathsf{pay\text{-}offered}, cid, \epsilon, \mathsf{strat})$ from $C_i$ for some $i \in \{1, \dots, \ell\}$ :
  // Channel must be open.
  if $(\nexists \mathbb{C}[cid]$ or $\mathbb{C}[cid].C \neq C_i$ or $\mathbb{C}[cid].\mathsf{status} \neq \mathsf{open})$ return
  Retrieve latest state as $(\cdot, B_C, B_M) = \mathbb{C}[cid].\mathsf{states\text{-}list}[\mathtt{LAST}]$
  // Honest payments have valid amounts.
  if $((C_i \notin \mathcal{I}$ or $\mathsf{strat} = \mathsf{honest})$ and $(B_C < \epsilon$ or $B_M < -\epsilon))$ return
  Sample random $pid$ and set $\mathbb{C}[cid].pid = pid$
  // Cases for honest merchant.
  if $(M \notin \mathcal{I})\{$
    // Honest payments go through.
    if $(C_i \notin \mathcal{I}$ or $\mathsf{strat} = \mathsf{honest})\{$
      Append $(\mathbb{C}[cid].pid, B_C - \epsilon, B_M + \epsilon)$ to $\mathbb{C}[cid].\mathsf{states\text{-}list}$
      Send $(\mathsf{pay}, \mathbb{C}[cid].pid, \epsilon)$ to $M$ and $C_i$
    // Invalid amount freezes before payment.
    $\}$ elif $(B_C < \epsilon$ or $B_M < -\epsilon)\{$
      Set $\mathbb{C}[cid].\mathsf{status} = \mathsf{frozen}$
      Send $(\mathsf{frozen}, \mathbb{C}[cid].pid, \epsilon)$ to $M$ and $C_i$
    // Valid amount freezes after payment and allows closure on previous state.
    $\}$ else $\{$
      Append $(\mathbb{C}[cid].pid, B_C - \epsilon, B_M + \epsilon)$ to $\mathbb{C}[cid].\mathsf{states\text{-}list}$
      Set $\mathbb{C}[cid].\mathsf{aborted\text{-}pay\text{-}close} = \mathsf{true}, \mathbb{C}[cid].\mathsf{status} = \mathsf{frozen}$
      Send $(\mathsf{frozen}, \mathbb{C}[cid].pid, \epsilon)$ to $M$ and $C_i$
    $\}$
  // Merchant is corrupt, and therefore can specify instructions for this payment.
  $\}$ else $\{$
    Set $\mathbb{C}[cid].\epsilon = \epsilon, \mathbb{C}[cid].\mathsf{cust\text{-}strat} = \mathsf{strat}, \mathbb{C}[cid].\mathsf{status} = \mathsf{pay\text{-}offered}$,
    Send $(\mathsf{pay\text{-}offered}, \mathbb{C}[cid].pid, \epsilon)$ to $M$
  $\}$
```

```
// Merchant instructions instr $\in \{\mathsf{reject}, \mathsf{freeze\text{-}after\text{-}pay}, \mathsf{honest\text{-}accept}\}$.
on $(\mathsf{pay\text{-}instr}, pid, \mathsf{instr})$ from $M$ :
  // Find channel and ensure merchant corrupt.
  if $(\nexists cid$ s.t. $\mathbb{C}[cid].pid = pid$ or $\mathbb{C}[cid].\mathsf{status} \neq \mathsf{pay\text{-}offered}$ or $M \notin \mathcal{I})$ return
  Retrieve latest state as $(\cdot, B_C, B_M) = \mathbb{C}[cid].\mathsf{states\text{-}list}[\mathtt{LAST}]$
  // Reject payment and freeze the channel.
  if $(\mathsf{instr} = \mathsf{reject})\{$
    $\mathbb{C}[cid].\mathsf{status} = \mathsf{frozen}$
    Send $(\mathsf{reject}, \mathbb{C}[cid].pid)$ to $M$ and $\mathbb{C}[cid].C$
  // Customer acts dishonestly.
  $\}$ elif $(\mathbb{C}[cid].\mathsf{cust\text{-}strat} = \mathsf{dishonest}$ and $\mathsf{instr} \in \{\mathsf{freeze\text{-}after\text{-}pay}, \mathsf{honest\text{-}accept}\})\{$
    // Invalid amount freezes before payment.
    if $(B_C < \mathbb{C}[cid].\epsilon$ or $B_M < -\mathbb{C}[cid].\epsilon)\{$
      Set $\mathbb{C}[cid].\mathsf{status} = \mathsf{frozen}$
      Send $(\mathsf{frozen}, \mathbb{C}[cid].pid, \mathbb{C}[cid].\epsilon)$ to $M$ and $\mathbb{C}[cid].C$
    // Valid amount freezes after payment and allows closure on previous state.
    $\}$ else $\{$
      Append $(\mathbb{C}[cid].pid, B_C - \epsilon, B_M + \epsilon)$ to $\mathbb{C}[cid].\mathsf{states\text{-}list}$.
      Set $\mathbb{C}[cid].\mathsf{aborted\text{-}pay\text{-}close} = \mathsf{true}, \mathbb{C}[cid].\mathsf{status} = \mathsf{frozen}$
      Send $(\mathsf{frozen}, \mathbb{C}[cid].pid, \mathbb{C}[cid].\epsilon)$ to $M$ and $\mathbb{C}[cid].C$
    $\}$
  $\}$ elif $(\mathbb{C}[cid].\mathsf{cust\text{-}strat} = \mathsf{honest}$ and $\mathsf{instr} = \mathsf{freeze\text{-}after\text{-}pay})\{$
    Append $(\mathbb{C}[cid].pid, B_C - \epsilon, B_M + \epsilon)$ to $\mathbb{C}[cid].\mathsf{states\text{-}list}$
    Set $\mathbb{C}[cid].\mathsf{status} = \mathsf{frozen}$
    Send $(\mathsf{frozen}, \mathbb{C}[cid].pid, \epsilon)$ to $M$ and $\mathbb{C}[cid].C$
  $\}$ else $\{$ // $\mathsf{instr} = \mathsf{honest\text{-}accept}$ and $\mathbb{C}[cid].\mathsf{cust\text{-}strat} = \mathsf{honest}$
    Append $(\mathbb{C}[cid].pid, B_C - \epsilon, B_M + \epsilon)$ to $\mathbb{C}[cid].\mathsf{states\text{-}list}$
    Set $\mathbb{C}[cid].\mathsf{cust\text{-}strat} = \perp, \mathbb{C}[cid].\mathsf{status} = \mathsf{open}$
    Send $(\mathsf{pay}, \mathbb{C}[cid].pid, \epsilon)$ to $M$ and $\mathbb{C}[cid].C$
  $\}$
```

B.3. Channel Closing. We finish by describing the ideal functionality's interface for closing a channel. Because we are only concerned with customer initiated closures in this context, there is only one interface for closing. The customer chooses a state in the history of the channel on which to close, specified by its index. Specifically, a honest customer can choose only the more recent state, whereas a corrupt customer may choose any index.

```
on $(\mathsf{close}, cid, \mathsf{index})$ from $C_i$ for some $i \in \{1, \ldots, \ell\}$ :
  if $(\nexists \mathbb{C}[cid]$ or $\mathbb{C}[cid].C \neq C_i$ or $\mathbb{C}[cid].\mathsf{status} \notin \{\mathsf{open}, \mathsf{pay\text{-}offered}, \mathsf{frozen}\})$ return
  // Honest close on last state.
  if $(\mathsf{index} = \mathtt{LAST})\{$
    Retrieve latest state as $(pid, B_C, B_M) = \mathbb{C}[cid].\mathsf{states\text{-}list}[\mathtt{LAST}]$
    Set $\mathbb{C}[cid].\mathsf{status} = \mathsf{closed}$
    Send $(\mathsf{closed}, cid, \mathsf{honest}, B_C, B_M)$ to $C_i, M$
  // Close on penultimate state.
  $\}$ elif $(C_i \in \mathcal{I}$ and $\mathbb{C}[cid].\mathsf{status} = \mathsf{frozen}$ and $\mathbb{C}[cid].\mathsf{aborted\text{-}pay\text{-}close} = \mathsf{true}$ and $\mathsf{index} = \mathtt{LAST} - 1)\{$
    Retrieve penultimate state as $(pid, B_C, B_M) = \mathbb{C}[cid].\mathsf{states\text{-}list}[\mathtt{LAST} - 1]$
    Set $\mathbb{C}[cid].\mathsf{status} = \mathsf{closed}$
    Send $(\mathsf{closed}, cid, \mathsf{honest}, B_C, B_M)$ to $C_i, M$
  // Close on stale state.
  $\}$ elif $(C_i \in \mathcal{I}$ and $0 \leq \mathsf{index} < \mathtt{LAST})\{$
    Retrieve closing state as $(pid, B_C, B_M) = \mathbb{C}[cid].\mathsf{states\text{-}list}[\mathsf{index}]$
    // Note: status is not set to closed.
    Send $(\mathsf{closed}, cid, \mathsf{dishonest}, pid, B_C, B_M)$ to $C_i, M$
  $\}$
```

Modeling Communication and Players. Throughout both of the following experiments, we assume that the messaging happens during synchronous rounds and than only one player sends a message in any given round. Specifically, each messaging round is assigned to a particular player, during which that player has the opportunity to send a messages; The assignment of each consecutive messaging round rotates between players. Additionally, we assume that all messages are sent over point-to-point channels that are fully secure; that is, the adversary cannot see messages sent between honest players or between honest players and an ideal functionality. This simplified communication model allows us to focus on analyzing what the cryptographic protocol we present achieves, and removes concerns about race conditions and asynchronous communication.

Additionally, we choose to model the players in our experiments as p.p.t. stateful algorithms. These algorithms are fed explicit strategies when initialized; during each time step, the algorithm can be queried to determine what messages, if any, should be sent during their messaging round. For simplicity, we assume that during a player's messaging round, the experiment orchestrator (or the environment) runs the algorithm on all messages received since its last messaging round. The algorithm then produces the message it wishes to send during the messaging round. The orchestrator then delivers that message to the appropriate entity during the appropriate messaging round. During a messaging round belonging to a corrupted player, the adversary instead chooses what message to send on behalf of the corrupted player.

Ideal Experiment. In the **Ideal** experiment, the parties interaction is mediated through $\mathcal{F}_{\mathsf{zkAbacus}}$. At experiment initialization, each party is assigned a strategy as input that determines the messages they will send.[5] Then, the ideal p.p.t. adversary $\mathcal{S}$ corrupts an admissible set $\mathcal{I} \subset \{C_1, \ldots, C_\ell, M\}$. Note that the special arbiter player J cannot be corrupted. The experiment proceeds as described above: the players take turn sending messages to the ideal functionality, which will often react by sending a message, possibly to a different player.

The adversary determines when the experiment ends. When the experiment ends, the players that were not corrupted generate some output. Specifically, customers output tuples of the form (cid, B^C, B^M) for each channel between them and the merchant that have not been closed, where cid is the channel identifier, and (B^C, B^M) is the current balance in the channel. Additionally, the customer outputs tuples of the same form for each closed channels, but include the closing balance for channel instead. The merchant outputs tuples of the same form for each open channel, but includes the initial balances of that channel instead. For each closed channel, the merchant outputs the same tuple as the customer. For each message that the arbiter received during the experiment, it outputs a tuple of the form $(cid, \{\mathsf{honest}, \mathsf{dishonest}\}, B^C, B^M)$ corresponding to the values contained in that message. Finally, the corrupt parties output nothing, but the adversary outputs any probabilistic polynomial-time computable function of the corrupted parties' views.

The output of the **Ideal** experiment, denoted by $\mathbf{Ideal}_{\mathcal{F}_{\mathsf{zkAbacus}}, \mathcal{S}(z), \mathcal{I}}(\lambda, \mathbf{x}, z)$, on input strategies $\mathbf{x}$, auxiliary input z to $\mathcal{S}$, admissible set of corrupted parties $\mathcal{I}$, and security parameter λ, consists of the tuple of outputs of the honest parties and the adversary.

Real Experiment. In the **Real** experiment, the parties interact with each other to run the real protocol Π_{zkAbacus}. At experiment initialization, each party is assigned a strategy as input that determines the messages they will send. Then, the real adversary $\mathcal{A}$ corrupts an admissible set $\mathcal{I} \subset \{C_1, \ldots, C_\ell, M\}$. Note that the special arbiter player J cannot be corrupted. The experiment proceeds as described above: the players take turn sending messages to the ideal functionality, which will often react by sending a message, possibly to a different player.

The adversary determines when the experiment ends. When the experiment ends, the players that were not corrupted generate some output. Specifically, customers output tuples of the form (cid, B^C, B^M) for each channel between them and the merchant that have not been closed, where cid is the channel identifier, and (B^C, B^M) is the current balance in the channel. Additionally, the

[5] If randomness is required, random coins can be supplied as additional input

customer outputs tuples of the same form for each closed channels, but include the closing balance for channel instead. The merchant outputs tuples of the same form for each open channel, but includes the initial balances of that channel instead. For each closed channel, the merchant outputs the same tuple as the customer. For each message that the arbiter received during the experiment, it outputs a tuple of the form $(cid, \{\mathsf{honest}, \mathsf{dishonest}\}, B^C, B^M)$ corresponding to the values contained in that message. Finally, the corrupt parties output nothing, but the adversary outputs any probabilistic polynomial-time computable function of the corrupted parties' views.

The output of the **Real** experiment, denoted by $\mathbf{Real}_{\Pi_{\mathsf{zkAbacus}},\mathcal{A}(z),\mathcal{I}}(\lambda, \mathbf{x}, z)$, on input strategies $\mathbf{x}$, auxiliary input z to $\mathcal{A}$, admissible set of corrupted parties $\mathcal{I}$, and security parameter λ, consists of the tuple of outputs of the honest parties and the adversary.

Theorem 1. *The protocol Π_{zkAbacus} securely realizes the ideal functionality $\mathcal{F}_{\mathsf{zkAbacus}}$ in the presence of malicious adversaries. That is, for every p.p.t. real-world adversary $\mathcal{A}$, there exist a ideal-world p.p.t. adversary $\mathcal{S}$ such that for every admissible set of corrupted parties $\mathcal{I}$*

$$\left\{\mathbf{Ideal}_{\mathcal{F}_{\mathsf{zkAbacus}},\mathcal{S}(z),\mathcal{I}}(\lambda, \mathbf{x}, z)\right\}_{\lambda,\mathbf{x},z} \overset{c}{\approx} \left\{\mathbf{Real}_{\Pi_{\mathsf{zkAbacus}},\mathcal{A}(z),\mathcal{I}}(\lambda, \mathbf{x}, z)\right\}_{\lambda,\mathbf{x},z}.$$

B.4. Proof. TODO NOTES

- Double check that we dont need to simulate the situation in which both the customer and the merchant are corrupt. In particular, this would mean that we dont have to simulate the case where the sim/merchant receives a FROZEN message during PAY.
- Customer Hybrid for ZK soundness?

B.4.1. *Simulating a Customer.* .

Setup. The simulator starts by sampling the merchant's keypairs pk, sk and pk',sk' on its own. Recall that we need not simulate the interactions between a corrupt client and a corrupt merchant, as there is no private information between them. The simulator then initializes a signature table $\mathcal{T}$ that maintains tuples of messages and signatures that it signs. This table is initialized as empty.

Simulating Initialization. When the customer starts the Initialize subprotocol on public inputs pk, cid, B_0^C, B_0^M, the simulator forwards the request to the ideal functionality and runs the merchant side of the protocol as appropriate. More formally, when the customer starts the Initialize subprotocol on public inputs pk, cid, B_0^C, B_0^M, $\mathcal{S}$ does the following:

- If cid has already been used, $\mathcal{S}$ halts.
- $\mathcal{S}$ received the message A', A'', π from the customer.
- $\mathcal{S}$ checks that π verifies with respect to A', A'', cid, $close$, B_0^C, and B_0^M. If the verification fails, $\mathcal{S}$ halts.
- Using the extractor of the NIZKPoK scheme, $\mathcal{S}$ extracts the witness to π as
$$(n_0, rl_0, \tau_0, \overline{\tau}_0)$$
. If the extractor fails, $\mathcal{S}$ aborts with the error $\mathsf{Error}_{\text{EXTRACT}}$
- $\mathcal{S}$ computes $\widetilde{\sigma_0} = \Pi_{\mathsf{ZKSig}}.\mathsf{BlindSign}(sk'_M, A')$ and sends $\widetilde{\sigma_0}$ to the customer.
- $\mathcal{S}$ computes $\sigma_0 = \Pi_{\mathsf{ZKSig}}.\mathsf{Unblind}(\widetilde{\sigma_0}, \overline{\tau})$ and records $(\sigma_0, \bar{s}_0)$ in $\mathcal{T}$.
- $\mathcal{S}$ updates $S_0 = S_0 \cup (cid, A'')$
- $\mathcal{S}$ sends $(\mathsf{initialize\text{-}channel}, cid, B_0^C, B_0^M)$ to $\mathcal{F}_{\mathsf{zkAbacus}}$, and receives $(\mathsf{initialized}, cid, C_i, B_C^0, B_M^0)$ in response.

Simulating Activation. When the merchant activates a channel, $\mathcal{F}_{\mathsf{zkAbacus}}$ sends a notification $(\mathsf{activated}, cid)$ to the simulator. $\mathcal{S}$ then executes the Activate subprotocol on public inputs (pk, cid), playing the role of the merchant. More formally, when $\mathcal{S}$ receives $(\mathsf{activated}, cid)$ from $\mathcal{F}_{\mathsf{zkAbacus}}$, $\mathcal{S}$ does the following:

- $\mathcal{S}$ retrieves (cid, A'') from S_0.
- Generate and send $\widetilde{pt_0} = \Pi_{\mathsf{ZKSig}}.\mathsf{BlindSign}(sk, A'')$ to C

- $\mathcal{S}$ computes $(pt_0 = \Pi_{\mathsf{ZKSig}}.\mathsf{Unblind}(\widetilde{pt_0}, \tau_0)$, and records (s_0, pt_0) in $\mathcal{T}$.

Simulating Pay Offer. When the customer starts the Pay subprotocol on public inputs $(\mathsf{pp}, pk, \epsilon)$, the simulator runs the subprotocol with the customer to determine the malicious customer's strategy. More formally,

- $\mathcal{S}$ receives the message n_i, π, A, A', A'' from the customer. $\mathcal{S}$ then performs the following checks:
  - If $n_i \in S_1$, $\mathcal{S}$ halts.
  - If π does not verify, $\mathcal{S}$ halts.
  - Using the extractor of the NIZKPoK scheme, $\mathcal{S}$ extracts the witness to π as
  $$(cid, B_i^C, B_i^M, rl_i, n_{i+1}, rl_{i+1}, \tau_{i+1}, \overline{\tau}_{i+1}, \rho_i, pt_i)$$
  . If the extractor fails, $\mathcal{S}$ aborts with the error $\mathsf{Error}_{\text{EXTRACT}}$
  - $\mathcal{S}$ check that an entry $((cid, n_i, rl_i, B_i^C, B_i^M), pt_i)$ exists in $\mathcal{T}$. If not, $\mathcal{S}$ aborts with the error $\mathsf{Error}_{\text{SIGN}}$.
- $\mathcal{S}$ computes and sends $\widetilde{\sigma_{i+1}} = \Pi_{\mathsf{ZKSig}}.\mathsf{BlindSign}(sk, A')$ to the customer.
- $\mathcal{S}$ computes $\sigma_{i+1} = \Pi_{\mathsf{ZKSig}}.\mathsf{Unblind}(\widetilde{\sigma_{i+1}}, \overline{\tau}_{i+1})$ and records $(\bar{s}_{i+1}, \sigma_{i+1})$ in $\mathcal{T}$
- Upon receiving a message $\hat{rl}_i$, $\hat{rs}_i$, $\hat{\rho}_i$ from the customer in the same session, $\mathcal{S}$ performs the following checks:
  - If $(rl_i, \cdot) \in S_2$, $\mathcal{S}$ sends $(\mathsf{pay\text{-}offered}, cid, \epsilon, \mathsf{dishonest})$ to $\mathcal{F}_{\mathsf{zkAbacus}}$ and halts.
  - If $\Pi_{\mathsf{com}}.\mathsf{Decommit}(A, rl_i, \rho_i) \neq \mathsf{true}$, $\mathcal{S}$ send $(\mathsf{pay\text{-}offered}, cid, \epsilon, \mathsf{dishonest})$ to $\mathcal{F}_{\mathsf{zkAbacus}}$ and halts.
  - If $\hat{rl}_i \neq rl_i$ extracted from π, $\mathcal{S}$ aborts with the error $\mathsf{Error}_{\text{BINDING}}$.
  - If $rl_i \neq \mathsf{H}(rs_i)$, $\mathcal{S}$ sends $(\mathsf{pay\text{-}offered}, cid, \epsilon, \mathsf{dishonest})$ to $\mathcal{F}_{\mathsf{zkAbacus}}$ and halts.
- $\mathcal{S}$ computes and sends $\widetilde{pt_{i+1}} = \Pi_{\mathsf{ZKSig}}.\mathsf{BlindSign}(sk, A'')$ to the customer.
- $\mathcal{S}$ computes $pt_{i+1} = \Pi_{\mathsf{ZKSig}}.\mathsf{Unblind}(\widetilde{pt_{i+1}}, \tau_{i+1})$ and records $(cid, n_{i+1}, rl_{i+1}, B_i^C - \epsilon, B_i^M + \epsilon, pt_{i+1})$ in $\mathcal{T}$
- $\mathcal{S}$ sends $(\mathsf{pay\text{-}offered}, cid, \epsilon, \mathsf{honest})$ to $\mathcal{F}_{\mathsf{zkAbacus}}$
- $\mathcal{S}$ updates $S_2' = S_2 \cup \{(rl_i, rs_i)\}$

Simulating Closure. When the customer starts the closure subprotocol, $\mathcal{S}$ determines the index into the states that the closure request represents. More formally, when $\mathcal{S}$ receives $((cid, close, rl_i, B_i^C, B_i^M), \sigma_i)$, $\mathcal{S}$ does the following:

- If $\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk, \bar{s}_i, \sigma_i) \neq \mathsf{true}$, $\mathcal{S}$ halts.
- If there is no entry $((cid, close, rl_i, B_i^C, B_i^M), \sigma_i)$ in $\mathcal{T}$, $\mathcal{S}$ aborts with an error $\mathsf{Error}_{\text{SIGN}}$.
- FIGURE OUT THE INDEX and send $(\mathsf{close}, cid, \mathsf{index})$ to $\mathcal{F}_{\mathsf{zkAbacus}}$.

B.4.2. *Hybrid argument.* .

We now show that real experiment and the ideal experiment are indistinguishable with a hybrid argument, starting with the real world interaction. We denote the distribution of the real world experiment to be $\mathcal{H}_0$.

$\mathcal{H}_1$: Let $\mathcal{H}_1$ be the same as $\mathcal{H}_0$, but the merchant key pair (pk, sk) is sampled by $\mathcal{S}$. Clearly $\mathcal{H}_0$ and $\mathcal{H}_1$ are distributed the same.

$\mathcal{H}_2$: Let $\mathcal{H}_2$ be the same as $\mathcal{H}_1$, but $\mathcal{S}$ extracts the witness from π using the extractor. If the extractor fails, $\mathcal{S}$ aborts with an error $\mathsf{Error}_{\text{EXTRACT}}$. By definition, the extractor fails with only negligible probability, and therefore the difference between $\mathcal{H}_1$ and $\mathcal{H}_2$ is negligible.

$\mathcal{H}_3$: Let $\mathcal{H}_3$ be the same as $\mathcal{H}_2$, but $\mathcal{S}$ aborts with an error $\mathsf{Error}_{\text{BINDING}}$ if the customer opens A to a different value than is extracted from π. By the binding property of the commitment scheme, the distance between $\mathcal{H}_3$ and $\mathcal{H}_2$ is negligible.

$\mathcal{H}_4$: Let $\mathcal{H}_4$ be the same as $\mathcal{H}_3$, but $\mathcal{S}$ aborts with an error $\mathsf{Error}_{\text{SIGN}}$ if the customer attempts to close using a signature σ_i that the $\mathcal{S}$ did not issue. By the unforagability property of the blind signature scheme, the distance between $\mathcal{H}_4$ and $\mathcal{H}_3$ is negligible.

$\mathcal{H}_5$: Let $\mathcal{H}_5$ be the same as $\mathcal{H}_4$, but $\mathcal{S}$ aborts with an error $\mathsf{Error}_{\text{SIGN}}$ if the signature pt_i extracted from π does not match any that the $\mathcal{S}$ did not issue. By the unforagability property of the blind signature scheme, the distance between $\mathcal{H}_5$ and $\mathcal{H}_4$ is negligible.

We note that $\mathcal{H}_5$ is distributed the same as $\mathcal{S}$ presented above. Thus, we conclude this part of the proof.

B.4.3. *Simulating a Merchant.* .

Setup. The simulator starts by initializing an empty table of rejected channels $\mathcal{B}$. This table is used to record when the merchant does not correctly initialize a channel. Additionally, the simulator initializes an empty table of signatures produced by the merchant during channel activation, $\mathcal{T}$.

Simulating Initialization. When a customer C_i initializes a channel with $\mathcal{F}_{\mathsf{zkAbacus}}$, $\mathcal{F}_{\mathsf{zkAbacus}}$ sends a notification $(\mathsf{initialized}, cid, C_i, B_0^C, B_0^M)$ to the simulator. $\mathcal{S}$ then executes the $\mathsf{Initialize}$ subprotocol on public inputs pk, cid, B_0^C, B_0^M, playing the role of the customer. More formally, when $\mathcal{S}$ receives $(\mathsf{initialized}, cid, C_i, B_0^C, B_0^M)$ from $\mathcal{F}_{\mathsf{zkAbacus}}$, $\mathcal{S}$ does the following:

- If cid has already been used, $\mathcal{S}$ aborts with an error $\mathsf{Error}_{\mathrm{CID}}$.
- $\mathcal{S}$ generates the message A', A'', π as the honest customer would in the protocol, and sends the message to the merchant.
- When $\mathcal{S}$ receives $\widetilde{\sigma_0}$ from the merchant, it performs the following checks:
  - If $\widetilde{\sigma_0} \notin \mathcal{Y}$, $\mathcal{S}$ records $(cid, \mathsf{blocked})$ in $\mathcal{B}$
  - $\mathcal{S}$ computes $\sigma_0 = \Pi_{\mathsf{ZKSig}}.\mathsf{Unblind}(\widetilde{\sigma_0}, \overline{\tau})$. If $\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk, \bar{s}_0, \sigma_0) \neq \mathsf{true}$, $\mathcal{S}$ records $(cid, \mathsf{blocked})$ in $\mathcal{B}$

Simulating Activation. When the merchant starts the $\mathsf{Activate}$ subprotocol on public inputs (pk, cid), the simulator forwards the activation request $(\mathsf{activated}, cid)$ to $\mathcal{F}_{\mathsf{zkAbacus}}$. More formally, when $\mathcal{S}$ starts the $\mathsf{Activate}$ subprotocol on public inputs (pk, cid), $\mathcal{S}$ does the following:

- If $(cid, \mathsf{blocked})$ in $\mathcal{B}$, $\mathcal{S}$ halts.
- When $\mathcal{S}$ receives $\widetilde{pt_0}$ from the merchant, it performs the following:
  - Computes $pt_0 = \Pi_{\mathsf{ZKSig}}.\mathsf{Unblind}(\widetilde{pt_0}, \tau)$
  - If $\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk, \bar{s}_0, \sigma_0) \neq \mathsf{true}$, $\mathcal{S}$ $\mathcal{S}$ records $(cid, \mathsf{blocked})$ in $\mathcal{B}$ and halts.
  - Adds (cid, σ_0) (generated during initialization, above) to $\mathcal{T}$.
- $\mathcal{S}$ sends $(\mathsf{activated}, cid)$ to $\mathcal{F}_{\mathsf{zkAbacus}}$

Simulating Pay Offer Response. When the customer initiates pay, the $\mathcal{S}$ will receive the notification $(\mathsf{pay\text{-}offered}, \mathbb{C}[cid].pid, \epsilon)$. $\mathcal{S}$ then executes the Pay subprotocol on public inputs $(\mathsf{pp}, pk, \epsilon)$ playing the role of the customer. More formally, when $\mathcal{S}$ received $(\mathsf{pay\text{-}offered}, \mathbb{C}[cid].pid, \epsilon)$, $\mathcal{S}$ does the following:

- $\mathcal{S}$ samples a fresh nonce n_i, a fresh revocation key pair rl_i, rs_i, and then computes A' and A'' on arbitrary values. Additionally, $\mathcal{S}$ forms a commitment A to rl_i under randomness ρ_i.
- $\mathcal{S}$ simulates the proof π
- $\mathcal{S}$ sends the message n_i, π, A, A', A'' to the merchant.
- When $\mathcal{S}$ receives the message $\widetilde{\sigma_{i+1}}$ from the merchant, it performs the following checks:
  - $\mathcal{S}$ computes $\sigma_{i+1} = \Pi_{\mathsf{ZKSig}}.\mathsf{Unblind}(\widetilde{\sigma_{i+1}}, \overline{\tau}_{i+1})$
  - If $\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk, \bar{s}_{i+1}, \sigma_{i+1}) \neq \mathsf{true}$, $\mathcal{S}$ sends $(\mathsf{pay\text{-}instr}, pid, \mathsf{reject})$ to $\mathcal{F}_{\mathsf{zkAbacus}}$ and halts.
- $\mathcal{S}$ sends rl_i, rs_i, ρ_i to the merchant.
- When $\mathcal{S}$ receives the message $\widetilde{pt_{i+1}}$ from the merchant, it performs the following checks:
  - $\mathcal{S}$ computes $pt_{i+1} = \Pi_{\mathsf{ZKSig}}.\mathsf{Unblind}(\widetilde{pt_{i+1}}, \tau_{i+1})$
  - If $\Pi_{\mathsf{ZKSig}}.\mathsf{Verify}(pk, s_{i+1}, pt_{i+1}) \neq \mathsf{true}$, $\mathcal{S}$ sends $(\mathsf{pay\text{-}instr}, pid, \mathsf{freeze\text{-}after\text{-}pay})$ to $\mathcal{F}_{\mathsf{zkAbacus}}$ and halts.
- $\mathcal{S}$ sends $(\mathsf{pay\text{-}instr}, pid, \mathsf{honest\text{-}accept})$ to $\mathcal{F}_{\mathsf{zkAbacus}}$ and halts.

Simulating Closure. When an honest customer closes a channel, $\mathcal{S}$ receives a notification $(\mathsf{closed}, cid, \mathsf{honest}, B_C, B_M)$ from $\mathcal{F}_{\mathsf{zkAbacus}}$. $\mathcal{S}$ needs to present a signed closure state to the merchant for channel cid on the balances B_C and B_M. However, $\mathcal{S}$ is unlikely to have received a signature on such a closing state. To retrieve such a signature, $\mathcal{S}$ rewinds the merchant to the last sucsessful run of the pay protocols and substitutes the values cid, B_C and B_M in the closing state. More formally, when $\mathcal{S}$ receives $(\mathsf{closed}, cid, \mathsf{honest}, B_C, B_M)$ from $\mathcal{F}_{\mathsf{zkAbacus}}$, $\mathcal{S}$ does the following:

- If there has been no successful runs of the pay protocol with the merchant:
  - $\mathcal{S}$ retrieves (cid, σ_0) from $\mathcal{T}$.
  - $\mathcal{S}$ sends $((cid, close, rl_0, B_C, B_M), \sigma_0)$ to the merchant and halts
- Otherwise, $\mathcal{S}$ rewinds the merchant to the most recent successful run of the pay protocol with the merchant and does the following:
  - $\mathcal{S}$ reruns the pay protocol with the merchant, but form the commitment A' as a commitment to the tuple $\bar{s} = (cid, close, rl, B_C, B_M)$ where rl is freshly sampled by running $\Pi_{\mathsf{Rlock}}.\mathsf{KeyGen}()$. Denote the signature recovered from this run of the pay protocol as σ.
  - $\mathcal{S}$ runs the entire simulation forward. If at any point the merchant deviates from its prior behavior, $\mathcal{S}$ aborts with the error $\mathsf{Error}_{\text{BLINDNESS}}$
  - $\mathcal{S}$ sends $((cid, close, rl_0, B_C, B_M), \sigma)$ to the merchant

B.4.4. *Hybrid argument.* .

We now show that real experiment and the ideal experiment are indistinguishable with a hybrid argument, starting with the real world interaction. We denote the distribution of the real world experiment to be $\mathcal{H}_0$.

$\mathcal{H}_1$: Let $\mathcal{H}_1$ be the same as $\mathcal{H}_0$, but $\mathcal{S}$ simulates the proof π in the pay subprotocol instead of generating it honestly. By the zero-knowledge property of the NIZK, the difference between $\mathcal{H}_0$ and $\mathcal{H}_1$ is negligible.

$\mathcal{H}_2$: Let $\mathcal{H}_2$ be the same as $\mathcal{H}_1$, but $\mathcal{S}$ commits to arbitrary values in A, A', A'' instead of honestly. By the hiding property of the commitment schemes, the difference between $\mathcal{H}_1$ and $\mathcal{H}_2$ is negligible.

$\mathcal{H}_3$: Let $\mathcal{H}_3$ be the same as $\mathcal{H}_2$, but $\mathcal{S}$ runs the blind signing protocol on the arbitrary contents within the commitments A, A', A''. By the blindness property of the signature scheme, the blinded version of the message reveals nothing about the message itself, and therefore the $\mathcal{H}_2$ and $\mathcal{H}_3$ are distributed the same.

$\mathcal{H}_4$: Let $\mathcal{H}_4$ be the same as $\mathcal{H}_3$, but instead of closing with signatures generated during regular runs of the pay protocol, $\mathcal{S}$ instead generates closing signatures σ by rewinding the merchant to the most recent successful run of the pay protocol and replacing the closing state with the desired values. $\mathcal{S}$ then runs the protocol forward again. If the merchant deviates from its behavior before the rewinding, $\mathcal{S}$ aborts with the error $\mathsf{Error}_{\text{BLINDNESS}}$ (As described above in the simulator, if there had been no successful run of the pay protocol, $\mathcal{S}$ uses the closing signature produced during initialization). By the blindness property of the signature scheme, the distribution of the changed message is the same. Therefore, by the forking lemma, the distance between $\mathcal{H}_3$ and $\mathcal{H}_4$ is negligible.

We note that $\mathcal{H}_4$ is distributed the same as $\mathcal{S}$ presented above. Thus, we conclude this part of the proof.

C. Draft $\Pi_{\mathsf{zkEscrowAgent}}$ ideal functionality

```
on $(\mathsf{Register}, cid, acct\text{-}id_C, acct\text{-}id_M, pk_C, pk_M, pk'_M, B_0^C, B_0^M)$ from $P \in \{M, C_1, \ldots, C_\ell\}$ :
  if $(\exists J[cid])$ return
  Set $J[cid] = (\mathsf{registered}, P, B_C^0, B_M^0, 0, \mathsf{unfunded}, \mathsf{unfunded})$
  Record $(\mathsf{Register}, cid, P, B_0^C, B_0^M)$ and send it to all players

on $(\mathsf{Fund}, cid, role, \sigma)$ from $P \in \{M, C_1, \ldots, C_\ell\}$ :
  if $(\nexists\ J[cid]J[cid].\mathsf{status} \neq \mathsf{registered})$ return
  if $(P = J[cid].C$ and $J[cid].\mathsf{funded\text{-}status}_C = \mathsf{unfunded})$ {
    if $(J[P] < J[cid].B_C^0)$ return
    Set $J[P] -= J[cid].B_C^0, J[cid].\mathsf{balance} += B_C^0$
    Set $J[cid].\mathsf{funded\text{-}status}_C = \mathsf{funded}$
  } elif $(P = M$ and $J[cid].\mathsf{funded\text{-}status}_M = \mathsf{unfunded})$ {
    if $(J[P] < J[cid].B_M^0)$ return
    Set $J[P] -= J[cid].B_M^0, J[cid].\mathsf{balance} += B_M^0$
    Set $J[cid].\mathsf{funded\text{-}status}_M = \mathsf{funded}$
  }
  if $(J[cid].\mathsf{funded\text{-}status}_C = \mathsf{funded}$ and $J[cid].\mathsf{funded\text{-}status}_M = \mathsf{funded})$ {
    Set $J[cid].\mathsf{status} = \mathsf{funded}$
  }

on $(\mathsf{Disburse}, cid, B_C, B^M, (rl, \sigma_{\mathsf{ZKSig}}, \sigma_{\mathsf{Sig}}^C), (rs, \sigma_{\mathsf{Sig}}^M))$ from $P \in \{M, C_1, \ldots, C_\ell\}$ :

on $(\mathsf{Expiry}, cid, \sigma_{\mathsf{Sig}}^M)$ from $P \in \{M, C_1, \ldots, C_\ell\}$ :
```

Figure C.2. Arbiter ideal functionality J.

Assumes access to a table $\mathbb{C}$ indexed by cid with entries of the form

$$(\mathsf{status}, C, \epsilon, pid, \mathsf{cust\text{-}strat}, \mathsf{states\text{-}list}, \mathsf{aborted\text{-}pay\text{-}close})$$

```
on $(\mathsf{Register}, cid, B_0^C, B_0^M)$ from $P \in \{M, C_1, \ldots, C_\ell\}$ :
  if $(\exists J[cid])$ return
  Set $J[cid] = (\mathsf{registered}, P, B_C^0, B_M^0, 0, \mathsf{unfunded}, \mathsf{unfunded})$
  Record $(\mathsf{Register}, cid, P, B_0^C, B_0^M)$ and send it to all players

on $(\mathsf{Fund}, cid)$ from $P \in \{M, C_1, \ldots, C_\ell\}$ :
  if $(\nexists\ J[cid] J[cid].\mathsf{status} \neq \mathsf{registered})$ return
  if $(P = J[cid].C$ and $J[cid].\mathsf{funded\text{-}status}_C = \mathsf{unfunded})$ {
    if $(J[P] < J[cid].B_C^0)$ return
    Set $J[P] -= J[cid].B_C^0, J[cid].\mathsf{balance} += B_C^0$
    Set $J[cid].\mathsf{funded\text{-}status}_C = \mathsf{funded}$
  } elif $(P = M$ and $J[cid].\mathsf{funded\text{-}status}_M = \mathsf{unfunded})$ {
    if $(J[P] < J[cid].B_M^0)$ return
    Set $J[P] -= J[cid].B_M^0, J[cid].\mathsf{balance} += B_M^0$
    Set $J[cid].\mathsf{funded\text{-}status}_M = \mathsf{funded}$
  }
  if $(J[cid].\mathsf{funded\text{-}status}_C = \mathsf{funded}$ and $J[cid].\mathsf{funded\text{-}status}_M = \mathsf{funded})$ {
    Set $J[cid].\mathsf{status} = \mathsf{funded}$
  }

on $(\mathsf{Expiry}, cid)$ from $M$ :
  if $(\nexists\ J[cid]$ or $J[cid].\mathsf{status} \neq \mathsf{funded})$ return
  Set $J[cid].\mathsf{status} = \mathsf{expiry}$
```

```
on $(\mathsf{Disburse}, cid, \mathsf{index})$ from $P \in \{M, C_1, \ldots, C_\ell\}$ :
  if $(\nexists\ J[cid]$ or $P \notin \{J[cid].C, M\})$ return
  Retrieve state as $(\cdot, B_C, B_M) = \mathbb{C}[cid].\mathsf{states\text{-}list}[\mathsf{index}]$
// Mutual Close
if $(P = J[cid].C$ and $J[cid].\mathsf{status} = \mathsf{funded}$ and $\mathbb{C}[cid].\mathsf{status} = \mathsf{closed})\{$
    Set $J[J[cid].C] \mathrel{+}= B_C, J[M] \mathrel{+}= B_M, J[cid].\mathsf{balance} \mathrel{-}= (B_C + B_M)$
    Set $J[cid].\mathsf{status} = \mathsf{closed}$
  $\}$
// Honest Customer Initiated Close
 elif $(P = J[cid].C$ and $J[cid].\mathsf{status} = \mathsf{funded}$ and $\mathsf{index} = \mathbb{C}[cid].\mathtt{LAST})\{$
    Set $J[M] \mathrel{+}= B_M, J[cid].\mathsf{balance} = 0$
    Set $J[cid].\mathsf{timer} = \mathsf{currentTime} + \delta$
    Set $J[cid].\mathsf{status} = \mathsf{closed}, \mathbb{C}[cid].\mathsf{status} = \mathsf{closed}$
    After $\delta$ time passes, set $J[J[cid].C] \mathrel{+}= B_C$
  $\}$
// Dishonest Customer Initiated Close
 elif $(P = J[cid].C$ and $P \in \mathcal{I}$ and $J[cid].\mathsf{status} = \mathsf{funded}$ and $\mathsf{index} < \mathbb{C}[cid].\mathtt{LAST})\{$
    Set $J[M] \mathrel{+}= B_M, J[cid].\mathsf{balance} \mathrel{-}= B_M$
    Set $J[cid].\mathsf{timer} = \mathsf{currentTime} + \delta$
    Set $J[cid].\mathsf{status} = \mathsf{pending}$
  $\}$
// Honest Customer Expiry Close
 elif $(P = J[cid].C$ and $J[cid].\mathsf{status} = \mathsf{expiry}$ and $\mathsf{index} = \mathbb{C}[cid].\mathtt{LAST})\{$
    Set $J[M] \mathrel{+}= B_M, J[cid].\mathsf{balance} = 0$
    Set $J[cid].\mathsf{timer} = \mathsf{currentTime} + \delta$
    Set $J[cid].\mathsf{status} = \mathsf{closed}, \mathbb{C}[cid].\mathsf{status} = \mathsf{closed}$
    After $\delta$ time passes, set $J[J[cid].C] \mathrel{+}= B_C$
  $\}$
// Dishonest Customer Expiry Close
 elif $(P = J[cid].C$ and $P \in \mathcal{I}$ and $J[cid].\mathsf{status} = \mathsf{expiry}$ and $\mathsf{index} < \mathbb{C}[cid].\mathtt{LAST})\{$
    Set $J[M] \mathrel{+}= B_M, J[cid].\mathsf{balance} \mathrel{-}= B_M$
    Set $J[cid].\mathsf{timer} = \mathsf{currentTime} + \delta$
    Set $J[cid].\mathsf{status} = \mathsf{pending}$
  $\}$
// Merchant Dispute
 elif $(P = M$ and $P \in \mathcal{I}$ and $J[cid].\mathsf{status} = \mathsf{pending})\{$
    Set $J[M] \mathrel{+}= J[cid].\mathsf{balance}, J[cid].\mathsf{balance} = 0$
    Set $J[cid].\mathsf{status} = \mathsf{closed}$
  $\}$
// Cashout
 elif $(J[cid].\mathsf{timer} < \mathsf{currentTime}$ and $((P = C$ and $J[cid].\mathsf{status} = \mathsf{pending}\})$
        or $(P = M$ and $J[cid].\mathsf{status} = \mathsf{expiry}))\{$
    Set $J[P] \mathrel{+}= J[cid].\mathsf{balance}, J[cid].\mathsf{balance} = 0$
    Set $J[cid].\mathsf{status} = \mathsf{closed}$
  $\}$
```

Modeling Communication and Players. The communication and player models are the same as in Appendix B.

Ideal Experiment. In the **Ideal** experiment, the parties interaction is mediated through $\mathcal{F}_{\mathsf{zkEscrowAgent}}$. At experiment initialization, each party is assigned a strategy as input that determines the messages they will send.[6] Then, the ideal p.p.t. adversary $\mathcal{S}$ corrupts an admissible set $\mathcal{I} \subset \{C_1, \ldots, C_\ell, M\}$. Note that the special arbiter player J cannot be corrupted. The experiment proceeds as described above: the players take turn sending messages to the ideal functionality, which will often react by sending a message, possibly to a different player.

The adversary determines when the experiment ends. When the experiment ends, the players that were not corrupted generate some output. Specifically, customers output tuples of the form (cid, B^C, B^M) for each channel between them and the merchant that have not been closed, where cid is the channel identifier, and (B^C, B^M) is the current balance in the channel. Additionally, the customer outputs tuples of the same form for each closed channels, but include the closing balance for channel instead. The merchant outputs tuples of the same form for each open channel, but includes the initial balances of that channel instead. For each closed channel, the merchant outputs the same tuple as the customer. For each message that the arbiter received during the experiment, it outputs a tuple of the form $(cid, \{\mathsf{honest}, \mathsf{dishonest}\}, B^C, B^M)$ corresponding to the values contained in that message. Finally, the corrupt parties output nothing, but the adversary outputs any probabilistic polynomial-time computable function of the corrupted parties' views.

The output of the **Ideal** experiment, denoted by $\mathbf{Ideal}_{\mathcal{F}_{\mathsf{zkEscrowAgent}}, \mathcal{S}(z), \mathcal{I}}(\lambda, \mathbf{x}, z)$, on input strategies $\mathbf{x}$, auxiliary input z to $\mathcal{S}$, admissible set of corrupted parties $\mathcal{I}$, and security parameter λ, consists of the tuple of outputs of the honest parties and the adversary.

Real Experiment. In the **Real** experiment, the parties interact with each other to run the real protocol Π_{zkAbacus}. At experiment initialization, each party is assigned a strategy as input that determines the messages they will send. Then, the real adversary $\mathcal{A}$ corrupts an admissible set $\mathcal{I} \subset \{C_1, \ldots, C_\ell, M\}$. Note that the special arbiter player J cannot be corrupted. The experiment proceeds as described above: the players take turn sending messages to the ideal functionality, which will often react by sending a message, possibly to a different player.

The adversary determines when the experiment ends. When the experiment ends, the players that were not corrupted generate some output. Specifically, customers output tuples of the form (cid, B^C, B^M) for each channel between them and the merchant that have not been closed, where cid is the channel identifier, and (B^C, B^M) is the current balance in the channel. Additionally, the customer outputs tuples of the same form for each closed channels, but include the closing balance for channel instead. The merchant outputs tuples of the same form for each open channel, but includes the initial balances of that channel instead. For each closed channel, the merchant outputs the same tuple as the customer. For each message that the arbiter received during the experiment, it outputs a tuple of the form $(cid, \{\mathsf{honest}, \mathsf{dishonest}\}, B^C, B^M)$ corresponding to the values contained in that message. Finally, the corrupt parties output nothing, but the adversary outputs any probabilistic polynomial-time computable function of the corrupted parties' views.

The output of the **Real** experiment, denoted by $\mathbf{Real}_{\Pi_{\mathsf{zkAbacus}}, \mathcal{A}(z), \mathcal{I}}(\lambda, \mathbf{x}, z)$, on input strategies $\mathbf{x}$, auxiliary input z to $\mathcal{A}$, admissible set of corrupted parties $\mathcal{I}$, and security parameter λ, consists of the tuple of outputs of the honest parties and the adversary.

THEOREM 2. *The protocol Π_{zkAbacus} securely realizes the ideal functionality $\mathcal{F}_{\mathsf{zkEscrowAgent}}$ in the presence of malicious adversaries. That is, for every p.p.t. real-world adversary $\mathcal{A}$, there exist a ideal-world p.p.t. adversary $\mathcal{S}$ such that for every admissible set of corrupted parties $\mathcal{I}$*

$$\left\{\mathbf{Ideal}_{\mathcal{F}_{\mathsf{zkEscrowAgent}}, \mathcal{S}(z), \mathcal{I}}(\lambda, \mathbf{x}, z)\right\}_{\lambda, \mathbf{x}, z} \overset{c}{\approx} \left\{\mathbf{Real}_{\Pi_{\mathsf{zkAbacus}}, \mathcal{A}(z), \mathcal{I}}(\lambda, \mathbf{x}, z)\right\}_{\lambda, \mathbf{x}, z}.$$

C.1. Proof. We will write a proof here!

[6] If randomness is required, random coins can be supplied as additional input