Blind Off-Chain Lightweight Transactions (BOLT)

Version 1.0

Authors

Matthew D. Green (mgreen@cs.jhu.edu)
Ian Miers (imiers@cs.jhu.edu)
J. Ayo Akinyele (ayo@yeletech.org)

Contents

Abstract

This document describes the design and implementation of the Blind Off-chain Lightweight Transactions (BOLT) library. The BOLT protocol comprises a number of techniques for enabling privacy-preserving unlinkable payment channels for decentralized crypto-currencies between pairs of individual parties. BOLT is designed to provide a "Layer 2" payment protocol for privacy-preserving crypto-currencies such as Zerocash (or Zcash) [5], by allowing individuals to establish and use payment channels for rapid or instantaneous payments that do not require an on-chain transaction. This document describes the cryptographic instantiations of the BOLT protocol according to the published paper by Matthew Green and Ian Miers [3].

The intended use of this document is for understanding BOLT and the associated software implementation in the Rust programming language. This document is hereby released to the public domain free of charge.

Chapter 1

Introduction

BOLT is a system for conducting privacy-preserving off chain payments between pairs of individual parties. We refer to these parties as "customers", "merchants" and "hubs", with definitions provided below. BOLT is designed to provide a "Layer 2" payment protocol for privacy-preserving cryptocurrencies such as Zcash, by allowing individuals to establish and use payment channels for rapid/instantaneous payments that do not require an on-chain transaction.

Parties and Terminology. Financial transactions in the BOLT system are conducted between two parties, possibly with the assistance of an intermediate third party. Each party runs the BOLT software. These parties fall into three categories, which we describe below:

- **Customers.** Customers are users who establish payment channels and initiate payment transactions to a merchant, possibly via an intermediate party known as a "hub". These payments may have positive or negative value, provided there are sufficient funds in the payment channel to allow the transaction.
- **Merchants.** Merchants are users who cooperate in the establishment of payment channels (with customers and hubs), and receive payments from customers or hubs.
- **Hubs.** In some settings, customers may pay merchants directly. In other settings, a customer may pay a merchant via an intermediate party known as a "hub". Hubs establish channels with both customer and merchant, and relay transactions (of positive or negative value) between two such parties.

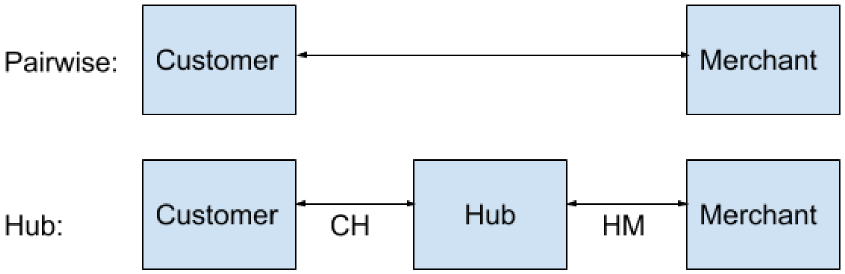
BOLT can be deployed in one of two settings, called "Pairwise" and "Hub" mode, as illustrated in Figure 1.1.

Note that these diagrams represent only one set of channel(s) between the parties. In practice, every party may have many relationships with different customers, merchants, or hubs.

Software components. Each BOLT participant requires must run the BOLT software, which consists of up to three pieces. These are:

- A full node for a BOLT-compatible cryptocurrency, *e.g.*, zcashd. This node is connected to the currency P2P network, and must support commands via an (*e.g.*, RPC) interface.

Figure 1.1: Pairwise and Hub modes for BOLT protocol



- A BOLT library (**libbolt**) that constructs and parses the messages required for interactive off-chain transactions with (one or more) remote BOLT participant(s), and interfaces with the cryptocurrency node via its interface.

The actual BOLT data transfer may be implemented by an application designer, via a channel such as HTTP or some alternative data transfer mechanism. Alternatively, parties can receive inbound connections using a dedicated daemon:

- A dedicated daemon (**boltd**) that implements BOLT communications with remote parties. This daemon uses HTTPS/JSON communications, incorporates libbolt and interfaces directly with the cryptocurrency node.

1.1 BOLT Privacy Guarantees

BOLT provides a more limited set of privacy guarantees than a standard payment network. This is inherent in the fact that BOLT uses payment channels, which are pairwise relationships that must be established between a Customer and Merchant before payment can take place.

The BOLT privacy guarantees can be summarized as follows:

- The Merchant will be aware of the number of channels she has open at any given time, and the total funding amount of each channel. We assuming an anonymous underlying currency network, so the Merchant does not know the identity of the Customer that opens each channel (this information should be protected by the underlying currency network).
- The Customer (who initiates transactions) always knows the identity of the Merchant she is paying, and the instantaneous balance of each of her open channels.
- The Merchant (who responds to transactions) does not learn the identity of the Customer, or which channel a payment is associated with. She knows only that the payment is associated with one of her current active payment channels.
- The sole exception to rule (3) is when a payment interaction fails or is disputed. In this case, the Merchant learns which channel is associated with the final (failed) payment, but cannot link this channel to any previous payments.

In practice this guarantee is sufficient to protect customer identities. Channels are not linked to customer identities (due to the anonymous payment network), and the merchant cannot link any previous payments to the failed payment or channel.

Chapter 2

BOLT Library

2.1 Overview

In this section, we describe the implementation of the BOLT library for an anonymous payment channel (APC). An APC is a construct established between two parties that interact via a payment network. An APC must be used in combination with a payment network capable of *conditionally* escrowing funds and *binding* these escrow transactions funds to some data. We assume the existence of such a payment network (*e.g.*, Zcash) and leave the details of the payment network outside the scope of this document.

2.2 Core Cryptographic Building Blocks

In this section, we describe the core cryptographic primitives required to implement the BOLT protocol. They include the following:

- Commitment Scheme
- Digital Signatures with efficient protocols
- Symmetric Key Encryption
- Pseudo-random Functions
- One-time Encryption
- Non-Interactive Zero-Knowledge Proofs of Knowledge

2.2.1 Commitment Scheme

BOLT instantiates a commitment primitive using the Pedersen commitment scheme [4]. The scheme has the following interface:

$\mathsf{CSetup}() \rightarrow PP$: the algorithm generates public parameters and outputs the PP.

$\mathsf{Commit}(PP, M; r) \rightarrow C$: On input parameters PP, a message $M \in \mathbb{G}$, and optional random coins r, the algorithm outputs a commitment C.

$\mathsf{Decommit}(PP, C, M) = \{0, 1\}$: On input parameters PP, and a tuple (C, M, r) outputs 1 if C is a valid commitment to the message or 0 otherwise.

Pedersen Commitments

We define Pedersen commitments over a vector of n messages as follows:

$\mathsf{CSetup}() \rightarrow PP$: the algorithm generates public parameters and outputs the $PP = (g_1, g_2, ..., g_n, h)$.

$\mathsf{Commit}(PP, M; r) \rightarrow C$: On input parameters PP, a message tuple $M = (m_1, m_2, ...m_n)$, and optional random coins r, the algorithm outputs a commitment $C = {g_1}^{m_1} \cdot {g_2}^{m_2} \ldots {g_n}^{m_n} \cdot h^r$ and r.

$\mathsf{Decommit}(PP, C, M) = \{0, 1\}$: On input parameters PP, and a tuple (C, M, r). Check if $C \stackrel{?}{=} {g_1}^{m_1} \cdot {g_2}^{m_2} \ldots {g_n}^{m_n} \cdot h^r$. Outputs 1 if C is a valid commitment to the message or 0 otherwise.

2.2.2 Symmetric-Key Encryption

BOLT includes an symmetric-key encryption (SymEnc) primitive (via XSalsa20-Poly1305) for providing confidentiality and integrity.

$\mathsf{Encrypt}(K, M, N) \rightarrow C$. The encryption algorithm takes as input a symmetric-key, a message $M \in \{0, 1\}^\ell$ and the associated nonce $N \in \{0, 1\}^k$. The algorithm outputs the ciphertext C.

$\mathsf{Decrypt}(K, C, N) = M \cup \perp$. The decryption algorithm takes as input a symmetric-key, the ciphertext C and the associated nonce N. The algorithm outputs the message M or returns $\perp$.

2.2.3 Digital Signatures with Efficient Protocols

BOLT includes signatures due to Camenisch and Lysyanskaya (CL) [1] which features: (1) a protocol for a user to obtain a signature on the value(s) in a commitment without the signer learning anything about the message(s), and (2) a non-interactive protocol for proving knowledge of a signature. We will first provide the definition of a signature scheme then provide the details of the signature and the associated protocols.

$\mathsf{SigKeygen}(\mathrm{PP}) \rightarrow (\mathrm{PK}, \mathrm{SK})$. The key generation algorithm takes as input a security parameter τ, runs the $\mathsf{ECSetup}(1^\tau)$ to select the elliptic curve parameters and outputs the public and secret key.

$\mathsf{Sign}(\mathrm{PP}, \mathrm{SK}, M) \rightarrow \sigma$. The signing algorithm takes as input public parameters, a secret key SK and message $m \in \{0, 1\}^*$ and outputs a signature σ.

$\mathsf{Verify}(\mathrm{PK}, M, \sigma) = \{true, false\}$. The verification algorithm takes as input a public key PK, the message m and the signature σ. The algorithm outputs $true$ if the signatures is valid with respect to M and PK. Otherwise, it outputs $false$.

CL Signature [1]

Run the Setup algorithm to generate public parameters $\mathrm{PP} = (q, \mathbb{G}_1, \mathbb{G}_2, g_1, g_2, e)$.

$\mathsf{SigKeygen}(\mathrm{PP}) \rightarrow (\mathrm{PK}, \mathrm{SK})$. The key generation algorithm chooses $x \leftarrow \mathbb{Z}_q$, $y \leftarrow \mathbb{Z}_q$ and for $1 \leq i \leq \ell$, choose $z_i \leftarrow \mathbb{Z}_q$. Let $X = g_1^x$, $Y = g_1^y$, for $1 \leq i \leq \ell$, set $Z_i = g_1^{z_i}$ and $W_i = Y^{z_i}$. Set $\mathrm{SK} = (x, y, z_1, \ldots, z_\ell)$ and $\mathrm{PK} = (X, Y, \{Z_i\}, \{W_i\})$.

Sign$(\mathrm{PP}, \mathrm{SK}, M) \rightarrow \sigma$. On input message $M = (m^{(0)}, m^{(1)}, \ldots, m^{(\ell)})$, and secret key $SK = (x, y, z_1, \ldots, z_\ell)$ and public parameters $\mathrm{PP} = (q, \mathbb{G}_1, \mathbb{G}_2, g_1, g_2, e)$, and computes the following:

1. choose a random $a \in \mathbb{G}_2$
2. Let $A_i = a^{z_i}$ for $1 \leq i \leq \ell$
3. Let $b = a^y$, $B_i = (A_i)^y$
4. Let $c = a^{x+xym^{(0)}} \cdot \prod_{i=1}^{\ell} A_i^{xym^{(i)}}$
5. Output signature $\sigma = (a, \{A_i\}, b, \{B_i\}, c)$

Verify$(\mathrm{PK}, M, \sigma) = \{true, false\}$. On input $\mathrm{PK} = (X, Y, \{Z_i\}, \{W_i\})$, message $M = (m^{(0)}, m^{(1)}, \ldots, m^{(\ell)})$ and signature $\sigma = (a, \{A_i\}, b, \{B_i\}, c)$, check the following:

1. $\{A_i\}$ were formed correctly: $e(Z_i, a) = e(g_1, A_i)$
2. b and $\{B_i\}$ were formed correctly: $e(Y, a) = e(g_1, b)$ and $e(Y, A_i) = e(g_1, B_i)$
3. c was formed correctly: $e(X, a) \cdot e(X, b)^{m^{(0)}} \cdot \prod_{i=1}^{\ell} e(X, B_i)^{m^{(i)}} = e(g_1, c)$

(1) Protocol for obtaining a signature on a committed value

Suppose $M = g_1^{m^{(0)}} \prod_{i=1}^{\ell} Z_i{}^{m^{(i)}}$ is a commitment to a vector of messages $(m^{(0)}, m^{(1)}, \ldots, m^{(\ell)})$ whose signature the user wishes to obtain. Then, the user and the signer execute the following protocol:

- The common input is the $\mathrm{PP} = (q, \mathbb{G}_1, \mathbb{G}_2, g_1, g_2, e)$, $\mathrm{PK} = (X, Y, \{Z_i\}, \{W_i\})$ and a commitment M.
- The user's input is the messages $m^{(0)}, m^{(1)}, \ldots, m^{(\ell)}$ that open to the commitment such that $M = g_1^{m^{(0)}} \prod_{i=1}^{\ell} Z_i{}^{m^{(i)}}$.
- The signer's input is the secret key $\mathrm{SK} = (x, y, z_1, \ldots, z_\ell)$.
- The user and signer engage in a zero-knowledge proof of knowledge of the opening to the commitment as follows:
  1. $PK\{(u^{(0)}, \cdots, u^{(\ell)}) : M = g_1^{m^{(0)}} \prod_{i=1}^{\ell} Z_i{}^{m^{(i)}}$
- The signer computes $\sigma = (a, \{A_i\}, b, \{B_i\}, c)$ as follows (using SK):
  1. Choose $\alpha \leftarrow \mathbb{Z}_q$,
  2. Set $a = g^\alpha$
  3. For $1 \leq i \leq \ell$, let $A_i = a^{z_i}$
  4. Set $b = a^y$
  5. For $1 \leq i \leq \ell$, let $B_i = A_i{}^y$
  6. Set $c = a^x M^{\alpha xy}$

(2) Protocol for proving knowledge of a signature

- The common input is the public parameters $\mathrm{PP} = (q, \mathbb{G}_1, \mathbb{G}_2, g_1, g_2, e)$ and public key $\mathrm{PK} = (X, Y, \{Z_i\}, \{W_i\})$
- The prover's input is the vector of messages $m^{(0)}, m^{(1)}, \ldots, m^{(\ell)}$ and signature $\sigma = (a, \{A_i\}, b, \{B_i\}, c)$
- The protocol is in three parts:
  1. The prover computes a blinded version of the signature σ as follows:
     - Chose random $r, r' \in \mathbb{Z}_q$
     - Compute $\hat{\sigma} = (\tilde{a}, \{\tilde{A}_i\}, \tilde{b}, \{\tilde{B}_i\}, \hat{c})$ as follows: $\tilde{a} = a^r$, $\tilde{b} = b^r$, $\hat{c} = (c^r)^{r'}$, $\tilde{A} = {A_i}^r$ and $\tilde{B}_i = {B_i}^r$ for $1 \leq i \leq \ell$
     - Send $\hat{\sigma}$ to verifier
  2. Both the prover and verifier compute v_x, v_{xy}, $\mathsf{v}_{\{xy,i\}}$ where $i = 1, \ldots, \ell$ and v_s as follows:
     (a) Compute $\mathsf{v}_x = e(X, \tilde{a})$
     (b) Compute $\mathsf{v}_{xy} = e(X, \tilde{b})$
     (c) Compute $\mathsf{v}_{\{xy,i\}} = e(X, \tilde{B}_i)$
     (d) Compute $\mathsf{v}_s = e(g_1, \hat{c})$
  3. The prover and verifier execute the following protocol:
     (a) Verify proof $\pi = PK\{(\mu^0, \ldots, \mu^\ell, \rho) : (\mathsf{v}_s)^\rho = \mathsf{v}_x {\mathsf{v}_{xy}}^{\mu^0} \prod_{i=1}^{\ell} (\mathsf{v}_{(xy,i)})^{\mu_i}\}$.
     (b) Check if $e(Z_i, \tilde{a}) \stackrel{?}{=} e(g_1, \tilde{A}_i)$ and $e(Y, \tilde{a}) \stackrel{?}{=} e(g_1, \tilde{b})$ and $e(Y, \tilde{A}_i) \stackrel{?}{=} e(g_1, \tilde{B}_i)$.

2.2.4 Pseudo-random Functions (PRF)

For the unidirectional construction, BOLT includes a pseudo-random function F that supports efficient proofs of knowledge. F is instantiated using the Dodis-Yampolskiy PRF [2], the public parameters are a group $\mathbb{G}_1$ of prime order q with generator g. The seed is a random value $s \in \mathbb{Z}_q$ and the function is computed as $F_s(x) = g^{1/(s+x)}$.

2.2.5 One-Time Encryption

For the bidirectional construction, BOLT includes a IND-CPA secure one-time encryption scheme with a keyspace that is also the range of the pseudo-random function (PRF) described in Section 2.2.4. In addition, the message space is the domain of the public key for the CL signature scheme instantiated in Section 2.2.3.

$\mathsf{OTKeyGen}(\tau) \rightarrow K$. On input parameters, the algorithm outputs a random key, $K \in \mathbb{G}_1$.

$\mathsf{OTEnc}(K, M) \rightarrow C$. The algorithm takes as input a one-time key K and a message tuple $(M_1, M_2) \in \mathbb{G}_1$ and outputs a ciphertext C.

$\mathsf{OTDec}(K, C) = M$ or $\perp$. The algorithm takes as input a key K and the ciphertext C and outputs the message tuple as M or $\perp$.

2.2.6 Non-Interactive Zero Knowledge Proofs (NIZKP)

BOLT features non-interactive zero-knowledge proofs of knowledge for the purposes of proving statements about committed values:

1. a proof of knowledge of a committed value
2. a proof that a committed value is in a range

2.3 Constructions

2.3.1 Unidirectional Scheme

The unidirectional payment construction only supports payments from a customer to a merchant and only supports transfer of fixed-value coins. It consists of a tuple of possibly probabilistic algorithms (Setup, KeyGen, $\mathsf{Init}_\mathsf{C}, \mathsf{Init}_\mathsf{M}$, Refund, Refute, $\mathsf{Resolve}$) and two interactive protocols ($\mathsf{Establish}$, Pay).

1. $\mathsf{Setup}(1^\lambda) \rightarrow \mathsf{PP}$. On input λ, optionally generate CRS parameters for (1) a secure commitment scheme (see Section 2.2.1), (2) a non-interactive zero knowledge proof system (see Section 2.2.6). Output all of these as PP.

2. $\mathsf{KeyGen}(\mathsf{PP}) \rightarrow (pk, sk)$.

   - Compute $(pk, sk) \leftarrow \prod_{\mathsf{sig}} .\mathsf{SigKeygen}(1^\lambda)$.

3. $\mathsf{Init}_\mathsf{C}(\mathsf{PP}, B_0^{\mathsf{cust}}, B_0^{\mathsf{merch}}, pk_c, sk_c) \rightarrow (\mathsf{T}_c, csk_c)$. On input a keypair (pk_c, sk_c), perform the following:

   - Uniformly sample two distinct PRF seeds k_1, k_2 and random coins r for the commitment scheme.
   - Compute $\mathsf{wCom} = \mathsf{Commit}(sk_c, k_1, k_2, B_0^{\mathsf{cust}}; r)$
   - For $i = 1$ to B_0^{cust}, sample $ck_i \rightarrow \mathsf{SymKeyGen}(1^\lambda)$ to form the vector $\vec{ck}$.
   - Output $\mathsf{T}_c = (\mathsf{wCom}, pk_c)$ and $csk_c = (sk_c, k_1, k_2, r, B_0^{\mathsf{cust}}, \vec{ck})$.

4. $\mathsf{Init}_\mathsf{M}(\mathsf{PP}, B_0^{\mathsf{cust}}, B_0^{\mathsf{merch}}, pk_m, sk_m) \rightarrow \mathsf{T}_m, csk_m$. On input a keypair (pk_m, sk_m), perform the following:

   - Output $\mathsf{T}_m = pk_m$ and $csk_m = (sk_m, B_0^{\mathsf{merch}})$.

5. $\mathsf{Establish}(C\{\mathsf{PP}, \mathsf{T}_m, csk_c)\}, \{M(\mathsf{PP}, \mathsf{T}_c, csk_m)\})$. On input public parameters and each of the initial channel tokens, the $\mathsf{Establish}$ protocol activates a channel between customer and merchant who have previously escrowed funds. If the interaction succeeds, the merchant receives $\mathsf{established}$ message and the customer receives a wallet w. Either party may receive an error denoted by $\perp$.

   The customer executes the following algorithm:

- Parse csk_c as $(pk_c, sk_c, k_1, k_2, r, B_0^{\mathsf{cust}})$.
- Sample $sk_0 \in \{0,1\}^\ell$.
- Generate $\pi_1 = PK\{(sk_c, k_1, k_2, r) : \mathsf{wCom} = \mathsf{Commit}(sk_c, k_1, k_2; r) \wedge (pk_c, sk_c) \in \mathsf{KeyGen}(1^\lambda)\}$
  - NIZK statement: $\mathsf{wCom} = {g_1}^{x_c} \cdot {g_2}^{y_c} \cdot {g_3}^{k_1} \cdot {g_4}^{k_2} \cdot h^r \wedge X_c = {g_1}^{x_c} \wedge Y_c = {g_2}^{y_c}$ where $sk_c = (x_c, y_c)$.
- For $j = 1$ to B:
  (a) Compute $s_j \leftarrow F_{k_1}(j), u_j \leftarrow F_{k_2}(j)$.
  (b) $\pi_j^r = PK\{(sk_c, k_1, k_2, r) : s_j \leftarrow F_{k_1}(j) \wedge u_j \leftarrow F_{k_2}(j)$
  $\wedge\, \mathsf{wCom} = \mathsf{Commit}(sk_c, k_1, k_2; r)$
  $\wedge\, (pk_c, sk_c) \in \mathsf{KeyGen}(1^\lambda)\}$
  (c) Compute internal signature $\hat{\sigma_j} = \mathsf{Sign}(sk_c, \mathsf{spend}||j||s_j||u_j||\pi_j^r||ck_{j+1})$.
  (d) Compute $C_j = \mathsf{SymEnc}(ck_j, j||s_j||u_j||\pi_j^r||\hat{\sigma_j}||ck_{j+1})$
  (e) Compute external signature $\sigma_j = \mathsf{Sign}(sk_c, \mathsf{coin}||j||C_j)$.
- Customer sends $\mathsf{wCom}, \pi, (C_1, \sigma_1, \ldots, C_B, \sigma_B)$ to the merchant.

The merchant executes the following algorithm in response:

(a) Verify the signature on T_c

(b) Check that $B_0^{\mathsf{cust}} = B$

(c) Verify π_1

(d) For $j = 1$ to B, verify the signature σ_j on C_j

(e) If any of the above conditions (1-4) do not hold, abort and output $\perp$

(f) Return a blind signature σ_w on the contents of wCom

The merchant sets state to $\mathsf{established}$ and the customer obtains $w = (sk_0, sk_c, k_1, k_2, r, B, \sigma_w, 1)$.

6. $\mathsf{Pay}(C\{\mathsf{PP}, \epsilon, w_{\mathsf{old}})\}, \{M(\mathsf{PP}, \epsilon, \mathbf{S}_{\mathsf{old}})\})$. On input parameters, a payment amount ϵ, and a wallet w_{old} from a customer, and the merchant's current state $\mathbf{S}_{\mathsf{old}}$ (initially set to 0) from the merchant: the customer receives a payment success bit R_{C} and the new wallet w_{new} on success. The merchant receives a payment success bit R_{M} and an updated $\mathbf{S}_{\mathsf{new}}$ on success.

   The customer executes the following algorithm:

   - Parse w_{old} as $(sk_0, sk_c, k_1, k_2, r, B, \sigma_w, i)$. Return $\perp$ if $i \geq B$.
   - Compute $s \leftarrow F_{k_1}(i)$
   - Compute $t \rightarrow \mathsf{OTEnc}(F_{k_2}(i), pk_c)$
   - Compute $\pi = PK\{(pk_c, sk_c, k_1, k_2, r, i, \sigma_w) : s = F_{k_1}(i) \wedge 0 < i \leq B$
   $\wedge\, t = \mathsf{OTEnc}(F_{k_2}(i), pk_c)$
   $\wedge\, \mathsf{Verify}(pk_m, (k_1, k_2, sk_c), \sigma_w)$
   $\wedge\, (pk_c, sk_c) \in \mathsf{KeyGen}(1^\lambda)\}$
   - Return (s, t, π)

   The merchant executes the following:

- Verify π and $(s, \cdot, \cdot) \notin \mathbf{S}$.
- If this holds, then set $\mathbf{S} \leftarrow \mathbf{S} \cup (s, t, \pi)$ and $R_M \leftarrow 1$.
- Otherwise, set $R_M \leftarrow \perp$.

The customer obtains a new wallet $w_{new} := (sk_0, sk_c, k_1, k_2, r, B, \sigma_w, i + 1)$.

7. $\mathsf{Refund}(\mathsf{PP}, \mathsf{T}_M csk_c, \sigma)$. On input wallet w, output a customer channel closure message rc_c. This algorithm is executed by the customer.

   - Parse w to obtain $\vec{ck}$ and the current coin index i.
   - Compute $\sigma \leftarrow \mathsf{Sign}(sk_c, \mathsf{refund}||\mathsf{cID}||i||ck_i)$.
     NOTE: cID uniquely identifies the channel being closed
   - Output $\mathsf{rc}_C := (\mathsf{cID}, i, ck_i, \sigma)$.

8. $\mathsf{Refute}(\mathsf{PP}, \mathsf{T}_C, \mathbf{S}, \mathsf{rc}_C)$. On input the merchant's current state $\mathbf{S}_{\mathsf{old}}$ and a customer channel closure message, output a merchant channel closure message rc_M and an updated merchant state $\mathbf{S}_{new}$. This algorithm is executed by the merchant.

   (a) Parse the customer's channel closure message rc_C as $(\mathsf{cID}, i, ck_i, \sigma)$.
   (b) Verify cID and the signature σ.
   (c) If signature is valid, obtain the ciphertexts $C_i, \ldots, C_B$ (from the $\mathsf{Establish}$ protocol)
   (d) For $j = i$ to B, compute $(j||s_j||u_j||\pi^r_j||ck_j||\hat{\sigma_j}) \leftarrow \mathsf{SymDec}(ck_j, C_j)$ and verify the signature $\hat{\sigma_j}$ and proof π^r_j.
   (e) Failure conditions: (1) fail to verify $\hat{\sigma_j}$ and proof π^r_j, or (2) decryption of any ciphertext C_j results in $\perp$, or (3) decrypted values $(s_j, u_j) \in \mathbf{S}$ where $\mathsf{OTDec}(u_j, t_j) = pk_c$.
   (f) If failure, then record invalid result $\mathsf{rc}_M = (\mathsf{fail}, \mathsf{cID})$
   (g) If success, then record valid result $\mathsf{rc}_M = (\mathsf{accept})$
   (h) Sign result using sk_m (verified by payment network)
   (i) For each valid C_j, set $\mathbf{S} \leftarrow \mathbf{S} \cup (s_j, t_b, \pi)$
   (j) Output $\mathbf{S}$ as new merchant state

9. $\mathsf{Resolve}(\mathsf{PP}, \mathsf{T}_C, \mathsf{T}_M, \mathsf{rc}_C, \mathsf{rc}_M)$. On input the customer and merchant channel tokens T_C and T_M, along with closure messages $\mathsf{rc}_C, \mathsf{rc}_M$ (where either message may be null), this algorithm outputs the final channel balance $B^{\mathsf{merch}}_{\mathsf{final}}$, $B^{\mathsf{cust}}_{\mathsf{final}}$.

   (a) Parse the customer and merchant closure messages and verify all signatures
   (b) If there are any invalid signatures, grant the balance of the channel to the opposing party.
   (c) If $\mathsf{rc}_C = (N, sk_N, \sigma)$ and $\mathsf{rc}_M = \mathsf{accept}$, then set $B^{\mathsf{cust}}_{\mathsf{final}} = (B^{\mathsf{cust}}_0 - N) + 1$.
   (d) Evaluate the merchant closure message to determine whether the customer misbehaved.
   (e) If so, assign the merchant the full balance of the channel.

2.3.2 Bidirectional Scheme

The bidirectional payment construction enables compact closure and compact wallets in addition to a single run of the Pay protocol to transfer arbitrary values (constrained by a maximum payment amount). It consists of a tuple of possibly probabilistic algorithms (Setup, KeyGen, $\mathsf{Init_C}, \mathsf{Init_M}$, Refund, Refute, $\mathsf{Resolve}$) and two interactive protocols ($\mathsf{Establish}$, Pay).

1. $\mathsf{Setup}(1^\lambda) \rightarrow \mathsf{PP}$. On input λ, optionally generate CRS parameters for (1) a secure commitment scheme (see Section 2.2.1), (2) a non-interactive zero knowledge proof system (see Section 2.2.6). Output all of these as PP.

2. $\mathsf{KeyGen}(\mathsf{PP}) \rightarrow (pk, sk)$. Compute $(pk, sk) \leftarrow \prod_{\mathsf{sig}}.\mathsf{SigKeygen}(1^\lambda)$.

3. $\mathsf{Init_C}(\mathsf{PP}, \mathsf{cID}, B_0^{\mathsf{cust}}, B_0^{\mathsf{merch}}, pk_c, sk_c) \rightarrow (\mathsf{T_c}, csk_\mathsf{c})$. On input a keypair (pk_c, sk_c), perform the following:

   (a) Customer generates wallet commitment by sampling random coins r.

   (b) Compute ephemeral keypair $(wpk, wsk) \leftarrow \mathsf{KeyGen}(\mathsf{PP})$.

   (c) Compute $\mathsf{wCom} = \mathsf{Commit}(\mathsf{cID}, wpk, B_0^{\mathsf{cust}}; r)$.

   (d) Output $\mathsf{T_c} = (pk_c, \mathsf{wCom})$ and retains secret $csk_\mathsf{c} = (sk_c, \mathsf{cID}, wpk, wsk, r, B_0^{\mathsf{cust}})$.

4. $\mathsf{Init_M}(\mathsf{PP}, B_0^{\mathsf{cust}}, B_0^{\mathsf{merch}}, pk_m, sk_m) \rightarrow \mathsf{T}_m, csk_m$. On input a keypair (pk_m, sk_m), perform the following:

   - Output $\mathsf{T_m} = pk_m$ and $csk_\mathsf{m} = (sk_m, B_0^{\mathsf{merch}})$.

5. $\mathsf{Establish}(C\{\mathsf{PP}, \mathsf{T}_m, csk_c)\}, \{M(\mathsf{PP}, \mathsf{T}_c, csk_m)\})$. On input public parameters and each of the initial channel tokens, the $\mathsf{Establish}$ protocol activates a channel between the two parties who have previously escrowed funds. If the interaction succeeds, the merchant receives $\mathsf{established}$ message and the customer receives a wallet w. Either party may receive an error denoted by $\perp$.

   The customer does the following:

   (a) Parse csk_c to obtain $(\mathsf{cID}, \mathsf{wCom}, wpk, wsk, r, B_0^{\mathsf{cust}})$

   (b) Generate a proof π_1 of the following statement:
   $\pi_1 = PK\{(wpk, wsk, r) : \mathsf{wCom} = \mathsf{Commit}(\mathsf{cID}, wpk, B_0^{\mathsf{cust}}; r) \wedge (wpk, wsk) \in \mathsf{KeyGen}(1^\lambda)\}$

   (c) Send proof π_1 to the merchant.

   The merchant does the following:

   (a) Parse T_C to obtain $B_0^{\mathsf{cust}}, \mathsf{wCom}$.

   (b) Verify proof π_1 is valid. If not, output $\perp$

   (c) Execute interactive protocol to compute **a blind signature** σ_w under sk_m on contents of wCom.

   (d) Customer obtains σ_w.

The customer obtains a wallet $w := (B_0^{\mathsf{cust}}, wpk, wsk, r, \sigma_w)$ and the merchant sets its state to $\mathsf{established}$ for the channel.

6. $\mathsf{Pay}(C\{\mathsf{PP}, \epsilon, w_{\mathsf{old}})\}, \{M(\mathsf{PP}, \epsilon, \mathbf{S}_{\mathsf{old}})\})$. On input parameters, a payment amount ϵ, and a wallet w_{old} from a customer, and the merchant's current state $\mathbf{S}_{\mathsf{old}}$ (initially set to 0) from the merchant: the customer receives a payment success bit R_{C} and the new wallet w_{new} on success. The merchant receives a payment success bit R_{M} and an updated $\mathbf{S}_{\mathsf{new}}$ on success.

   In the first phase, the customer does the following:

   (a) Parse w_{old} as $(\mathsf{cID}, B, wpk, wsk, r, \sigma_w)$.

   (b) Sample $(wpk', wsk') \leftarrow \mathsf{KeyGen}(\text{PP})$.

   (c) Sample random coins r'.

   (d) Generate $\mathsf{wCom}' \leftarrow \mathsf{Commit}(\mathsf{cID}, wpk', B - \epsilon; r')$

   (e) Generate proof π_2 as follows:
   $\pi_2 = PK\{(wpk', B, r', \sigma_w) : \mathsf{wCom}' = \mathsf{Commit}(\mathsf{cID}, wpk', B - \epsilon; r')$
   $\wedge\, \mathsf{Verify}(pk_m, (wpk, B), \sigma_w) = 1$
   $\wedge\, 0 \le (B - \epsilon) \le \mathsf{val}_{\mathsf{max}}\}$

   - Compute $C_1 = g^B \cdot h^{r_1}$
   - Compute $C_2 = C_1 / g^{\epsilon}$
   - Compute $C_3 = g_1^{x_1} \cdot g_2^{x_2} \cdot h^{r_3}$
   - Compute $\mathsf{wCom}'' = C_2 \cdot C_3$. Keep wCom' private.

   (f) Send $(\epsilon, \mathsf{wCom}', wpk, \pi_2)$ to the merchant.

   In response, the merchant does the following for the first phase:

   (a) Verify π_2, ensure that $(wpk, \cdot) \notin \mathbf{S}$ and $\epsilon_{\mathsf{min}} \le \epsilon \le \epsilon_{\mathsf{max}}$

   (b) If these conditions do not hold, abort and output $\perp$

   (c) Set $\mathbf{S}_{\mathsf{new}} := \mathbf{S}_{\mathsf{old}} \cup \{(wpk, \perp)\}$.

   (d) If $\epsilon < 0$, then $R_M \leftarrow 1$ otherwise $R_M \leftarrow \perp$.

   (e) Execute interactive protocol to generate a **partially blind signature** $rt_{w'}$ under sk_m on the message $(\mathsf{refund}||wpk'||B - \epsilon)$.
   NOTE: wpk' and $B - \epsilon$ are the contents of wCom'.

   (f) The customer obtains $rt_{w'}$ at the end of this phase.

   In the second phase, the customer does the following:

   (a) Check that $\mathsf{Verify}(pk_m, (\mathsf{refund}||wpk'||B - \epsilon), rt_{w'}) = 1$

   (b) If verification failure or message does not arrive, abort and output $rt_{w'}$.

   (c) Otherwise, compute $\sigma_{rev} = \mathsf{Sign}(wsk, \mathsf{revoke}||wpk)$.

   (d) Send σ_{rev} to the merchant.

   In the second phase, the merchant does the following:

(a) Ensure $\mathsf{Verify}(wpk, (\mathsf{revoke}||wpk), \sigma_{rev}) = 1$.

(b) If so, set $\mathbf{S}_{\mathsf{new}} := \mathbf{S}_{\mathsf{old}} \cup \{(wpk, \sigma_{rev}\}$ and $R_M \leftarrow 1$.

(c) Execute interactive protocol to generate a **blind signature** $\sigma_{w'}$ on the contents of wCom' using sk_m.

(d) If this completes, set $R_M \leftarrow 2$.

(e) Send $\sigma_{w'}$ back to the customer.

The customer obtains a new wallet $w_{\mathsf{new}} := (B - \epsilon, wpk', wsk', r', \sigma_{w'})$ and the merchant keeps $\mathbf{S}_{\mathsf{new}}, R_M$.

7. $\mathsf{Refund}(\mathsf{PP}, \mathsf{T}_M, csk_C, w) \rightarrow (m, \sigma)$.

(a) If the customer has not invoked the Pay protocol, then $m := (\mathsf{refundUnsigned}, (wpk, B), r)$.

(b) Otherwise, set $m := (\mathsf{refundToken}, (wpk, B), rt_w)$.

(c) Compute $\sigma = \mathsf{Sign}(sk_c, m)$.

(d) Output $\mathsf{rc}_C = (m, \sigma)$.

8. $\mathsf{Refute}(\mathsf{PP}, \mathsf{T}_C, \mathbf{S}, \mathsf{rc}_C)$.

(a) If a merchant sees a channel closure message, it first parses T_C to obtain pk_c.

(b) Parse tuple $(m, \sigma) \leftarrow \mathsf{rc}_C$

(c) $\mathsf{Verify}(pk_c, m, \sigma) = 1$ or $\perp$

(d) If signature σ verifies, then parse tuple $(\mathsf{cID}, wpk, B) \leftarrow m$.

(e) Verify that cID matches the channel.

(f) If previous record of $(wpk, \sigma_{rev}) \in \mathbf{S}$, then output $\mathsf{rc}_M = ((\mathsf{revoked}, \sigma_{rev}), \sigma)$.
NOTE: σ is a valid signature over $(\mathsf{revoked}, \sigma_{rev})$

(g) Otherwise, add new key $\mathbf{S} = \mathbf{S} \cup wpk$.

9. $\mathsf{Resolve}(\mathsf{PP}, \mathsf{T}_C, \mathsf{T}_M, \mathsf{rc}_C, \mathsf{rc}_M)$.

(a) Verify that both tokens $\mathsf{rc}_C, \mathsf{rc}_M$ are signed by the customer and merchant keys pk_c and pk_m respectively.

(b) Verify that both tokens contain the same channel identifier cID and matches the one from T_C and T_M.

(c) If either of the tokens fail this test, then replace with $\perp$. Let $B_{\mathsf{total}} = B_0^{\mathsf{cust}} + B_0^{\mathsf{merch}}$.

(d) If rc_C is $\perp$, output all the funds to the merchant.

(e) Parse $(pk_c, \mathsf{wCom}) = \mathsf{T}_C$.

(f) m should have the following structure $(\mathsf{type}, (\mathsf{cID}, wpk, B), \mathsf{Token})$

(g) Parse $(\mathsf{revoked}, wpk, \sigma_{rev}) = m$

(h) Ensure that $\mathsf{Verify}(wpk, (\mathsf{revoked}||\mathsf{cID}||wpk), \sigma) = 1$.

(i) If verification check fails, then output $B_{\mathsf{final}}^{\mathsf{cust}} = B_{\mathsf{total}}$ and $B_{\mathsf{final}}^{\mathsf{merch}} = 0$.

(j) Check the refund validity:

a. If $\mathsf{type} = \mathsf{refundUnsigned}$, check $\mathsf{wCom} = \mathsf{Commit}(wpk, B; \mathsf{Token})$ and that merchant's token contains σ_{rev}.

b. If $\mathsf{type} = \mathsf{refundToken}$, check Token is a valid refund token on (wpk, B).

c. If either **(a)** or **(b)** fails, abort and output $B^{\mathsf{cust}}_{\mathsf{final}} = 0$ and $B^{\mathsf{merch}}_{\mathsf{final}} = B_{\mathsf{total}}$.

(k) Check the refutation's validity by checking that $\mathsf{Verify}(wpk, \mathsf{revoke}||wpk, \sigma_{rev}) = 1$.

a. If valid, abort and output $B^{\mathsf{cust}}_{\mathsf{final}} = 0$ and $B^{\mathsf{merch}}_{\mathsf{final}} = B_{\mathsf{total}}$.

b. If invalid, pay the claimed balance to the customer ($B^{\mathsf{cust}}_{\mathsf{final}} = B$) and the remainder to Merchant ($B^{\mathsf{merch}}_{\mathsf{final}} = B_{\mathsf{total}} - B$).

Chapter 3

BOLT Usage

3.1 Overview

BOLT is a payment channel protocol. In order to use BOLT, a Customer and Merchant must establish and fund a payment channel on the network of a compatible cryptocurrency. Customer and Merchant represent defined roles in the BOLT system.

Pairwise channels. A standard pairwise BOLT interaction consists of five phases. At a high level they are as follows:

1. **Channel negotiation.** To initiate an interaction, Customer and Merchant agree on the initial balances of the channel, which we denote by A (customer initial balance) and B (merchant initial balance) respectively. The Merchant provides a public key and signed channel opening transaction to the Customer.

2. **Channel funding.** Customer transmits the channel opening transaction to the currency network, which causes the Customer and Merchant to fund the channel with (**A**, **B**) units of currency respectively. This funding is conducted *on-chain*, and should be conducted using a privacy-preserving currency like Zcash so as to protect the customer's identity.

3. **Channel activation.** Once the channel has been funded and the transaction confirmed on the network, the parties now interact directly to activate the channel and prepare it for online payments.

4. **Payment.** This step may occur many times. To initiate a payment, the customer initiates an off-chain payment of **D** units of currency (of positive or negative value) to the merchant. This payment maintains the total channel balance, but updates the Customer and Merchant's ownership of the balances. The merchant does not learn which Customer or Channel was involved in the payment. This produces updated state at each party.

5. **Channel closure.** At the conclusion of a channel interaction, the customer or merchant may initiate the closure of the channel. If the parties dispute the balance of the channel, each party transmits a "closure token" to the currency network. The payment network must include logic to evaluate the tokens to determine the correct balances **(A, B)** to pay out to

the Customer and Merchant respectively. The parties may now withdraw their shares of the resulting channel.

A key design consideration of BOLT is that no party should ever be at risk of losing their funds because the other party has become unresponsive or has submitted invalid information. At each of the above steps, either party may abort and close the channel using the most recent balance information. The cryptocurrency network must enforce a *time-delay before releasing funds*, in order to ensure that both parties have the opportunity to dispute closure.

We include a discussion of the precise requirements for the cryptocurrency network further below.

Hub channels. BOLT also supports a “Hub” mode in which a Customer and Merchant interact via a payment hub. The use of a payment hub significantly increases the flexibility of BOLT, by enabling a hub and spoke model without the need for direct payment channel relationships between each individual Customer and Merchant. More significantly, the Hub does not learn the identity of the Customer or Merchant, nor does it see the payment amount.

The use of a Hub between a given Customer and Merchant requires the creation of two separate channels, one Customer $\longleftrightarrow$ Hub (“CH”) channel, and one Hub $\longleftrightarrow$ Merchant (“HM”) channel. The steps in opening and closing each channel are similar to steps (1), (2), (3) and (5) in the description above. However, the payment step (4) differs as follows:

- **(Hub-based) Payment.** The Customer initiates an off-chain payment of D units of currency (of positive or negative value) to the Merchant, via the Hub. This payment atomically updates the CH and HM channels such that the Hub’s balance on the CH channel is increased by D units, and the Merchant’s balance on the HM channel is increased by D units. If either channel update fails, the entire transaction fails and both channels fall back to the previous channel balances.

The Hub learns only that a transaction took place, but not the amount or the identities of the Customer or Merchant.

Bibliography

[1] 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.

[2] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *Public Key Cryptography - PKC 2005*, pages 416–431, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[3] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 473–489, New York, NY, USA, 2017. ACM.

[4] 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.

[5] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, May 2014.