Oblivious Accounting

Stavros Kaparelos
Supervisor: Dr. George Danezis
MSc. in Information Security
University College London
2016

This report is submitted as part requirement for the MSc. in Information Security at University College London. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

We design and implement a new scheme named *Oblivious Payment* that shares many characteristics with electronic cash, but it does not require a bank to hold and know each user's balance. Our construction is between a service, a user, and a merchant and allows the user to transact with the merchant anonymously and unlinkably. The user can transact with the merchant in an offline manner without the service learning the transacted amount, the user or the merchant identities, and how often and when a user or a merchant uses the system. Our construction is accountless, i.e. users do not create an account with the service, and each user keeps his/her balance locally, therefore, the service cannot know each user's balance. In addition, our scheme supports transferability of coins, online double spending detection, any denomination, and can be used to add anonymity and unlinkability in the transactions of any currency.

Contents

1 Introduction **10**

1.1 Structure 11

2 Background **12**

2.1 Related Work & Comparison 12

2.1.1 Electronic Cash 12

2.1.2 Anonymous Credentials 13

2.2 Preliminaries 14

2.2.1 Negligible Function 14

2.2.2 Pedersen Commitments 15

2.2.3 Zero Knowledge Proofs 15

2.2.4 Zero Knowledge Proofs of Range 16

2.2.5 Anonymous Credentials Light 18

3 Design **22**

3.1 Overview 22

3.2 Syntax 23

3.3 Security Policy & Threat Model 26

3.4 Construction 26

3.5 Ensuring that an ACL issuer is the verifier 34

3.6 Integration with existing currencies 35

3.7 Design Choices 35

4 Implementation **39**

4.1 Implemented Protocols 39

4.2 Cryptographic Framework 39

4.3 Technology Stack 40

4.4 Storage 41

4.5 Liveness 42

4.6 Algorithms 42

4.6.1 Proof that a committed number is a bit 43

4.6.2 Proof that a number lies in a interval 44

4.6.3 ACL Verification 46

4.6.4 Double Spending Detection 47

5 Evaluation **48**

5.1 Security Evaluation 48

5.1.1 Service Owned by the Adversary 48

5.1.2 Maintaining Bitcoin Privacy 49

5.1.3 Connection Errors & System State 49

5.2 Theoretical Analysis 50

5.2.1 Coin & Database Sizes 50

5.2.2 Erroneous double spending 50

5.3 Experimental Analysis 51

5.3.1 Size of Transmitted Data 51

5.3.2 Time results 52

5.3.3 Comparison With Naive Spending Schemes 53

5.3.4 Concurrency 54

5.4 Comparison with Hinterwälder et al. [33] 55

5.5 Unit tests 55

6 Conclusion & Future Work **57**

6.1 Future work 57

List of Figures

1 Protocol execution time using different curves. 53

2 Execution time within each protocol for curve *secp224r1* 54

List of Tables

1 NIZK π for commitment to $m \in \{0,1\}$ 17

2 The ACL construction 19

3 Our modified ACL construction 21

4 The `Deposit` Protocol 27

5 The `Withdraw` Protocol 28

6 The `Split` Protocol. 29

7 Maximum Value for $v_1 + v_2$ 31

8 The `Combine` Protocol 32

9 The `Spend` Protocol 33

10 The `DoubleSpent` Protocol 34

11 Comparison of Constructions 38

12 Bytes transmitted for each protocol (except `Spend`) for curves *secp224r1*, *secp256r1* 51

13 Bytes transmitted for each protocol (except `Spend`) for curves *secp384r1*, *secp521r1* 52

14 Bytes transmitted for the `Spend` protocol 52

15 Protocol execution time in seconds over different curve choices 53

16 Concurrent times per protocol for curve *secp224r1* 55

17 Comparison of our scheme and the one by Hinterwälder et al. in [33] in terms of numbers of exponentiations per protocol. (E = number of exponentiations) 55

List of Algorithms

1 ProveBit Proof that a committed number is a bit as presented by Groth and Danezis in [30] and [23] respectively. Run by the user. 43

2 VerifyBit Verification that a committed number is a bit as presented by Groth and Danezis in [30] and [23] respectively. Run by the verifier. 44

45

46

5 ACLVerify ACL Verification run by the verifier 47

6 checkDoubleSpend Double Spending Detection, run by the verifier . 47

Acknowledgements

I would like to thank my supervisor, Dr. George Danezis, for the support and guidance he offered throughout this project. The project would not have been possible without his help. It has been a pleasure.

1 Introduction

Electronic cash (e-cash) schemes, the digital equivalent of cash, allow users to transact with a merchant in an anonymous and unlinkable manner, however, they do not hide from the bank each user's total balance, nor the account activity. In a more privacy preserving scenario, the bank would not know anything about the balance of each user and his[1] account activity. Nevertheless, it could be the case that one does not trust a bank to store his balance and wants to store everything in his own wallet, locally[2]. This is our construction's main contribution. Namely, to the extent we are aware of, we create the first scheme that has properties of an e-cash scheme, but allows users to store their balances locally and does not require users to have accounts in the service.

In our construction, the user loads an amount of money to the service and in exchange gets a coin that corresponds to that amount which can be stored locally. Our construction is based on anonymous credentials, and therefore, the service does not know the coin's denomination, nor can link the coin with its issuance. Since a coin can be of any denomination and the user might engage in a transaction that requires a different denomination, he must be able to split and combine coins in order to create a coin of the right denomination. The split and combine functions require the involvement of the service, while our spend protocol does not. In addition, the scheme supports online double spending detection, where the service checks whether a coin has been spent against a database of coins.

We base our work on "Anonymous Credentials Light" (ACL) [5], which are single show anonymous credentials, and work in the elliptic curve setting. Elliptic curve cryptography, due to the small number of bits required, makes our construction very efficient and fast for personal computers, and, also, makes it a good candidate for less powerful devices, such as smartcards, RFID cards, and mobile devices.

Our prototype is a desktop application consisting of a client and a server. The client application is the user's wallet, where he can store his coins, while the server application is the service that is contacted to deposit, withdraw, split, and combine coins. Our spend protocol requires 70ms to complete, while the size of the client's wallet is $1024n$ bytes and the size of the service's database is $68n$ bytes, where n represents the number of coins.

[1] The choice to use a masculine gender was made on the basis of a coin flip.

[2] Not using a bank is not an absurd assumption, in fact it is a widely used one: Bitcoin private keys are stored locally.

1.1 Structure

In section 2, we discuss related work, we compare our work to it and explain the cryptographic primitives required to understand the rest of the sections. In section 3, we present the design of our construction in detail, explain the security behind our choices, define our security policy and threat model, explain how can our construction integrate with both fiat and crypto currencies, and, finally, we compare our construction with alternative design choices. In section 4, we explain the implementation details, such as the cryptographic framework, the technology stack, liveness issues, and present important algorithms in our construction. In section 5, we evaluate our prototype implementation and discuss about our findings. Finally, in section 6 we conclude and identify future work.

2 Background

In this section we, first, describe and compare our work to related work and, then, explain the cryptographic building blocks necessary to understand our construction.

2.1 Related Work & Comparison

2.1.1 Electronic Cash

Electronic cash, introduced by Chaum [17], [19] in 1982, is the digital equivalent of physical cash. An e-cash scheme involves three entities: a bank $\mathcal{B}$, a user $\mathcal{U}$, and a merchant $\mathcal{M}$. Once the user has an account at the bank, he can withdraw an amount of coins that he can spend with the merchant, who, at a later time, can deposit the received coins in his bank account. The withdrawal and deposit phases, ideally, should be constructed in such a way that are unlinkable, i.e. the bank cannot link a withdrawal phase with the corresponding deposit phase. *Unlinkability* is one of the desirable properties of e-cash schemes, along with *unforgeability*, the property which guarantees that an adversary cannot forge coins.

E-cash schemes come at a variety. A *transferable* e-cash scheme [6] is one that allows a user to transfer an amount of coins to a merchant who can later transfer them to another merchant (or user), and so on until a merchant deposits the coins in the bank. In non-transferable e-cash schemes, the merchant upon receiving an amount of coins from the user has to deposit them in the bank and not spend them any further to other merchants or users. A *compact* e-cash scheme, initially proposed by Camenisch et al. [12], is one where a user can withdraw an amount of coins (often considered to be a power of 2) 2^l and can spend these coins one by one, making thus compact e-cash schemes not so favourable in several practical scenarios. A *divisible* e-cash scheme ([41], [40], [15], [34]) solves this problem, as it allows a user to withdraw an amount of 2^l coins and then spend an amount 2^k, where $k \leq l$.

A major problem in e-cash schemes is the problem of double spending detection. Since coins are represented by bits, an adversary could copy and spend a coin an arbitrary amount of times. Double spending detection can be done either *offline* or *online*. In the offline case, withdrawn coins are transferred from a user to a merchant and, once the merchant deposits them back in the bank, the bank can detect double spending and

can reveal the identity of the fraudulent user. In the online case, the bank verifies all transactions at the time they take place and, thus, it detects double spending exactly when it happens.

Comparison with e-cash schemes. E-cash schemes are close to our scheme, however, there are two profound differences. First, in e-cash schemes the bank knows the total balance of each user at each point in time, while in our scheme the service is not aware of each user's balance. Second, e-cash schemes involve the concept of an account, while our scheme is accountless, i.e. the user does not maintain an account in the service and his wallet is kept by him locally. Our construction shares the entities involved in an e-cash scheme, the only difference being that we have an entity called the service, instead of a bank. In addition, similar to some e-cash schemes, our scheme offers online double spending detection, allows the transferability of coins, and, finally, allows users to use denominations of their preference instead of being bounded to a fixed set.

2.1.2 Anonymous Credentials

Credential systems allow a user to obtain credentials from a party and prove possession of these credentials whenever needed. Credential systems involve three entities: the *issuer* who issues the credentials, the *prover* who gets issued a credential and proves its possession to the *verifier*, who verifies the validity of the credential. Notice that the issuer and the verifier might be the same entity or two distinct entities. Anonymous credentials [18], [13] allow a user to prove to a verifier possession of certain attributes (credentials) without exposing the prover's identity and by providing unlinkability among shows of the same credential.

In anonymous credentials, during the phase of *issuance*, the issuer issues a credential to a number of attributes the prover possesses. The prover gives the attributes to the issuer in a privacy-preserving manner, so that the issuer does not know the credentials he certified, who outputs a certified credential which gives to the prover. At a later time, in the phase of *showing*, the prover proves nothing else but the ownership of a credential to the verifier.

Credential systems can be either *single-show* or *multi-show*. In the former, every time the prover wants to prove ownership of a credential must have it reissued by the issuer, while in the latter, the prover gets a credential issued only once and is able to show it multiple times while maintaining unlinkability between each show. Depending on the

context of the application one might prefer one property over the other. In our context, the single-show property is a feature, as it inherently handles the double spending problem.

Baldimitsi and Lysyanskaya in "Anonymous Credentials Light" [5] proposed a single-show credential construction with a proof of security in the Random Oracle model. The ACL construction is based on Elliptic Curve Cryptography (ECC) making it very efficient for less powerful devices, such as smartcards, RFID cards, and mobile devices. The ACL construction will be discussed in greater depth, once we define the cryptographic primitives that it is based on, later in this section.

Hinterwälder et al. in [33] realised the first e-cash scheme based on ACL (which they also used in [32]), with offline double spending detection, focusing on NFC payments in the transportation setting using mobile devices. Their system consists of three entities: the transportation authority (TA) (vending machines), the users (U), and the payment machines (M). Users, after registering an account with the transportation authority, can use the vending machines to load their balance and get in exchange an ACL with the loaded amount as an attribute. To use the transportation system, users have to use the payment machines where they prove ownership of their credentials and are allowed or denied access to the transportation system. Multiple denominations are supported in the naive way, namely if a user wants to withdraw k coins, he must engage in a withdrawal protocol k times, once for each coin.

Our construction is based on ACL, however, it is different than the one presented by Hinterwälder et al. We utilise ACLs in a different way which we explain, once we explain the ACL paper in greater depth later in this section.

2.2 Preliminaries

2.2.1 Negligible Function

We use the following definition of a *negligible function* from [35] and provide it here as is: "*A function f from the natural numbers to the non-negative real numbers is negligible if for every positive polynomial p there is an N such that for all integers $n > N$ it holds that $f(n) < \frac{1}{p(n)}$*".

2.2.2 Pedersen Commitments

To understand Pedersen commitments the reader has to be familiar with the notion of commitment schemes. A commitment scheme allows a user to commit to a value or to a set of values with the potential of revealing the value later. Commitment schemes have two properties: *binding* and *hiding*. Binding means that it is hard for a user to find another commitment to a different value that is equal to the first commitment, therefore not allowing the user to change the committed value. Hiding means that when another user is given a commitment, he cannot know what the committed value is. Pedersen commitments are both computationally and information theoretically hiding.

The Pedersen commitment scheme is such a scheme whose security is based on the hardness of the Discrete Logarithm. A commitment C, based on the Pedersen commitment scheme over a prime order group G with order q, where g, h are generators of G, is defined as $C = g^m h^r$ or $Com(m, r) = g^m h^r$, where $m \in \mathbb{Z}_q$ is the value one wants to hide, and $r \in_R \mathbb{Z}_q$ is a blinding factor.

A generalised version of the Pedersen commitment scheme, as explained in the ACL paper, that allows a commitment to set of messages $(L_1, ..., L_n)$ is defined as follows: $Com(L_1, ..., L_n) = h^R \prod_{i=1}^{n} {h_i}^{L_i}$, where $R \in_R \mathbb{Z}_q$, $L_i \in \mathbb{Z}_q$ and $h, h_1, ..., h_n$ are generators of G.

In the ACL paper, they noticed that given a Pedersen commitment $C = {h_0}^R {h_1}^{L_1} ... {h_n}^{L_n}$, the values (z^γ, C^γ) are also a commitment to the same values with randomness (R, γ). Based on this observation they defined the *blinded* Pedersen commitment scheme, which is a new commitment scheme that is both unconditionally hiding and binding and is defined as $Com^B(L_1, ..., L_n; R, \gamma) = (z^\gamma, Com(L_1, ..., L_n); R)^\gamma)$.

2.2.3 Zero Knowledge Proofs

We assume the reader to be familiar with Zero Knowledge proofs. For a short description we refer the reader to [5], whereas for an extended formal treatment to [7]. When referring to Zero Knowledge proofs we will use the notation introduced in [14] by Camenisch and Stadler. For example, the following proof represents a non-interactive zero knowledge proof (NIZK) of knowledge of secrets m, r within a commitment C.

$$NIZK\{(m, r) : C = g^m h^r\}$$

In this proof the items in the parenthesis are secret, while everything else, namely C, g, h, is supposed to be public. In [14] they use Greek letters to denote the secret elements in the parentheses, however, we do not follow this convention.

2.2.4 Zero Knowledge Proofs of Range

Zero knowledge proofs of range or range proofs for short, allow a prover to convince a verifier that a committed value lies in a range, e.g. $x \in [0, 100]$. There are many proofs in the literature that serve different purposes with different efficiency considerations. Famous works in the area involve the *classic proof* by bit decomposition initially presented in [37], Boudot's [10], Lipmaa's [36], and Groth's [29] work on square decomposition, Camenisch et al. [11] work on signature based proofs, as well as proofs which prove that a committed value is less than another committed value [16].

The proof we present here and use in the rest of this document is the classic proof by bit decomposition. According to Boudot [10], this proof is an expensive one, however, among those we considered (those presented in [10]) it was the only one that proved to a verifier that a number lies in the exact range of our preference without increasing the range boundaries.

Proof that a committed value is 0 or 1
A well known example of an interactive zero knowledge proof between a prove $\mathcal{P}$ and a verifier $\mathcal{V}$ is given in [30] by Groth and can be seen at Table 1: $NIZK\{(m, r) : C = Com(m, r) \wedge m \in \{0, 1\}\}$. We use the non-interactive version of this proof by applying to it the Fiat-Shamit heuristic [28].

Proof that a committed number lies in $[0, 2^l - 1]$
It is often the case that we will need to prove that a committed number is positive instead of just proving a bit. Boudot in [10] presents methods to prove that a committed value lies in an interval and gives a detailed description on their size and efficiency. The one we present here is the folklore method between a prover (Bob) and a verifier (Alice) based on [10] and [37]. Given an l-bit number x, Bob treats x as $x = x_0 2^0 + x_1 2^1 + ... + x_{l-1} 2^{l-1} = \sum_{i=0}^{l-1} x_i 2^i$, where $x_i \in \{0, 1\}$. Bob chooses a random r_i for each $i \in \{0, ..., l-1\}$ then calculates r, such that $r = \sum_{i=0}^{l-1} r_i 2^i$. Bob commits to $Com(x, r)$ and sends the commitment to Alice. In addition, Bob forms the commitments $Com(x_i, r_i)$ for each i and proves by forming a zero-knowledge proof π_i for each of them that they open to a value in the set $\{0, 1\}$ as described before. Subsequently, Bob sends

Table 1: NIZK π for commitment to $m \in \{0,1\}$

$\mathcal{P}(pparams, C, m, r)$	$\mathcal{V}(pparams, C)$
$a, s, t \xleftarrow{\$} \mathbb{Z}_q$	
$C_a = Com(a, s)$	
$C_b \leftarrow Com(am, t)$	
$\xrightarrow{c_a, c_b}$	
$\xleftarrow{x \leftarrow \{0,1\}^\lambda}$	
$f \leftarrow mx + a$	
$z_a \leftarrow rx + s$	
$z_b \leftarrow r(x - f) + t$	
$\xrightarrow{f, z_a, z_b}$	
	Accept if:
	$c_a, c_b \in \mathcal{C}_{pparams}, f, z_a, z_b \in \mathbb{Z}_q$
	$C^x C_a = Com(f, z_a)$
	$C^{x-f} C_b = Com(0, z_b)$

all $Com(x_i, r_i), \pi_i$ to Alice. Alice verifies that indeed all are commitments to 0 or 1, and checks whether $Com(x, r) \stackrel{?}{=} \prod_{i=0}^{l-1} Com(x_i, r_i)^{2^i}$.

If Bob has not cheated, then $Com(x, r) \stackrel{?}{=} \prod_{i=0}^{l-1} Com(x_i, r_i)^{2^i}$ is true, since:

$$\begin{aligned} Com(x, r) &= g^x h^r \\ &= g^{x_0 2^0 + x_1 2^1 + \ldots + x_{l-1} 2^{l-1}} h^{r_0 2^0 + r_1 2^1 + \ldots + r_{l-1} 2^{l-1}} \\ &= g^{x_0 2^0} g^{x_1 2^1} \ldots\ g^{x_{l-1} 2^{l-1}} h^{r_0 2^0} h^{r_1 2^1} \ldots\ h^{r_{l-1} 2^{l-1}} \\ &= g^{x_0 2^0} h^{r_0 2^0} g^{x_1 2^1} h^{r_1 2^1} \ldots\ g^{x_{l-1} 2^{l-1}} h^{r_{l-1} 2^{l-1}} \\ &= (g^{x_0} h^{r_0})^{2^0} (g^{x_1} h^{r_1})^{2^1} \ldots\ (g^{x_{l-1}} * h^{r_{l-1}})^{2^{l-1}} \\ &= Com(x_0, r_0)^{2^0} Com(x_1, r_1)^{2^1} \ldots\ Com(x_{l-1}, r_{l-1})^{2^{l-1}} \\ &= \prod_{i=0}^{l-1} Com(x_i, r_i)^{2^i} \end{aligned} \tag{1}$$

After these checks, Alice knows that the number Bob committed to lies in $[0, 2^l - 1]$.

The above construction can be turned into the form of a NIZK proof as follows:

$$\begin{aligned} NIZK\{(x, r, x_0, x_1, ..., x_{l-1}, r_0, r_1, ..., r_{l-1}) : & \ Com(x, r) \ \wedge \\ & \ Com(x_0, r_0) \ \wedge x_0 \in \{0,1\} \ \wedge \\ & \ Com(x_1, r_1) \ \wedge x_1 \in \{0,1\} \ \wedge \\ & \ \ldots \ \wedge \\ & \ Com(x_{l-1}, r_{l-1}) \ \wedge x_{l-1} \in \{0,1\}\} \end{aligned} \tag{2}$$

where each $Com(x_i, r_i)\ \wedge x_0 \in \{0, 1\}$ refer to the proof explained in table 1.

Due the space this proof occupies, we will refer to it by : $NIZK\{(x, r) : C = g^x h^r \wedge x \in [0, 2^l - 1]\}$

2.2.5 Anonymous Credentials Light

In "Anonymous Credentials Light" , the authors, Baldimitsi and Lysyanskaya, define a blind signature scheme named *blind signatures with attributes* and realise a single-show anonymous credential system based on that scheme. Their scheme works in the elliptic group setting and it is based on the decisional Diffie-Hellman assumption. Working on the elliptic curve setting makes the scheme mobile and smart-card friendly, as it allows for faster calculations due to the smaller number of bits needed compared to other anonymous credentials that use the RSA group or groups with pairings.

In order for the user to obtain a credential, he must commit to the set of attributes he possesses, thus generating a commitment C that he gives to the issuer along with a zero-knowledge proof, which proves that C opens to the correct attributes and convinces the verifier to issue a blinded, unlinkable credential $\tilde{C}$ to the same attributes, as well as a signature to $\tilde{C}$.

ACL comes with two security properties in the Random Oracle model: *blindness* and *unforgeability*. Blindness guarantees the unlinkability of a signature to its issuing, while unforgeability guarantees that a user cannot forge a signature.

We will now present the ACL construction as presented in the ACL paper and focus on explaining some key parts that will be used later. For a complete treatment of the ACL construction the reader is referred to the original ACL paper. Table 2 depicts the ACL construction. We have tried to save some space keeping the table's total length less than a page by writing more than one steps in the same line, however, the construction remains the same.

The ACL construction is between a User and a Signer (issuer), whose private key is x and public key is $y = g^x$, respectively and consists of three phases: registration, preparation, and validation. At the end of the protocol, the user will have obtained a signature σ by the issuer on $(m, \zeta, \zeta_1, \zeta_2, \rho, \omega, {\rho_1}', {\rho_2}', \omega', \mu)$.

Let us provide an example to ease explanation of how the ACL construction works:

assume a user with one attribute L_1 who wants to obtain a credential for that attribute. The user chooses a secret random number $R \in_R \mathbb{Z}_q$, forms the commitment $C = {h_0}^R {h_1}^{L_1}$, thus hiding L_1 from the issuer, and also proves in zero-knowledge knowledge of secrets in the commitment. The issuer chooses an rnd to make z_1 and z_2 "one-time" tag keys. rnd is sent to the user in order to be convinced that the issuer does not know $\log_g z_1$. In his turn, the user chooses the blinding factor $\gamma \in_R \mathbb{Z}_q$ which he uses to blind z into $\zeta = z^\gamma$, z_1 into $\zeta_1 = {z_1}^\gamma$, and z_2 into $\zeta_2 = \zeta/\zeta_1$. (ζ, ζ_1) is a blinded Pedersen commitment which corresponds to the unlinkable commitment $\tilde{C}$ mentioned above.

Table 2: The ACL construction

Signer$(params, x, C)$		User$(params, y, m, (L_1, ...L_n; R))$
Registration		
		$C = {h_0}^R {h_1}^{L_1} {h_2}^{L_2} ... \ {h_n}^{L_n}$
	$\overset{\pi_1}{\longleftrightarrow}$	
Preparation		
$rnd \in_R \mathbb{Z}_q$		
$z_1 = Cg^{rnd}, z_2 = z/z_1$		
	$\overset{rnd}{\longrightarrow}$	
		check if $rnd \neq 0$
		$z_1 = Cg^{rnd}, \gamma \in_R {\mathbb{Z}_q}^*$
		$\zeta = z^\gamma, \zeta_1 = {z_1}^\gamma, \zeta_2 = \zeta/\zeta_1$
		$\tau \in_R \mathbb{Z}_q, \eta = z^\tau$
Validation		
$u, {\rho_1}', {\rho_2}', c' \in_R \mathbb{Z}_q$		
$a = g^u, {a_1}' = g^{{\rho_1}'} {z_1}^{c'}, {a_2}' = h^{{\rho_2}'} {z_2}^{c'}$		
	$\overset{a, {a_1}', {a_2}'}{\longrightarrow}$	
		check if $a, {a_1}', {a_2}' \in G$
		$t_1, t_2, t_3, t_4, t_5 \in_R \mathbb{Z}_q$
		$\alpha = a g^{t_1} y^{t_2}, {\alpha_1}' = {{a_1}'}^\gamma g^{t_3} {\zeta_1}^{t_4}$
		${\alpha_2}' = {{a_2}'}^\gamma h^{t_5} {\zeta_2}^{t_4}$
		$\varepsilon = \mathcal{H}_2(\zeta, \zeta_1, \alpha, {\alpha_1}', {\alpha_2}')$
		$e = (\varepsilon - t_2 - t_4) \mod q$
	$\overset{e}{\longleftarrow}$	
$c = e - c' \mod q, r = u - cx \mod q$		
	$\overset{c, r, c', {r_1}', {r_2}'}{\longrightarrow}$	
		$\rho = r + t_1 \mod q, \omega = c + t_2 \mod q$
		${\rho_1}' = \gamma {r_1}' + t_3 \mod q, {\rho_2}' = \gamma {r_2}' + t_5 \mod q$
		$\omega' = c' + t_4 \mod q, \mu = \tau - \omega' \gamma \mod q$
End of the ACL construction		

ACL Verification. A signature verifies if $\zeta \neq 1$ and

$$\omega + \omega' = \mathcal{H}(\zeta, \zeta_1, g^{\rho} y^{\omega}, g^{\rho_1{}'} \zeta_1{}^{\omega'}, h^{\rho_2{}'} \zeta_2{}^{\omega'}, z^{\mu} \zeta^{\omega'}, m) \mod q$$

The ACL construction presented thus far has not been optimised. Specifically, the preparation and a part of the validation phases can be executed together reducing the rounds of communication by one round. We present our modified version in table 3. This optimisation is also mentioned presented in the ACL paper. We will distinguish between our modified ACL version and the one presented in the ACL paper by referring to the latter as the *original* ACL construction.

Comparison with Hinterwälder et al. [33]. The construction in [33] utilises ACLs in a different way than we do. Particularly, the registration phase is executed only once for each user. This approach, also described in the original ACL paper, is beneficial in some scenarios, however we deviate from it as it poses a major restriction: the value of each coin cannot be encoded as an attribute at the registration phase, since it only happens once and users might want to buy coins of any denomination. To solve this problem in [33], they made the assumption that an ACL corresponds to a minimum coin value and if a user wants k coins, he must execute the protocol k times. We solve this problem by treating the registration phase differently than what was envisioned in the ACL paper and to what the authors did in [33]. That is we do not register users and execute the registration phase every time. This allows us to encode as an attribute the value of each coin, and therefore requiring only one execution to obtain a coin of value k, rather than k executions. Notice that in the average case[3] our approach is far less expensive than the one in [33]. On the other hand, the drawback of this approach is that we cannot have offline double spending detection, as we do not register our users. Finally, we identify as future work to see if the *rnd* value of the ACL protocol has a valuable role in the way we treat the scheme, since it is basically used to make a registered commitment C unlinkable from show to show.

[3]That is the case where k is not a number very close to 1.

Table 3: Our modified ACL construction

Signer$(params, x, C)$	User$(params, y, m, (L_1, ...L_n; R))$
Registration	
	$C = {h_0}^R {h_1}^{L_1} {h_2}^{L_2} ... \ {h_n}^{L_n}$
$\xleftrightarrow{\pi_1}$	
Preparation - Validation 1	
$rnd \in_R \mathbb{Z}_q$	
$z_1 = Cg^{rnd}, z_2 = z/z_1$	
$u, {\rho_1}', {\rho_2}', c' \in_R \mathbb{Z}_q$	
$a = g^u, {a_1}' = g^{{\rho_1}'} {z_1}^{c'}, {a_2}' = h^{{\rho_2}'} {z_2}^{c'}$	
$\xrightarrow{rnd, a, {a_1}', {a_2}'}$	
	check if $rnd \neq 0$
	$z_1 = Cg^{rnd}, \gamma \in_R {\mathbb{Z}_q}^*$
	$\zeta = z^\gamma, \zeta_1 = {z_1}^\gamma, \zeta_2 = \zeta/\zeta_1$
	$\tau \in_R \mathbb{Z}_q, \eta = z^\tau$
	check if $a, {a_1}', {a_2}' \in G$
	$t_1, t_2, t_3, t_4, t_5 \in_R \mathbb{Z}_q$
	$\alpha = a g^{t_1} y^{t_2}, {\alpha_1}' = {{a_1}'}^\gamma g^{t_3} {\zeta_1}^{t_4}$
	${\alpha_2}' = {{a_2}'}^\gamma h^{t_5} {\zeta_2}^{t_4}$
	$\varepsilon = \mathcal{H}_2(\zeta, \zeta_1, \alpha, {\alpha_1}', {\alpha_2}')$
	$e = (\varepsilon - t_2 - t_4) \mod q$
$\xleftarrow{e}$	
Validation 2	
$c = e - c' \mod q, r = u - cx \mod q$	
$\xrightarrow{c, r, c', {r_1}', {r_2}'}$	
	$\rho = r + t_1 \mod q, \omega = c + t_2 \mod q$
	${\rho_1}' = \gamma {r_1}' + t_3 \mod q, {\rho_2}' = \gamma {r_2}' + t_5 \mod q$
	$\omega' = c' + t_4 \mod q, \mu = \tau - \omega' \gamma \mod q$
End of the modified ACL construction	

3 Design

In this section we will describe the design of our scheme. We will explain its syntax, the protocols and algorithms that it consists of, the security properties that must hold, integration with existing currencies, as well as the design choices we considered and the reason we concluded to the current one.

3.1 Overview

We construct a new scheme named *Oblivious Payment* ($\mathcal{OP}$) that is a tuple of protocols and algorithms consisting of at least three entities: a service $\mathcal{S}$, a user $\mathcal{U}$, and a merchant $\mathcal{M}$. Our scheme is based on the ACL construction, specifically ACL credentials represent electronic coins with the denomination encoded as an attribute. We call the electronic coins in our system ACLcoins and denote them by $\mathcal{AC}$. An $\mathcal{AC}$ corresponds to a signature in the original ACL paper, where ζ_1 is a blinded commitment signed by the issuer where the L_1 attribute is equal to the amount of money this coin represents. Each user maintains a wallet denoted by $\mathcal{W}$ that contains all ACLcoins he possesses. If a user deposits 10$ in the service, then he will obtain a credential containing $\zeta_1 = ({h_0}^R {h_1}^{10} g^{rnd})^\gamma$. Notice that due to the blindness property of the ACL credentials[4], the service cannot link a coin with the user that issued it. In addition, due to the unforgeability property, a malicious user cannot create fake ACLcoins.

The system allows users to anonymously and unlinkably transact using any currency. Transactions occur in an offline manner and, thus, they do not involve the service. The service is only involved with coin operations (i.e. coin mintage, coin division, coin combination, and coin withdrawal) by verifying their validity and preventing double spending.

Only two entities can interact per protocol, either a user with the service, or the user with the merchant. The service can be used for the following four operations: a) by a user to exchange currency to an $\mathcal{AC}$ via a `Deposit` protocol, b) by a user to exchange $\mathcal{AC}$s to currency via a `Withdraw` protocol, c) by a user to split an $\mathcal{AC}$ into two $\mathcal{AC}$s whose denominations will be equal to the initial $\mathcal{AC}$ via a `Split` protocol, and d) by a user to combine two $\mathcal{AC}$s into one, whose value will be the sum of the two via a `Combine` protocol. The service maintains no state apart from a list of the coins that have been

[4]For more information the reader is advised to the background section of this document.

spent, which is used to check for double spending[5]. Specifically, $\mathcal{AC}$s and wallets are stored by each user locally. In addition, there is no concept of accounts in the scheme, i.e., a user does not have to create an account in the service to be able to use it.

A user can obtain an $\mathcal{AC}$ in two ways: a) either by exchanging currency to the equivalent ACLcoin, or b) by receiving an ACLcoin from another user. If a user has an $\mathcal{AC}$ worth of 10 and wants to send 6 to his friend, then he can split the $\mathcal{AC}$ into two $\mathcal{AC}$s worth 4 and 6 respectively and send the one worth 6.

Double spending detection happens from the service by checking whether the given $\mathcal{AC}$ exists it the list of $\mathcal{AC}$s it maintains. As stated in [33], we cannot expect the database to store coins forever. The solution to this problem is to either have a coin expiration date encoded as an attribute or change the service's public key at specific time intervals, thus, invalidating all coins. This is a matter of implementation and will be discussed in the implementation section.

Finally, we should note that the size of our wallet is $\mathrm{O}(s \cdot N)$ where s represents the number of bits that represent the size of the coin and N represents the total number of coins a user possesses.

3.2 Syntax

We write $y = g^x$ to denote assignment, $x \leftarrow \mathbb{Z}_q$ to denote that x was chosen from a set whose elements are integers and has order q. We write $x \xleftarrow{\$} \mathbb{Z}_q$, or $x \in_R \mathbb{Z}_q$ to denote that x is randomly chosen from the set $\mathbb{Z}_q$. Finally, we write `Protocol` to denote a protocol.

An $\mathcal{OP}$ scheme is a tuple of protocols and algorithms $\mathcal{OP}$ = (`ParamGen`, `SKeyGen`, `Deposit`, `Withdraw`, `Split`, `Combine`, `Spend`). Unless explicitly defined all algorithms are probabilistic polynomial time.

- `ParamGen` (1^λ): The parameter generation algorithm. We assume that this algorithm is executed once by the service $\mathcal{S}$, which we assume to be a trusted party, and its output, *params* is public and is considered default input in all of our algorithms.

[5]In contrast to other schemes, our scheme does not reveal the identity of the user that tries to double spend, it merely does not let the transaction proceed.

– `SKeyGen` (1^λ): The service's key generation algorithm. It outputs the service's secret $SK_{\mathcal{S}}$ which must remain only known to the service, as well as the service's public key $PK_{\mathcal{S}}$.

– `Deposit`: $\mathcal{U}(PK_{\mathcal{S}}, amount, account_{\mathcal{U}}, account_{\mathcal{S}})$; $\mathcal{S}()$
If a user wants deposit money to use within our construction, he must engage in a `Deposit` protocol, where he exchanges currency with an ACLcoin, $\mathcal{AC}$. The user takes as input the service's public key, the amount of money he wants exchange ($amount$), as well as his own and the service's bank accounts ($account_{\mathcal{U}}, account_{\mathcal{S}}$). At the end of the the protocol, the user has transferred $amount$ from $account_{\mathcal{U}}$ to $account_{\mathcal{S}}$ and has obtained an ACLcoin $\mathcal{AC}$ whose balance is equal to $amount$.

– `Withdraw`: $\mathcal{U}(PK_{\mathcal{S}}, \mathcal{AC}, \gamma, rnd, R, BankAccount)$; $\mathcal{S}()$
If a user wants to get his money out of the service, he must engage in a `Withdraw` protocol, which is a protocol between a user $\mathcal{U}$ and the service $\mathcal{S}$. $\mathcal{U}$ takes as input the service's public key, an ACLcoin $\mathcal{AC}$ that he owns, the openings to the blinded commitment, as well as a bank account $BankAccount$. The service takes no inputs. At the end of the protocol, the service has invalidated $\mathcal{AC}$ and has made a bank transfer to $account$ worth the balance of $\mathcal{AC}$.

– `Split`: $\mathcal{U}(PK_{\mathcal{S}}, \mathcal{AC}, \gamma, rnd, L_1, splitAmount, l)$; $\mathcal{S}(l)$
If a user has a coin that he wants to split, he must engage in a `Split` protocol between a user $\mathcal{U}$ and the service $\mathcal{S}$. $\mathcal{U}$ takes as input the service's public key $PK_{\mathcal{S}}$, an ACLcoin $\mathcal{AC}$ that he owns, its openings γ, rnd, L_1, the amount of money that he wants to split the coin by, $splitAmount$, as well as a positive integer l, such that 2^l denotes the maximum bit representation that the value of $splitAmount$ can take. We assume that l is set once and never changes. The service takes as input l. At the end of the protocol, $\mathcal{S}$ has invalidated $\mathcal{AC}$, and $\mathcal{U}$ has obtained two fresh ACLcoins $\mathcal{AC}_1$, $\mathcal{AC}_2$ whose balance equals that of $\mathcal{AC}$.

– `Combine`: $\mathcal{U}(PK_{\mathcal{S}}, \widetilde{\mathcal{AC}}, \widetilde{rnd}, \tilde{\gamma}, \tilde{v_1}, \widehat{\mathcal{AC}}, \widehat{rnd}, \hat{\gamma}, \hat{v_2})$; $\mathcal{S}()$
If a user has two coins whose value he wants to combine, he must engage in a `Combine` protocol between a user $\mathcal{U}$ and the service $\mathcal{S}$. $\mathcal{U}$ takes as input the service's public key $PK_{\mathcal{S}}$, and two ACLcoins $\widetilde{\mathcal{AC}}$, $\widehat{\mathcal{AC}}$ that he owns along with their openings $(\widetilde{rnd}, \tilde{\gamma}, \tilde{v_1}, \widehat{rnd}, \hat{\gamma}, \hat{v_2})$. The service takes no inputs. At the end of the protocol, $\mathcal{S}$ has invalidated both $\widetilde{\mathcal{AC}}$, $\widehat{\mathcal{AC}}$ and $\mathcal{U}$ has obtained a fresh ACLcoin $\mathcal{AC}$, whose balance equals that of $\widetilde{\mathcal{AC}} + \widehat{\mathcal{AC}}$.

– `Spend`: $\mathcal{U}(PK_{\mathcal{M}}, \mathcal{AC}, rnd, \gamma, R, L_1)$; $\mathcal{M}()$

A protocol between 2 entities: a user $\mathcal{U}$, and a merchant $\mathcal{M}$, where $\mathcal{U}$ transfers an ACLcoin $\mathcal{AC}$ to $\mathcal{M}$. $\mathcal{U}$ takes as input the ACLcoin $\mathcal{AC}$, its openings (γ, rnd, R), as well as $\mathcal{M}$'s public key. $\mathcal{M}$ takes as input his secret and public key pair. At the end of the protocol, $\mathcal{M}$ has obtained $\mathcal{AC}$. We should note that $\mathcal{M}$ has no way of knowing whether the coin he just received has already been used or not. A technique that can be used to check that is to engage in a `Split` protocol where the *splitAmount* is 0, thus, if not spent before, creating two coins: one of value 0 and the other of value L_1.

We will now informally describe some basic security properties of our system. As we will explain in the "Security Policy & Threat Model" section, we assume an honest service. If the service is not honest, it can behave in arbitrary ways that we cannot predict.

- *Blindness.* The service cannot view the messages and the attributes it signs and it cannot link a coin to its issuing [5].
- *Unforgeability.* An adversary must not be able to forge a coin, and therefore, create money.
- *Double spending detection.* The service can detect if a given coin has already been spent or not.
- *Transaction Correctness.* If an honest user engages in a `Withdraw` protocol with the service, the user obtains a valid coin. If an honest user engages in a `Split` with the service, the user obtains two valid coins. If an honest user engages in a `Combine` protocol with the service, the user obtains one valid coin. If an honest user engages in a `Spend` protocol with an honest merchant, the merchant accepts the coin.
- *User anonymity.* The service cannot learn the identity of the user that is interacting with. This property must hold in `Split`, `Combine`, and `Spend` protocols.
- *Unlinkability.* The service cannot tell whether two coins that were not sent together were sent by the same or by different users. This property must hold in `Split`, `Combine`, and `Spend` protocols.
- *Hidden Balance.* If the service interacts with an honest user it cannot tell the denomination of each coin that it receives in `Split` and `Combine` protocols.

3.3 Security Policy & Threat Model

We assume the service always to be honest, and communication channels to be anonymous and private. A dishonest service can behave in an arbitrary fashion that we cannot control. Anonymous communication channels ensure that the service does not know the identity of the user that it is interacting with, while private communication channels ensure that an adversary cannot read the communication between the service and an honest user.

We now define the principals of our system, the assets that must be protected, and our threat model.

Principals
Our system consists of three principals: the service provider, the user, and the merchant.

Assets & Properties
The assets and properties that we are trying to protect against adversaries are the confidentiality and integrity of the service's private key, the integrity of the service's database containing the already spent coins, the availability of the service, as well as the confidentiality and integrity of the communication channels.

Threat Model
Our threat model cannot be very different than the threat model in the ACL paper, since we use ACLs as our basic building block. We assume a probabilistic polynomial time adversary who has in depth knowledge of the workings of the system, can play the roles of the user and the merchant, but cannot be the service, can observe traffic (which, as described before, is assumed to be encrypted), and, finally, cannot delete or modify traffic.

3.4 Construction

-ParamGen. Parameters are public and fixed, they only need to be generated once and we assume them as default input in every protocol. In this scenario we assume that the service, $\mathcal{S}$, is a trusted party. This is similar to the setup of the original ACL construction. Let G be a group of order q with a generator g. Also let $\mathcal{H} : \{0,1\}^* \to \mathbb{Z}_q$ be a hash function. The service chooses $z, h, h_0, h_1 \in G$ and sets $params = (G, g, q, z, h, h_0, h_1)$.

-SKeyGen. The service chooses its secret and public key. It chooses for secret key $x \xleftarrow{\$} \mathbb{Z}_q$ and sets its public key $y = g^x \mod q$. We assume that each service knows its public and private keys.

-Deposit. For this protocol we slightly twist the Registration phase of the original ACL construction. Recall that L_1 is the amount of money. In this scenario L_1 is sent by the user to the service along with the commitment $C = {h_0}^R {h_1}^{L_1}$. The user makes L_1 public, hides R, and includes a NIZK proving knowledge of the secret of the commitment, i.e. proving knowledge of R. Table 4 depicts the protocol.

Table 4: The `Deposit` Protocol

$\mathcal{U}(PK_{\mathcal{S}}, amount, account_{\mathcal{U}}, account_{\mathcal{S}})$		$\mathcal{S}()$
Registration		
$L_1 = amount$		
$C = {h_0}^R {h_1}^{L_1}$		
	$\xrightarrow{C, L_1, {h_1}'}$	
	$\xrightarrow{NIZK\{(R):\ {h_1}' = {h_1}^{L_1} \ \wedge\ C = {h_0}^R {h_1}'\}}$	
		Preparation - Validation 1
		If $\neg NIZK$, then $\perp$
	$\xleftrightarrow{Rest\ of\ ACL(C)}$	
Makes ACLcoin		
Return ACLcoin		

The importance of making R a secret. Instead of the user having to create the commitment C and send it to the verifier along with the proof, it could be the case that the issuer created C and sent C, R to the user who would verify that $C \stackrel{?}{=} {h_0}^R {h_1}^{L_1}$ is true. This would have been faster, but it would make the ACLcoin created using `Deposit` linkable. This is due to the fact that when the verifier would be given the blinded version of C, which would be $\zeta_1 = ({h_0}^{\gamma})^R ({h_1}^{\gamma})^{L_1} (g^{\gamma})^{rnd}$, the verifier would know everything apart from γ. However that would not be a problem, since the verifier (as will be seen in other protocols) will be given the individual values of ${h_0}^{\gamma}, {h_1}^{\gamma}$ and g^{γ} that he could try raising to all the R, L_1, rnd values he had generated in the past and check whether this matches the given ζ_1 or not. If ζ_1 was indeed a coin generated using the `Deposit` protocol, the service could know the owner of that ACLcoin. Saying "could" because the service has no way of knowing whether that coin has been transferred before being presented or not.

-Withdraw. If a user wants to exchange an $\mathcal{AC}$ back to currency and, therefore, take

money out of the service, he must engage in a `Withdraw` protocol. He must convince the service of the ownership of a coin by presenting the coin and its denomination along with a bank account, in order for the service to transfer the corresponding amount. `Withdraw` protocol can be seen at table 5.

Table 5: The `Withdraw` Protocol

$\mathcal{U}(PK_{\mathcal{S}}, \mathcal{AC}, \gamma, rnd, R, BankAccount)$	$\mathcal{S}()$
$(m, \zeta, \zeta_1, \zeta_2, \omega, \omega', \rho, {\rho_1}', {\rho_2}', \mu) = \mathcal{AC}$ ${h_0}' = {h_0}^{\gamma}, {h_1}' = {h_1}^{\gamma}, g' = g^{\gamma}$	
$\xrightarrow{\mathcal{AC}, {h_0}', {h_1}', g', L_1, BankAccount}$	
$NIZK\Big\{(R,\gamma,rnd):$ $h_0{}'=h_0{}^{\gamma} \wedge h_1{}'=h_1{}^{\gamma} \wedge g'=g^{\gamma} \wedge$ $\zeta=z^{\gamma} \wedge \zeta_1=h_0{}'^{R} h_1{}'^{L_1} g'^{rnd}\Big\}$	
$\longrightarrow$	
	If $\neg NIZK$, then $\perp$ transfer L_1 to $BankAccount$

The importance of keeping γ, rnd, and R secret. Alternatively to using a NIZK, we could have revealed the openings γ, rnd, and R to the service, however, if the service kept a list of the rnd values it has generated, it would be able to link our coin with its issuance, thus, devaluing the unlinkability property of our scheme. By proving knowledge of the openings to the verifier and only revealing L_1, we do not allow the service to link the received coin with its issuance, since of the openings of $\zeta_1 = {h_0}'^{R} {h_1}'^{L_1} g'^{rnd}$, the verifier cannot learn R and γ, which are secret and only known to the user.

-Split. Assuming that a user has obtained an $\mathcal{AC} = (m, \zeta, \zeta_1, \zeta_2, \omega, \omega', \rho, {\rho_1}', {\rho_2}', \mu)$, and wants to get 2 coins of values v_1 and v_2, the user must present the following NIZK proof to the service:

$$
\begin{aligned}
NIZK\Big\{\Big(R, L_1, \gamma, R', v_1, R'', v_2, rnd\Big) : \zeta = z^{\gamma} \wedge \zeta_1 = ({h_0}^{R} {h_1}^{L_1} g^{rnd})^{\gamma} \wedge \\
C' = {h_0}^{R'} {h_1}^{v_1} \wedge C'' = {h_0}^{R''} {h_1}^{v_2} \wedge \\
L_1 = v_1 + v_2 \wedge v_1 \in [0, 2^l - 1] \wedge \\
v_2 \in [0, 2^l - 1]\Big\}
\end{aligned}
\tag{3}
$$

Notice that although γ is a secret, we can make public the following ${h_0}^{\gamma}, {h_1}^{\gamma}, g^{\gamma}$, due to the fact that γ is a random number and deducing it from these values would be equivalent to solving the discrete logarithm problem.

(3) after the appropriate substitutions are made becomes:

$$NIZK\Big\{\Big(R,\gamma,R',v_1,R'',v_2,rnd\Big) : \zeta = z^{\gamma} \wedge {h_0}' = {h_0}^{\gamma} \wedge {h_1}' = {h_1}^{\gamma} \wedge g' = g^{\gamma} \wedge \\ \zeta_1 = {h_0}'^{R} {h_1}'^{v_1} {h_1}'^{v_2} {g'}^{rnd} \wedge C' = {h_0}^{R'} {h_1}^{v_1} \wedge \\ C'' = {h_0}^{R''} {h_1}^{v_2} \wedge v_1 \in [0, 2^l - 1] \wedge \\ v_2 \in [0, 2^l - 1]\Big\} \tag{4}$$

The importance of atomicity. The `Split` protocol we have designed takes as input one ACLcoin $\mathcal{AC}$ and produces two ACLcoins $\mathcal{AC}_1, \mathcal{AC}_2$ for C' and C'' respectively by executing a modified version of the ACL protocol that creates two ACLcoins instead of one. Alternatively, we could have engaged in two separate executions of the ACL protocol that are sequential in time, one in order to obtain $\mathcal{AC}_1$ for C' and one to obtain $\mathcal{AC}_2$ for C''. There are two problems with the alternative approach: a) it is difficult to ensure atomicity (i.e that both ACL protocol executions complete or nothing completes), and b) it involves more rounds of communication. The main problem with atomicity, is the case where the first ACL protocol execution might complete, while the second one not. In that case the user would have obtained $\mathcal{AC}_1$, but not $\mathcal{AC}_2$, and the service would have to invalidate the input $\mathcal{AC}$, as otherwise the user could spend it again. This scenario has no outcome that can satisfy both parties. For this reason, we created a modification of the ACL protocol that takes as input an $\mathcal{AC}$, runs once and at the end produces $\mathcal{AC}_1$ and $\mathcal{AC}_2$. Now, if the execution was successful, the service invalidates $\mathcal{AC}$, while if otherwise, it does not.

The `Split` protocol can be seen at table 6.

Table 6: The `Split` Protocol.

$\mathcal{U}(PK_{\mathcal{S}}, \mathcal{AC}, \gamma, rnd, L_1, splitAmount, l)$	$\mathcal{S}(l)$
Registration	
$(m, \zeta, \zeta_1, \zeta_2, \omega, \omega', \rho, {\rho_1}', {\rho_2}', \mu) = \mathcal{AC}$	
${h_0}' = {h_0}^{\gamma}, {h_1}' = {h_1}^{\gamma}, g' = g^{\gamma}$	
$v_1 = L_1 - splitAmount, v_2 = splitAmount$	
$\zeta_1 = {h_0}'^{R} {h_1}'^{v_1} {h_1}'^{v_2} {g'}^{rnd}$	
$R', R'' \xleftarrow{\$} \mathbb{Z}_q$	
$\xrightarrow{{h_0}', {h_1}', g', \mathcal{AC}, C', C''}$	
$NIZK\Big\{(R,\gamma,R',v_1,R'',v_2,rnd):$	
${h_0}' = {h_0}^{\gamma} \wedge {h_1}' = {h_1}^{\gamma} \wedge g' = g^{\gamma}$	
$\wedge \zeta = z^{\gamma} \wedge \zeta_1 = {h_0}'^{R} {h_1}'^{v_1} {h_1}'^{v_2} {g'}^{rnd} \wedge C' = {h_0}^{R'} {h_1}^{v_1}$	
$\wedge C'' = {h_0}^{R''} {h_1}^{v_2} \wedge v_1 \in [0, 2^l - 1] \wedge v_2 \in [0, 2^l - 1]\Big\}$	
$\longrightarrow$	
	Preparation - Validation 1

`Split` protocol continued

If $\neg NIZK$, then $\perp$
$rnd, \widetilde{rnd} \in_R \mathbb{Z}_q$
$z_1 = C' g^{rnd}, \tilde{z_1} = C'' g^{\widetilde{rnd}}$
$z_2 = z/z_1, \widetilde{z_2} = z/\tilde{z_1}$
$u, {r_1}', {r_2}', c' \in_R \mathbb{Z}_q$
$\tilde{u}, \widetilde{r_1'}, \widetilde{r_2'}, \tilde{c'} \in_R \mathbb{Z}_q$
$a = g^u, \tilde{a} = g^{\tilde{u}}$
${a_1}' = g^{{r_1}'} {z_1}^{c'}, \widetilde{a_1'} = g^{\widetilde{r_1'}} {z_1}^{\tilde{c'}}$
${a_2}' = h^{{r_2}'} {z_2}^{c'}, \widetilde{a_2'} = h^{\widetilde{r_2'}} {z_2}^{\tilde{c'}},$

$\xleftarrow{rnd, \widetilde{rnd}, a, \tilde{a}, {a_1}', \widetilde{a_1'}, {a_2}', \widetilde{a_2'}}$

Validation 2

check if $rnd, \widetilde{rnd} \neq 0$
check if $a, \tilde{a}, {a_1}', \widetilde{a_1'}, {a_2}', \widetilde{a_2'} \in G$
$z_1 = C' g^{rnd}, \tilde{z_1} = C'' g^{\widetilde{rnd}}$
$\gamma, \tilde{\gamma} \in_R \mathbb{Z}_q^*$
$\zeta = z^{\gamma}, \tilde{\zeta} = z^{\tilde{\gamma}}$
$\zeta_1 = {z_1}^{\gamma}, \tilde{\zeta_1} = \tilde{z_1}^{\tilde{\gamma}}$
$\zeta_2 = \zeta/\zeta_1, \tilde{\zeta_2} = \tilde{\zeta}/\tilde{\zeta_1}$
$\tau, \tilde{\tau} \in_R \mathbb{Z}_q$
$\eta = z^{\tau}, \tilde{\eta} = z^{\tilde{\tau}}$
$t_1, t_2, t_3, t_4, t_5, \tilde{t_1}, \tilde{t_2}, \tilde{t_3}, \tilde{t_4}, \tilde{t_5} \in_R \mathbb{Z}_q$
$\alpha = a g^{t_1} y^{t_2}, \tilde{\alpha} = \tilde{a} g^{\tilde{t_1}} y^{\tilde{t_2}}$
${\alpha_1}' = {a_1}'^{\gamma} g^{t_3} {\zeta_1}^{t_4}, \widetilde{\alpha_1'} = \widetilde{a_1'}^{\tilde{\gamma}} g^{\tilde{t_3}} \tilde{\zeta_1}^{\tilde{t_4}}$
${\alpha_2}' = {a_2}'^{\gamma} h^{t_5} {\zeta_2}^{t_4}, \widetilde{\alpha_2'} = \widetilde{a_2'}^{\tilde{\gamma}} h^{\tilde{t_5}} \tilde{\zeta_2}^{\tilde{t_4}}$
$\varepsilon = \mathcal{H}_2(\zeta, \zeta_1, \alpha, {\alpha_1}', {\alpha_2}', \eta, m)$
$\tilde{\varepsilon} = \mathcal{H}_2(\tilde{\zeta}, \tilde{\zeta_1}, \tilde{\alpha}, \tilde{\alpha_1'}, \tilde{\alpha_2'}, \tilde{\eta}, \tilde{m})$
$\tilde{e} = (\tilde{\varepsilon} - \tilde{t_2} - \tilde{t_4}) \mod q$

$\xrightarrow{e, \tilde{e}}$

$c = e - c' \mod q, \tilde{c} = \tilde{e} - \tilde{c'} \mod q$
$r = u - cx \mod q, \tilde{r} = \tilde{u} - \tilde{c}x \mod q$

$\xleftarrow{c, \tilde{c}, r, \tilde{r}, c', \tilde{c'}, {r_1}', \widetilde{r_1'}, {r_2}', \widetilde{r_2'}}$

$\rho = r + t_1 \mod q, \tilde{\rho} = \tilde{r} + \tilde{t_1} \mod q$
$\omega = c + t_2 \mod q, \tilde{\omega} = \tilde{c} + \tilde{t_2} \mod q$
${\rho_1}' = \gamma {r_1}' + t_3 \mod q, \widetilde{\rho_1'} = \tilde{\gamma} \widetilde{r_1'} + \tilde{t_3} \mod q$
${\rho_2}' = \gamma {r_2}' + t_5 \mod q, \widetilde{\rho_2'} = \tilde{\gamma} \widetilde{r_2'} + \tilde{t_5} \mod q$
$\omega' = c' + t_4 \mod q, \tilde{\omega'} = \tilde{c'} + \tilde{t_4} \mod q$
$\mu = \tau - \omega' \gamma \mod q, \tilde{\mu} = \tilde{\tau} - \tilde{\omega'} \tilde{\gamma} \mod q$
$\mathcal{AC}_1 = (m, \zeta, \zeta_1, \zeta_2, \omega, \omega', \rho, {\rho_1}', {\rho_2}', \mu)$
$\mathcal{AC}_2 = (\tilde{m}, \tilde{\zeta}, \tilde{\zeta_1}, \tilde{\zeta_2}, \tilde{\omega}, \tilde{\omega'}, \tilde{\rho}, \tilde{\rho_1'}, \tilde{\rho_2'}, \tilde{\mu})$
Return $\mathcal{AC}_1, \mathcal{AC}_2$

End of `Split`

The importance of $\zeta = z^{\gamma}$. Recall that z is a public parameter, raised to the γ which is a secret. It is of paramount importance that this is included in the proof, otherwise an attack can take place. We first describe the attack and then explain why including $\zeta = z^{\gamma}$ prevents it. Assume that the verifier is given a $\zeta_1 = {h'_0}^{R} {h'_1}^{v_1} {h'_1}^{v_2} {g'}^{rnd}$. The verifier at this point knows that h'_0 is supposed to be h_0^{γ}, but he cannot be sure that he is indeed given h_0 raised to the γ. An adversary could have used a different base and a different exponent such that $h_k^{\lambda} = h_0^{\gamma}$, where λ can represent an amount that he wished to have. Notice that this is not equivalent to solving the discrete logarithm problem. Instead, high school knowledge of mathematics suffices to solve this. To see how could this work in practice, assume that the original $h_0 = 4$ and $\gamma = 5$. An adversary could pick another base $h_k = 2$ and $\lambda = 10$. Notice that $4^5 = (2^2)^5 = 2^{10}$, which eventually leads to adding more money to his account. By including $\zeta = z^{\gamma}$, we force the adversary to use that γ, since in the proof the same γ that is an exponent to z is also used as an exponent in the bases h_0, h_1, and g. Notice that in order for the adversary to cheat, he must find another value τ', such that $z^{\tau'} = \zeta = z^{\gamma}$, which requires solving the discrete logarithm problem.

The importance of $v_1 \in [0, 2^l - 1] \wedge v_2 \in [0, 2^l - 1]$. Since we are working in a group whose order is q, operations are $\bmod q$. If we are not careful, this can lead to attacks where the adversary can create an arbitrary amount of money that does not correspond to money that he has indeed deposited. Initially, we will explain the attack that would occur if we did not include the range proofs and, then, describe how adding the range proofs prevents the attack. If we did not have the proofs in place, an attacker could present $\zeta_1 = {h_0}'^{R} {h_1}'^{L_1} {g'}^{rnd}$, as well as $C' = {h_0}^{R'} {h_1}^{v_1} \wedge \ C'' = {h_0}^{R''} {h_1}^{v_2}$, where he would show that $L_1 = v_1 + v_2$. To make the example more concrete, assume that $L_1 = 100$ which corresponds to money that he has indeed deposited and has obtained and ACLcoin for. The attacker could chose $v_1 = q - 1$, and $v_2 = 101$, so that $L_1 = v_1 + v_2 = q - 1 + 101 = 100(\mod q)$. Therefore, the attacker could now create two ACLcoins of denominations $q - 1$ and 101 respectively. By including a range proof we do not allow for that kind of attacks, since the issuer can check that both v_1, v_2 lie in $[0, 2^l - 1]$. From the inequality addition depicted in table 7, we can conclude that if we set for l a value such that $2^{l+1} < q$, then the attack described is restricted.

Table 7: Maximum Value for $v_1 + v_2$

$$\begin{array}{rc} & 0 \leq v_1 < 2^l \\ + & 0 \leq v_2 < 2^l \\ \hline & 0 \leq v_1 + v_2 < 2^{l+1} \end{array}$$

-Combine. In addition, we present the `Combine` protocol. Prior to that we present the NIZK that is needed by the prover to prove to the verifier that indeed he is building a new ACL whose balance is the sum of the other 2. `Combine` takes as input 2 ACLs and produces a new ACL whose balance is the sum of the entered ACL balances.

Table 8: The `Combine` Protocol

$\mathcal{U}(\widetilde{\mathcal{AC}}, \widetilde{rnd}, \tilde{\gamma}, \tilde{v_1}, \widehat{\mathcal{AC}}, \widehat{rnd}, \hat{\gamma}, \hat{v_2})$	$\mathcal{S}()$
Registration	
$\tilde{h_0} = {h_0}^{\tilde{\gamma}}, \tilde{h_1} = {h_1}^{\tilde{\gamma}}, \tilde{g} = g^{\tilde{\gamma}}$	
$\tilde{\zeta_1} = {\tilde{h_0}}^{\tilde{R}} {\tilde{h_1}}^{\tilde{v_1}} \widetilde{g^{rnd}}$	
$\hat{h_0} = {h_0}^{\hat{\gamma}}, \hat{h_1} = {h_1}^{\hat{\gamma}}, \hat{g} = g^{\hat{\gamma}}$	
$\hat{\zeta_1} = {\hat{h_0}}^{\hat{R}} {\hat{h_1}}^{\hat{v_2}} \widehat{g^{rnd}}$	
$R \xleftarrow{\$} \mathbb{Z}_q$	
$\xrightarrow{\tilde{h_0}, \tilde{h_1}, \tilde{g}, \hat{h_0}, \hat{h_1}, \hat{g}, C}$	
$\xrightarrow{\widetilde{\mathcal{AC}} = (\tilde{m}, \tilde{\zeta}, \tilde{\zeta_1}, \tilde{\zeta_2}, \tilde{\omega}, \tilde{\omega'}, \tilde{\rho}, \tilde{\rho_1}', \tilde{\rho_2}', \tilde{\mu})}$	
$\xrightarrow{\widehat{\mathcal{AC}} = (\hat{m}, \hat{\zeta}, \hat{\zeta_1}, \hat{\zeta_2}, \hat{\omega}, \hat{\omega'}, \hat{\rho}, \hat{\rho_1}', \hat{\rho_2}', \hat{\mu})}$	
$NIZK\{(\tilde{R}, \tilde{\gamma}, \tilde{v_1}, \widetilde{rnd}, \hat{R}, \hat{\gamma}, \hat{v_2}, \widehat{rnd}, R):$	
$\hat{h_0} = {h_0}^{\hat{\gamma}} \wedge \hat{h_1} = {h_1}^{\hat{\gamma}} \wedge \hat{g} = g^{\hat{\gamma}} \wedge$	
$\tilde{h_0} = {h_0}^{\tilde{\gamma}} \wedge \tilde{h_1} = {h_1}^{\tilde{\gamma}} \wedge \tilde{g} = g^{\tilde{\gamma}} \wedge$	
$\tilde{\zeta} = z^{\tilde{\gamma}} \wedge \hat{\zeta} = z^{\hat{\gamma}} \wedge$	
$\tilde{\zeta_1} = {\tilde{h_0}}^{\tilde{R}} {\tilde{h_1}}^{\tilde{v_1}} \widetilde{g^{rnd}} \wedge$	
$\hat{\zeta_1} = {\hat{h_0}}^{\hat{R}} {\hat{h_1}}^{\hat{v_2}} \widehat{g^{rnd}} \wedge$	
$C = {h_0}^{R} {h_1}^{\tilde{v_1}} {h_1}^{\hat{v_2}}\}$	
$\longrightarrow$	
	Prep., Val. 1, Val. 2
	If $\neg NIZK$, then $\perp$
$\xleftarrow{Rest\ of\ ACL(C)}$	
Makes $\mathcal{AC}$	
Return $\mathcal{AC}$	
End of `Combine`	

-Spend. If a user $\mathcal{U}$ wants to pay a merchant $\mathcal{M}$ he must engage in a `Spend` protocol, which requires the former to provide the ACL he has obtained from the service, as well as its value L_1 so that $\mathcal{M}$ can use it. Table 9 depicts the `Spend` protocol.

Table 9: The `Spend` Protocol

$\mathcal{U}(PK_{\mathcal{M}}, \mathcal{AC}, rnd, \gamma, R, L_1)$		$\mathcal{M}()$
$(m, \zeta, \zeta_1, \zeta_2, \omega, \omega', \rho, {\rho_1}', {\rho_2}', \mu) = \mathcal{AC}$		
	$\xrightarrow{\mathcal{AC}, L_1, {h_0}', {h_1}', g', z'}$	
	$NIZK\{(R,\gamma,rnd):$ $h_0{}' = h_0{}^{\gamma} \wedge h_1{}' = h_1{}^{\gamma} \wedge g' = g^{\gamma} \wedge$ $\zeta = z^{\gamma} \wedge \zeta_1 = {h_0}'^{R} {h_1}'^{L_1} g'^{rnd} \wedge z' = z^{L_1}\}$	
	$\longrightarrow$	
		Return $\mathcal{AC}$

According to the protocol at table 9, the merchant never learns all of the openings of a coin. This is to maintain the unlinkability property of our scheme. We cannot trust the merchant and reveal the openings γ, rnd, R, since it could be the case that the merchant and the service are the same entity, or that the merchant and the service collude with each other, and therefore, the service could learn the rnd value and link the coin to its issuance, thus devaluing the unlinkability property of our scheme.

-DoubleSpent. If a merchant $\mathcal{M}$ has been given an $\mathcal{AC}$ by a user $\mathcal{U}$ and wants to check if the coin has already been spent, and if not, then obtain new one to which he would know the openings, he must engage in a `DoubleSpent` protocol. Table 10 depicts the `DoubleSpend` protocol.

In table 10 $NIZK_1$ is the same NIZK sent by the user to the merchant in the `Spend` protocol, while $NIZK_2$ is a new NIZK that is used to prove the openings of C. Also by including $z' = z^{L_1}$ in $NIZK_2$, we allow to prove equality between the L_1 in ζ_1 of $NIZK_1$ and the L_1 value used in the $NIZK_2$.

Table 10: The `DoubleSpent` Protocol

$\mathcal{M}(\mathcal{AC}, R', L_1)$		$\mathcal{S}()$
$(m, \zeta, \zeta_1, \zeta_2, \omega, \omega', \rho, {\rho_1}', {\rho_2}', \mu) = \mathcal{AC}$		
	$\xrightarrow{\mathcal{AC}}$	
		Check if $\mathcal{AC}$ has been double spent. If yes, reply 1, otherwise 0.
	$\xleftarrow{\{0,1\}}$	
if 1, then $\perp$. if 0, continue.		
	$\xrightarrow{L_1, C, {h_0}', {h_1}', g', \zeta, \zeta_1, z', {h_1}''}$	
	$NIZK_1\{(R,\gamma,rnd): {h_0}'={h_0}^{\gamma} \wedge {h_1}'={h_1}^{\gamma} \wedge g'=g^{\gamma} \wedge \zeta=z^{\gamma} \wedge \zeta_1={h_0}'^{R}{h_1}'^{L_1}g'^{rnd} \wedge z'=z^{L_1}\}$	
	$\longrightarrow$	
	$NIZK_2\{(R'): {h_1}''={h_1}^{L_1} \wedge z'=z^{L_1} \wedge C={h_0}^{R'}{h_1}''\}$	
	$\longrightarrow$	
		If $\neg NIZK_1$ or $\neg NIZK_2$, then $\perp$
	$\xleftrightarrow{Rest\ of\ ACL(C)}$	
Makes $\mathcal{AC}$		
Return $\mathcal{AC}$		

3.5 Ensuring that an ACL issuer is the verifier

A feature of ACL is that the issuer and the verifier do not have to be the same entity. If a verifier has the public key of an issuer, he is able to do the ACL verification. Therefore, services that are owned by different entities can verify ACLcoins for other services. Adhering to this is a matter of preference and trust. As we will see later in our implementation, we chose to be on the safe side and not to allow this, since a verifier has to trust that the issuer is not malicious. A malicious issuer could have generated ACLcoins that do not actually represent real money that have been deposited in the service. If the verifier verified such a fake ACLcoin, money would be created that do not really exist. For that reason, we suggest that an ACLcoin can be used only within the same service. i.e. assume two distinct services $\mathcal{S}_1$ and $\mathcal{S}_2$. If a user converts currency to ACL using $\mathcal{S}_1$, then he cannot use $\mathcal{S}_2$ to engage in any of the protocols.

3.6 Integration with existing currencies

To exchange currency to $\mathcal{AC}$ for usage within the service, the service $\mathcal{S}$ has to accept payments in a variety of currencies. Once the payment is initiated, the user and the service will engage in a `Deposit` protocol at the end of which the user will have obtained an $\mathcal{AC}$ to the amount of the payment.

When a user wants to convert an $\mathcal{AC}$ back to currency, he must engage in a `Withdraw` protocol at the end of which the service makes a transfer to the user's bank account whose value is the value in the $\mathcal{AC}$.

Decimal Currencies. Most currencies are decimal currencies, therefore, currency denominations belong in the set of positive rational numbers ($\mathbb{Q}_{\geq 0}$), and not in the set of positive integers with order q ($\mathbb{Z}_q$) that we work. By multiplying each deposited value by 100 (or with the equivalent "cent" value of the currency used) we provide a mapping from $\mathbb{Q}_{\geq 0}$ to $\mathbb{Z}_q$.

Bitcoin Integration. Bitcoin's smallest unit is the *shatosi*, which corresponds to 10^{-8} BTCs. Since all balances in the service must be in $\mathbb{Z}_q$, we multiply the balance by 10^8 to make it an element in $\mathbb{Z}_q$. After doing so, we give to the user an ACLcoin which he can use in the system as explained thus far. When a user wants to convert an ACLcoin to Bitcoin, he engages in a `Withdraw` protocol where he presents the ACLcoin to the service, which divides its balance by 10^{-8} to get the corresponding Bitcoin value, asks the user for a Bitcoin address to sent the Bitcoin to. The service generates a new Bitcoin address, to which it transfers the right amount of BTCs from other accounts it owns. Finally, the service using this newly generated address sends the Bitcoins to the address entered by the user. For the security of the Bitcoin integration the reader is referred to the evalutation section of this document.

3.7 Design Choices

The $\mathcal{OP}$ scheme we have been discussing so far was not our initial construction. In this part we discuss the alternative candidate constructions that we came up with and why we decided on our current construction. We present four constructions including our final $\mathcal{OP}$ construction, discuss their functionality, their advantages and disadvantages, as well as their properties. At the end of our discussion, we provide a comparison between the three alternatives and our current choice in the form of a table.

Before explaining the alternatives we should note that all of them were designed to work in a server-client architecture in a centralised scenario. In addition, for the properties we will explain, we assume that the active users using the system are at least two. If there is only one user in a system, then we cannot provide anonymity.

Construction 1. For our first construction we designed a scheme where each user would have an account and the service would maintain all users' balances encrypted under each user's public key using an additively homomorphic scheme, such as the Benaloh cryptosystem [8][6]. This means that all balances would be encrypted and the service would not be able to read them. In addition, due to the homomorphic property of the scheme, operations could be done on ciphertexts, thus a transfer of an encrypted amount $E(x)$ from user to a merchant would require the service to homomorphically add $E(x)$ to merchant's balance and add $E(-x)$ to user's balance, effectively reducing his balance. Such an action would be initiated only by the user given that he would provide a NIZK proving that he has sufficient balance and that x is a positive value (as otherwise he would add money to his account). The service would maintain a database holding each user's id, username, as well as their encrypted balance. Upon creation of an account the user would create a private/public key pair and would give the public key to the service that in its turn would encrypt the value 0 and store this as the encrypted balance of the user. Later, when a user would like to add money to his account, he would pay the service and the service would add the newly added amount to his balance. This system hides the amount of money transferred from user to user, and, therefore, potentially can hide a user's total balance. The main drawback with this approach was the exposure of the anonymity of the user and the merchant.

Construction 2. For our second construction we tried to hide the user and merchant identities, therefore we designed an accountless system [7]. In this scenario, the service still maintains the encrypted balances of all users, however, users do not maintain accounts with usernames as in the first construction. Instead, they only maintain public keys. We distinguish between two kinds of public keys: those that can be linked to a user's identity (known public keys) and those that cannot (unknown public keys). The former can be used by the user to add money to the service or to receive money by others, while the latter must not be linked with his identity and should be only used in this service. The service maintains a publicly readable database with 2 columns:

[6]Such a scheme could be implemented in the form of a website application where each user has an account.

[7]Such a system would not need a website anymore and it could be implemented using a networked desktop application that communicates with the service.

public keys and encrypted balances. Each user reads the database using trivial PIR [21] or (1-n) oblivious transfer [27], so that the service does not learn the entry he is interested in. When the user wants to transfer an amount x to a merchant, he encrypts $E(x), E(-x)$, and $E(0)$. Since he can read the whole database, he can read all of the public keys and use them to encrypt values. Doing so, the user adds $E(0)$ to all entries apart from his and the merchant's, adds $E(x)$ to the merchant's account and $E(-x)$ to his account, effectively "touching" all accounts and disallowing the service to know the two accounts that were involved in the transaction due to the rerandomisation property of the Benaloh cryptosystem. Of course these additions require the user to create a number of NIZK proofs linear to the number of the entries in the database. This construction hides the amount of money being transferred from user to merchant, the user's and the merchant's identity, as well as when and how often a user uses the service. The main drawback of this construction is scalability. The database can soon become very large and downloading it would take time. In addition, due to the number of the NIZK proofs the transaction time would increase as the number of entries increased.

Construction 3. In our third construction we tried to tackle the scalability issue of our previous construction. For that we introduced ACLs in the form of coins. The service would still maintain the encrypted balances, but transactions would require withdrawing a coin out of an account, passing it to someone else who then could deposit it into his account. In this scenario, users have to indicate to which encrypted balance to add the coin's value. In order to maintain users' anonymity, an anonymity set must be formulated in a pool or threshold manner [44]. This construction solves the scalability issue, since the coin is a constant time operation in the sense that it does not depend on the number of the users in the system. In addition, this system becomes more robust as the number of active users increases, since it increases the anonymity set. The drawback of this construction is that the anonymity set depends on others, i.e. others must use the system in order to function properly, and is prone to Sybil attacks [26], i.e. the type of attack where the attacker can control a large number of the user set and eventually learn to which account an honest user deposits his coin.

Construction 4 (Current). Our final construction is the $\mathcal{OP}$ scheme that we focused on in this 'Design' section. This construction's operations, as in the previous construction, do not depend on the number on the number of the users or on the number of database entries. However, in addition to the previous construction, it does not require the generation of an anonymity set, therefore is not prone to Sybil attacks. In this scenario, state is removed from the service and is maintained in the client. The service

does not keep the encrypted balances of each user, but instead they are encoded as attributes in ACL credentials forming ACL coins. The service is the only entity allowed to do operations on coins, therefore, the operations of minting, dividing or combining coins to form coins of different value must be done via the service. This construction allows for constant time operations in the sense that they do not depend on the number of users. Also, anonymity does not depend on a set of active users.

Table 11 provides a comparison between the four constructions indicating the properties each one maintains. As it can be seen our current construction fulfils our requirements.

Table 11: Comparison of Constructions

	Con. 1	Con. 2	Con. 3	Con. 4 (Current)
Anonymous	✗	✓	✓	✓
Anonymity is not based on a set of currently active users	n.a.	✗	✗	✓
Payment unlinkability	✗	✓	✓	✓
Transacted amount hidden	✓	✓	✓	✓
Service does not maintain state	✗	✗	✗	✓
Order of size of operations given n users	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Not susceptible to sybil attacks	✓	✗	✗	✓

4 Implementation

In this section we provide an overview of the characteristics of our implementation. We explain the technologies used and the rationale behind their choice.

4.1 Implemented Protocols

We implemented a subset of the protocols that we discussed in the design section of this document. Namely, the `Split` protocol, the `Combine` protocol, the `Spend` protocol, and a partial `Deposit` protocol, i.e. a `Deposit` protocol where users do not actually have to pay in some currency in order to receive the corresponding ACLcoin, instead they can use the protocol to generate an arbitrary amount of ACLcoins[8]. The only protocol not implemented is the `Withdraw` protocol, since it is almost the same as the `Spend` protocol and we can use that to get an idea of its performance.

4.2 Cryptographic Framework

Our prototype consists of a client and a server that both have been implemented in Python. Both the client and the server use *petlib* for cryptography, which is an OpenSSL wrapper for Python specifically designed for use in Privacy Enhancing Technologies projects [24]. We use elliptic curve cryptography and chose the NIST/SECG recommended [31] curve *secp224r1* over a 224 bit prime field as a default curve for our implementation[9]. As a basis for our prototype we used an existing implementation of Anonymous Credentials Light written in *petlib* by Danezis available at [25]. We changed large portions of the existing code to match our needs. Specifically, we made the necessary changes needed to network the code, to reduce the amount of communication between the client and the server as presented in table 3, as well as to optimise for execution speed and storage efficiency.

[8] This was only done for testing purposes and we do not consider it part of our final solution.

[9] In the evaluation section we test for a variety of curves and discuss the results.

4.3 Technology Stack

The server is a virtual private server hosted on the cloud by Digital Ocean located in London. It has one CPU (Intel Xeon CPU E5-2630L v2 @ 2.40GHz), 512 MB of RAM, 20GB SSD, running the server edition of 14.04 x64 Ubuntu Linux LTS.

The server is built using the Flask web framework [43]. According to the Flask documentation, Flask's built-in web server is not suitable for production as by default handles one request per time and does not scale well. For that reason we deployed Flask on top of Gunicorn [20] with Nginx as our front-end reverse proxy, which is one of the proposed configurations on deploying Flask for production use[10] [11]. Currently, Nginx proxies all requests to Gunicorn which manages a set of asynchronous workers to avoid blocking that deal with client requests.

Communication between the client and the server takes place using HTTP. The server is designed to work in a RESTful manner to the extent that it is possible, since by design we have a very limited range of features. An alternative approach for communication would have been to use raw sockets, however, we based our choice on extensibility and robustness. Particularly, if we wanted to add SSL/TLS or compression to our implementation we would have to write the code on top of our custom socket implementation. By choosing to use HTTP we immediately solve these problems. Recall that we have made an assumption that all communication channels are private. To achieve this, we must use SSL/TLS, since data should not be visible to anyone, as this could lead to money theft. In addition, by using SSL/TLS man-in-the-middle attacks can be avoided that would otherwise lead to coin compromisation.

In order for the client to send requests to the server, we use the library `requests` [42], which we found very easy to use. In our application the client sends data to the server using POST requests. For this case `requests` is ideal as it provides one-liner HTTP methods. Here we provide an example of a POST request:

```
1 from requests import post
2 r = post('http://IP/acl/split/', data = {'data':'test'})
```

[10] http://flask.pocoo.org/docs/0.11/deploying/wsgi-standalone/

[11] Initially, we had deployed Flask on top of Apache using *mod_wsgi*, which is another proposed configuration for deploying Flask. However, after doing some benchmarks we found this configuration to be quite limited, as it could only handle about 300 HTTP req/s. Our current configuration can handle more than 2500 HTTP req/s.

Extending our code to include SSL/TLS after installing the certificate would merely adjust to changing the client's HTTP request methods from using `http` to `https`.

Both the initial and our modified 2-phase ACL (see table 3) require the issuer to keep state. Particularly, in the validation 2 phase the issuer must fetch the correct c', u to answer to the user. Since HTTP is stateless, we either need the server to keep state internally or pass the necessary data to the client encrypted and signed so that reading and tampering is not possible by the client. We chose the former, i.e. that the issuer stores data in a database and fetches them according to a unique -signed by the server- session id. We set the session id's duration to one minute and made it large enough (128 bytes) so that any attacks trying to find a session id within a minute fail. When a client initialises the first phase, a session id is generated and stored in the database to help indexing. Upon entrance on the second phase, the client sends e, as well as its session id. The server/issuer uses the given session id to load the right c', u values based on which he creates a response which he sends after deleting the contents of the session id, as they are no longer needed. Notice that, since the session id lasts for one minute and due to the fact that it can be deleted by the client, it is not possible for the server to link back the user to a particular issuing when verification takes place at a later time.

4.4 Storage

Both the server and the client use *SQLite* [22] as a database. SQLite was chosen on the basis of being a cross-platform, self-contained, and zero-configuration database. By being cross-platform allows client implementations on a variety of devices, while by being self-contained and zero-configuration makes it ideal for a "plug and play" client implementation requiring no additional libraries and no setup procedures as in other databases. Last but not least remains the fact that SQLite allows concurrent usage by different applications, therefore, allowing each Gunicorn worker to access the database in a non blocking manner.

The maximum integer number SQLite can store is 8 bytes[12], which is 64 bits, and therefore the maximum representation assuming unsigned integers is 2^{64} which is smaller than 2^{224} which is the order of our default group. As a result, we store the order of the group as text (varchar) occupying 1 byte per digit. 2^{224} consists of 68 digits, therefore

[12]https://www.sqlite.org/datatype3.html

68 bytes. A rough estimation can be computed as follows: $2^{224} = 2^{220}2^4 = (2^{10})^{22}2^4 \approx (10^3)^{22}2^4 \approx 10^{66}10 \approx 10^{67}$.

To store our data we first use *msgpack*[13] for packing which outputs binary data that we then encode in base64 to be suitable for storage. Encoding using base64 provides us with an overhead in size of approximately 4/3 of the size of the binary data [1].

4.5 Liveness

So far we have not considered two liveness problems: *timeout* and *crash and reboot*. Timeout refers to the scenario that the service is up, but it does not receive the user's message, or the user does not receive the client's response, or the network has stopped working. Crash and reboot refers to the scenario that the service crashes while executing a protocol, reboots, and has to remember its state to continue the protocol execution. If there is no safety measure, such a scenario could lead to coin loss. We stress that we have not implemented solutions for the crash and reboot scenario, while for the timeout scenario, we are safe to the extent HTTP is.

In order maintain tolerance in crash and reboot situations, a system must know its pre-crash state by storing it in non-volatile memory. State must be saved before the execution of any action, such as sending a reply over the network. Otherwise, i.e. if state is saved after an action, a crash might occur exactly after the execution before saving the correct state. This can lead to undesirable scenarios, e.g. a client might execute a protocol twice. Using the state system explained, in case of a crash the system can check its disk and load its pre-crash state.

4.6 Algorithms

We present what we consider as the most important algorithms in our implementation other than those explained in Design section (section 3) of this document (e.g. `Deposit`, `Split`, `Combine`, `Spend`, and `Withdraw`). Although in the Design section we did not present them in the form of algorithms, we believe the conversion is straight forward, thus we do not present them again here. We assume the public parameters of our scheme $params = (G, g, q, z, h, h_0, h_1)$ to be available to all of the following algorithms.

[13] https://pypi.python.org/pypi/msgpack-python

4.6.1 Proof that a committed number is a bit

Here we present the non-interactive version of the bit proof presented in table 1. The ProveBit algorithm takes as input a bit commitment (among other inputs) and checks that indeed this is the case, otherwise returns $\perp$. The prover selects three random numbers a, s, t, forms two commitments c_a, c_b, and computes the challenge along with the required responses. The prover then sends the output of this function to the verifier who runs the VerifyBit algorithm to verify that indeed he received a bit commitment. Notice that $c^{x-f}c_b$ is a commitment to $B(x-f)+aB = B(1-B)x - Ba + Ba = B(1-B)x$, which is equal to 0 if $B \in \{0,1\}$ [30].

Algorithm 1: ProveBit Proof that a committed number is a bit as presented by Groth and Danezis in [30] and [23] respectively. Run by the user.

Data: a commitment *com*, a bit B, a random number r

Result: Proof that a committed number is a bit

```
1  begin
2      if (B ≠ 0 and B ≠ 1) or com ≠ Com(B,r) then
3          return ⊥
4      a ⟵ random(0, q), s ⟵ random(0, q), t ⟵ random(0, q)
5      c_a ⟵ Com(a, s)
6      c_b ⟵ Com(aB, t)
7      x ⟵ H(g, h, c_a, c_b)  mod q
8      f ⟵ (B · x + a)  mod q
9      z_a ⟵ (r · x + s)  mod q
10     z_b ⟵ (r · (x − f) + t)  mod q
11     return x, f, z_a, z_b
```

Algorithm 2: VERIFYBIT Verification that a committed number is a bit as presented by Groth and Danezis in [30] and [23] respectively. Run by the verifier.

Data: a commitment com, a challenge x, and responses f, z_a, z_b

Result: 1 if the committed number is a bit, $\perp$ otherwise

```
1 begin
2     if x, f, z_a, z_b ∉ [0, q) then
3         return ⊥
4     c_a ⟵ Com(f, z_a) com^{-x}
5     c_b ⟵ Com(0, z_b) com^{-(x-f)}
6     if H(g, h, c_a, c_b) ≠ x then
7         return ⊥
8     return 1
```

4.6.2 Proof that a number lies in a interval

Here we explain the classic method to prove that a number lies in an interval by bit decomposition. We also explained this method in the 'Background' section of this document. We provide here a similar, but more concise explanation of the algorithm.

Given an l-bit number x, we decompose it such that $x = x_0 2^0 + x_1 2^1 + ... + x_{l-1} 2^{l-1} = \sum_{i=0}^{l-1} x_i 2^i$, where $x_i \in \{0, 1\}$. We select a random r_i for each $i \in \{0, ..., l-1\}$ and calculate r, such that $r = \sum_{i=0}^{l-1} r_i 2^i$. Subsequently, we form the commitments $Com(x, r)$, $Com_i(x_i, r_i)$ for each i, and form the zero knowledge proofs π_i that each Com_i is a bit commitment using the PROVEBIT algorithm (algorithm 1). Finally, we send all $Com(x, r)$, $Com_i(x_i, r_i)$, π_i to the verifier, who, using the algorithm 4 VERIFYRANGE, verifies that indeed all Com_i are commitments to 0 or 1, and checks whether $Com(x, r) \stackrel{?}{=} \prod_{i=0}^{l-1} Com(x_i, r_i)^{2^i}$. Finally, we should note that in our prototype implementation we have set l to 32.

Algorithm 3: ProveRange Range Proof $[0, 2^l - 1]$ as explained in [10] and [37].[a]
Run by the user

Data: a positive integer x, a max range l

Result: Proof that a committed number lies in a range

begin
  if $x < 0$ *or* $x > 2^l - 1$ *or type(x)* $\neq$ *type(int)* **then**
    return $\perp$
  xBitList[] $\longleftarrow$ reverse(get$l$BitRepr(x))
  commitmentList[] $\longleftarrow \varepsilon$
  proofList[] $\longleftarrow \varepsilon$
  $r \longleftarrow 0$
  for $i = 0$ *to* $l - 1$ **do**
    $r_i \longleftarrow$ random$(0, q)$
    $r \longleftarrow r + r_i \cdot 2^i$
    $com_i \longleftarrow$ Com(xBitList[i], r_i)
    commitmentList.append(com_i)
    $proof_i \longleftarrow$ ProveBit(com_i, xBitList[i], r_i)
    proofList.append($proof_i$)
  $r \longleftarrow r \mod q$
  com $\longleftarrow$ Com(x,r)
  proofCom $\longleftarrow$ proveOpenningsCom(com, x, r)
  return com, proofCom, r, commitmentList, proofList

[a] This algorithm was also described in the Background section of this document.

Algorithm 4: VERIFYRANGE Verification that a committed number lies in $[0, 2^l - 1]$ as explained in [10] and [37].[a] Run by the verifier

Data: a commitment *com*, a commitment list *comList*, a proof list *proofList*, and a max range l

Result: 1 if the committed number lies in the correct range

```
1  begin
2      if VerifyBit(comList[0], proofList[0]) ≠ 1 then
3          return ⊥
4      productCom ⟵ comList[0]
5      for i = 1 to l − 1 do
6          if VerifyBit(comList[i], proofList[i]) ≠ 1 then
7              return ⊥
8          productCom ⟵ productCom + (comList[i])^(2^i)
9      if productCom ≠ com then
10         return ⊥
11     return 1
```

[a] This algorithm was also described in the Background section of this document.

4.6.3 ACL Verification

Algorithm 5 illustrates the ACL Verification explained in the 'Background' section of this document. Recall that a signature $\sigma = (m, \zeta, \zeta_1, \zeta_2, \rho, \omega, \rho_1{}', \rho_2{}', \omega', \mu)$ verifies if $\zeta \neq 1$ and $\omega + \omega' = \mathcal{H}(\zeta, \zeta_1, g^{\rho} y^{\omega}, g^{\rho_1{}'} \zeta_1{}^{\omega'}, h^{\rho_2{}'} \zeta_2{}^{\omega'}, z^{\mu} \zeta^{\omega'}, m) \mod q$. If verification succeeds, a check is made to see if this signature has already been verified, by calling the CHECKDOUBLESPEND presented as algorithm 6.

Algorithm 5: ACLVerify ACL Verification run by the verifier

Data: σ, a database *db*, the verifier's public key y

Result: 1 if verification succeeded, $\perp$ otherwise

```
1 begin
2     rhs ⟵ H(ζ, ζ_1, g^ρ y^ω, g^ρ1' ζ_1^ω', h^ρ2' ζ_2^ω', z^μ ζ^ω', m)  mod q
3     lhs ⟵ (ω + ω')  mod q
4     if rhs ≠ lhs then
5         return ⊥
6     if checkDoubleSpend(rhs, db) == 1 then
7         return ⊥
8     insert(db, rhs)
9     return 1
```

4.6.4 Double Spending Detection

The checkDoubleSpend algorithm takes as input a verification and a database instance, and checks whether the hash exists in the database.

Algorithm 6: checkDoubleSpend Double Spending Detection, run by the verifier

Data: $coinHash$, and a database *db*

Result: 1 if double spending was detected, $\perp$ otherwise

```
1 begin
2     if coinHash in db then
3         return 1
4     else
5         return ⊥
```

5 Evaluation

5.1 Security Evaluation

In this section we consider the security of our construction as a whole. Discuss potential attacks, as well as methods and techniques to prevent attacks and potential errors.

5.1.1 Service Owned by the Adversary

First of all, we would like to consider an assumption that we have made in our threat model that does not always resemble real life situations and affects our scheme and e-cash schemes in general. That is that the adversary cannot be the service. This is very difficult, if not impossible, to control, for two reasons: first, it is difficult to control the distribution of the code of the service, and, second, if the service is represented by an organisation or a company, none of the people working there should be malicious. Besides the obvious denial of service attacks where the attacker denies users' access to the service's features, we should mention another attack based on the fact that he who knows the private key of the service can mint coin. An attacker, therefore, might not try to steal from others, but instead might mint coin for himself. In addition, as we discussed in section 3.5, a service should not verify another's service coins due to the fact that an adversary might be controlling it and minting coin in an arbitrary fashion.

We should also take some time to explain the scenario where an adversary owns the service and only receives coins and invalidates them, without giving a new coin in return. This is a scenario that we can solve by using a fair exchange protocol [3], [4]. In this case, a fair exchange protocols would ensure that both the service and the user get what they want or they get nothing. To achieve this, fair exchange protocols introduce a trusted third party to resolve conflicts. We have not implemented a fair exchange protocol, however, we identify such an addition as future work.

Finally, we stress that even if an honest service is compromised by an adversary, the service cannot delete a user's balance, since it is maintained locally and not in the service.

5.1.2 Maintaining Bitcoin Privacy

A second important aspect to focus on is that the service should not betray the anonymity of the users. We present a sketch of the implementation of the Bitcoin integration focusing on the privacy aspects. Bitcoin [39] is a decentralised cryptocurrency that does not protect privacy [2], [38], [9], since anyone can examine the blockchain and study the details of the transactions. We show how our implementation can integrate with Bitcoin and make Bitcoin's transactions anonymous and unlinkable by following methods that restrict deanonymization techniques.

Suppose that the service maintains a web service where users can exchange ACLcoins to Bitcoins. To protect privacy, every time a payment is to be done, the service must generate a new Bitcoin address to receive Bitcoins, instead of using a permanent address. The rationale is that a permanent address linked to a service makes it easy for an adversary to see how many transactions occurred with this service, and perhaps calculate an approximation of the received Bitcoin's merely by examining the public transaction ledger. By constantly changing addresses, we do not allow such kind of information to be leaked. Moreover, if Bitcoin users use only one address to transact and have made this address linkable to their identity by posting it online or otherwise, the described technique ensures that an examination of that user's activity in the Bitcoin transaction ledger does not reveal that the user used this service. Therefore, the trick of constantly using different addresses protects the privacy of both the users and the service.

5.1.3 Connection Errors & System State

We utilised a variety of tools and methods to ensure that our implementation remains robust under various circumstances. We used HTTP to transmit data and allow for faster, and perhaps more robust, SSL/TLS extensions. We used the `requests` library that supports SSL/TLS validation, automatic compression and decompression, cookies and sessions, and deals with connection drops and timeouts. Finally, we described, but not implemented, a state system to ensure that a crash does not affect the integrity and robustness of our application

5.2 Theoretical Analysis

5.2.1 Coin & Database Sizes

In order to detect double spending, the server stores the verification hash

$$\mathcal{H}(\zeta, \zeta_1, g^{\rho} y^{\omega}, g^{\rho_1{}'} \zeta_1{}^{\omega'}, h^{\rho_2{}'} \zeta_2{}^{\omega'}, z^{\mu} \zeta^{\omega'}, m) \mod q \tag{5}$$

Since this operation is reduced $\mod q$ it cannot exceed the size of q, which, in the default curve of our implementation, is 68 bytes[14]. Therefore, the database requires $68n$ bytes to store n coins resulting to $\Theta(n)$ space complexity. This result looks promising, since in order to store 1 billion coins our service would require $10^9 \cdot 68$ bytes = 68 GB of storage.

The coin size on the side of the client is larger, since the client has to save the signature $(m, \zeta, \zeta_1, \zeta_2, \rho, \omega, \rho_1{}', \rho_2{}', \omega', \mu)$, as well as the openings R, γ, rnd. Notice from table 2 that $\rho, \omega, \rho_1{}', \rho_2{}', \omega', \mu, R, \gamma, rnd \in \mathbb{Z}_q$, therefore each of them needs at most 68 bytes. In addition, ζ, ζ_1, ζ_2 are points in the curve and require 52 bytes of storage, while in our implementation we leave m empty resulting to $9 \cdot 68 + 3 \cdot 52 = 768$ bytes per coin. These are 768 bytes in binary representation which we convert into base64 leading to a $\approx 4/3$ increase in storage making the size of each coin ≈ 1024 bytes. Therefore, the client requires $\approx 1024n$ bytes to store n coins.

5.2.2 Erroneous double spending

An important question to address is when the first faulty double spend will occur. Double spending detection is done at the service by checking whether the result of equation 5 already exists in the database. In our implementation we use a hash function with 224 output bits, i.e the same as the order of our default group. We stress that due to the birthday attack, a collision can be found after $2^{112} \approx 5 \cdot 10^{33}$ coins have been used. To see how big this number is, we note that 10^{12} corresponds to one trillion and that numbers beyond 2^{80} (including the collision reduction) are considered safe at the time of writing from exhaustive search attacks [35].

[14]Recall from the implementation section that we save each number as text (thus 68 bytes). In addition, since this is double spending detection, the service does not need to pack and encode data, so the size remains 68 bytes and is not increased by 1/3.

5.3 Experimental Analysis

Our tests were conducted from a virtual machine with 2 CPUs (@ 2.0 GHz), 1024 MB of RAM, 10 GB of storage, running the 64 bit version of Linux Debian 8 (Jessie) using a home broadband connection. Timing results were calculated using Python's `time` function.

5.3.1 Size of Transmitted Data

Tables 12, 13 present the number of bytes that are sent between the client and the server at each phase of the `Deposit`, `split`, and `combine` protocols for four different curve choices. Each phase corresponds to a phase in table 3. Since a not very small number of random numbers that range from 0 to q is included in each protocol making the number of the transmitted bytes to fluctuate, we present the average set of values obtained after 500 executions.

Table 14 presents the number of bytes that are sent between the client and the server for the `Spend` protocol. Again, we present the average set of values obtained after 500 executions.

As it can be seen from the tables, the ranking based on the transmitted size has as follows: the `Deposit` protocol requires the least number of bytes, the `Spend` protocol the second least, the `Combine` protocol the third least, while the `Split` protocol requires the most. This is due to the range proof involved in the `Split` protocol.

Table 12: Bytes transmitted for each protocol (except `Spend`) for curves *secp224r1*, *secp256r1*

Curve	*secp224r1*			*secp256r1*		
Protocol	Dep.	Spl.	Comb.	Dep.	Spl.	Comb.
Registration	52	15596	2920	60	18096	3272
Preparation - Validation 1	244	436	244	280	504	280
Validation 2 from client to server	44	88	44	52	100	52
Validation 2 from server to client	216	432	216	244	484	244
Total (bytes)	**556**	**16552**	**3424**	**636**	**19184**	**3848**

Table 13: Bytes transmitted for each protocol (except `Spend`) for curves *secp384r1*, *secp521r1*

Curve	*secp384r1*			*secp521r1*		
Protocol	Dep.	Spl.	Comb.	Dep.	Spl.	Comb.
Registration	80	24280	4380	104	31088	5608
Preparation - Validation 1	388	696	388	508	908	504
Validation 2 from client to server	72	140	72	96	188	96
Validation 2 from server to client	352	700	352	468	932	468
Total (bytes)	**892**	**25816**	**5192**	**1176**	**33116**	**6676**

Table 14: Bytes transmitted for the `Spend` protocol

Protocol	`Spend`			
Curve	*secp224r1*	*secp256r1*	*secp384r1*	*secp521r1*
Total (bytes)	**1312**	**1480**	**2016**	**2608**

5.3.2 Time results

We conducted tests to measure the execution speed of our construction and, also, observe the effect of the choice of the curve on the execution time. Figure 1 depicts our results in a graph form, while table 15 contains the raw values of this graph. The values presented are the average values of 500 executions. As it can be seen, all protocols, except the `Split` protocol, need less than 750ms to complete using any of the four curves we tried. The execution time of the `Split` protocol increases on average by 600ms per curve change. This is due to the range proof, as it involves a large number of computations and variables whose size increases a lot due to the curve choice. In addition, the time it takes for the proof to be transmitted to the server increases as well, since the choice of the curve has an impact on the proof size, as tables 12, 13 demonstrate.

Figure 2 depicts where was the time spent for each protocol using the default curve *secp224r1*. As it can be deduced, most of the time is spent on processing by the server, then client computation, and, finally, communication between the client and the server takes the least time.

Figure 1: Protocol execution time using different curves.

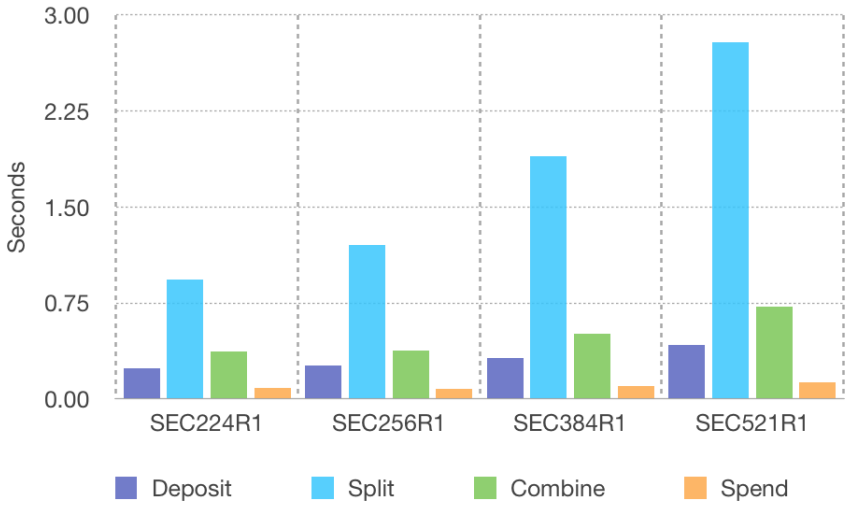


Table 15: Protocol execution time in seconds over different curve choices

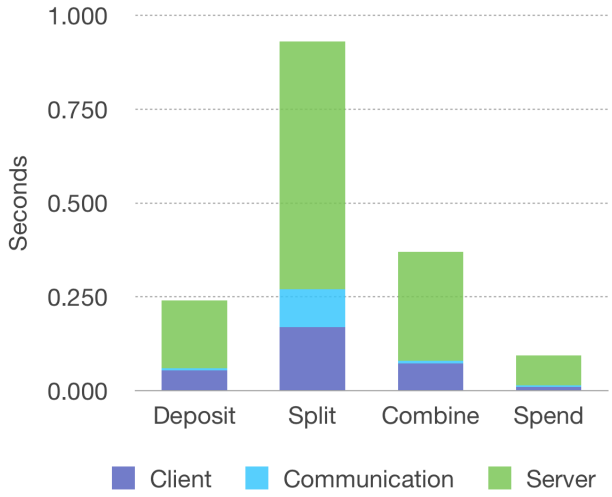
	secp224r1	*secp256r1*	*secp384r1*	*secp521r1*
`Deposit`	0.24s	0.26s	0.32s	0.42s
`Split`	0.93s	1.2s	1.9s	2.79s
`Combine`	0.37s	0.38s	0.51s	0.72s
`Spend`	0.07s	0.08s	0.10s	0.13s

5.3.3 Comparison With Naive Spending Schemes

Recall from the 'Background' section that some schemes, such as the one by Hinterwälder et al. [33], use a coin of unary value and have users do spending in the naive way, i.e. by spending as many unary-valued coins needed until they have reached the required amount. We compare our scheme's choice of having `Split` and `Combine` protocols to the naive spending choice both in terms of time and size and demonstrate that the choice we made is more efficient and better suited to the average case.

By observing the times each protocol requires to execute in table 15 for the default curve (*secp224r1*), we can see that the `Spend` protocol requires about one tenth of the time required for the `Split` protocol, and that together they need 1s. That means that if we had coins of unary value, spending them would only be faster if we had to spend $1/0.07 \approx 14$ coins. Assuming a currency like the euro or the pound, the unary coin should represent 0.01, which is the smallest unit possible. This restriction makes the naive spending method beneficial only if the transacted amount is less than 0.14 cents, as at the same time our method could have created any denomination. In addition, by

Figure 2: Execution time within each protocol for curve *secp224r1*



studying the tables showing the transmitted sizes per protocol (12, 13, 3), we notice that the size of the bytes transmitted in the `Spend` protocol is about 13 times that of the `Spend` protocol. That means that if we had coins of unary value, spending them would require sending fewer amount of bytes only if we had to spend less than 13 coins, which makes the unary spending method beneficial in terms of the transmitted size only if the transacted amount is less than 0.13 cents. Therefore, we stress that the design choice we made to have a `Split` and a `Combine` protocol is in fact better suited for the average case, i.e. the case where the transacted amount exceeds 0.14 cents.

5.3.4 Concurrency

So far all the times we have looked at are generated by a single client who makes requests in a sequential order. We now examine how our construction behaves as we add more clients who perform requests simultaneously. From figure 2 we can observe that the server spent 0.180s on the `Deposit` protocol. Therefore, based on the sequential scenario, if we increase the number of clients, it should be able to deal with a maximum of $1000/180 = 5.5$ requests per second. Table 16 shows the sequential scenario expectancy per protocol and the actual results obtained by running a different number of users with 100 concurrent requests. Extremely high CPU usage measurements on the service while engaging in a protocol indicate that the service is CPU bounded. This is as expected, since our service consists of 1 CPU.

As it can be deduced from table 16, as the number of users increases the req/s served

by the service slightly decrease. This is due to the fact that traffic increases and the server spends more time dealing with the incoming requests.

Table 16: Concurrent times per protocol for curve *secp224r1*

	`Deposit`	`Split`	`Combine`	`Spend`
sequential req/s	1000/180=5.5	1000/660=1.5	1000/290=3.5	1000/52=19.2
req/s for 2 users with 100 reqs	4.30	1.34	3.05	20.10
req/s for 5 users with 100 reqs	4.21	1.42	2.83	38.50
req/s for 10 users with 100 reqs	3.96	1.36	2.51	37.03

5.4 Comparison with Hinterwälder et al. [33]

In table 17 we present the computational cost[15] of our $\mathcal{OP}$ construction and compare it to the ACL e-cash scheme by Hinterwälder et al. in [33], which is the closest construction to ours. Our scheme needs more exponentiations in all of the protocols, due to the fact that we do the registration phase of the ACL protocol for every coin in the protocols `Deposit`, `Split`, `Combine`, while in [33] they do the registration only once for all coins.

Table 17: Comparison of our scheme and the one by Hinterwälder et al. in [33] in terms of numbers of exponentiations per protocol. (E = number of exponentiations)

	H. et al.	**Our scheme**		**H. et al.**	**Our scheme**
Deposit			**Spend**		
User	0E	16E	User	10E	16E
Service	1E	10E	Service	19E	14E
Split			**Withdrawal**		
User	n/a	604E	User	12E	14E
Service	n/a	460E	Service	6E	12E
Combine			**Account Openning**		
User	n/a	46E	User	4E	n/a
Service	n/a	34E	Service	3E	n/a

5.5 Unit tests

We implemented a number of unit tests to ensure that the various parts of the system work as expected and were described in the rest of this document. These tests can be executed upon startup of the program, or can be executed manually by the user.

[15]The coin size consists of about the same items, however they use the 160 bit curve *secp160r1*, so their end results are smaller.

Cryptographic Primitives and petlib Tests

Tests to ensure the correctness of the petlib library were devised in addition to the ones already shipped with the library in order to verify that the cryptographic primitives function as expected.

Proof and Verification Tests

Proof and verification tests ensure that the output of the implemented proof mechanisms verifies if entered to the corresponding verification mechanism. This is very useful as it ensures that the proof was built correctly and, probably, the server will not return an error. Specifically, this check ensures that the range proof, bit proof, knowledge of secrets in a commitment proof, `Split` proof, `Combine` proof, `Deposit` proof, as well as `Spend` proof work as expected.

Database Tests & Space Tests

Database tests ensure that the database has not been altered by doing a hash based integrity check, as well as that saving and loading operations can occur without causing any issues and that the user has the right permissions to read and write to the database. In addition, prior to execution a check is made that there is enough space for writing.

Integration Tests

Integration tests ensure that the sub-components of the system work properly when working together. Integration tests in our scenario need to be executed manually by the user and execute a full protocol with the service for a dummy value in order to ensure that both the service and the client work properly. This is not a local test, in contrast with the 'proof and verification tests', and results are saved in the database. For instance, the user can engage in a `Split` protocol, to split a coin worth of 0$ to two coins worth 0$ and 0$. This gives him back two coins and convinces him that both the client and the service are operational without risking losing coins of some value.

6 Conclusion & Future Work

This work introduced a new scheme called *Oblivious Payment*, a scheme between a service, a user, and a merchant that shares many properties of e-cash schemes, however, it differentiates itself by allowing users to maintain their wallets locally instead of being stored in the service. Our scheme is based on "Anonymous Credentials Light" [5]. As the evaluation results demonstrate, all protocols, except the `Split` protocol, have execution times that are acceptable. The protocol that we assume to be used the most, i.e. the `Spend` protocol, has an execution time of 70ms and requires 1KB of transfer. Space complexity is also within acceptable bounds, specifically for storage the server requires 68 bytes per coin, while the client 1KB.

6.1 Future work

Finally, we identify improvements that must be made in order to make the scheme presented more efficient, complete and robust.

Range Proofs We used the range proof method by bit decomposition which is very expensive and it is the reason that our `Split` protocol takes so long compared to the rest protocols. In order to make our `Split` protocol more efficient, we need to replace bit decomposition by a more efficient range proof, such as a signature based proof.

Definition and Proof of Security In the scheme we presented, we identified informally security properties that must hold in the system and described them. However, we did not define them, neither we proved their security. We believe that this is a very important aspect to work on, as a proof of security would be a direct way to convince the cryptographic community about the security of our scheme.

Coin Expiration Date as an Attribute We cannot expect to save coins for ever, since this would require a great amount of storage. The solution we use in our construction is to change the service's public key per specific time intervals, thus, invalidating all coins that use the previous key by having users engage in a procedure where they send their coins and in exchange they get a new one valid under the new public key. However, this method is harsh and may result to coin loss, as it requires all users being active. A better alternative would be to encode an expiration year as an attribute within the coin.

Fair Exchange The addition of a fair exchange protocol would allow users to use the service's features even if it was controlled by an adversary. Particularly, a fair exchange protocol would guarantee that the users always get back a valid credential.

Access Control We could extend our scheme to include other types of attributes besides the value of a coin, e.g. we could include a person's age or citizenship. Then a merchant could allow a user to buy something only if he proves that he is over 18 or that he is citizen of a specific country.

References

[1] Gisle Aas. *Perl Base64 Documentation*. 2010. URL: http://perldoc.perl.org/MIME/Base64.html (visited on 07/03/2016).

[2] Elli Androulaki et al. "Evaluating user privacy in bitcoin". In: *Financial Cryptography and Data Security*. Springer, 2013, pp. 34–51.

[3] N. Asokan, Matthias Schunter, and Michael Waidner. "Optimistic Protocols for Fair Exchange". In: *Proceedings of the 4th ACM Conference on Computer and Communications Security*. CCS '97. Zurich, Switzerland: ACM, 1997, pp. 7–17. ISBN: 0-89791-912-2. DOI: 10.1145/266420.266426. URL: http://doi.acm.org/10.1145/266420.266426.

[4] Nadarajah Asokan, Victor Shoup, and Michael Waidner. "Optimistic fair exchange of digital signatures". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 1998, pp. 591–606.

[5] Foteini Baldimtsi and Anna Lysyanskaya. "Anonymous credentials light". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 1087–1098.

[6] Foteini Baldimtsi et al. "Anonymous transferable e-cash". In: *IACR International Workshop on Public Key Cryptography*. Springer. 2015, pp. 101–124.

[7] Mihir Bellare and Oded Goldreich. "On Defining Proofs of Knowledge". In: *Advances in Cryptology — CRYPTO' 92: 12th Annual International Cryptology Conference Santa Barbara, California, USA August 16–20, 1992 Proceedings*. Ed. by Ernest F. Brickell. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 390–420. ISBN: 978-3-540-48071-6. DOI: 10.1007/3-540-48071-4_28. URL: http://dx.doi.org/10.1007/3-540-48071-4_28.

[8] Josh Benaloh. "Dense probabilistic encryption". In: *Proceedings of the workshop on selected areas of cryptography*. 1994, pp. 120–128.

[9] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. "Deanonymisation of clients in Bitcoin P2P network". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 15–29.

[10] Fabrice Boudot. "Efficient proofs that a committed number lies in an interval". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2000, pp. 431–444.

[11] Jan Camenisch, Rafik Chaabouni, et al. “Efficient protocols for set membership and range proofs”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2008, pp. 234–252.

[12] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. “Compact e-cash”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2005, pp. 302–321.

[13] Jan Camenisch and Anna Lysyanskaya. “An efficient system for non-transferable anonymous credentials with optional anonymity revocation”. In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2001, pp. 93–118.

[14] Jan Camenisch and Markus Stadler. “Efficient group signature schemes for large groups”. In: *Annual International Cryptology Conference*. Springer. 1997, pp. 410–424.

[15] Sébastien Canard and Aline Gouget. “Divisible e-cash systems can be truly anonymous”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2007, pp. 482–497.

[16] Sébastien Canard, Aline Gouget, and Emeline Hufschmitt. “A handy multi-coupon system”. In: *International Conference on Applied Cryptography and Network Security*. Springer. 2006, pp. 66–81.

[17] David Chaum. “Blind Signatures for Untraceable Payments”. In: *Advances in Cryptology: Proceedings of CRYPTO ’82*. Plenum, 1982, pp. 199–203.

[18] David Chaum. “Blind signatures for untraceable payments”. In: *Advances in cryptology*. Springer. 1983, pp. 199–203.

[19] David Chaum. “Blind signature system”. In: *Advances in cryptology*. Springer. 1984, pp. 153–153.

[20] Benoit Chesneau. *Gunicorn Documentation*. 2016. URL: `http://docs.gunicorn.org/en/stable/` (visited on 07/03/2016).

[21] Benny Chor et al. “Private information retrieval”. In: *Journal of the ACM (JACM)* 45.6 (1998), pp. 965–981.

[22] SQLite Community. *sqlite*. 2016. URL: `https://sqlite.org/`.

[23] George Danezis. 2015. URL: `https://github.com/gdanezis/petlib/blob/master/examples/GK15ringsig.py#L41` (visited on 07/03/2016).

[24] George Danezis. 2015. URL: `https://conspicuouschatter.wordpress.com/2015/07/01/introducing-the-petlib-library/` (visited on 07/03/2016).

[25] George Danezis. 2015. URL: https://github.com/gdanezis/petlib/blob/master/examples/BLcred.py (visited on 07/03/2016).

[26] John R Douceur. "The sybil attack". In: *Peer-to-peer Systems*. Springer, 2002, pp. 251–260.

[27] Shimon Even, Oded Goldreich, and Abraham Lempel. "A randomized protocol for signing contracts". In: *Communications of the ACM* 28.6 (1985), pp. 637–647.

[28] Amos Fiat and Adi Shamir. "How to prove yourself: Practical solutions to identification and signature problems". In: *Conference on the Theory and Application of Cryptographic Techniques*. Springer. 1986, pp. 186–194.

[29] Jens Groth. "Non-interactive zero-knowledge arguments for voting". In: *International Conference on Applied Cryptography and Network Security*. Springer. 2005, pp. 467–482.

[30] Jens Groth and Markulf Kohlweiss. "One-out-of-many proofs: Or how to leak a secret and spend a coin". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2015, pp. 253–280.

[31] Network Working Group. *Elliptic Curve Cryptography Subject Public Key Information*. 2009. URL: https://www.ietf.org/rfc/rfc5480.txt.

[32] Gesine Hinterwälder, Felix Riek, and Christof Paar. "Efficient E-cash with Attributes on MULTOS Smartcards". In: *International Workshop on Radio Frequency Identification: Security and Privacy Issues*. Springer. 2015, pp. 141–155.

[33] Gesine Hinterwälder et al. "Efficient e-cash in practice: Nfc-based payments for public transportation systems". In: *International Symposium on Privacy Enhancing Technologies Symposium*. Springer. 2013, pp. 40–59.

[34] Malika Izabachene and Benoît Libert. "Divisible e-cash in the standard model". In: *International Conference on Pairing-Based Cryptography*. Springer. 2012, pp. 314–332.

[35] J. Katz and Y. Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC Cryptography and Network Security Series. Taylor & Francis, 2014. ISBN: 9781466570269. URL: https://books.google.co.uk/books?id=OWZYBQAAQBAJ.

[36] Helger Lipmaa. "Statistical Zero-Knowledge Proofs from Diophantine Equations." In: *IACR Cryptology ePrint Archive* 2001 (2001), p. 86.

[37] Wenbo Mao. "Guaranteed correct sharing of integer factorization with off-line shareholders". In: *International Workshop on Public Key Cryptography*. Springer. 1998, pp. 60–71.

[38] Sarah Meiklejohn et al. "A fistful of bitcoins: characterizing payments among men with no names". In: *Proceedings of the 2013 conference on Internet measurement conference*. ACM. 2013, pp. 127–140.

[39] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2008.

[40] Tatsuaki Okamoto. "An efficient divisible electronic cash scheme". In: *Annual International Cryptology Conference*. Springer. 1995, pp. 438–451.

[41] Tatsuaki Okamoto and Kazuo Ohta. "Universal electronic cash". In: *Annual International Cryptology Conference*. Springer. 1991, pp. 324–337.

[42] Kenneth Reitz. *Requests: HTTP for Humans*. 2016. URL: `http://docs.python-requests.org/en/master/` (visited on 07/03/2016).

[43] Armin Ronacher. *Flask Documentation*. 2015. URL: `http://flask.pocoo.org/` (visited on 07/03/2016).

[44] Andrei Serjantov. *On the anonymity of anonymity systems*. Tech. rep. UCAM-CL-TR-604. University of Cambridge, Computer Laboratory, Oct. 2004. URL: `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-604.pdf`.