Coconut: Threshold Issuance Selective Disclosure Credentials with Applications to Distributed Ledgers

Alberto Sonnino
University College London

Mustafa Al-Bassam
University College London

Shehar Bano
University College London

George Danezis
University College London
The Alan Turing Institute

Abstract

We present Coconut, a novel selective disclosure credential scheme supporting distributed threshold issuance, public and private attributes, re-randomization, and multiple unlinkable selective attribute revelations. Coconut can be used by modern blockchains to ensure confidentiality, authenticity and availability even when a subset of credential issuing authorities are malicious or offline. We implement and evaluate a generic Coconut smart contract library for Chainspace and Ethereum; and present three applications related to anonymous payments, electronic petitions, and distribution of proxies for censorship resistance. Coconut uses short and computationally efficient credentials, and our evaluation shows that most Coconut cryptographic primitives take just a few milliseconds on average, with verification taking the longest time (10 milliseconds).

1 Introduction

Selective disclosure credentials [15, 17] allow the issuance of a credential to a user, and the subsequent unlinkable revelation (or 'showing') of some of the attributes it encodes to a verifier for the purposes of authentication, authorization or to implement electronic cash. However, established schemes have shortcomings. Some entrust a single issuer with the credential signature key, allowing a malicious issuer to forge any credential or electronic coin. Other schemes do not provide the necessary re-randomization or blind issuing properties necessary to implement modern selective disclosure credentials. No existing scheme provides all of threshold distributed issuance, private attributes, re-randomization, and unlinkable multi-show selective disclosure.

The lack of full-featured selective disclosure credentials impacts platforms that support 'smart contracts', such as Ethereum [40], Hyperledger [14] and Chainspace [3]. They all share the limitation that verifiable smart contracts may only perform operations recorded on a public blockchain. Moreover, the security models of these systems generally assume that integrity should hold in the presence of a threshold number of dishonest or faulty nodes (Byzantine fault tolerance); it is desirable for similar assumptions to hold for multiple credential issuers (threshold aggregability).

Issuing credentials through smart contracts would be very desirable: a smart contract could conditionally issue user credentials depending on the state of the blockchain, or attest some claim about a user operating through the contract—such as their identity, attributes, or even the balance of their wallet. This is not possible, with current selective credential schemes that would either entrust a single party as an issuer, or would not provide appropriate re-randomization, blind issuance and selective disclosure capabilities (as in the case of threshold signatures [5]). For example, the Hyperledger system supports CL credentials [15] through a trusted third party issuer, illustrating their usefulness, but also their fragility against the issuer becoming malicious.

Coconut addresses this challenge, and allows a subset of decentralized mutually distrustful authorities to jointly issue credentials, on public or private attributes. Those credentials cannot be forged by users, or any small subset of potentially corrupt authorities. Credentials can be re-randomized before selected attributes being shown to a verifier, protecting privacy even in the case all authorities and verifiers collude. The Coconut scheme is based on a threshold issuance signature scheme, that allows partial claims to be aggregated into a single credential. Mapped to the context of permissioned and semi-permissioned blockchains, Coconut allows collections of authorities in charge of maintaining a blockchain, or a side chain [5] based on a federated peg, to jointly issue selective disclosure credentials.

Coconut uses short and computationally efficient credentials, and efficient revelation of selected attributes and verification protocols. Each partial credentials and the

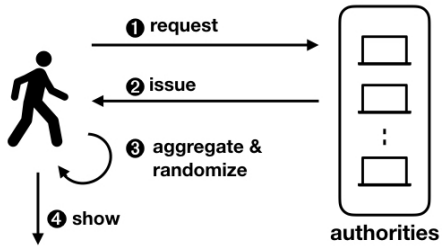

Figure 1: Overview of the Coconut general architecture.

consolidated credential is composed of exactly two group elements. The size of the credential remains constant, and the attribute showing and verification are $O(1)$ in terms of both cryptographic computations and communication of cryptographic material—irrespective of the number of attributes or authorities/issuers. Our evaluation of the Coconut primitives shows very promising results. Verification takes about 10ms, while signing an attribute is 15 times faster. The latency is about 600 ms when the client aggregates partial credentials from 10 authorities distributed across the world.

This paper makes three key contributions:

- We describe the signature schemes underlying Coconut, including how key generation, distributed issuance, aggregation and verification of signatures operate (Sections 2 and 3). The scheme is an extension and hybrid of the Waters signature scheme [39], the BGLS signature [8], and the signature scheme of Pointcheval *et al.* [31]. This is the first fully distributed threshold issuance, re-randomizable, multi-show credential scheme we are aware of.
- We use Coconut to implement a generic smart contract library for Chainspace [3] and one for Ethereum [40], performing public and private attribute issuing, aggregation, randomization and selective disclosure (Section 4). We evaluate their performance, and cost within those platforms (Section 6).
- We design three applications using the Coconut contract library: a coin tumbler providing payment anonymity; a privacy preserving electronic petitions; and a proxy distribution system for a censorship resistance system (Section 5). We implement and evaluate the two former ones on the Chainspace platform, and provide a security and performance evaluation (Section 6).

2 Overview of Coconut

Coconut is a fully featured selective disclosure credential system, supporting threshold credential issuance of public and private attributes, re-randomization of credentials to support multiple unlikable revelations, and the ability to selectively disclose a subset of attributes. It is embedded into a smart contract library, that can be called from other contracts to issue credentials.

The Coconut architecture is illustrated in Figure 1. Any Coconut user may send a Coconut *request* command to a set of Coconut signing authorities; this command specifies a set of public or encrypted private attributes to be certified into the credential (❶). Then, each authority answers with an *issue* command delivering a partial credentials (❷). Any user can collect a threshold number of shares, aggregate them to form a consolidated credential, and re-randomize it (❸). The use of the credential for authentication is however restricted to a user who knows the private attributes embedded in the credential—such as a private key. The user who owns the credentials can then execute the *show* protocol to selectively disclose attributes or statements about them (❹). The showing protocol is publicly verifiable, and may be publicly recorded. Coconut has the following design goals:

- **Threshold authorities:** Only a subset of the authorities is required to issue partial credentials in order to allow the users to generate a consolidated credential [7]. The communication complexity of the *request* and *issue* protocol is thus $O(t)$, where t is the size of the subset of authorities.
- **Non-interactivity:** The authorities may operate independently of each other, following a simple key distribution and setup phase to agree on public security and cryptographic parameters—they do not need to synchronize or further coordinate their activities.
- **Blindness:** The authorities issue the credential without learning any additional information about the private attributes included in the credential [17].
- **Unlinkability:** It is impossible to link multiple showing of the credentials with each other, or the issuing transcript, even if all the authorities collude [30].
- **Efficiency:** The credentials and all zero-knowledge proofs involved in the protocols are short and computationally efficient. After aggregation and re-randomization, the attribute showing and verification only involve a single consolidated credential, and are therefore $O(1)$ in terms of both cryptographic computations and communication of cryptographic material—no matter the number of authorities or the number of attributes embedded in the credentials.

As a result, a large number of authorities may be used to

issue credentials, without significantly affecting the efficient operation of other operations.

3 Cryptographic Constructions

We introduce the cryptograhic primitives supporting the Coconut architecture, step by step from a simple construction to the full scheme. The exact definitions of these primitives can be found in Appendix A.

Step 1: We first present (Section 3.2) a toy signature scheme on a single public attribute m, called the *Coconut signature scheme*; i.e., users can request a partial signature σ_i from each authority on a public (clear text) attribute m, and then aggregate them all into a single consolidated credential σ. This scheme sets the foundations for the subsequent steps.

Step 2: We extend the signature scheme (Section 3.3) in order to support private attributes; the resulting scheme is called the *Coconut anonymous credentials scheme*. In a nutshell, it allows a user to request a partial credential from each authority on a private (encrypted) attribute; each authority can issue a partial credential without seeing the actual content of m (such schemes are also called *blind* signatures).

Step 3: We introduce (Section 3.4) the *Coconut threshold credentials scheme*, which has all the properties of the schemes above, but allows the user to collect partial credentials from only a subset of the authorities to form a credential. If we have a set of n authorities, the user only needs to collect $t \leq n$ partial credentials in order to aggregate them into a consolidated credential.

Step 4: Finally, we extend (Section 3.5) our schemes to support public or private issuance on q distinct attributes $(m_0, \ldots, m_{q-1})$ at the same time.

Design Goals. In addition to the design goals in Section 2, the Coconut threshold credentials scheme has the following properties:

- **Short signatures:** Each partial credential—as well as the consolidated credential—is composed of exactly two group elements.
- **Constant signature size:** The size of the credential is constant in the number of attributes or with the number of authorities.
- **Unforgeability:** An adversary who is given credentials for a few attributes of their choice is not able to produce a credential for a new attribute (more formally *unforgeability under chosen-message attack*) [10].
- **Robustness:** It is impossible to generate a consolidated credential from fewer than t partial credentials [7].

3.1 Notations and Assumptions

We present the notation used in the rest of the paper, as well as the security assumptions on which our primitives rely.

Zero-knowledge proofs. Coconut uses non-interactive zero-knowledge proofs to assert knowledge and relations over discrete logarithm values. We represent these zero-knowledge proofs with the notation introduced by Camenisch *et al.* [16]:

$$\text{NIZK}\{(x, y, \ldots) : \text{statements about x, y, } \ldots\},$$

which denotes proving in zero-knowledge that the secret values $(x, y, \ldots)$ (all other values are public) satisfy the statements after the colon.

Security assumptions. Our credential scheme requires groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ of prime order p with a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ and satisfying the following properties: (*i*) *Bilinearity* means that for all $g_1 \in \mathbb{G}_1$, $g_2 \in \mathbb{G}_2$ and $(x, y) \in \mathbb{F}_p^2$, $\quad e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$; (*ii*) *Non-degeneracy* means that for all $g_1 \in \mathbb{G}_1$, $g_2 \in \mathbb{G}_2$, $e(g_1, g_2) \neq 1$; (*iii*) *Efficiency* implies the map e is efficiently computable; (*iv*) furthermore, $\mathbb{G}_1 \neq \mathbb{G}_2$, and there is no efficient homomorphism between $\mathbb{G}_1$ and $\mathbb{G}_2$. Those type 3 parings are efficient [20]. They support the XDH assumption which implies the difficulty of the Computational co-Diffie-Hellman (co-CDH) problem in $\mathbb{G}_1$ and $\mathbb{G}_2$, and the difficulty of the Decisional Diffie-Hellman (DDH) problem in $\mathbb{G}_1$ [9].

Coconut also relies on a cryptographically secure hash function H, hashing an element of group $\mathbb{G}_1$ into another element of $\mathbb{G}_1$, namely $H : \mathbb{G}_1 \to \mathbb{G}_1$. Boneh *et al.* [9] provide a detailed description of such functions.

3.2 The Coconut Signature Scheme

We introduce the *Coconut signature scheme* allowing users to obtain a partial credential σ_i on a single public attribute m from each of the n authorities; and then aggregate them all into a consolidated credential σ.

Scheme definition. The Coconut signature scheme works in a bilinear map group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ of type 3, with a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ as described in Section 3.1. The secret key is a pair $(x, y) \in \mathbb{F}_p^2$ and the verification key is the triplet $(g_2, g_2^x, g_2^y) \in \mathbb{G}_2^3$.

❖ **Setup(1^λ):** Choose a bilinear group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ with order p, where p is an λ-bit prime number. Let g_1 be a generator of $\mathbb{G}_1$, and g_2 generator of $\mathbb{G}_2$. The system parameters are $params = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2)$.

❖ **KeyGen($params$):** Choose a random secret key $sk = (x, y) \in \mathbb{F}_p^2$. Parse $params = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2)$, and publish the verification key $vk = (g_2, \alpha, \beta) = (g_2, g_2^x, g_2^y)$ along with the proof π_k showing knowledge of the secret key:

$$\pi_k = \text{NIZK}\{(x, y) : \alpha = g_2^x \wedge \beta = g_2^y\}$$

❖ **Sign($params, sk, m$):** Parse $sk = (x, y)$ and $params = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2)$. Compute $h = H(g_1^m)$ and $\sigma = (h, h^{x+m \cdot y})$. Output σ.

❖ **AggregateSign($\sigma_0, \ldots, \sigma_{n-1}$):** Parse each $\sigma_i = (h, \varepsilon_i)$ for $i \in [0, \ldots, n-1]$; compute $\sigma = (h, \prod_{i=0}^{n-1} \varepsilon_i)$. Output σ.

❖ **AggregateKey($vk_0, \ldots, vk_{n-1}$):** Parse each $vk_i = (g_2, \alpha_i, \beta_i)$ for $i \in [0, \ldots, n-1]$; compute $vk = (g_2, \prod_{i=0}^{n,-1} \alpha_i, \prod_{i=0}^{n-1} \beta_i)$. Output vk.

❖ **Verify(vk, m, σ):** Parse $vk = (g_2, \alpha, \beta)$ and $\sigma = (h, \varepsilon)$. Accept if $h \neq 1$ and $e(h, \alpha\beta^m) = e(\varepsilon, g_2)$.

Correctness and explanation. The Setup algorithm generates the public parameters. Credentials are elements of $\mathbb{G}_1$, while public keys are elements of $\mathbb{G}_2$. On key generation, the proof π_k asserts knowledge of the secret key and prevents attacks from corrupted authorities [9]. In order to obtain a credential on a public attribute $m \in \mathbb{F}_p$, the user submits the same attribute m to all the n signing authorities. Each signer i generates a partial signature $\sigma_i = h^{x_i + m y_i}$, where $h = H(g_1^m)$, and (x_i, y_i) is its private key. Then, the user aggregates the n signatures to compute σ as shown below:

$$\sigma = \prod_{i=0}^{n-1} \sigma_i = \prod_{i=0}^{n-1} h^{x_i} \left(\prod_{i=0}^{n-1} h^{y_i} \right)^m = h^{x+my}$$

where

$$x = \sum_{i=0}^{n-1} x_i \quad \text{and} \quad y = \sum_{i=0}^{n-1} y_i$$

Note that every authority must operate on the same element h. Intuitively, generating h from $h = H(g_1^m)$ is equivalent to computing $h = g_1^r$ where $r \in \mathbb{F}_p$ is unknown by both users and authorities. However, since h is deterministic, every authority can uniquely derive it in isolation and forgeries are prevented since different m_0 and m_1 cannot lead to the same value of h.[1]

[1] If an adversary $\mathscr{A}$ can obtain two credentials σ_0 and σ_1 on respectively $m_0 = 0$ and $m_1 = 1$ with the same value h as follows: $\sigma_0 = h^x$ and $\sigma_1 = h^{x+y}$; then $\mathscr{A}$ could forge a new credential σ_2 on $m_2 = 2$: $\sigma_2 = (\sigma_0)^{-1} \sigma_1 \sigma_1 = h^{x+2y}$.

To verify the credentials, a verifier collects and aggregates the verification keys of every authority to obtain an aggregated verification key vk:

$$vk = (g_2, \prod_{i=0}^{n-1} g_2^{x_i}, \prod_{i=0}^{n-1} g_2^{y_i}) = (g_2, g_2^x, g_2^y)$$

Upon reception of the tuple (m, σ), the verifier checks the pairing of the signature using the aggregated key. Since h is an element of $\mathbb{G}_1$, we can express it as $h = g_1^{r'} \mid r' \in \mathbb{F}_p$. The left-hand side of the pairing verification can be expanded as:

$$e(h, \alpha\beta^m) = e(h, g_2^{x+my}) = e(g_1, g_2)^{r'(x+my)}$$

and the right-hand side:

$$e(\varepsilon, g_2) = e(h^{x+my}, g_2) = e(g_1, g_2)^{r'(x+my)}$$

Therefore, $e(h, \alpha\beta^m) = e(\varepsilon, g_2)$, from where the correctness of the Verify algorithm follows.

Theorem 1. *If the co-CDH assumption holds in $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$, then the Coconut signature scheme is existentially unforgeable under chosen-message attack (EUF-CMA) [22].*

A sketch of the proof of Theorem 1 can be found in Appendix B.1.

3.3 The Coconut Anonymous Credentials Scheme

The *Coconut anonymous credentials scheme* extends the previous scheme to support issuance on private attributes.

Scheme definition. This scheme is defined by the same set of algorithms presented in Section 3.2 but by replacing the Sign and Verify algorithms by two protocols, PrepareBlindSign(m) $\leftrightarrow$ BlindSign(sk, c_m, c, π_s) and ShowBlindSign(vk, m, σ') $\leftrightarrow$ BlindVerify($\kappa, \nu, \sigma', \pi_v$). The BlindSetup algorithm slightly modifies the Setup algorithm to provide an additional generator $h_1 \in \mathbb{G}_1$, and the scheme is extended with two additional algorithms: Unblind and Randomize.

❖ **BlindSetup(1^λ):** Call Setup(1^λ), let h_1 be an other generator of $\mathbb{G}_1$; the system parameters are $params = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, h_1, g_2)$.

❖ **PrepareBlindSign(m):** Pick a random $o \in \mathbb{F}_p$. Build the commitment c_m and hash value h as follows:

$$c_m = g_1^m h_1^o \quad \text{and} \quad h = H(c_m)$$

Generate an El-Gamal key-pair $(d, \gamma = g_1^d)$ and pick a random $k \in \mathbb{F}_p$; then compute an El-Gamal encryption of m as below:

$$c = Enc(h^m) = (g_1^k, \gamma^k h^m)$$

Output (c_m, c, π_s), where π_s is defined by:

$$\pi_s = \text{NIZK}\{(m,k,o) : \gamma = g_1^d \wedge c_m = g_1^m h_1^o \wedge c = (g_1^k, \gamma^k h^m)\}$$

❖ **BlindSign(sk, c_m, c, π_s):** Parse $sk = (x,y)$ and $c = (a,b)$. Recompute $h = \text{H}(c_m)$. Verify the proof π_s. If the proof is valid, build $\tilde{c} = (a^y, h^x b^y)$ and output $\tilde{\sigma} = (h, \tilde{c})$; otherwise output $\perp$.

❖ **Unblind($\tilde{\sigma}, d$):** Parse $\tilde{\sigma} = (h, \tilde{c})$ and $\tilde{c} = (\tilde{a}, \tilde{b})$; compute $\sigma = (h, \tilde{b}(\tilde{a})^{-d})$. Output σ.

❖ **Randomize(σ):** Parse $\sigma = (h, \varepsilon)$. Pick a random $t \in \mathbb{F}_p$, set $\sigma' = (h^t, \varepsilon^t)$. Output σ'.

❖ **ShowBlindSign(vk, m, σ'):** Parse $vk = (g_2, \alpha, \beta)$ and $\sigma' = (h', \varepsilon')$. Pick a random $r \in \mathbb{F}_q$, build $\kappa = \alpha\beta^m g_2^r$ and $\nu = (h')^r$. Output $(\kappa, \nu, \sigma', \pi_v)$, where π_v is:

$$\pi_v = \text{NIZK}\{(m,r) : \kappa = \alpha\beta^m g_2^r \wedge \nu = \left(h'\right)^r\}$$

❖ **BlindVerify(κ, ν, σ', π_v):** Parse $\sigma' = (h', \varepsilon')$. Accept if the proof π_v verifies, $h' \neq 1$ and $e(h', \kappa) = e(\varepsilon' \nu, g_2)$

Correctness and explanation. Figure 2 illustrates the Coconut anonymous credentials protocol. To keep an attribute $m \in \mathbb{F}_p$ hidden from the authorities, the user and the authorities first execute the PrepareBlindSign(m) $\leftrightarrow$ BlindSign(sk, c_m, c, π_s) protocol. The user generates a Pedersen commitment $c_m = g_1^m h_1^o$ on the attribute m with randomness $o \in \mathbb{F}_p$, $(g_1, h_1) \in \mathbb{G}_1^2$, and $h = H(c_m)$. Then, the user creates an El-Gamal keypair $(d, \gamma = g_1^d)$ and computes the encryption of h^m as:

$$c = Enc(h^m) = (a,b) = (g_1^k, \gamma^k h^m),$$

where $k \in \mathbb{F}_p$. Finally, the user sends c and c_m to the signer, as well as a zero-knowledge proof π_s ensuring knowledge of m, and correctness of the encryption c and of the commitment c_m (❶).

To blindly sign the attribute, each authority i verifies π_s and uses the homomorphic properties of El-Gamal to generate an encryption $\tilde{c}$ of $h^{x_i}(h^m)^{y_i}$ as:

$$\tilde{c} = (a^y, h^{x_i} b^{y_i}) = (g_1^{ky_i}, \gamma^{ky_i} h^{x_i + my_i})$$

Upon reception of $\tilde{c}$, the users decrypt it using their private El-Gamal key d to recover the partial credentials

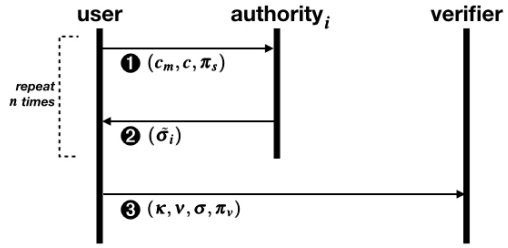

Figure 2: Coconut anonymous credential protocol.

$\sigma_i = (h, h^{x_i + my_i})$ (❷). Then, the users can call the AggregateSign algorithm described in Section 3.2 to aggregate all the partial credentials.

Verification is implemented by the Randomize algorithm and the ShowBlindSign(vk, m, σ') $\leftrightarrow$ BlindVerify($\kappa, \nu, \sigma', \pi_v$) protocol. First, the user randomizes the signature through Randomize; then, the user computes $\kappa = \alpha\beta^m g_2^r$ from the aggregated verification key, $\nu = h^r$, and sends $(\kappa, \nu, \sigma, \pi_v)$ to the verifier where π_v is a zero-knowledge proof asserting the correctness of κ and ν (❸). The proof π_v ensures that the user actually knows m and that κ has been built using the correct verification keys and blinding factors. The pairing verification is similar to the previous section; express $h = g_1^{r'} \mid r' \in \mathbb{F}_p$, the left-hand side of the pairing verification can be expanded as:

$$e(h, \kappa) = e(h, g_2^{(x+my)r}) = e(g_1, g_2)^{(x+my)rr'}$$

and the right-hand side:

$$e(\varepsilon\nu, g_2) = e(h^{(x+my)r}, g_2) = e(g_1, g_2)^{(x+my)rr'}$$

From where the correctness of BlindVerify follows.

Theorem 2. *If the co-CDH assumption holds in $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$, then the Coconut anonymous credential scheme is existentially unforgeable under chosen-message attack; and if the DDH problem holds in $\mathbb{G}_1$, the scheme ensures the properties of unlinkability [30] and blindness [17].*

A sketch of the proof of Theorem 2 can be found in Appendix B.2.

3.4 The Coconut Threshold Credentials Scheme

In the previous constructions, the users collect exactly n partial credentials—one from each authority to obtain a consolidated credential. A t-out-of-n threshold credentials scheme offers more flexibility as the users needs to collect only $t \leq n$ of these partial credentials in order to recompute the consolidated credential (both t and n are

scheme parameters). This section presents the *the Coconut Threshold Credentials Scheme*—an extension of our previous schemes applying the techniques developed by Boldyreva [7].

Scheme definition. The Coconut threshold credentials scheme is similar to the previous ones except for the key generation and the aggregation algorithms. For the sake of simplicity, we describe below a key generation algorithm TTPKeyGen as executed by a trusted third party; this protocol can however be execute in a distributed way as illustrated by Gennaro *et al.* [21]. Adding and removing authorities implies to re-run the key generation algorithm—this limitation is inherited from the underlying Shamir's secret sharing protocol [35].

❖ **TTPKeyGen(**$params, t, n$**):** Choose two polynomials v, w of degree $t-1$ and set $(x,y) = (v(0), w(0))$. Issue to each signer i a secret key $sk_i = (x_i, y_i) = (v(i), w(i))$ (for each $i \in [1,\ldots,n]$), and publish their public key $vk_i = (g_2, \alpha_i, \beta_i) = (g_2, g_2^{x_i}, g_2^{y_i})$.

❖ **AggregateThSign(**$\sigma_1, \ldots, \sigma_t$**):** Parse each σ_i as (h, ε_i) for $i \in [1,\ldots,t]$. Output $(h, \prod_{i=1}^{t} \varepsilon_i^{l_i})$, where:

$$l_i = \left[\prod_{i=1, j\neq i}^{t} (0-j)\right]\left[\prod_{i=1, j\neq i}^{t} (i-j)\right]^{-1} \bmod p$$

Correctness and explanation. In this scheme, the AggregateSign algorithm is replaced by AggregateThSign algorithm using the Lagrange basis polynomial l which allows to reconstruct the original $v(0)$ and $w(0)$ through polynomial interpolation;

$$v(0) = \sum_{i=1}^{t} v(i) l_i \quad \text{and} \quad w(0) = \sum_{i=1}^{t} w(i) l_i$$

One can easily verify the correctness of AggregateThSign as below.

$$\begin{aligned} h^{x+my} &= h^{v(0)+mw(0)} = \prod_{i=1}^{t} h^{(x_i l_i)} \prod_{i=1}^{t} h^{m(y_i l_i)} \\ &= \prod_{i=1}^{t} (h^{x_i})^{l_i} \prod_{i=1}^{t} (h^{my_i})^{l_i} = \prod_{i=1}^{t} \left(h^{x_i + my_i}\right)^{l_i} \end{aligned}$$

Theorem 3. *If the co-CDH assumption holds in* $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ *and the signers produced less than t signatures, then the Coconut threshold credentials scheme is robust and existentially unforgeable under chosen-message attack.*

Robustness ensures the scheme maintains all the properties described in Section 2 as long as at least t out of n authorities are honest. Intuitively, this can be observed from the fact that t partial credentials are required to recover a signature σ. The formal security definition and security proof is analog to the threshold extension of the BGLS Signature [8] presented by Boldyreva [7].

3.5 Multi-Attributes Signatures

We expand the previous scheme to sign multiple attributes; this generalization follows directly from the Waters signature scheme [39]. The authorities key pair becomes:

$$sk = (x, y_0, \ldots, y_{q-1}) \quad \text{and} \quad vk = (g_2, g_2^{x}, g_2^{y_0}, \ldots, g_2^{y_{q-1}})$$

where q is the number of attributes. The multi-attribute credential is then generalized to

$$\sigma = (h, h^{x + \sum_{j=0}^{q-1} m_j y_j})$$

It is noticeable that the credentials size does not increase with the number of attributes, and is two group elements. The security proof of the multi-attribute scheme relies on a reduction against the single-attribute scheme and is analog to [31]. Moreover, it is also possible to combine public and private attributes to keep only a subset of the attributes hidden from the authorities, while revealing some others; the BlindSign algorithm only verifies the proof π_s on the private attributes.

4 Implementation

We implement a Python library containing the cryptographic primitives described in Section 3 and publish the code on GitHub as an open-source project [2]. We also implement a smart contract library on Chainspace to enable other application-specific smart contracts (see Section 5) to conveniently use our cryptographic primitives. We present the design and implementation of the Coconut smart contract library in Section 4.1. In addition, we implement and evaluate some of the functionality of the Coconut smart contract library in Ethereum (Section 4.2). Finally, we show how to integrate Coconut into existing semi-permissioned blockchains.

4.1 The Coconut Smart Contract Library

We implement the Coconut smart contract in Chainspace[3] (which can be used by other application-specific smart contracts) as a library to issue and verify randomizable threshold credentials through cross-contract calls. The contract has four functions, (Create,

[2] https://github.com/asonnino/coconut

[3] https://github.com/asonnino/coconut/tree/master/coconut-chainspace

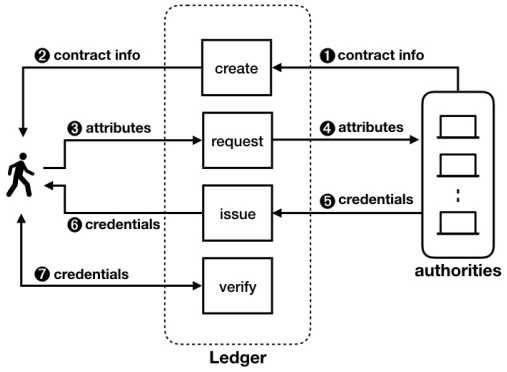

Figure 3: The Coconut smart contract library.

Request, Issue, Verify), and is illustrated in Figure 3. First, a set of authorities call the Create function to initialize a Coconut instance defining the *contract info*; i.e., their verification key, the number of authorities and the threshold parameter (❶). The initiator smart contract can specify a callback contract that needs to be executed by the user in order to request credentials; e.g., this callback can be used for authentication. The instance is public and can be read by the user (❷); any user can request a credential through the Request function by executing the specified callback contract, and providing the public and private *attributes* to include in the credentials (❸). The public attributes are simply a list of clear text strings, while the private attributes are encrypted as described in Section 3.3. Each signing authority monitors the blockchain at all times, looking for credential requests. If the request appears on the blockchain (i.e., a transaction is executed), it means that the callback has been correctly executed (❹); each authority issues a partial *credential* on the specified attributes by calling the Issue procedure (❺). In our implementation, all partial credentials are in the blockchain; however, these can also be provided to the user off-chain. Users collect a threshold number of partial credentials, and aggregate them to form a full credential (❻). Then, the users locally randomize the credential. The last function of the Coconut library contract is Verify that allows the blockchain—and anyone else—to check the validity of a given credential (❼).

A limitation of this architecture is that it is not efficient for the authorities to continuously monitor the blockchain. Section 4.3 explains how to overcome this limitation by embedding the authorities into the nodes running the blockchain.

4.2 Ethereum Smart Contract Library

To make Coconut more widely available, we also implement it in Ethereum—a popular permissionless smart contract blockchain [40]. The Coconut Ethereum smart contract library is written in Solidity, a high-level JavaScript-like language that compiles down to Ethereum Virtual Machine (EVM) assembly code, and we released it as open source library[4]. Ethereum recently hardcoded a pre-compiled smart contract in the EVM for performing pairing checks and elliptic curve operations on the alt_bn128 curve [13, 33], for efficient verification of zkSNARKs. The execution of an Ethereum smart contract has an associated 'gas cost', a fee that is paid to miners for executing a transaction. Gas cost is calculated based on the operations executed by the contract; i.e., the more operations, the higher the gas cost. The pre-compiled contracts have lower gas costs than equivalent native Ethereum smart contracts.

We use the pre-compiled contract for performing a pairing check, in order to implement Coconut verification within a smart contract. The Ethereum code only implements elliptic curve addition and scalar multiplication on $\mathbb{G}_1$, whereas Coconut requires operations on $\mathbb{G}_2$ to verify credentials. Therefore, we implement an elliptic curve addition and scalar multiplication on $\mathbb{G}_2$ as an Ethereum smart contract library[5] written in Solidity. This is a practical solution for many Coconut applications, as verifying credentials with one revealed attribute only requires one addition and one scalar multiplication. It would not be practical however to verify credentials with attributes that will not be revealed, as this requires three $\mathbb{G}_2$ multiplications using our elliptic curve implementation, which would exceed the current Ethereum block gas limit (8M as of February 2018).

We can however use the Ethereum contract to design a federated peg for side chains, or a coin tumbler as an Ethereum smart contract, based on credentials that reveal one attribute. We go on to describe and implement this tumbler using the Coconut Chainspace library in Section 5.1, however the design for the Ethereum version differs slightly to avoid the use of attributes that will not be revealed, which we describe in Appendix C.

The library shares the same functions as the Chainspace library described in Section 4.1, except for Request and Issue, as these simply act as a communication channel between users and authorities, so users can directly communicate with authorities off the blockchain to request tokens, thus saving significant gas costs that would be incurred by storing Request and Issue events on the blockchain. The Verify function simply verifies tokens against Coconut instances created by the Create function.

[4] https://github.com/asonnino/coconut/tree/master/coconut-ethereum

[5] https://github.com/musalbas/solidity-BN256G2

4.3 Deeper Blockchain Integration

The designs of Section 4.1 and Section 4.2 rely on authorities on-the-side for issuing credentials. We present designs that incorporate Coconut authorities within the infrastructure of a number of semi-permissioned blockchains. This enables the issuance of credentials as a side effect of the normal system operations, taking no additional dependency on extra authorities. It remains an open problem how to embed Coconut into unpermissioned systems, based on proof of work or stake. Those systems have a highly dynamic set of nodes maintaining the state of their blockchains, which cannot readily be mapped into stable authorities.

Hyperledger Fabric [14], a permissioned blockchain platform, can incorporate Coconut straightforwardly. Fabric contracts run on private sets of compute nodes—and use the Fabric protocols for cross contract calls. In this setting, Coconut issuing authorities can coincide with the fabric smart contract authorities. Upon a contract setup they perform the setup and key distribution, and then issue partial credentials when authorized by the contract. For issuing Coconut credentials, the only secret maintained are the private issuing keys; all other operations of the contract can be logged and publicly verified. Coconut has obvious advantages over using traditional CL credentials relying on a single authority—as present in the Hyperledger roadmap[6]. The threshold trust assumption—namely that integrity is guaranteed under the corruption of a subset of authorities is preserved, and prevents forgeries by a single corrupted node.

We can also naturally embed Coconut into sharded scalable blockchains, as exemplified by Chainspace [3], which supports general smart contracts, and Omniledger [25], that supports digital tokens. In those transactions are distributed and executed on 'shards' of authorities, whose membership and public keys are known. Coconut authorities can naturally coincide with the nodes within a shard, and a special transaction type, in Omniledger, or a special object, in Chainspace can signal to them that issuing a credential is authorized. Then the authorities, would issue the partial signature necessary to reconstruct the Coconut credential, and attach it to the transaction they are processing anyway. Users, can aggregate, re-randomize and show the credential.

5 Applications

In this section, we present three applications—a coin tumbler (Section 5.1), a privacy-preserving petition system (Section 5.2), and a system for censorship-resistant distribution of proxies (Section 5.3)—that leverage Coconut to offer improved security and privacy properties. For generality, the applications assume authorities external to the blockchain, but these can also be embedded into the blockchain as described in Section 4.3.

[6] http://nick-fabric.readthedocs.io/en/latest/idemix.html

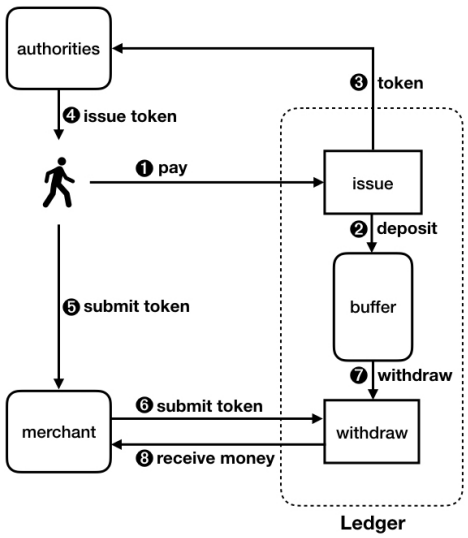

Figure 4: The coin tumbler application.

5.1 Coin Tumbler

We implement a coin tumbler (or mixer) on Chainspace as depicted in Figure 4. Coin tumbling is a method to mix cryptocurrency associated with an address visible in a public ledger with other addresses, to "clean" the coins and obscure the trail back to the coins' original source address. In relation to previous similar schemes [6, 11, 23, 28, 29, 34, 38] that are either centralized (i.e. there is a central authority that operates the tumbler, which may go offline) or decentralized (i.e. there is no central authority, but users must coordinate with each other), the Coconut tumbler is distributed, in that its security relies on a set of multiple authorities that are collectively trusted to contain at least t honest ones.

The tumbler uses Coconut to instantiate a pegged sidechain [5], providing stronger value transfer anonymity than the original cryptocurrency platform, through unlinkability between issuing a credential representing an e-coin [18], and spending it. The tumbler application is based on the Coconut contract library and an application specific smart contract called 'tumbler'.

A set of authorities jointly create an instance of the Coconut smart contract as described in Section 4.1 and specify the smart contract handling the coins of the underlying blockchain as callback. Specifically, the callback requires a coin transfer to a buffer account. Then, users execute the callback and *pay* v coins to the buffer to

ask a signature on the public attribute v, and on two private attributes: the user's private key k and a randomly generated sequence number s (❶). Note that to prevent tracing traffic analysis, v should be limited to a specific set of possible values (similar to cash denominations). The request is accepted by the blockchain only if the user *deposited* v coins to the buffer account (❷).

Each authority monitors the blockchain and detects the *request* (❸); and issues a partial *credential* to the user (either on chain or off-chain) (❹). The user aggregates all partial credentials into a consolidated credential, re-randomizes it, and *submits* it as money token to a merchant. First, the user produces a zk-proof of knowledge of its private key by binding the proof to the merchant's address $addr$; then, the user provides the merchant with the proof along with the sequence number s and the consolidated credential (❺). The coins can only be spent with knowledge of the associated sequence number and by the owner of $addr$. To accept the above as payment, the merchant *submits* the token by showing the signature and the sequence number to the tumbler contract (❻). To prevent double spending, the tumbler contract keeps a record of all the sequence numbers that have already been shown. Upon showing a fresh (unspent) sequence number s, the contract verifies that the signature checks and that s doesn't already appear in the spent list. Then, it *withdraws* v coins from the buffer (❼), sends them to be *received* by the merchant account determined by $addr$, and adds s to the spent list (❽). For the sake of simplicity, we keep the transfer value v in clear-text (treated as a public attribute), but this could be easily hidden by integrating a range proof; this can be efficiently implemented using the technique developed by Bünz *et al.* [12].

Security considerations. Coconut provides blind issuance which allows the user to obtain a signature on the sequence number s without the authorities learning its value. Without blindness, any authority seeing the user key k could potentially out-speed the user and the merchant, and spend it—blindness prevents authorities from stealing the token. Furthermore, Coconut provides unlinkability between the *pay* phase (❶) and the *submit* phase (❺) (see Figure 4), and prevents any authority or third parties from keeping track of the user's transactions. As a result, a merchant can receive payments for good/services offered, yet not identify the purchasers. Finally, this application prevents a single authority from creating coins to steal all the money in the buffer. The threshold property of Coconut implies that the adversary needs to corrupt at least t authorities for this attack to be possible. This property also prevents a single authority blocking the issuance of a token—the service is guaranteed to be available as long as at least t authorities are running.

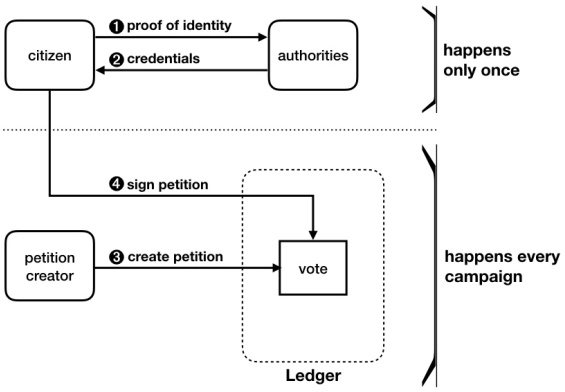

Figure 5: The petition application.

5.2 Privacy-preserving petition

This application extends the work of Diaz *et al.* [19]. We consider the scenario where a city C wishes to issue some long term credentials to its citizens to enable any third party to organize a privacy-preserving petition. All citizens of C are allowed to participate, but should remain anonymous and unlinkable across petitions. This application is based on the Coconut library contract and a simple smart contract called "petition". There are three types of parties: a set of signing authorities representing C, a peition initiator, and the citizens of C. The signing authorities create an instance of the Coconut smart contract as described in Section 4.1. As shown in Figure 5, the citizen provides a *proof of identity* to the authorities (❶). The authorities check the citizen's identity, and issue a blind and long-term signature on her private key k. This signature, which the citizen needs to obtain only once, acts as her long term *credential* to sign any petition petition (❷).

Any third party can *create a petition* by creating a new instance of the petition contract and become the "owner" of the petition. The petition instance specifies an identifier s unique to the petition, and the verification key of the authorities issuing the credentials, as well as any application specific parameters (e.g., the options and current votes) (❸). In order to *sign* a petition, the citizens compute a value ζ as follows:

$$\zeta = (H(s))^k$$

Then they adapt the zero-knowledge proof of the ShowBlindSign algorithm of Section 3.3 to show that ζ is built from the same attribute k in the credential; the petition contract checks the proof π and the credentials, and checks that the signature is fresh by verifying that ζ is not part of a spent list. If all the checks pass, it adds the citizens' signatures to a list of records and adds ζ to the spent list to prevents a citizen from signing the same petition multiple times (prevent double spending) (❹). Also,

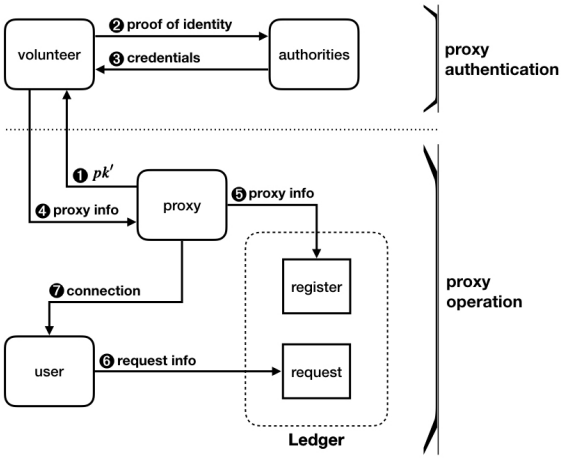

Figure 6: The censorship-resistant proxy distribution system.

the proof π ensures that ζ has been built from a signed private key k; this means that the users correctly executed the callback to prove that they are citizens of C.

Security consideration. Coconut's blindness property prevents the authorities from learning the citizen's secret key, and misusing it to sign petitions on behalf of the citizen. Another benefit is that it lets citizens sign petitions anonymously; citizens only have to go through the issuance phase once, and can then re-use credentials multiple times while staying anonymous and unlinkable across petitions. Coconut allows for distributed credentials issuance, removing a central authority and preventing a single entity from creating arbitrary credentials to sign petitions multiple times.

5.3 Censorship-resistant distribution of proxies

Proxies can be used to bypass censorship, but often become the target of censorship themselves. We present a system based on Coconut for censorship-resistant distribution of proxies (CRS). In our CRS, the volunteer V runs proxies, and is known to the Coconut authorities through its long-term public key. The authorities establish reputability of volunteers (identified by their public keys) through an out of band mechanism. The user U wants to find proxy IP addresses belonging to reputable volunteers, but volunteers want to hide their identity. As shown in Figure 6, V gets an ephemeral public key pk' from the proxy (❶), provides *proof of identity* to the authorities (❷), and gets a *credential* on three private attributes: the proxy IP address, pk', and the time period δ for which it is valid (❸). V shares the credential with the concerned proxy (❹), which creates the *proxy info* including pk', δ, and the credential; the proxy 'registers' itself by appending this information to the blockchain along with a zero-knowledge proof and the material necessary to verify the validity of the credential (❺).

The users U monitor the blockchain for proxy registrations. When a registration is found, U indicates the intent to use a proxy by publishing to the blockchain a *request info* which looks as follows: User IP address encrypted under pk' inside the registration blockchain entry (❻). The proxy continuously monitors the blockchain, and upon finding a user request addressed to itself, connects to U and presents proof of knowledge of the private key associated with pk' (❼). U verifies the proof, the proxy IP address and its validity period, and then starts relaying its traffic through the proxy.

Security consideration. A common limitation of censorship resistance schemes is assuming volunteers that are inherently resistant to coercion: either (*i*) the volunteer is a large, commercial organisation (e.g., Amazon or Google) over which the censor cannot exert its influence; and/or (*ii*) the volunteer is located outside the country of censorship. However, both these assumptions were proven wrong [36, 37]. The proposed CRS overcomes this limitation by offering coercion-resistance to volunteers from censor-controlled users and authorities. Due to Coconut's blindness property, the volunteer can get a credential on their IP address and ephemeral public key without revealing those to the authorities. The users get a proxy IP address run by the volunteer, while being unable to link it to the volunteer's long-term public key. Moreover, the authorities operate independently and can be controlled by different entities, and are resilient against a threshold number of authorities being dishonest or taken down.

6 Evaluation

We present the evaluation of the Coconut threshold credentials scheme; first we present a benchmark of the cryptographic primitives described in Section 3 and then we evaluate the smart contracts described in Section 5.

6.1 Cryptographic Primitives

We implement the primitives described in Section 3 in Python using petlib [1] and bplib [2]. The bilinear pairing is defined over the Barreto-Naehrig [24] curve, using OpenSSL as the arithmetic backend.

Timing benchmark. Table 1 shows the mean (μ) and standard deviation ($\sqrt{\sigma^2}$) of the execution of each procedure described in section Section 3. Each entry is the result of 10,000 runs measured on an Octa-core Dell

Operation	μ [ms]	$\sqrt{\sigma^2}$ [ms]
Keygen	2.392	$\pm$ 0.006
Sign	0.445	$\pm$ 0.001
AggregateSign	0.004	$\pm$ 0.000
AggregateKeys	0.017	$\pm$ 0.000
Randomize	0.545	$\pm$ 0.002
Verify	6.714	$\pm$ 0.005
PrepareBlindSign	2.633	$\pm$ 0.003
BlindSign	3.356	$\pm$ 0.002
ShowBlindSign	1.388	$\pm$ 0.001
BlindVerify	10.497	$\pm$ 0.002
AggregateThSign	0.454	$\pm$ 0.000

Table 1: Execution times for the cryptographic primitives described in Section 3. Measured over 10,000 runs.

Number of authorities: n, Signature size: 132 bytes **Transaction**	**complexity**	**size [B]**
Signature on public attribute:		
❶ request credential	$O(n)$	32
❷ issue credential	$O(n)$	132
❸ verify credential	$O(1)$	162
Signature on private attribute:		
❶ request credential	$O(n)$	516
❷ issue credential	$O(n)$	132
❸ verify credential	$O(1)$	355

Table 2: Communication complexity and transaction size for the Coconut credentials scheme when signing one public and one private attribute (see Figure 2 of Section 3).

desktop computer, 3.6GHz Intel Xeon. Signing is much faster than verifying signatures—due to the pairing operation in the latter; verification takes about 10ms; signing a public attribute is 15 times faster; and signing a private attribute is about 3 times faster.

Communication complexity and packets size. Table 2 shows the communication complexity and the size of each exchange involved in the Coconut credentials scheme, as presented in Figure 2. The communication complexity is expressed as a function of the number of signing authorities (n), and the size of each attribute is limited to 32 bytes as the output of the SHA-2 hash function. The size of a signature is 132 bytes. The highest transaction type is a requests for a signature on a private attribute; this is due to the proofs π_s and π_v (see Section 3). The proof π_s is approximately 318 bytes and π_v is 157 bytes.

Client-perceived latency. We evaluate the client-perceived latency for Coconut threshold credentials scheme for authorities deployed on Amazon AWS [4] when issuing partial credentials on one public and one private attribute. The client requests a partial credential from 10 authorities, and latency is defined as the time it waits to receive t-out-of-10 partial signatures. Figure 7 presents measured latency for a threshold parameter t ranging from 1–10. The dots correspond to the average latency and the error-bars represent the normalized standard deviation, computed over 100 runs. The client is located in London while the 10 authorities are geographically distributed across the world; US East (Ohio), US West (N. California), Asia Pacific (Mumbai), Asia Pacific (Singapore), Asia Pacific (Sydney), Asia Pacific (Tokyo), Canada (Central), EU (Frankfürt), EU (London), and South America (São Paulo). All machines are running a fresh 64-bit Ubuntu distribution, the client runs on a *large* AWS instance and the authorities run on *nano* instances.

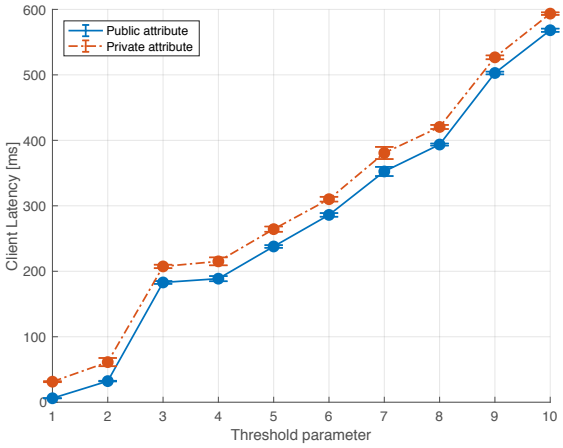

Figure 7: Client-perceived latency for Coconut threshold credentials scheme with geographically distributed authorities.

As expected, we observe that the further the authorities are from the client, the higher the latency due to higher response times; the first authorities to respond are always those situated in Europe, while Sidney and Tokyo are the latest. Latency grows linearly, with the exception of a large jump (of about 150 ms) when t increases from 2 to 3—this is due to the 7 remaining authorities being located outside Europe. The latency overhead between credential requests on public and private attributes remains constant.

6.2 Chainspace Implementation

We evaluate the Coconut smart contract library implemented in Chainspace, as well as the the coin tumbler (Section 5.1) and the privacy-preserving e-petition (Section 5.2) applications that use this library. As expected, Table 3 shows that the most time consuming procedures are the checker of Create and the checker of Verify; i.e., they call the BlindVerify primitives which takes about 10

Coconut smart contract library			
Operation	μ **[ms]**	$\sqrt{\sigma^2}$ **[ms]**	**size [kB]**
Create [g]	0.195	$\pm$ 0.065	$\sim$ 1.38
Create [c]	12.099	$\pm$ 0.471	-
Request [g]	7.094	$\pm$ 0.641	$\sim$ 3.77
Request [c]	6.605	$\pm$ 0.559	-
Issue [g]	4.382	$\pm$ 0.654	$\sim$ 3.08
Issue [c]	0.024	$\pm$ 0.001	-
Verify [g]	5.545	$\pm$ 0.859	$\sim$ 1.76
Verify [c]	10.814	$\pm$ 1.160	-

Table 3: Timing and transaction size of the Chainspace implementation of the Coconut smart contract library described in Section 4.1, measured over 10,000 runs. The notation [g] denotes the execution the procedure and [c] denotes the execution of the checker.

Coin tumbler			
Operation	μ **[ms]**	$\sqrt{\sigma^2}$ **[ms]**	**size [kB]**
InitTumbler [g]	0.235	$\pm$ 0.065	$\sim$ 1.38
InitTumbler [c]	19.359	$\pm$ 0.773	-
Pay [g]	11.939	$\pm$ 0.792	$\sim$ 4.28
Pay [c]	6.625	$\pm$ 0.559	-
Redeem [g]	0.132	$\pm$ 0.012	$\sim$ 3.08
Redeem [c]	11.742	$\pm$ 0.757	-

Table 4: Timing and transaction size of the Chainspace implementation of the coin tumbler smart contract (described in Section 5.1), measured over 10,000 runs; the contract has been tested with two authorities, and one public and one hidden attribute. The notation [g] denotes the execution the procedure and [c] denotes the execution of the checker.

ms (see Table 1).

Similarly, the most time consuming procedure of the coin tumbler (Table 4) application and of the privacy-preserving e-petition (Table 5) are the checker of InitTumbler and the checker of SignPetition, respectively; these two checkers call the BlindVerify primitive involving pairing checks. The Pay procedure of the coin tumbler presents the highest transaction size as it is composed of two distinct transactions: a coin transfer transaction and a Request transaction from the Coconut contract library. However, they are all practical, i.e., they all run in a few milliseconds.

6.3 Ethereum Implementation

We evaluate the Coconut Ethereum smart contract library described in Section 4.2 using the Go implementation of Ethereum on an Intel Core i5 laptop with 12GB of RAM running Ubuntu 17.10. Table 6 shows the execution times and gas costs for different procedures in the smart contract. The execution times for Create and Verify are higher than the execution times for the Chainspace version (Table 3) of the library, due to the different implementations. The arithmetic underlying Coconut in Chainspace is performed through Python naively binding to C libraries, while in Ethereum arithmetic is defined in solidity and executed by the EVM.

Privacy-preserving e-petition			
Operation	μ **[ms]**	$\sqrt{\sigma^2}$ **[ms]**	**size [kB]**
InitPetition [g]	3.260	$\pm$ 0.209	$\sim$ 1.50
InitPetition [c]	3.677	$\pm$ 0.126	-
SignPetition [g]	7.999	$\pm$ 0.467	$\sim$ 3.16
SignPetition [c]	15.801	$\pm$ 0.537	-

Table 5: Timing and transaction size of the Chainspace implementation of the privacy-preserving e-petition smart contract (described in Section 5.2), measured over 10,000 runs. The notation [g] denotes the execution the procedure and [c] denotes the execution of the checker.

Coconut Ethereum smart contract library			
Operation	μ **[ms]**	$\sqrt{\sigma^2}$ **[ms]**	**gas**
Create	27.45	$\pm$ 3.054	$\sim 23,000$
Verify	120.17	$\pm$ 25.133	$\sim 2,150,000$

Table 6: Timing and gas cost of the Ethereum implementation of the Coconut smart contract library (described in Section 4.2). Measured over 100 runs, for one public attribute.

We also observe that the Verify function has a significantly higher gas cost than Create. This is mostly due the implementation of elliptic curve multiplication as a native Ethereum smart contract—the elliptic curve multiplication alone costs around $\sim 1,700,000$ gas, accounting for the vast majority of the gas cost, whereas the pairing operation using the pre-compiled contract costs only 260,000 gas. The actual fiat USD costs corresponding to those gas costs, fluctuate wildly depending on the price of Ether—Ethereum's internal value token—the load on the network, and how long the user wants to wait for the transaction to be mined into a block. As of February 7th 2018, for a transaction to be confirmed within 6 minutes, the transaction fee for Verify is \$1.74, whereas within 45 seconds, the transaction fee is \$43.5.[7]

The bottleneck of our Ethereum implementation is the high-level arithmetic in $\mathbb{G}_2$. However, Ethereum provides a pre-compiled contract for arithmetic operations in $\mathbb{G}_1$. We could re-write our cryptographic primitives by swapping all the operations in $\mathbb{G}_1$ and $\mathbb{G}_2$, at the cost of relying on the SXDH assumption [32] (which is stronger than the XDH assumption that we are currently using).

[7] https://ethgasstation.info/

Scheme	Blindness	Unlinkable	Aggregable	Threshold	Signature Size
[39] Waters Signature	✗	✗	○	✗	2 Elements
[26] LOSSW Signature	✗	✗	◒	✗	2 Elements
[8] BGLS Signature	✗	✗	●	✓	1 Element
[15] CL Signature	✓	✓	○	✗	$O(q)$ Elements
[31] Pointcheval *et al.*	✓	✓	◒	✗	2 Elements
[Section 3] Coconut	✓	✓	●	✓	2 Elements

Table 7: Comparison of Coconut with other relevant cryptographic constructions. The aggregability of the signature scheme reads as follows; ○: not aggregable, ◒: sequentially aggregable, ●: aggregable. The signature size is measured in terms of the number of group elements it is made of, and q indicates the number of signed messages.

7 Comparison with Related Works

We compare the Coconut cryptographic constructions and system with related work in Table 7, along the dimensions of key properties offered by Coconut—blindness, unlinkability, aggregability (i.e., whether multiple authorities are involved in issuing the credential), threshold aggregation (i.e., whether a credential can be aggregated using signatures issued by a subset of authorities), and signature size (see Sections 2 and 3).

The Waters signature scheme [39] provides the bone structure of our primitive, and introduces a clever solution to aggregate multiple attributes into short signatures. However, the original Water's signature does not allow blind issuance or unlinkability, and is not aggregable since it has not been built for use in a multi-authority setting. Lu *et al.* scheme, commonly known as LOSSW signature scheme [26], is also based on Water's scheme and comes with the improvement of being sequentially aggregable. In a sequential aggregate signature scheme, the aggregate signature is built in turns by each signing authority; this requires the authorities to communicate with each other resulting in increased latency and cost.

The BGLS signature [8] scheme is built upon the BLS signature and is remarkable because of its short signature size—signatures are composed of only one group element. The BGLS scheme has a number of desirable properties as it is aggregable without needing coordination between the signing authorities, and can be extended to work in a threshold setting [7]. Moreover, Boneh *et al.* show how to build verifiably encrypted signatures [8] which is close to our requirements, but not suitable for anonymous credentials. The CL Signature scheme [15] provides the most well-known building blocks for anonymous credentials protocols. It provides blind issuance and unlikability through randomization; but signatures are not short since their size grows linearly with the number of signed attributes, and are not aggregable. Pointcheval *et al.* [31] present a construction which is the missing piece of the BGLS signature scheme; it achieves blindness by allowing signatures on committed values and unlinkability through signature randomization. However, it only supports sequential aggregation and does not provide threshold aggregation.

We extend these previous works by presenting a short, aggregable, and randomizable signature scheme; allowing threshold and blind issuance, and a multi-authority anonymous credentials scheme. Our primitive does not require sequential aggregation, that is the aggregate operation does not have to be performed by each signer in turn. Any independent party can aggregate any threshold number of partial signatures into a single aggregate credential, and verify its validity.

As a final remark, for anonymous credentials in a setting where the signing authorities are also verifiers (i.e., without public verifiability), Chasse *et al.* [17] develop an efficient protocol. Its 'GGM' varient has a similar structure to Coconut, but forgoes the pairing operation by using a message authentication code (MACs).

8 Conclusion

Existing selective credential disclosure schemes may be useful, but do not provide the full set of desired properties, particularly when it comes to issuing fully functional selective disclosure credentials without sacrificing desirable distributed trust assumptions. In this paper, we presented Coconut—a novel scheme that supports distributed threshold issuance, public and private attributes, re-randomization, and multiple unlinkable selective attribute revelations. We provided an overview of the Coconut system, and the cryptographic primitives underlying Coconut; an implementation and evaluation of Coconut as a smart contract library in Chainspace and Ethereum, a sharded and a permissionless blockchain respectively; and three diverse and important application to anonymous payments, petitions and censorship resistance. The Coconut fills an important gap in the literature and enables selective disclosure credentials—an important privacy enhancing technology—to be embedded into modern transparent computation platforms.

Acknowledgements. George Danezis, Shehar Bano and Alberto Sonnino are supported in part by EPSRC Grant EP/N028104/1 and the EU H2020 DECODE project under grant agreement number 732546. Mustafa Al-Bassam is supported by a scholarship from The Alan Turing Institute. We extend our thanks to Jonathan Bootle, Andrea Cerulli, and Natalie Eskinazi for helpful suggestions on early manuscripts.

References

[1] petlib. "https://github.com/gdanezis/petlib", 2017 (version July 20, 2017).

[2] bplib. "https://github.com/gdanezis/bplib", 2017 (version October 12, 2017).

[3] AL-BASSAM, M., SONNINO, A., BANO, S., HRYCYSZYN, D., AND DANEZIS, G. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778* (2017).

[4] AMAZON WEB SERVICES, I. Aws whitepapers. "https://aws.amazon.com/whitepapers/", 2017 (version April, 2017).

[5] BACK, A., CORALLO, M., DASHJR, L., FRIEDENBACH, M., MAXWELL, G., MILLER, A., POELSTRA, A., TIMÓN, J., AND WUILLE, P. Enabling blockchain innovations with pegged sidechains. *http://www.opensciencereview.com/papers/123/enablingblockchain-innovations-with-pegged-sidechains* (2014).

[6] BISSIAS, G., OZISIK, A. P., LEVINE, B. N., AND LIBERATORE, M. Sybil-resistant mixing for bitcoin. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society* (New York, NY, USA, 2014), WPES '14, ACM, pp. 149–158.

[7] BOLDYREVA, A. Efficient threshold signature, multisignature and blind signature schemes based on the gap-diffie-hellman-group signature scheme. *IACR Cryptology ePrint Archive 2002* (2002), 118.

[8] BONEH, D., GENTRY, C., LYNN, B., AND SHACHAM, H. Aggregate and verifiably encrypted signatures from bilinear maps. In *Eurocrypt* (2003), vol. 2656, Springer, pp. 416–432.

[9] BONEH, D., LYNN, B., AND SHACHAM, H. Short signatures from the weil pairing. *Advances in CryptologyASIACRYPT 2001* (2001), 514–532.

[10] BONEH, D., SHEN, E., AND WATERS, B. Strongly unforgeable signatures based on computational diffie-hellman. In *Public Key Cryptography* (2006), vol. 3958, Springer, pp. 229–240.

[11] BONNEAU, J., NARAYANAN, A., MILLER, A., CLARK, J., KROLL, J. A., AND FELTEN, E. W. Mixcoin: Anonymity for bitcoin with accountable mixes. In *Financial Cryptography 2014* (2014).

[12] BÜNZ, B., BOOTLE, J., BONEH, D., POELSTRA, A., WUILLE, P., AND MAXWELL, G. Bulletproofs: Short proofs for confidential transactions and more.

[13] BUTERIN, V., AND REITWIESSNER, C. Ethereum improvement proposal 197 - precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-197.md, 2017.

[14] CACHIN, C. Architecture of the hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers* (2016).

[15] CAMENISCH, J., AND LYSYANSKAYA, A. Signature schemes and anonymous credentials from bilinear maps. In *Annual International Cryptology Conference* (2004), Springer, pp. 56–72.

[16] CAMENISCH, J., AND STADLER, M. Proof systems for general statements about discrete logarithms.

[17] CHASE, M., MEIKLEJOHN, S., AND ZAVERUCHA, G. Algebraic macs and keyed-verification anonymous credentials. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 1205–1216.

[18] CHAUM, D., FIAT, A., AND NAOR, M. Untraceable electronic cash. In *Conference on the Theory and Application of Cryptography* (1988), Springer, pp. 319–327.

[19] DIAZ, C., KOSTA, E., DEKEYSER, H., KOHLWEISS, M., AND NIGUSSE, G. Privacy preserving electronic petitions. *Identity in the Information Society 1*, 1 (2008), 203–219.

[20] GALBRAITH, S. D., PATERSON, K. G., AND SMART, N. P. Pairings for cryptographers. *Discrete Applied Mathematics 156*, 16 (2008), 3113–3121.

[21] GENNARO, R., JARECKI, S., KRAWCZYK, H., AND RABIN, T. Secure distributed key generation for discrete-log based cryptosystems. In *Eurocrypt* (1999), vol. 99, Springer, pp. 295–310.

[22] GOLDWASSER, S., MICALI, S., AND RIVEST, R. L. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing 17*, 2 (1988), 281–308.

[23] HEILMAN, E., ALSHENIBR, L., BALDIMTSI, F., SCAFURO, A., AND GOLDBERG, S. Tumblebit: An untrusted bitcoin-compatible anonymous payment hub. In *NDSS 2017* (2016).

[24] KASAMATSU, K. Barreto-naehrig curves. "https://tools.ietf.org/id/draft-kasamatsu-bncurves-01.html", 2014 (version August 14, 2014).

[25] KOKORIS-KOGIAS, E., JOVANOVIC, P., GASSER, L., GAILLY, N., AND FORD, B. Omniledger: A secure, scale-out, decentralized ledger. *IACR Cryptology ePrint Archive 2017* (2017), 406.

[26] LU, S., OSTROVSKY, R., SAHAI, A., SHACHAM, H., AND WATERS, B. Sequential aggregate signatures, multisignatures, and verifiably encrypted signatures without random oracles. *Journal of cryptology 26*, 2 (2013), 340–373.

[27] LYSYANSKAYA, A., MICALI, S., REYZIN, L., AND SHACHAM, H. Sequential aggregate signatures from trapdoor permutations. In *Eurocrypt* (2004), vol. 3027, Springer, pp. 74–90.

[28] MAXWELL, G. Coinjoin: Bitcoin privacy for the real world. https://bitcointalk.org/index.php?topic=279249, 2013.

[29] MEIKLEJOHN, S., AND MERCER, R. Möbius: Trustless tumbling for transaction privacy. In *Proceedings of Privacy Enhancing Technologies* (2018).

[30] PFITZMANN, A., AND KÖHNTOPP, M. Anonymity, unobservability, and pseudonymitya proposal for terminology. In *Designing privacy enhancing technologies* (2001), Springer, pp. 1–9.

[31] POINTCHEVAL, D., AND SANDERS, O. Short randomizable signatures. In *Cryptographers Track at the RSA Conference* (2016), Springer, pp. 111–126.

[32] RAMANNA, S. C., AND SARKAR, P. Efficient adaptively secure ibbe from the sxdh assumption. *IEEE Transactions on Information Theory 62*, 10 (2016), 5709–5726.

[33] REITWIESSNER, C. Ethereum improvement proposal 196 - precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128. `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-196.md`, 2017.

[34] RUFFING, T., MORENO-SANCHEZ, P., AND KATE, A. Coinshuffle: Practical decentralized coin mixing for bitcoin. In *ESORICS (2)* (2014), vol. 8713 of *Lecture Notes in Computer Science*, Springer, pp. 345–364.

[35] SHAMIR, A. How to share a secret. *Communications of the ACM 22*, 11 (1979), 612–613.

[36] THE GUARDIAN. History of 5-Eyes Explainer. `http://www.theguardian.com/world/2013/dec/02/history-of-5-eyes-explainer`, 2013.

[37] THE TOR PROJECT. meek-google suspended for terms of service violations (how to set up your own), 2016. `https://lists.torproject.org/pipermail/tor-talk/2016-June/041699.html`.

[38] VALENTA, L., AND ROWAN, B. Blindcoin: Blinded, accountable mixes for bitcoin. In *Financial Cryptography and Data Security* (Berlin, Heidelberg, 2015), M. Brenner, N. Christin, B. Johnson, and K. Rohloff, Eds., Springer Berlin Heidelberg, pp. 112–126.

[39] WATERS, B. Efficient identity-based encryption without random oracles. In *Eurocrypt* (2005), vol. 3494, Springer, pp. 114–127.

[40] WOOD, G. Ethereum: A secure decentralised generalised transaction ledger eip-150 revision. "`http://gavwood.com/paper.pdf`", 2016 (visited August 9, 2017).

A Primitives Definitions

We present an overview of the cryptographic primitives described in Section 3:

- ❖ **Setup(1^λ):** defines the system parameters *params* with respect to the security parameter λ. These *params* are publicly available.
- ❖ **KeyGen(*params*):** from the public *params*, each signer generates its own secret key *sk* and verification key *vk*.
- ❖ **Sign(sk, m):** generates a signature on a public attribute m using the secret key *sk*.
- ❖ **AggregateSign($\sigma_0, \ldots, \sigma_{n-1}$):** aggregates n partial credentials into one consolidated credential.
- ❖ **AggregateKey($vk_0, \ldots, vk_{n-1}$):** aggregates the verification keys of n authorities into a single key.
- ❖ **Randomize(σ):** randomizes the credential σ.
- ❖ **Verify(vk, m, σ):** verifies the validity of credential σ on the attribute m using the verification key *vk*.

When handling private attributes (Section 3.3), we replace the algorithms Sign and Verify by the following protocols:

- ❖ **PrepareBlindSign(m) $\leftrightarrow$ BlindSign(sk, c_m, c, π_s):** the user requests a blind signature on a private attribute m to an authority by proving in zero-knowledge (π_s) the correctness of the commitment c_m and El-Gamal encryption c of m.
- ❖ **ShowBlindSign(vk, m) $\leftrightarrow$ BlindVerify($\kappa, \nu, \sigma, \pi_v$):** to show possession of a signed attribute σ without revealing it, we follow the standard approach where the algorithm ShowBlindSign is a proof of knowledge of the credential; i.e., using the proof π_v and the group element κ instead of the plain-text values to verify the credentials.

The Coconut threshold credentials scheme replace the algorithm AggregateSign by AggregateThSign relying on a threshold of authorities, rather than all of them:

- ❖ **AggregateThSign($\sigma_1, \ldots, \sigma_t$):** aggregates any subset of t partial credentials into one consolidated credential.

B Sketch of Security Proofs

This appendix sketches the security proofs of the cryptographic construction described in Section 3.

B.1 Security of the Coconut Signature Scheme

First, the co-CDH problem can be rephrased in this context as follows: it is computationally unfeasible for an algorithm $\mathscr{C}$ knowing only $(g_2, g_2^a) \in \mathbb{G}_2^2$ and $h \in \mathbb{G}_1$ (where $a \in \mathbb{F}_p$) to output $h^a \in \mathbb{G}_1$.

Then, Pointcheval *et al.* [31] proposed an assumption based on the LRSW Assumption [27] and on the co-CDH problem that can be rephrased in our context as follows. Considering $vk = (g_2, g_2^x, g_2^y) \in \mathbb{G}_2^3$ where $x, y \in \mathbb{F}_p^2$, an oracle $\mathscr{O}(m)$ on input $m \in \mathbb{F}_p$ chooses a random $h \in \mathbb{G}_1 \setminus 1$ and outputs the pair $\sigma = (h, \varepsilon) = (h, h^{x+my})$; given *vk* and unlimited access to $\mathscr{O}$, it is computationally unfeasible to output σ for a new $m' \in \mathbb{F}_p$ that has not been queried to $\mathscr{O}$.

Finally, we create a modified oracle $\mathscr{O}'$ that acts exactly as $\mathscr{O}$ but doesn't generate h at random; it computes $h = H(g_1^m)$ instead. Under the Random Oracle Assumption, the EUF-CMA security of our scheme follows from the above since the modified oracle $\mathscr{O}'$ is perfectly equivalent to a signing oracle.

B.2 Security of the Coconut Anonymous Credentials Scheme

The following paragraphs argue about the unforgeability, unlinkability and blindness of the Coconut anonymous credentials scheme.

Unforgeability. The unforgeability of the Coconut anonymous credentials scheme relies on the unforgeability of the underlying signature scheme (see Theorem 1). It can be shown that if there is a forger $\mathscr{A}$ capable of forging a credential, then an algorithm $\mathscr{C}$ can query $\mathscr{A}$ to break the underlying signature scheme. Intuitively, $\mathscr{C}$ would execute PrepareBlindSign and get a forgery from $\mathscr{A}$ on a hidden attribute m; then uses her private El-Gamal key to call Unblind on the credential and output a valid forgery on the signature scheme.

Unlinkability. The unlinkability property means that the verifier cannot link multiple executions of the ShowSign $\leftrightarrow$ BlindVerify protocol between each other or with the execution of PrepareBlindSign $\leftrightarrow$ BlindSign (for a given attribute m). This property is enabled by the possibility to re-randomize the signature. Intuitively, given two randomized signatures, σ_0 and σ_1 on the attributes m_0 and m_1, respectively; there is no adversary capable to distinguish which one is a signature on m_0 and which one is a signature on m_1, since both signature are randomly distributed over $\mathbb{G}_1^2$. More specifically, considering signature σ on the attribute m, one can pick a random $t \in \mathbb{F}_p$ and randomized this signature as follows:

$$\sigma' = \mathsf{Randomize}(\sigma) = (h^t, \varepsilon^t)$$

Therefore, we can argue that since t is randomly distributed in $\mathbb{F}_p$, σ' is randomly distributed in $\mathbb{G}_1^2$.

Blindness. Blindness ensures that the signer will not learn any additional information about the messages m during the execution of BlindSign. This property is guaranteed by the security properties of the El-Gamal encryption system since the input of BlindSign is an El-Gamal encryption of m. Also, the ShowBlindSign algorithm does not reveal any information about m neither by the zero-knowledge property of the proof π_v.

C Ethereum tumbler

We extend the example of the tumbler application described in Section 5.1 to the Ethereum version of the Coconut library, with a few modifications to reduce the gas costs incurred and to adapt the system for Ethereum. Instead of having v (the number of coins) as an attribute, which would increase the number of elliptic curve multipications required to verify the credentials, we allow for a fixed number of instances of Coconut to be setup for different denominations for v. The Tumbler has a Deposit method, where users can deposit Ether into the contract, and then send an issuance request to authorities on one private attribute: $addr||s$, where $addr$ is the destination address of the merchant, and s is a randomly generated sequence number (1). It is necessary for $addr$ to be a part of the attribute because once the attribute is revealed, the credential can be spent by anyone with knowledge of the attribute (including any peers monitoring the blockchain for transactions), therefore the credential must be bounded to a specific recipient address before it is revealed. This issuance request is signed by the Ethereum address that deposited the Ether into the smart contract, as proof that the request is associated with a valid deposit, and sent to the authorities (2). As $addr$ and s will be both revealed at the same time when withdrawing the token, we concatenate these in one attribute to save elliptic curve operations.

After the authorities issued the credentials to the users (3), they aggregate them and re-randomize them as usual. The resulting token can then be passed to the Withdraw function, where the withdrawer reveals $addr$ and s (4). As usual, the contract maintains a map of s values associated with tokens that have already been withdrawn to prevent double-spending. After checking that the token's credentials verifies and the it has not already been spent, the contract sends v to the Ethereum destination address $addr$ (5).