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

Alberto Sonnino*, Mustafa Al-Bassam*, Shehar Bano*, Sarah Meiklejohn* and George Danezis*†
* University College London, United Kingdom
† The Alan Turing Institute

***Abstract*—Coconut is a novel selective disclosure credential scheme supporting distributed threshold issuance, public and private attributes, re-randomization, and multiple unlinkable selective attribute revelations. Coconut integrates with 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).**

I. Introduction

Selective disclosure credentials [17], [20] 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 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 [51], Hyperledger [16] 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 issuance).

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 [17] 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 credential and the consolidated credential is composed of exactly two group elements. The size of the credential remains constant regardless of the number of attributes or authorities/issuers. Furthermore, after a one-time setup phase where the users collect and aggregate a threshold number of verification keys from the authorities, the attribute showing and verification are $O(1)$ in terms of both cryptographic computations and communication of cryptographic material—irrespective of the number of authorities. Our evaluation of the Coconut primitives shows very promising results. Verification takes about 10ms, while signing a private attribute is about 3 times faster. The latency is about 600 ms when the client aggregates partial credentials from 10 authorities distributed across the world.

Contribution. 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 II and III). The scheme is an extension and hybrid of the Waters signature scheme [50], the BGLS signature [11], and the signature scheme of Pointcheval and Sanders [41]. This is the first fully distributed threshold issuance, re-randomizable, multi-show credential scheme of which we are aware.
- We use Coconut to implement a generic smart contract library for Chainspace [3] and one for Ethereum [51], performing public and private attribute issuing, aggregation, randomization and selective disclosure (Section IV).

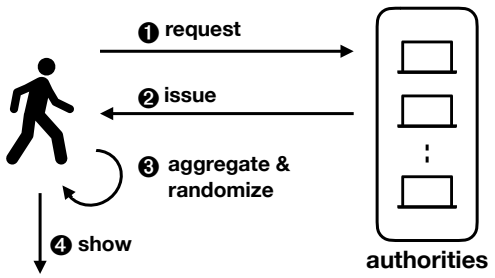

Fig. 1: A high-level overview of Coconut architecture.

We evaluate their performance, and cost within those platforms (Section VI).

- 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 V). We implement and evaluate the first two applications on the Chainspace platform, and provide a security and performance evaluation (Section VI).

II. Overview of Coconut

Coconut is a selective disclosure credential system, supporting threshold credential issuance of public and private attributes, re-randomization of credentials to support multiple unlinkable 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 credential (❷). Any user can collect a threshold number of shares, aggregate them to form a single 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 [10]. The communication complexity of the *request* and *issue* protocol is thus $O(t)$, where t is the size of the subset of authorities. Furthermore, it is impossible to generate a consolidated credential from fewer than t partial credentials.
- **Blind issuance & Unlinkability:** The authorities issue the credential without learning any additional information about the private attributes embedded in the credential. Furthermore, it is impossible to link multiple showings of the credentials with each other, or the issuing transcript, even if all the authorities collude (see Section III-B).
- **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.
- **Liveness:** Coconut guarantees liveness as long as a threshold number of authorities remains honest and weak synchrony assumptions holds for the key distribution [32].
- **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.
- **Short credentials:** Each partial credential—as well as the consolidated credential—is composed of exactly two group elements, 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 efficiency.

III. The Coconut Construction

We introduce the cryptograhic primitives supporting the Coconut architecture, step by step from the design of Pointcheval and Sanders [41] and Boneh *et al.* [12], [11] to the full Coconut scheme.

- **Step 1:** We first recall (Section III-C) the scheme of Pointcheval *et al.* [41] for single-attribute credentials. We present its limitations preventing it from meeting our design goals presented in Section II, and we show how to incorporate principles from Boneh *et al.* [12] to overcome them.
- **Step 2:** We introduce (Section III-D) the *Coconut threshold credentials scheme*, which has all the properties of Pointcheval and Sanders [41] and Boneh *et al.* [12], and allows us to achieve all our design goals.
- **Step 3:** Finally, we extend (Section III-E) our schemes to support credentials embedding q distinct attributes $(m_1, \ldots, m_q)$ simultaneously.

A. 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.

a) Zero-knowledge proofs: Our credential scheme uses non-interactive zero-knowledge proofs to assert knowledge and relations over discrete logarithm values. We represent these non-interactive zero-knowledge proofs with the notation introduced by Camenisch *et al.* [18]:

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

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

b) Security settings: Coconut 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 $(x, y) \in \mathbb{F}_p^2$, $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$; (*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 $\mathbb{G}_2$. The type-3 pairings are efficient [25]. 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 $\mathbb{G}_1$ [12].

Coconut also relies on a cryptographically secure hash function H, hashing an element $\mathbb{G}_1$ into an other element of $\mathbb{G}_1$, namely $H : \mathbb{G}_1 \to \mathbb{G}_1$. We implement this function by serializing the (x, y) coordinates of the input point and applying a full-domain hash function to hash this string into an element of $\mathbb{G}_1$ (as Boneh *et al.* [12]).

B. Scheme Definitions and Security Properties

We present the protocols that comprise a threshold credentials scheme:

- **Setup(1^λ) → ($params$):** defines the system parameters $params$ with respect to the security parameter λ. These parameters are publicly available.
- **KeyGen($params$) → (sk, vk):** run by the authorities to generate their secret key sk and verification key vk from the public $params$.
- **AggKey($vk_1, \ldots, vk_t$) → (vk):** run by whoever wants to verify a credential to aggregate any subset of t verification keys vk_i into a single consolidated verification key vk. AggKey needs to be run only once.
- **IssueCred(m, ϕ) → (σ):** Interactive protocol between a user and each authority, by which the user obtains a credential σ embedding the private attribute m satisfying the statement ϕ.
- **AggCred($\sigma_1, \ldots, \sigma_t$) → ($\sigma$):** run by the user to aggregate any subset of t partial credentials σ_i into a single consolidated credential.
- **ProveCred(vk, m, ϕ') → (Θ, ϕ'):** run by the user to compute a proof Θ of possession of a credential certifying that the private attribute m satisfies the statement ϕ' (under the corresponding verification key vk).
- **VerifyCred(vk, Θ, ϕ') → ($true/false$):** run by whoever wants to verify a credential embeding a private attribute satisfying the statement ϕ', using the verification key vk and cryptographic material Θ generated by ProveCred.

A threshold credential scheme must satisfy the following security properties:

Unforgeability: It must be unfeasible for an adversarial user to convince an honest verifier that they are in possession of a credential if they are in fact not (i.e., if they have not received valid partial credentials from at least t authorities).

Blindness: It must be unfeasible for an adversarial authority to learn any information about the attribute m during the execution of the IssueCred protocol, except for the fact that m satisfies ϕ.

Unlinkability / Zero-knowledge: It must be unfeasible for an adversarial verifier (potentially working with an adversarial authority) to learn anything about the attribute m, except that it satisfies ϕ', or to link the execution of ProveCred with either another execution of ProveCred or with the execution of IssueCred (for a given attribute m).

C. Foundations of Coconut

Before giving the full Coconut construction, we first recall the credentials scheme proposed by Pointcheval and Sanders [41]; their construction has the same properties as CL-signatures [17] but is more efficient. The scheme works in a bilinear 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 III-A.

- **P.Setup(1^λ) → ($params$):** Choose a bilinear group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ with order p, where p is a λ-bit prime number. Let g_1 be a generator of $\mathbb{G}_1$, and g_2 a generator of $\mathbb{G}_2$. The system parameters are $params = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2)$.
- **P.KeyGen($params$) → (sk, vk):** 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 $vk = (g_2, \alpha, \beta) = (g_2, g_2^x, g_2^y)$.
- **P.Sign($params, sk, m$) → (σ):** Parse $sk = (x, y)$. Pick a random $r \in \mathbb{F}_p$ and set $h = g_1^r$. Output $\sigma = (h, s) = (h, h^{x+y \cdot m})$.

The signature $\sigma = (h, s)$ is randomizable by choosing a random $r' \in \mathbb{F}_p$ and computing $\sigma' = (h^{r'}, s^{r'})$. The above scheme can be modified to obtain credentials on a private attribute: to run IssueCred the user first picks a random $t \in \mathbb{F}_p$, computes the commitment $c_p = g_1^t Y$ where $Y = g_1^y$, and sends it to a single authority along with a zero-knowledge proof of the opening of the commitment. The authority verifies the proof, picks a random $u \in \mathbb{F}_p$, and returns $\sigma' = (h', s') = (g^u, (X c_p)^u)$ where $X = g_1^x$. The user unblinds the signature by computing $\sigma = (h', s'(h')^{-t})$, and this value acts as the credential.

This scheme provides blindness, unlinkability, efficiency and short credentials; but it does not support threshold issuance and therefore does not achieve our design goals. This limitation comes from the P.Sign algorithm—the issuing authority computes the credentials using a private and self-generated random number r which prevents the scheme from being efficiently distributed to a multi-authority setting[1]. To overcome that limitation, we take advantage of a concept introduced by BLS signatures [12]; exploiting a hash function $H : \mathbb{F}_p \to \mathbb{G}_1$ to compute the group element $h = H(m)$. The next section describes how Coconut incorporates these concepts to achieve all our design goals.

D. The Coconut Threshold Credential Scheme

We introduce the *Coconut* threshold credential scheme, allowing users to obtain a partial credential σ_i on a private

[1]The original paper of Pointcheval and Sanders [41] proposes a sequential aggregate signature protocol that is unsuitable for threshold credentials issuance (see Section VII).

or public attribute m. In a system with n authorities, a t-out-of-n threshold credentials scheme offers great flexibility as the users need to collect only $n/2 < t \leq n$ of these partial credentials in order to recompute the consolidated credential (both t and n are scheme parameters).

a) Cryptographic primitives: 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 executed in a distributed way as illustrated by Gennaro *et al.* [26] under synchrony assumption, and as illustrated by Kate *et al.* [32] under weak synchrony assumption. Adding and removing authorities implies a re-run of the key generation algorithm—this limitation is inherited from the underlying Shamir's secret sharing protocol [46] and can be mitigated using techniques introduced by Herzberg *et al.* [28].

❖ **Setup**(1^λ) → ($params$)**:** Choose a bilinear group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ with order p, where p is a λ-bit prime number. Let g_1, h_1 be generators of $\mathbb{G}_1$, and g_2 a generator of $\mathbb{G}_2$. The system parameters are $params = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2, h_1)$.

❖ **TTPKeyGen**($params, t, n$) → (sk, vk)**:** Pick[2] two polynomials v, w of degree $t-1$ with coefficients in $\mathbb{F}_p$, and set $(x, y) = (v(0), w(0))$. Issue to each authority $i \in [1, \ldots, n]$ a secret key $sk_i = (x_i, y_i) = (v(i), w(i))$, and publish their verification key vk_i = $(g_2, \alpha_i, \beta_i) = (g_2, g_2^{x_i}, g_2^{y_i})$.

❖ **IssueCred**(m, ϕ) → (σ)**:** Credentials issuance is composed of three algorithms:

❖ PrepareBlindSign(m, ϕ) → (d, Λ, ϕ): The users generate an El-Gamal key-pair $(d, \gamma = g_1^d)$; pick a random $o \in \mathbb{F}_p$, compute the commitment c_m and the group element $h \in \mathbb{G}_1$ as follows:

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

Pick a random $k \in \mathbb{F}_p$ and compute an El-Gamal encryption of m as below:

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

Output $(d, \Lambda = (\gamma, c_m, c, \pi_s), \phi)$, where ϕ is an application-specific predicate satisfied by m, and π_s is defined by:

$$\pi_s = \text{NIZK}\{(d, m, o, k) : \gamma = g_1^d \ \wedge \ c_m = g_1^m h_1^o \wedge c = (g_1^k, \gamma^k h^m) \ \wedge \ \phi(m) = 1\}$$

❖ BlindSign(sk_i, Λ, ϕ) → ($\tilde{\sigma}_i$): The authority i parses $\Lambda = (\gamma, c_m, c, \pi_s)$, $sk_i = (x, y)$, and $c = (a, b)$. Recompute $h = H(c_m)$. Verify the proof π_s using γ, c_m and ϕ; if the proof is valid, build $\tilde{c} = (a^y, h^x b^y)$ and output $\tilde{\sigma}_i = (h, \tilde{c})$; otherwise output $\perp$ and stop the protocol.

❖ Unblind($\tilde{\sigma}_i, d$) → (σ_i): The users parse $\tilde{\sigma}_i = (h, \tilde{c})$ and $\tilde{c} = (\tilde{a}, \tilde{b})$; compute $\sigma_i = (h, \tilde{b}(\tilde{a})^{-d})$. Output σ_i.

❖ **AggCred**($\sigma_1, \ldots, \sigma_t$) → ($\sigma$)**:** Parse each σ_i as (h, s_i) for $i \in [1, \ldots, t]$. Output $(h, \prod_{i=1}^{t} s_i^{l_i})$, where l is the Lagrange coefficient:

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

❖ **ProveCred**(vk, m, σ, ϕ') → (Θ, ϕ')**:** Parse $\sigma = (h, s)$ and $vk = (g_2, \alpha, \beta)$. Pick at random $r', r \in \mathbb{F}_p^2$; set $\sigma' = (h', s') = (h^{r'}, s^{r'})$; build $\kappa = \alpha\beta^m g_2^r$ and $\nu = (h')^r$. Output $(\Theta = (\kappa, \nu, \sigma', \pi_v), \phi')$, where ϕ' is an application-specific predicate satisfied by m, and π_v is:

$$\pi_v = \text{NIZK}\{(m, r) : \kappa = \alpha\beta^m g_2^r \wedge \nu = (h')^r \wedge \phi'(m) = 1\}$$

❖ **VerifyCred**(vk, Θ, ϕ') → ($true/false$)**:** Parse $\Theta = (\kappa, \nu, \sigma', \pi_v)$ and $\sigma' = (h', s')$; verify π_v using vk and ϕ'. Output $true$ if the proof verifies, $h' \neq 1$ and $e(h', \kappa) = e(s'\nu, g_2)$; otherwise output $false$.

b) Correctness and explanation: The Setup algorithm generates the public parameters. Credentials are elements of $\mathbb{G}_1$, while verification keys are elements of $\mathbb{G}_2$. Figure 2 illustrates the protocol exchanges.

To keep an attribute $m \in \mathbb{F}_p$ hidden from the authorities, the users run PrepareBlindSign to produce $\Lambda = (\gamma, c_m, c, \pi_s)$. They create an El-Gamal keypair $(d, \gamma = g_1^d)$, pick a random $o \in \mathbb{F}_p$, and compute a commitment $c_m = g_1^m h_1^o$. Then, the users compute $h = H(c_m)$ and the encryption of h^m as below:

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

where $k \in \mathbb{F}_p$. Finally, the users send (Λ, ϕ) to the signer, where π_s is a zero-knowledge proof ensuring that m satisfies the application-specific predicate ϕ, and correctness of γ, c_m, c (❶). Note that all the zero-knowledge proofs required by Coconut are based on standard sigma protocols to show knowledge of representation of discrete logarithms; they are based on the DH assumption [18] and do not require any trusted setup.

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

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

Note that every authority must operate on the same element h. Intuitively, generating h from $h = H(c_m)$ is equivalent to computing $h = g_1^{\tilde{r}}$ where $\tilde{r} \in \mathbb{F}_p$ is unknown by the users (as in Pointcheval and Sanders [41]). 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.[3] As described in Section III-C, the blind signature scheme of Pointcheval and Sanders builds the credentials directly from a commitment of the attribute and a blinding factor secretly chosen by the authority; this is unsuitable for issuance of threshold credentials. We circumvent that problem by introducing the El-Gamal ciphertext c in our scheme and exploiting its homomorphism, as described above.

[2]This algorithm can be turned into the KeyGen and AggKey algorithms described in Section III-B using techniques illustrated by Gennaro *et al.* [26] or Kate *et al.* [32].

[3]If an adversary $\mathcal{A}$ 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 $\mathcal{A}$ could forge a new credential σ_2 on $m_2 = 2$: $\sigma_2 = (\sigma_0)^{-1}\sigma_1\sigma_1 = h^{x+2y}$.

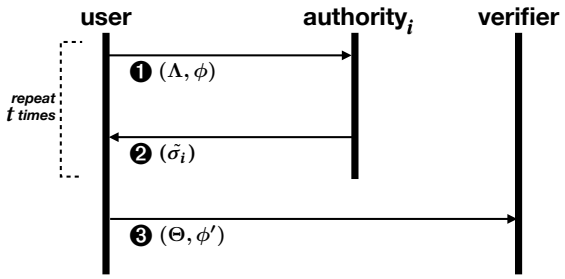

Fig. 2: Coconut threshold credentials protocol exchanges.

Upon reception of $\tilde{c}$, the users decrypt it using their ElGamal private key d to recover the partial credentials $\sigma_i = (h, h^{x_i + y_i \cdot m})$; this is performed by the Unblind algorithm (❷). Then, the users can call the AggCred algorithm to aggregate any subset of t partial credentials. This algorithm uses the Lagrange basis polynomial l which allows to reconstruct the original $v(0)$ and $w(0)$ through polynomial interpolation;

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

However, this computation happens in the exponent—neither the authorities nor the users should know the values $v(0)$ and $w(0)$. One can easily verify the correctness of AggCred of t partial credentials $\sigma_i = (h_i, s_i)$ as below.

$$\begin{aligned} s &= \prod_{i=1}^{t} (s_i)^{l_i} = \prod_{i=1}^{t} \left(h^{x_i + y_i \cdot m}\right)^{l_i} \\ &= \prod_{i=1}^{t} (h^{x_i})^{l_i} \prod_{i=1}^{t} (h^{y_i \cdot m})^{l_i} = \prod_{i=1}^{t} h^{(x_i l_i)} \prod_{i=1}^{t} h^{(y_i l_i) \cdot m} \\ &= h^{v(0) + w(0) \cdot m} = h^{x + y \cdot m} \end{aligned}$$

Before verification, the verifier collects and aggregates the verifications keys of the authorities—this process happens only once and ahead of time. The algorithms ProveCred and VerifyCred implement verification. First, the users randomize the credentials by picking a random $r' \in \mathbb{F}_p$ and computing $\sigma' = (h', s') = (h^{r'}, s^{r'})$; then, they compute κ and ν from the attribute m, a blinding factor $r \in \mathbb{F}_p$ and the aggregated verification key:

$$\kappa = \alpha \beta^m g_2^r \qquad \text{and} \qquad \nu = (h')^r$$

Finally, they send $\Theta = (\kappa, \nu, \sigma', \pi_v)$ and ϕ' to the verifier where π_v is a zero-knowledge proof asserting the correctness of κ and ν; and that the private attribute m embedded into σ satisfies the application-specific predicate ϕ' (❸). The proof π_v also ensures that the users actually know m and that κ has been built using the correct verification keys and blinding factors. The pairing verification is similar to Pointcheval and Sanders [41] and Boneh *et al.* [12]; expressing $h' = g_1^{\tilde{r}} \mid \tilde{r} \in \mathbb{F}_p$, the left-hand side of the pairing verification can be expanded as:

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

and the right-hand side:

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

From where the correctness of VerifyCred follows.

c) Security: The proof system we require is based on standard sigma protocols to show knowledge of representation of discrete logarithms, and can be rendended non-interactive using the Fiat-Shamir heuristic [24] in the random oracle model. As our signature scheme is derived from the ones due to Pointcheval and Sanders [41] and BLS [12], we inherit their assumptions as well; namely, LRSW [36] and XDH [12].

Theorem 1. *Assuming LRSW, XDH, and the existence of random oracles, Coconut is a secure threshold credentials scheme, meaning it satisfies unforgeability, blindness, and unlinkability.*

A sketch of this proof, based on the security of the underlying components of Coconut, can be found in Appendix A.

E. Multi-Attribute Credentials

We expand our scheme to embed multiple attributes into a single credential without increasing its size; this generalization follows directly from the Waters signature scheme [50] and Pointcheval and Sanders [41]. The authorities key pairs becomes:

$$sk = (x, y_1, \ldots, y_q) \quad \text{and} \quad vk = (g_2, g_2^x, g_2^{y_1}, \ldots, g_2^{y_q})$$

where q is the number of attributes. The multi-attribute credential is derived from the commitment c_m and the group element h as below:

$$c_m = g_1^o \prod_{j=1}^{q} h_j^{m_j} \qquad \text{and} \qquad h = H(c_m)$$

and the credential generalizes as follows:

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

Note that the credential's size does not increase with the number of attributes or authorities—it is always composed of two group elements. The security proof of the multi-attribute scheme relies on a reduction against the single-attribute scheme and is analog to Pointcheval and Sanders [41]. 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 (similar to Chase *et al.* [20]). The full primitives of the multi-attribute cryptographic scheme are presented in Appendix B.

Note that if the credentials only include non-random attributes, the verifier could guess its value by brute-forcing the verification algorithm[4]. This issue is prevented by always embedding a private random attribute into the credentials, that can also act as the authorization key for the credential.

[4]Let assume for example that some credentials include a single attribute m representing the age of the user; the verifier can run the verification algorithm $e(h, \kappa(\alpha \cdot \beta^m)^{-1}) = e(\nu, g_2)$ for every $m \in [1, 100]$ and guess the value of m.

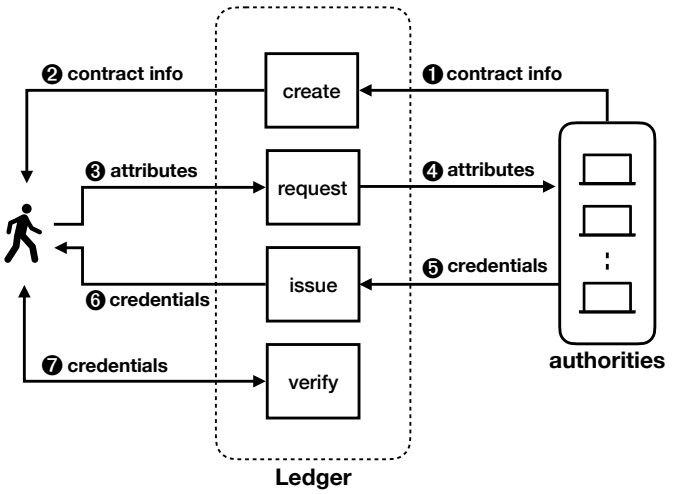

Fig. 3: The Coconut smart contract library.

IV. Implementation

We implement a Python library for Coconut as described in Section III and publish the code on GitHub as an open-source project [5]. We also implement a smart contract library in Chainspace [3] to enable other application-specific smart contracts (see Section V) to conveniently use our cryptographic primitives. We present the design and implementation of the Coconut smart contract library in Section IV-A. In addition, we implement and evaluate some of the functionality of the Coconut smart contract library in Ethereum [51] (Section IV-B). Finally, we show how to integrate Coconut into existing semi-permissioned blockchains (Section IV-C).

A. The Coconut Smart Contract Library

We implement the Coconut smart contract in Chainspace[6] (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, Request, Issue, Verify), as 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 III-D. 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 IV-C explains how to overcome this limitation by embedding the authorities into the nodes running the blockchain.

B. Ethereum Smart Contract Library

To make Coconut more widely available, we also implement it in Ethereum—a popular permissionless smart contract blockchain [51]. We release the Coconut Ethereum smart contract as an open source library[7]. The library is written in Solidity, a high-level JavaScript-like language that compiles down to Ethereum Virtual Machine (EVM) assembly code. Ethereum recently hardcoded a pre-compiled smart contract in the EVM for performing pairing checks and elliptic curve operations on the alt_bn128 curve [15], [43], 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 elliptic curve addition and scalar multiplication on $\mathbb{G}_2$ as an Ethereum smart contract library written in Solidity that we also release open source[8]. 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—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 V-A, 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 IV-A, except for Request and Issue which are computed off the blockchain to save gas costs. As Request and Issue functions simply act as a

[5]https://github.com/asonnino/coconut
[6]https://github.com/asonnino/coconut-chainspace
[7]https://github.com/musalbas/coconut-ethereum
[8]https://github.com/musalbas/solidity-BN256G2

communication channel between users and authorities, users can directly communicate with authorities off the blockchain to request tokens. This saves significant gas costs that would be incurred by storing these functions on the blockchain.

The Verify function simply verifies tokens against Coconut instances created by the Create function.

C. Deeper Blockchain Integration

The designs described in Section IV-A and Section IV-B rely on authorities on-the-side for issuing credentials. In this section, 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 permissionless systems, based on proof of work or stake. These systems have a highly dynamic set of nodes maintaining the state of their blockchains, which cannot readily be mapped into Coconut issuing authorities.

Integration of Coconut into Hyperledger Fabric [16]—a permissioned blockchain platform—is straightforward. Fabric contracts run on private sets of computation 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 secrets 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 currently present in the Hyperledger roadmap[9]. The threshold trust assumption—namely that integrity and availability 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 [33] (which supports digital tokens). In these systems, 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—a special transaction type in Omniledger, or a special object in Chainspace, can signal to them that issuing a credential is authorized. The authorities, then 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.

V. Applications

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

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

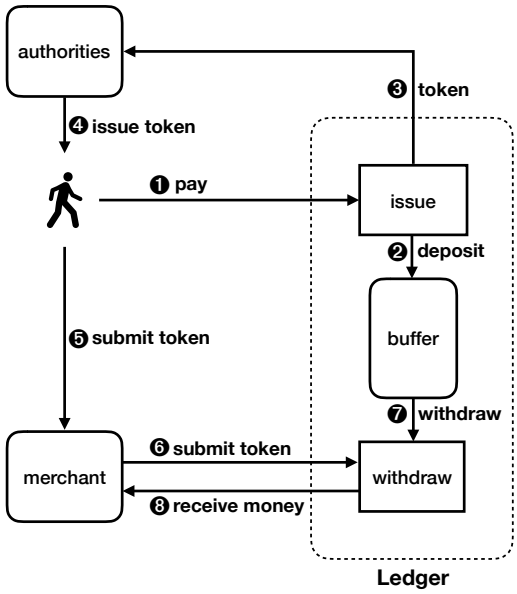

Fig. 4: The coin tumbler application.

A. 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. A limitation of previous similar schemes [13], [49], [27], [37], [44], [9], [38] is that they are either centralized (i.e., there is a central authority that operates the tumbler, which may go offline), or require users to coordinate with each other. The Coconut tumbler addresses these issues *via* a distributed design (i.e., security relies on a set of multiple authorities that are collectively trusted to contain at least t honest ones), and does not require users to coordinate with each other. Zcash [45] achieves a similar goal; it theoretically hides the totality of the transaction but at a huge computational cost, and offers the option to cheaply send transactions in clear. In practice, the computational overhead of sending hidden transactions makes it impractical, and only a few users take advantage of the optional privacy provided by Zcash; as a result, transactions are easy to de-anonymize [29]. Coconut provides efficient proofs taking only a few milliseconds (see Section VI), and makes hidden transactions practical. Trust assumptions in Zcash are different from Coconut. However, instead of assuming a threshold number of honest authorities, Zcash relies on zk-SNARKs which assumes a setup algorithm executed by a single trusted authority.

Our 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 [21], 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 IV-A 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 credential 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 coin 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 credential and a group element $\zeta = g_1^s \in \mathbb{G}_1$ to the tumbler contract along with a zero-knowledge proof ensuring that ζ is well-formed (❻). To prevent double spending, the tumbler contract keeps a record of all elements ζ that have already been shown. Upon showing a ζ embedding a fresh (unspent) sequence number s, the contract verifies that the credential and zero-knowledge proofs check, and that ζ 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 ζ 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.* [14].

Security consideration. Coconut provides blind issuance which allows the user to obtain a credential on the sequence number s without the authorities learning its value. Without blindness, any authority seeing the user key k could potentially race 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 or services offered, yet not identify the purchasers. Keeping a spent list of all elements ζ prevents double-spending attacks [30] without revealing the sequence number s; this prevents an attacker from exploiting a race condition in the *submit token* phase (❻) and lock user's funds[10]. 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. A small subset of authorities cannot block the issuance of a token—the service is guaranteed to be available as long as at least t authorities are running.

[10]An attacker observing a sequence number s during a *submit token* phase (❻) could exploit a race condition to lock users fund by quickly buying a token using the same s, and spending it before the original *submit token* phase is over.

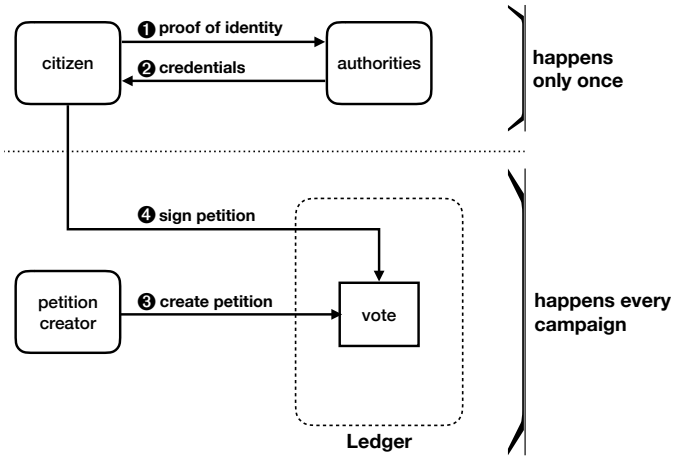

Fig. 5: The petition application.

B. Privacy-preserving petition

We consider the scenario where several authorities managing the country *C* wish 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 extends the work of Diaz *et al.* [23] which does not consider threshold issuance of credentials.

Our petition system 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 petition initiator, and the citizens of *C*. The signing authorities create an instance of the Coconut smart contract as described in Section IV-A. 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 (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 $g_s \in \mathbb{G}_1$ unique to the petition where its representation is unlinkable to the other points of the scheme[11], as well as the verification key of the authorities issuing the credentials and any application specific parameters (e.g., the options and current votes) (❸). In order to *sign* a petition, the citizens compute a value $\zeta = g_s^k$. They then adapt the zero-knowledge proof of the ProveCred algorithm of Section III-D to show that ζ is built from the same attribute k in the credential; the petition contract checks the proofs 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, the zero-knowledge proof ensures that ζ has been built from a signed private key k; this means that

[11]This identifier can be generated through a hash function $\mathbb{F}_p \rightarrow \mathbb{G}_1 : \widetilde{H}(s) = g_s \mid s \in \mathbb{F}_p$.

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.

C. 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 two 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* message which looks as follows: user IP address encrypted under pk' which is embedded in 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 relying on volunteers that are *assumed* to be 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 [48], [47]. The proposed CRS overcomes this limitation by offering coercion-resistance to volunteers from censor-controlled users and authorities. Due to Coconut's blindness property, a volunteer can get a credential on its IP address and ephemeral public key without revealing those to the authorities. The users get proxy IP addresses 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.

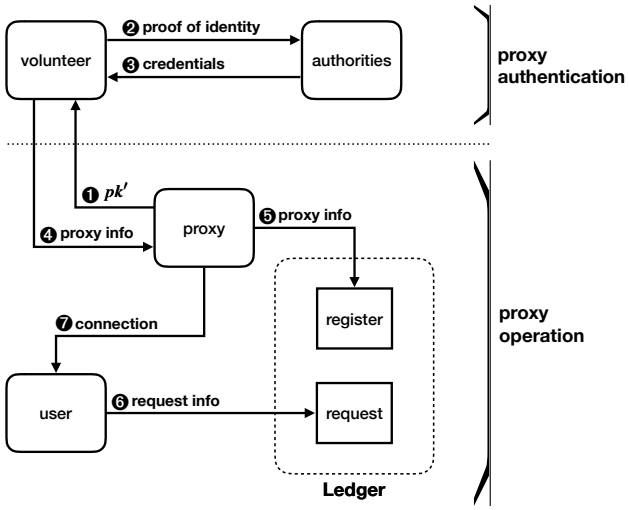

Fig. 6: The censorship-resistant proxy distribution system.

Operation	μ [ms]	$\sqrt{\sigma^2}$ [ms]
PrepareBlindSign	2.633	$\pm$ 0.003
BlindSign	3.356	$\pm$ 0.002
Unblind	0.445	$\pm$ 0.002
AggCred	0.454	$\pm$ 0.000
ProveCred	1.544	$\pm$ 0.001
VerifyCred	10.497	$\pm$ 0.002

TABLE I: Execution times for the cryptographic primitives described in Section III, measured for one private attribute over 10,000 runs. AggCred is computed assuming two authorities; the other primitives are independent of the number of authorities.

VI. Evaluation

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

A. Cryptographic Primitives

We implement the primitives described in Section III in Python using petlib [1] and bplib [2]. The bilinear pairing is defined over the Barreto-Naehrig [31] curve, using OpenSSL as arithmetic backend.

a) Timing benchmark: Table I shows the mean (μ) and standard deviation ($\sqrt{\sigma^2}$) of the execution of each procedure described in section Section III. Each entry is the result of 10,000 runs measured on an Octa-core Dell desktop computer, 3.6GHz Intel Xeon. Signing is much faster than verifying credentials—due to the pairing operation in the latter; verification takes about 10ms; signing a private attribute is about 3 times faster.

b) Communication complexity and packets size: Table II shows the communication complexity and the size of each exchange involved in the Coconut credentials scheme,

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

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

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 credential is 132 bytes. The highest transaction sizes are to request and verify credentials embedding a private attribute; this is due to the proofs π_s and π_v (see Section III). The proof π_s is approximately 318 bytes and π_v is 157 bytes.

c) Client-perceived latency: We evaluate the client-perceived latency for the 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.

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.

B. Chainspace Implementation

We evaluate the Coconut smart contract library implemented in Chainspace, as well as the the coin tumbler (Section V-A) and the privacy-preserving e-petition (Section V-B) applications that use this library. As expected, Table III shows that the most time consuming procedures are the checker of Create and the checker of Verify; i.e., they call the VerifyCred primitives which takes about 10 ms (see Table I). Table III is computed assuming two authorities; the transaction size

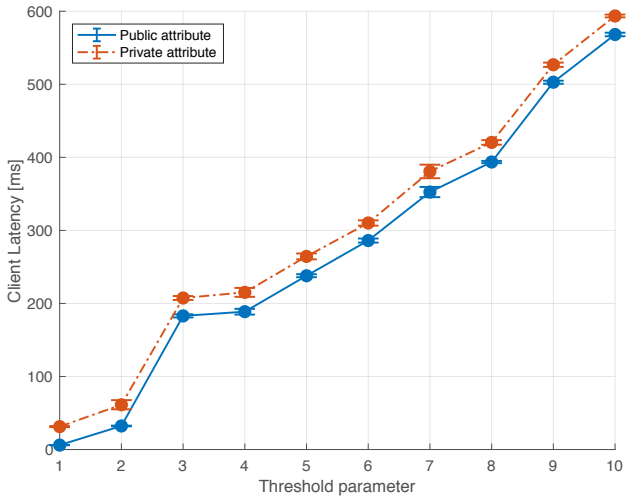

Fig. 7: Client-perceived latency for Coconut threshold credentials scheme with geographically distributed authorities, measured for one attribute over 100 runs.

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

TABLE III: Timing and transaction size of the Chainspace implementation of the Coconut smart contract library described in Section IV-A, measured for two authorities and one private attributes over 10,000 runs. The notation [g] denotes the execution the procedure and [c] denotes the execution of the checker.

of Issue increases by about 132 bytes (*i.e.,* the size of the credentials) for each extra authority[12] while the other transactions are independent of the number of authorities.

Similarly, the most time consuming procedure of the coin tumbler (Table IV) application and of the privacy-preserving e-petition (Table V) 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, and they all run in a few milliseconds. These transactions are independent of the number of authorities as issuance is either handled off-chain or by the Coconut smart contract library.

C. Ethereum Implementation

We evaluate the Coconut Ethereum smart contract library described in Section IV-B using the Go implementation of Ethereum on an Intel Core i5 laptop with 12GB of RAM

[12]The Request and Issue procedures are only needed in the case of on-chain issuance (see Section IV-A).

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

TABLE IV: Timing and transaction size of the Chainspace implementation of the coin tumbler smart contract (described in Section V-A), measured over 10,000 runs. The transactions are independent of the number of authorities. The notation [g] denotes the execution the procedure and [c] denotes the execution of the checker.

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

TABLE V: Timing and transaction size of the Chainspace implementation of the privacy-preserving e-petition smart contract (described in Section V-B), measured over 10,000 runs. The transactions are independent of the number of authorities. The notation [g] denotes the execution the procedure and [c] denotes the execution of the checker.

running Ubuntu 17.10. Table VI 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 III) 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.

We also observe that the Verify function has a significantly higher gas cost than Create. This is mostly due to the implementation of elliptic curve multiplication as a native Ethereum smart contract—the elliptic curve multiplication alone costs around $1,700,000$ 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.[13]

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 [42] (which is stronger than the standard XDH assumption that we are currently using).

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 VI: Timing and gas cost of the Ethereum implementation of the Coconut smart contract library described in Section IV-B. Measured over 100 runs, for one public attribute. The transactions are independent of the number of authorities.

VII. Comparison with Related Works

We compare the Coconut cryptographic constructions and system with related work in Table VII, 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 II and III).

a) Short and aggregable signatures: The Waters signature scheme [50] 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 as it has not been built for use in a multi-authority setting. Lu *et al.* scheme, commonly known as LOSSW signature scheme [35], 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 [11] 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 [10]. Moreover, Boneh *et al.* show how to build verifiably encrypted signatures [11] which is close to our requirements, but not suitable for anonymous credentials.

b) Anonymous credentials: CL Signatures [17], [34] and Idemix [8] are amongst the most well-known building blocks that inspired applications going from direct anonymous attestations [22], [7] to electronic cash [19]. They provide blind issuance and unlikability through randomization; but come with significant computational overhead and credentials are not short as their size grows linearly with the number of signed attributes, and are not aggregable. U-Prove [39] and Anonymous Credentials Light (ACL) [6] are computationally efficient credentials that can be used once unlinkably; therefore the size of the credentials is linear in the number of unlinkable uses. Pointcheval and Sanders [41] 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. For anonymous credentials in a setting where the signing authorities are also verifiers (i.e., without

[13] https://ethgasstation.info/

Scheme	Blindness	Unlinkable	Aggregable	Threshold	Signature Size
[50] Waters Signature	✗	✗	○	✗	2 Elements
[35] LOSSW Signature	✗	✗	◒	✗	2 Elements
[11] BGLS Signature	✓	✗	●	✓	1 Element
[17] CL Signature	✓	✓	◒	✗	$O(q)$ Elements
[8] Idemix	✓	✓	○	✗	$O(q)$ Elements
[39] U-Prove	✓	✓	○	✗	$O(v)$ Elements
[6] ACL	✓	✓	○	✗	$O(v)$ Elements
[41] Pointcheval and Sanders	✓	✓	◒	✗	2 Elements
[Section III] Coconut	✓	✓	●	✓	2 Elements

TABLE VII: 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 asymptotically or in terms of the number of group elements it is made of (for constant-size credentials); q indicates the number of attributes embedded in the credentials and v the number of times the credential may be shown unlinkably.

public verifiability), Chasse *et al.* [20] develop an efficient protocol. Its 'GGM' variant has a similar structure to Coconut, but forgoes the pairing operation by using message authentication codes (MACs). None of the above schemes support threshold issuance.

c) Short and threshold issuance anonymous credentials: Coconut extends 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.

VIII. Limitations

Coconut has a number of limitations that are beyond the scope of this work, and deferred to future work.

Adding and removing authorities implies to re-run the key generation algorithm—this limitation is inherited from the underlying Shamir's secret sharing protocol [46] and can be mitigated using techniques coming from proactive secret sharing introduced by Herzberg *et al.* [28]. However, credentials issued by authorities with different key sets are distinguishable, and therefore frequent key rotation reduces the privacy provided.

As any threshold system, Coconut is vulnerable if more than the threshold number of authorities are malicious; colluding authorities could create coins to steal all the coins in the buffer of the coin tumbler application (Section V-A), create fake identities or censor legitimate users of the electronic petition application (Section V-B), and defeat the censorship resistance of our proxy distribution application (Section V-C). Note that users' privacy is still guaranteed under colluding authorities, or an eventual compromise of their keys.

Implementing the Coconut smart contract library on Ethereum is expensive (Table VI) as Ethereum does not provide pre-compiled contracts for elliptic curve arithmetic in $\mathbb{G}_2$; re-writing our cryptographic primitives by swapping all the operations in $\mathbb{G}_1$ and $\mathbb{G}_2$ would dramatically reduce the gas cost, at the cost of relying on the SXDH assumption [42].

IX. Conclusion

Existing selective credential disclosure schemes 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. This limits their applicability in distributed settings such as distributed ledgers, and prevents security engineers from implementing separation of duty policies that are effective in preserving integrity.

In this paper, we present Coconut—a novel scheme that supports distributed threshold issuance, public and private attributes, re-randomization, and multiple unlinkable selective attribute revelations. We provide 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. Coconut fills an important gap in the literature and enables selective disclosure credentials—an important privacy enhancing technology—to be used in settings with no natural single trusted third party to issue them, and to interoperate with modern transparent computation platforms.

References

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

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

[3] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A Sharded Smart Contracts Platform," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[4] I. Amazon Web Services, "Aws whitepapers," "https://aws.amazon.com/whitepapers/", 2017 (version April, 2017).

[5] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, "Enabling blockchain innovations with pegged sidechains," *http://www.opensciencereview.com/papers/123/enablingblockchain-innovations-with-pegged-sidechains*, 2014.

[6] F. Baldimtsi and A. Lysyanskaya, "Anonymous credentials light," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1087–1098.

[7] D. Bernhard, G. Fuchsbauer, E. Ghadafi, N. P. Smart, and B. Warinschi, "Anonymous attestation with user-controlled linkability," *International Journal of Information Security*, vol. 12, no. 3, pp. 219–249, 2013.

[8] P. Bichsel, C. Binding, J. Camenisch, T. Groß, T. Heydt-Benjamin, D. Sommer, and G. Zaverucha, "Cryptographic protocols of the identity mixer library," *Tech. Rep. RZ 3730, Tech. Rep.*, 2009.

[9] G. Bissias, A. P. Ozisik, B. N. Levine, and M. Liberatore, "Sybil-resistant mixing for bitcoin," in *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, ser. WPES '14. New York, NY, USA: ACM, 2014, pp. 149–158. [Online]. Available: http://doi.acm.org/10.1145/2665943.2665955

[10] A. Boldyreva, "Efficient threshold signature, multisignature and blind signature schemes based on the gap-diffie-hellman-group signature scheme." *IACR Cryptology ePrint Archive*, vol. 2002, p. 118, 2002.

[11] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps," in *Eurocrypt*, vol. 2656. Springer, 2003, pp. 416–432.

[12] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," *Advances in CryptologyASIACRYPT 2001*, pp. 514–532, 2001.

[13] J. Bonneau, A. Narayanan, A. Miller, J. Clark, J. A. Kroll, and E. W. Felten, "Mixcoin: Anonymity for bitcoin with accountable mixes," in *Financial Cryptography 2014*, 2014.

[14] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," 2018.

[15] V. Buterin and C. Reitwiessner, "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.

[16] C. Cachin, "Architecture of the hyperledger blockchain fabric," in *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016.

[17] J. Camenisch and A. Lysyanskaya, "Signature schemes and anonymous credentials from bilinear maps," in *Annual International Cryptology Conference*. Springer, 2004, pp. 56–72.

[18] J. Camenisch and M. Stadler, "Proof systems for general statements about discrete logarithms," 1997.

[19] S. Canard, D. Pointcheval, O. Sanders, and J. Traoré, "Divisible e-cash made practical," in *IACR International Workshop on Public Key Cryptography*. Springer, 2015, pp. 77–100.

[20] M. Chase, S. Meiklejohn, and G. Zaverucha, "Algebraic macs and keyed-verification anonymous credentials," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1205–1216.

[21] D. Chaum, A. Fiat, and M. Naor, "Untraceable electronic cash," in *Conference on the Theory and Application of Cryptography*. Springer, 1988, pp. 319–327.

[22] L. Chen, D. Page, and N. P. Smart, "On the design and implementation of an efficient daa scheme," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2010, pp. 223–237.

[23] C. Diaz, E. Kosta, H. Dekeyser, M. Kohlweiss, and G. Nigusse, "Privacy preserving electronic petitions," *Identity in the Information Society*, vol. 1, no. 1, pp. 203–219, 2008.

[24] A. Fiat and A. 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.

[25] S. D. Galbraith, K. G. Paterson, and N. P. Smart, "Pairings for cryptographers," *Discrete Applied Mathematics*, vol. 156, no. 16, pp. 3113–3121, 2008.

[26] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," in *Eurocrypt*, vol. 99. Springer, 1999, pp. 295–310.

[27] E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg, "Tumblebit: An untrusted bitcoin-compatible anonymous payment hub," in *NDSS 2017*, 2016.

[28] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, "Proactive secret sharing or: How to cope with perpetual leakage," in *Annual International Cryptology Conference*. Springer, 1995, pp. 339–352.

[29] G. Kappos, H. Yousaf, M. Maller, and S. Meiklejohn, "An empirical analysis of anonymity in zcash," 2018.

[30] G. O. Karame, E. Androulaki, and S. Capkun, "Double-spending fast payments in bitcoin," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 906–917.

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

[32] A. Kate, Y. Huang, and I. Goldberg, "Distributed key generation in the wild," Cryptology ePrint Archive, Report 2012/377, 2012, https://eprint.iacr.org/2012/377.

[33] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger." *IACR Cryptology ePrint Archive*, vol. 2017, p. 406, 2017.

[34] K. Lee, D. H. Lee, and M. Yung, "Aggregating cl-signatures revisited: Extended functionality and better efficiency," in *International Conference on Financial Cryptography and Data Security*. Springer, 2013, pp. 171–188.

[35] S. Lu, R. Ostrovsky, A. Sahai, H. Shacham, and B. Waters, "Sequential aggregate signatures, multisignatures, and verifiably encrypted signatures without random oracles," *Journal of cryptology*, vol. 26, no. 2, pp. 340–373, 2013.

[36] A. Lysyanskaya, R. L. Rivest, A. Sahai, and S. Wolf, "Pseudonym systems," in *International Workshop on Selected Areas in Cryptography*. Springer, 1999, pp. 184–199.

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

[38] S. Meiklejohn and R. Mercer, "Möbius: Trustless tumbling for transaction privacy," in *Proceedings of Privacy Enhancing Technologies*, 2018.

[39] C. Paquin and G. Zaverucha, "U-prove cryptographic specification v1. 1," *Technical Report, Microsoft Corporation*, 2011.

[40] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Annual International Cryptology Conference*. Springer, 1991, pp. 129–140.

[41] D. Pointcheval and O. Sanders, "Short randomizable signatures," in *Cryptographers Track at the RSA Conference*. Springer, 2016, pp. 111–126.

[42] S. C. Ramanna and P. Sarkar, "Efficient adaptively secure ibbe from the sxdh assumption," *IEEE Transactions on Information Theory*, vol. 62, no. 10, pp. 5709–5726, 2016.

[43] C. Reitwiessner, "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.

[44] T. Ruffing, P. Moreno-Sanchez, and A. Kate, "Coinshuffle: Practical decentralized coin mixing for bitcoin," in *ESORICS (2)*, ser. Lecture Notes in Computer Science, vol. 8713. Springer, 2014, pp. 345–364.

[45] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 459–474.

[46] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[47] The Guardian, "History of 5-Eyes Explainer," http://www.theguardian.com/world/2013/dec/02/history-of-5-eyes-explainer, 2013.

[48] 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.

[49] L. Valenta and B. Rowan, "Blindcoin: Blinded, accountable mixes for bitcoin," in *Financial Cryptography and Data Security*, M. Brenner, N. Christin, B. Johnson, and K. Rohloff, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 112–126.

[50] B. Waters, "Efficient identity-based encryption without random oracles." in *Eurocrypt*, vol. 3494. Springer, 2005, pp. 114–127.

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

APPENDIX A
SKETCH OF SECURITY PROOFS

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

a) Unforgeability: There are two possible ways for an adversary to forge a proof of a credential: (*i*) an adversary without a valid credential nevertheless manages to form a proof such that VerifyCred passes; and (*ii*), an adversary that has successfully interacted with fewer than t authorities generates a valid consolidated credential (of which they then honestly prove possession using ProveCred).

Unforgeability in scenario (*i*) is ensured by the soundness property of the zero-knowledge proof. For scenario (*ii*), running AggCred involves performing Lagrange interpolation. If an adversary has fewer than t partial credentials, then they have fewer than t points, which makes the resulting polynomial (of degree $t-1$) undetermined and information-theoretically impossible to compute. The only option available to the adversary is thus to forge the remaining credentials directly. This violates the unforgeability of the underlying blind signature scheme, which was proved secure by Pointcheval and Sanders [41] under the LRSW assumption [36].

b) Blindness: Blindness follows directly from the blindness of the signature scheme used during IssueCred, which was largely proved secure by Pointcheval and Sanders [41] under the XDH assumption [12]. There are only two differences between their protocol and ours.

First, the Coconut authorities generate the credentials from a group element $h = H(c_m)$ instead of from $g_1^{\tilde{r}}$ for random $\tilde{r} \in \mathbb{F}_p$. The hiding property of the commitment c_m, however, ensures that $H(c_m)$ does not reveal any information about m. Second, Pointcheval and Sanders use a commitment to the attributes as input to BlindSign (see Section III-C), whereas Coconut uses an encryption instead. The IND-CPA property, however, of the encryption scheme ensures that the ciphertext also reveals no information about m.

Concretely, Coconut uses Pedersen Commitments [40] for the commitment scheme, which is secure under the discret logarithm assumption. It uses El-Gamal for the encryption scheme in $\mathbb{G}_1$, which is secure assuming DDH. Finally, it relies on the blindness of the Pointcheval and Sanders signature, which is secure assuming XDH [12]. As XDH implies both of the previous two assumptions, our entire blindness argument is implied by XDH.

c) Unlinkability / Zero-knowledge: Unlinkability and zero-knowledge are guaranteed under the XDH assumption [12]. The zero-knowledge property of the underlying proof system ensures that ProveCred does not on its own reveal any information about the attribute m (except that it satisfies ϕ'). The fact that credentials are re-randomized at the start of ProveCred in turn ensures unlinkability, both between different executions of ProveCred and between an execution of ProveCred and of IssueCred.

APPENDIX B
MULTI-ATTRIBUTES CREDENTIALS

We present the cryptographic primitives behind the multi-attribute Coconut threshold issuance credential scheme described in Section III-E. As in Section III-D, 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 Kate *et al.* [32].

❖ **Setup**$(1^\lambda, q) \rightarrow (params)$**:** 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 $g_1, h_1, \ldots, h_q$ be generators of $\mathbb{G}_1$, and g_2 a generator of $\mathbb{G}_2$. The system parameters are $params = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2, h_1, \ldots, h_q)$.

❖ **TTPKeyGen**$(params, t, n, q) \rightarrow (sk, vk)$**:** Choose $(q+1)$ polynomials of degree $(t-1)$ with coefficients in $\mathbb{F}_p$, noted $(v, w_1, \ldots, w_q)$, and set:

$$(x, y_1, \ldots y_q) = (v(0), w_1(0), \ldots, w_q(0))$$

Issue a secret key sk_i to each authority $i \in [1, \ldots, n]$ as below:

$$sk_i = (x_i, y_{i,1}, \ldots, y_{i,q}) = (v(i), w_1(i), \ldots, w_q(i))$$

and publish their verification key vk_i computed as follows:

$$vk_i = (g_2, \alpha_i, \beta_{i,1}, \ldots, \beta_{i,q})) = (g_2, g_2^{x_i}, g_2^{y_{i,1}}, \ldots, g_2^{y_{i,1q}})$$

❖ **IssueCred**$(m_1, \ldots, m_q, \phi) \rightarrow (\sigma)$**:** Credentials issuance is composed of three algorithms:

❖ PrepareBlindSign$(m_1, \ldots, m_q, \phi) \rightarrow (d, \Lambda, \phi)$: The users generate an El-Gamal key-pair $(d, \gamma = g_1^d)$; pick a random $o \in \mathbb{F}_p$ compute the commitment c_m and the group element $h \in \mathbb{G}_1$ as follows:

$$c_m = g_1^o \prod_{j=1}^{q} h_j^{m_j} \qquad \text{and} \qquad h = H(c_m)$$

Pick at random $(k_1, \ldots, k_q) \in \mathbb{F}_p^q$ and compute an El-Gamal encryption of each m_j for $\forall j \in [1, \ldots, q]$ as below:

$$c_j = Enc(h^{m_j}) = (g_1^{k_j}, \gamma^{k_j} h^{m_j})$$

Output $(d, \Lambda = (\gamma, c_m, c_j, \pi_s), \phi)$ $\forall j \in [1, \ldots, q]$, where π_s is defined by:

$$\begin{aligned} \pi_s \quad = \quad & \text{NIZK}\{(d, m_1, \ldots, m_q, o, k_1, \ldots, k_q) : \gamma = g_1^d \\ & \wedge c_m = g_1^o \prod_{j=1}^{q} h_j^{m_j} \ \wedge \ c_j = (g_1^{k_j}, \gamma^{k_j} h^{m_j}) \\ & \wedge \phi(m_1, \ldots, m_q) = 1\} \quad \forall j \in [1, \ldots, q] \end{aligned}$$

❖ BlindSign$(sk, \Lambda, \phi) \rightarrow (\tilde{\sigma}_i)$: The authority i parses $\Lambda = (\gamma, c_m, c_j, \pi_s)$ and $c_j = (a_j, b_j)$ $\forall j \in [1, \ldots, q]$, and $sk_i = (x, y_1, \ldots, y_q)$. Recompute $h = H(c_m)$. Verify the proof π_s using γ, c_m and ϕ. If the proof is invalid, output $\perp$ and stop the protocol; otherwise output $\tilde{\sigma}_i = (h, \tilde{c})$, where $\tilde{c}$ is defined as below:

$$\tilde{c} = \left(\prod_{j=1}^{q} a_j^{y_j}, h^x \prod_{j=1}^{q} b_j^{y_j} \right)$$

❖ Unblind$(\tilde{\sigma}_i, d) \rightarrow (\sigma_i)$: The users parse $\tilde{\sigma}_i = (h, \tilde{c})$ and $\tilde{c} = (\tilde{a}, \tilde{b})$; compute $\sigma_i = (h, \tilde{b}(\tilde{a})^{-d})$. Output σ_i.

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

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

❖ **ProveCred**($vk, m_1, \ldots, m_q, \sigma, \phi'$) $\rightarrow$ (σ', Θ, ϕ')**:** Parse $\sigma = (h, s)$ and $vk = (g_2, \alpha, \beta_1, \ldots, \beta_q)$. Pick at random $r', r \in \mathbb{F}_q^2$; set $\sigma' = (h', s') = (h^{r'}, s^{r'})$, and build κ and ν as below:

$$\kappa = \alpha \prod_{j=1}^{q} \beta_j^{m_j} g_2^r \qquad \text{and} \qquad \nu = (h')^r$$

Output $(\Theta = (\kappa, \nu, \sigma', \pi_v), \phi')$, where π_v is:

$$\begin{aligned} \pi_v \quad = \quad & \text{NIZK}\{(m_1, \ldots, m_q, r) : \kappa = \alpha \prod_{j=1}^{q} \beta_j^{m_j} g_2^r \\ & \wedge \ \nu = (h')^r \ \wedge \ \phi(m_1, \ldots, m_q) = 1\} \end{aligned}$$

❖ **VerifyCred**(vk, Θ, ϕ') $\rightarrow$ ($true/false$)**:** Parse $\Theta = (\kappa, \nu, \sigma', \pi_v)$ and $\sigma' = (h', s')$; verify π_v using vk and ϕ'; Output $true$ if the proof verifies, $h' \neq 1$ and $e(h', \kappa) = e(s'\nu, g_2)$; otherwise output $false$.

APPENDIX C
ETHEREUM TUMBLER

We extend the example of the tumbler application described in Section V-A 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 on 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 that it has not already been spent, the contract sends v to the Ethereum destination address $addr$ (5).