MULTIDARKROOM

ZERO KNOWLEDGE MULTI PARTY SIGNATURES WITH APPLICATION TO DISTRIBUTED LEDGERS

Denis Roio
Dyne.org foundation
Amsterdam, 1013AK
J@Dyne.org

Alberto Ibrisevic
Laboratory of Cryptography
Trento University (stage)
bettowski@dyne.org

Andrea D'Intino
Dyne.org foundation
Amsterdam, 1013AK
Andrea@Dyne.org

February 28, 2021

ABSTRACT

Multidarkroom is a novel signature scheme supporting unlinkable signatures by multiple parties authenticated by means of a zero-knowledge credential scheme. Multidarkroom integrates with blockchains to ensure condidentiality, authenticity and availability even when credential issuing authorities are offline. We implement and evaluate a Multidarkroom smart contract for Zenroom and present an application related to multiple anonymous signatures by authenticated parties and their non-interactive verification Multidarkroom uses short and computationally efficient authentication credentials and signatures application scale linearly over multiple participants.

I Introduction

Multi-party computation applied to the signing process allows the issuance of signatures without requiring any of the participating parties to disclose secret signing keys to each other, nor requires the presence of a trusted third-party to receive them and compose the signatures. However, established schemes have shortcomings. Existing protocols do not provide the necessary efficiency, re-randomization or blind issuance properties necessary for the application to trustless distributed systems. Those managing to implement such privacy preserving features are prone to rogue-key attacks [Boneh et al., 2020] since they cannot grant that signatures are produced by legitimate key holders.

The lack of efficient, scalable and privacy-preserving signature schemes impacts distributed ledger technologies that support 'smart contracts' as decentralized or federated architectures where trust is not shared among all participants, but granted by one or more authorities through credential issuance for the generation of non-interactive and unlinkable proofs.

Multidarkroom applies to the signature process a mechanism of credential issuance by one or more authorities for the generation of non-interactive and unlinkable proofs, resulting in short and computationally efficient signatures composed of exactly two group elements that are linked to each other. The size of the signature remains constant regardless of the number of parties that are signing, while the credential is verified and discarded after signature aggregation. While being signed, duplicates may be avoided by collecting unlinkable fingerprints of signing parties, as they would invalidate the final result. Before being able to sign, a one-time setup phase is required where the signing party collects and aggregates a signed credential from one or more autorities. 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 [Sonnino et al., 2018].

Our evaluation of the Multiparty primitives shows very promising results. Session creation takes about $20ms$, while signing $73ms$ and verification $40ms$ on average consumer hardware.

Applications Multidarkroom provides a production-ready implementation that is easy to embed in end-to-end encryption applications. By making it possible for multiple parties to anonymously authenticate and produce untraceable signatures, its goal is to leverage privacy-by-design scenarios that minimize the information exchange needed for document authentication.

The participation to a signature will be governed by one or more issuers holding keys for the one-time setup of signature credentials:

1. Participant generates keys and a credential request
2. Issuer signs the credential request
3. Participant stores the credential in its keyring

Following this setup any participant will be able to produce a zero-knowledge proof of possession of the credential, giving access to the signature process.

The base application of Multidarkdroom is obviously the collective signature of digital documents, following such a flow:

1. One may choose a document and decide to indict its signature
2. One selects the participants to the signature and publishes it
3. Participants may chose to add their signature to the document
4. Anyone may verify the document is signed by all and only all participants

Going beyond a single document signing scenario, Multidarkroom can be used to implement a material passport for circular economy applications [Luscuere, 2017] to maintain the genealogy of a specific product, providing authenticated information about the whole set of actors, tools, collaborations, agreements, efforts and energy involved in its production, transportation and disposal [Dyne.org, 2020].

This paper makes three key contributions

- We describe the signature scheme underlying Multidarkroom, including how key generation, signing and verification operate (Section II). The scheme is an application of the BLS signature scheme [Boneh et al., 2018] fitted with features to grant the unlinkability of signatures and to secure it against rogue-key attacks.
- We describe the credential scheme underlying Multidarkroom, including how key generation, issuance, aggregation and verification of credentials operate (Section III). The scheme is an application of the Coconut credential scheme [Sonnino et al., 2018] that is general purpose and can be scaled to a fully distributed threshold issuance that is re-randomizable.
- We implement a Zencode scenario of Multidarkroom to be executed on and off-chain by the Zenroom VM, complete with functions for public credential issuance, signature session creation and multi-party non-interactive signing (Section IV). We evaluate the performance and cost of this implementation on on-site and on-line platforms leveraging end-to-end encryption (Section V).

We will adopt the following notations

- $\mathbb{F}_p$ is the prime finite field with p elements (i.e. of prime order p). In our case the prime is long 383 bits;
- E denotes the (additive) group of points of the curve BLS-383 [Scott, 2017] of order n which can be described with the Weierstrass form $y^2 = x^3 + 16$;
- E_T represents instead the group of points of the twisted curve of BLS-383, with embedding degree $k = 12$. The order of this group is also n;

We also require defining the notion of a cryptographic pairing. Basically it is a function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ are all groups of same order n, such that satisfies the following properties:

i. *Bilinearity*, i.e. given $P, Q \in \mathbb{G}_1$ and $R, S \in \mathbb{G}_2$, we have

$$e(P + Q, R) = e(P, R) \cdot e(Q, R)$$
$$e(P, R + S) = e(P, R) \cdot e(P, S)$$

ii. *Non-degeneracy*, meaning that for all $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$, $e(g_1, g_2) \neq 1_{\mathbb{G}_T}$, the identity element of the group $\mathbb{G}_T$;

iii. *Efficiency*, so that the map e is easy to compute;

iv. $\mathbb{G}_1 \neq \mathbb{G}_2$, and moreover, that there exist no efficient homomorphism between $\mathbb{G}_1$ and $\mathbb{G}_2$.

For the purpose of our protocol we will have $\mathbb{G}_1 = E_T$ and $\mathbb{G}_1 = E$, and $\mathbb{G}_T \subset \mathbb{F}_{p^{12}}$ is the subgroup containing the n-th roots of unity. Instead $e : E_T \times E \to \mathbb{G}_T$ is the *Miller pairing*, which in our code is referred as the method `miller(ECP2 P, ECP Q)`.

II Signature

A *BLS signature* is a signature scheme whose design exploits a cryptographic pairing. As for other well known algorithm such as ECDSA, it will work following these four main steps:

- **Key Generation phase.** For a user who wants to sign a message m, a secret key sk is randomically chosen uniformly in $\mathbb{F}_n$, where n is the order of the groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$. The corresponding public key pk is the element $sk \cdot G_2 \in E_T$;
- **Signing phase.** The message m is first hashed into the $U \in E$, which in our scheme is done by the method `hashtopoint`; the related signature is then given by $\sigma = sk \cdot P$;
- **Verification phase.** For an other user that wants to verify the authenticity and the integrity of the message m, it needs to
  1. parse m, pk and σ
  2. hash the message m into the point U and then check if the following identity holds,

$$e(pk, U) = e(G_2, \sigma)$$

If verification passes it means that σ is a valid signature for m and the protocol ends without errors.

Proof of the verification algorithm: By using the definitions of the elements involved and exploiting the property of the pairing e we have

$$\begin{aligned} e(pk, U) &= e(sk \cdot G_2, U) \\ &= e(G_2, U)^{sk} \\ &= e(G_2, sk \cdot U) \\ &= e(G_2, \sigma) \end{aligned}$$

□

BLS signatures present some interesting features. For instance, the length of the output σ competes to those obtained by ECDSA and similar algorithms; in our specific case, by using BLS-383 [Scott, 2017], it will be 32 Bytes long, which is typically a standard nowadays. Then, since this curve is also pairing-friendly, with the assumption made on e, the signature is produced in a very short time. Moreover, BLS supports also aggregation, that is the ability to aggregate a collection of multiple signatures σ_i (each one related to a different message m_i) into a singular new object σ, that can be validated using the respective public keys pk_i. This is possible thanks to the fact that $\sigma_i \in G_1 \forall i$, giving to the algorithm an homomorphic property. We will show now how this last feature can be attained in the context of a multi-party computation using the same message m but different partecipants.

Session Generation After the key generation step we introduce a new phase called **session generation**, where the signature is initialized; anyone willing to start a signing session on a message m will create:

1. a random r and its corresponding point $R = r \cdot G_2$
2. the sum of R and all pk supposed to participate to the signature such as $P = R + \sum_i pk_i$
3. the unique identifier `UID` of the session calculated as hash to point of the message m, such as $U = H(m) \in E$, where H is a combination of a cryptographic hash function (treated as a random oracle) together with an encoding into elliptic curve points procedure
4. the first layer of the signature $\sigma \leftarrow r \cdot U$, later to be summed with all other signatures in a multi-party computation setup resulting in the final signature as $\sigma \leftarrow r \cdot U + \sum_i sk_i \cdot U$
5. the array of unique fingerprints ζ_i of each signature resulting from the credential authentication (see section III)

After this phase is terminated, every partecipants involved in the session start their own signing phase, producing (from the same message m) their respective σ_i's. The final signature σ is then computed in this way: first of all let us call $\sigma_0 = R$, then supposing k partecipants have already aggrgated their σ_i, obtaing a partial signature S_k, the $(k+1)$-th one will compute

$$S_{k+1} = S_k + \sigma_k = \sum_{i=0}^{k+1} \sigma_i$$

Finally, the resulting output will be $\sigma = S_N$, where N is the total number of the signers of the session. In order to verify that σ is valid we compute $P = pk_R + \sum_{i=0}^{N} pk_i$, where $pk_R = r \cdot G_2$, that works as a public key with respect to the nonce r, which instead is kept secret. Verification is then performed by checking if the following identity holds

$$e(P, U) = e(G_2, \sigma)$$

If verification passes without errors it means that σ is a valid aggregated signature of m.

Proof. For simplicity we will prove the above statement for $N = 2$, since the general case can be obtained by an inductive argument. By recalling that $\sigma = R + \sigma_1 + \sigma_2$, $P = pk_R + pk_1 + pk_2$, by using the property of the pairing e we have

$$\begin{aligned} e(P, U) &= e(pk_R + pk_1 + pk_2, U) \\ &= e((r + sk_1 + sk_2) \cdot G_2, U) \\ &= e(G_2, U)^{r+sk_1+sk_2} \\ &= e(G_2, (r + sk_1 + sk_2) \cdot U) \\ &= e(G_2, R + \sigma_1 + \sigma_2) \\ &= e(G_2, \sigma) \end{aligned}$$

□

We conclude this section with a final consideration on this feature. We recall that in the generation of the aggregated signature σ we used as a starting point the variable $R = r \cdot G_2$, but in literature it is also common to find instead simply the base point G_2. The choice of randomizing it (providing that the random number generator acts as an oracle) helps in preventing replay attacks, since the signature generated by the process is linked to the session in which is produced, for if an attacker managed to get some information from σ, it would be difficult to use it in order to forge new signatures.

III Credential

Following the guidelines of Coconut, the credentials issuing scheme works as follows:

1. the issuer generates its own keypair (s_k, v_k), where $s_k = (x, y) \in \mathbb{Z}^2$ is the pair of secret scalars (the signing key) and $v_k = (\alpha, \beta) = (x \cdot G_2, y \cdot G_2)$ is the verifying key, made by the related pair of public points over the twisted elliptic curve of BLS-383 of embedding degree $k = 12$;
2. the user i, with its respective keys (sk_i, PK_i) make a credential request on its secret attribute $ck_i \in \mathbf{Z}$ to the issuer, sending λ which contains a zero-knowledge proof π_s of the authenticity of user i;
3. the issuer, after having received λ, verifies the proof π_s at its inside, and if it passes, then releases to user i a credential $\tilde{\sigma}$ signed used its own key sk.

Step 1. is self-explanatory. Steps 2. and 3. require a bit more effort, in fact in order to build a valid request λ, and so also a valid proof π_s, first of all the user must produce an hash digest $H(ck_i)$ for the attribute ck_i, that we call h, then computes two more variables c and s_h defined as

$$c = r \cdot G_1 + h \cdot HS$$
$$s_h = (a, b) = (k \cdot G_1, k \cdot \gamma + h \cdot c)$$

where r and k are fresh randomly generated integers and HS is an hard-encoded point on the curve E. These two variables are alleged in the credential request λ produced in `prepare_blind_sign` and are needed to the verifier to assure the authenticity of the user through the proof π_s, which requires as input h, k, r, c and $ck_i \cdot G_1$ (called also γ). The Non-Iteractive Zero Knowledge proof (for short NIZK proof) π_s generated by the function `blind_sign` is computed as follows:

- **Randomization phase.** Three new nonces $w_h, w_k, w_r \in \mathbb{Z}$ are generated, each one related to the input variables h, k, r respectively as we will show soon;
- **Challenge phase.** The protocol creates three commitment values, namely A_w, B_w, C_w defined as follows

  $$A_w = w_k \cdot G_1$$
  $$B_w = w_k \cdot \gamma + w_h \cdot c$$
  $$C_w = w_r \cdot G_1 + w_h \cdot HS$$

  Then these variables are used as input of a function φ producing an integer $c_h = \varphi(\{c, A_w, B_w, C_w\})$;
- **Response phase.** In order that the proof can be verified the protocol generates three more variables which are alleged inside the proof itself and link the nonces w_h, w_k, w_r with h, k, r, i.e:

  $$r_h = w_h - c_h h$$
  $$r_k = w_k - c_h k$$
  $$r_r = w_r - c_h r$$

So basically, the proof π_s contains the three response variables r_h, r_k, r_r and also the commitment value c_h, that can be used for a predicate ϕ which is true when computed on m. Once the verifier receives the request λ, in order to check if the proof is valid it should be able to reconstruct A_w, B_w, C_w by doing these computations,

$$\widehat{A}_w = c_h \cdot a + r_k \cdot G_1$$
$$\widehat{B}_w = c_h \cdot b + r_k \cdot \gamma + r_h \cdot c$$
$$\widehat{C}_w = c_h \cdot c + r_r \cdot G_1 + r_h \cdot HS$$

If the request is correct, then we will have that

$$\varphi(\{c, \widehat{A}_w, \widehat{B}_w, \widehat{C}_w\}) = \varphi(\{c, A_w, B_w, C_w\}) = c_h \quad (1)$$

and verification is thus complete, meaning that the verifier has right to believe that the prover actually owns the secret attribute ck_i associated to the public variable γ and that consequently has produced a valid commitment c and (ElGamal) encryption s; in other words,

$$\begin{aligned}\pi_s = \text{ NIZK}\{&(ck_i, h, r, k) : \\ &\gamma = ck_i \cdot G_1, \\ &c = r \cdot G_1 + h \cdot HS, \\ &s_h = (k \cdot G_1, k \cdot \gamma + h \cdot c), \\ &\phi(h) = 1\}\end{aligned}$$

At this point the user will now have a blind credential $\tilde{\sigma} = (c, \tilde{a}, \tilde{b})$ issued by the authority, where

$$\tilde{a} = y \cdot a$$
$$\tilde{b} = x \cdot c + y \cdot b$$

The user then will have to unblind it using its secret credential key, obtaining $\sigma = (c, s) = (c, \tilde{b} - ck_i(\tilde{a}))$, which will use to prove its identity when signing a message. The procedure is similar to the one seen before with some extra details:

- **Setup.** As for the the BLS signature, an elliptic point U, associated to the hash of the message to sign, is required as an Unique Identifier (`UID`) for the signing session;
- **Credential proving.** The user produces two cryptographical objects θ (containing a new proof π_v) and ζ (which is unequivocally associated to U) through `prove_cred_uid`, taking as input its own credential σ, the related secret attribute c_k, the authority public key $v_k = (\alpha, \beta)$ and the session point U.

The new objects θ and ζ are derived as follows:

- as before the user hashes ck into h, and this time generates two random values r and r';
- next, it randomizes its credential σ into $\sigma' = (c', s') = (r' \cdot c, r' \cdot s)$ and then computes two elliptic curve points κ and ν as

$$\kappa = \alpha + h \cdot \beta + r \cdot G_2$$
$$\nu = r \cdot c'$$

- finally, the resulting θ is the t-uple $(\kappa, \nu, \pi_v, \phi')$, where π_v is a valid zero knowledge proof of the following form

$$\begin{aligned}\pi_v = \text{ NIZK}\{&(h, r) : \\ &\kappa = \alpha + h \cdot \beta + r \cdot G_2 \\ &\nu = r \cdot c' \\ &\phi'(h) = 1\}\end{aligned}$$

with ϕ' being a predicate which is true on h, while ζ is simply the scalar multiplication $h \cdot U$.

Building the proof π_v requires similar steps as seen for π_s, in fact we create three commitment values A_w, B_w, C_w defined as

$$A_w = \alpha + w_h \cdot \beta + w_r \cdot G_2$$
$$B_w = w_r \cdot c'$$
$$C_w = w_h \cdot U$$

where w_h, w_r are fresh generated nonces; then we set the challenge as the computation of $c_h = \varphi(\{\alpha, \beta, A_w, B_w, C_w\})$ with the related responses

$$r_h = w_h - hc$$
$$r_r = w_r - rc$$

The values of r_h and r_r are stored inside π_v which will be then sent through θ (together with ζ) from the prover to the verifier. In order to check that the user has legitimately generated the proof and at the same time is the owner of the credential the following steps must be made:

1. extracting κ, ν, π_v (which is (r_h, r_r, c_h)) from θ;
2. reconstructing the commitments A_w, B_w, C_w as

$$\widehat{A}_w = c_h \cdot \kappa + r_r \cdot G_2 + (1 - c_h) \cdot \alpha + r_h \cdot \beta$$
$$\widehat{B}_w = r_r \cdot c' + c_h \cdot \nu$$
$$\widehat{C}_w = r_h \cdot U + c_h \cdot \zeta$$

3. checking either that

$$\begin{aligned}\varphi(\{\alpha, \beta, \widehat{A}_w, \widehat{B}_w, \widehat{C}_w\}) = \\ \varphi(\{\alpha, \beta, A_w, B_w, C_w\}) = c_h,\end{aligned} \quad (2)$$

that $c' \neq O$, the point at infinity, and that

$$e(\kappa, c') = e(G_2, s' + \nu) \quad (3)$$

where e is the Miller pairing function defined as

$$e : E_T \times E \to \mathbb{F}_{p^{12}}$$

Actually the predicate ϕ in the definition of pi_v can be thought as performing steps 2. and 3. and, if any of these fails the protocol will abort returning a failure, otherwise verification passes and the user can finally produce the signature.

Proof of the verification algorithms. We now show for the proof π_s that actually by using the responses r_h, r_k and r_r, together with c_h and the other parameters inside λ, i.e. $s = (a, b), c$ and γ, using also the hard coded point HS, it is possible to reconstruct the commitments A_w, B_w, C_w:

$$\begin{aligned}
\widehat{A}_w &= c_h \cdot a + r_k \cdot G_1 = c_h k \cdot G_1 + r_k \cdot G_1 \\
&= (c_h k + r_k) \cdot G_1 = (c_h k + w_k - c_h k) \cdot G_1 \\
&= w_k \cdot G_1 = A_w \\
\widehat{B}_w &= c_h \cdot b + r_k \cdot \gamma + r_h \cdot c \\
&= c_h \cdot (k \cdot \gamma + h \cdot c) + r_k \cdot \gamma + r_h \cdot c \\
&= (c_h k + r_k) \cdot \gamma + (c_h h + r_h \cdot c) \\
&= (c_h k + w_k - c_h k) \cdot \gamma + (c_h h + w_h - c_h h) \cdot c) \\
&= w_k \cdot \gamma + w_h \cdot c = B_w \\
\widehat{C}_w &= c_h \cdot c + r_r \cdot G_1 + r_h \cdot HS \\
&= c_h \cdot (r \cdot G_1 + h \cdot HS) + r_r \cdot G_1 + r_h \cdot HS \\
&= (c_h r + r_r) \cdot G_1 + (c_h h + r_h) \cdot HS \\
&= (c_h r + w_r - c_h r) \cdot G_1 + (c_h h + w_h - c_h h) \cdot HS \\
&= w_r \cdot G_1 + w_h \cdot HS = C_w
\end{aligned}$$

Regarding the second proof π_v, we have to prove both the identities (2) and (3) hold. We will focus only on the latter since the former requires a similar approach on what we have have done for π_s, but with different parameters involved (κ, ν, etc.). The LHS of the relation can be expressed as

$$\begin{aligned}
e(\kappa, c') &= e(\alpha + h \cdot \beta + r \cdot G_2, c') \\
&= e(x \cdot G_2 + hy \cdot G_2 + r \cdot G_2, \tilde{r} \cdot G_1) \\
&= e((x + hy + r) \cdot G_2, \tilde{r} \cdot G_1) \\
&= e(G_2, G_1)^{(x+hy+r)\tilde{r}}
\end{aligned}$$

using the substitution $c' = \tilde{r} \cdot G_1$, with $\tilde{r} \in \mathbb{F}_p$ since we know that $c' \in E$. For the RHS we have instead

$$\begin{aligned}
e(G_2, s' + \nu) &= e(G_2, r' \cdot s + r \cdot c') \\
&= e(G_2, r'(\tilde{b} - ck_i y(\tilde{a})) + r \cdot c') \\
&= e(G_2, r'(x \cdot c + y \cdot b - ck_i y \cdot a) + r \cdot c') \\
&= e(G_2, r'(x \cdot c + y \cdot b - ck_i y \cdot a + r \cdot c))
\end{aligned}$$

The second argument of the pairing can be rewritten as

$$\begin{aligned}
&r'(x \cdot c + y \cdot b - ck_i y \cdot a + r \cdot c) = \\
&r'(x \cdot c + y(k \cdot \gamma + h \cdot c) - (ck_i)yk \cdot G_1 + r \cdot c) = \\
&r'(x \cdot c + yh \cdot c + yk(ck_i) \cdot G_1 - yk(ck_i) \cdot G_1 + r \cdot c) = \\
&r'(x + yh + r) \cdot c
\end{aligned}$$

So, at the end

$$\begin{aligned}
e(G_2, s' + \nu) &= e(G_2, r'(x \cdot c + y \cdot b - dy \cdot a + r \cdot c)) \\
&= e(G_2, r'(x + yh + r) \cdot c) \\
&= e(G_2, (x + hy + r)\tilde{r} \cdot G_1) \\
&= e(G_2, G_1)^{(x+hy+r)\tilde{r}}
\end{aligned}$$

and (3) is finally proved. □

Security considerations. As mentioned in Coconut [Sonnino et al., 2018], BLS signatures and the proof system obtained with credentials are considered secure by assuming the existence of random oracles [Koblitz and Menezes, 2015], together with the decisional Diffie-Hellman Problem (DDH) [Boneh, 1998], the external Diffie-Hellman Problem (XDH), and with the Lysyanskaya-Rivest-Sahai-Wol Problem (LRSW) [Lysyanskaya et al., 1999], which are connected to the Discrete Logarithm. In fact, under these assumptions, we have that our protocol satisfies uforgeability, blindness, and unlinkability.

IV Implementation

In this section we illustrate our implementation of Multidarkroom keygen, sign and verify operations outlining for each:

1. the communication sequence diagram (figure)
2. the Zenroom code (Lua dialect script)
3. the Zencode statements

In addition to the above, a Setup operation will be briefly illustrated without the sequence diagram, as it includes the local creation of a keypair for the Issuer who will validate the credentials.

Setup: Generate the Issuer keys for credential signature

```
x = INT.random()
y = INT.random()
sk = { x = x,
       y = y  }
vk = { alpha = G2 * x,
       beta  = G2 * y  }
return { sign = sk,
         verify = vk }
```

Executed by the Zencode utterance:

When I create the issuer keypair

It will create a new *issuer keypair* that can be used to sign each new *credential request*. Its public member *.verify* should be public and know to anyone willing to verify the credentials of signers.

Keygen: Generate a credential request and have it signed by an Issuer, as well generate a BLS keypair used to sign documents. This procedure will generate private keys that should not be comunicated, as well public BLS keys that can be aggregated for signature verification.

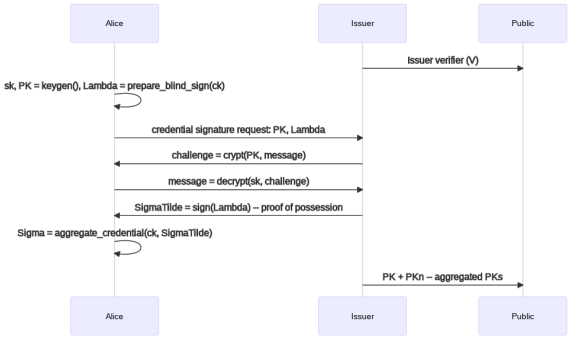
Figure 1 illustrates how this process consists of two different function calls: `keygen` to create an ElGamal keypair and `prepare_blind_sign` to generate a Coconut credential request. An interactive exchange takes place between the Signer and the Issuer that verifies the possession of the secret ElGamal key and signs a credential to witness this condition.

The public key of the Issuer which is used to sign the credential should have been public and known by the Signer at the beginning of the keygen process, while at the end of this process the public ElGamal key should be published, i.e. on a distributed ledger.

The following Zenroom implementation makes use of the Coconut built-in extension for zero-knowledge proof credentials.

```
ZK = require_once('crypto_abc')
```

Figure 1: Keygen process sequence diagram




```
issuer = ZK.issuer_keygen() -- setup
sk = INT.random() -- signing key
ck = INT.random() -- credential key
PK = G2 * sk      -- signature
    verifier
Lambda =
 ZK.prepare_blind_sign(ck * G1, ck)
SigmaTilde =
 ZK.blind_sign(issuer.sign, Lambda)
Sigma =
 ZK.aggregate_creds(ck, {SigmaTilde
    })
```

This code is executed in multiple steps by the Zencode utterances:

1. **When I create the credential keypair**
   will create a new *credential keypair* object containing members *public* (ECP) and *private* (BIG).
2. **When I create the credential request**
   will use the *credential keypair* to create a new *credential request* complex schema object for ZK proof.
3. **When I create the credential signature**
   will be executed by the Issuer after the proof-of-possession challenge is positive (exchange and confirmation of an encrypted message using BLS public keys) to sign the credential.
4. **When I create the credentials**
   will aggregate one or more *credential signature* (SigmaTilde) together with the *private* member of the *credential keypair* and finally create *credentials* capable of producing Zero-Knowledge proofs of possession.

Sign:

Verify:

```
---------
-- SETUP
---------
```

Figure 2: Signing process sequence diagram

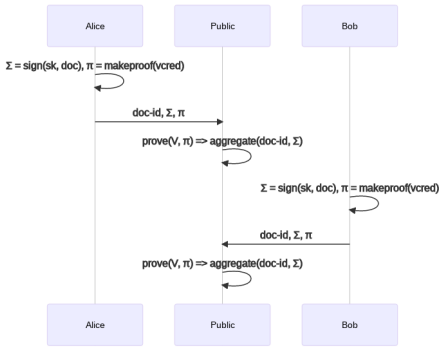
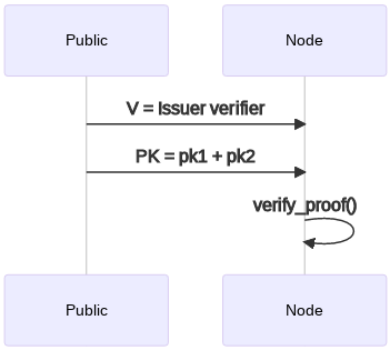


Figure 3: Verification process sequence diagram




```
G1 = ECP.generator()
G2 = ECP2.generator()

-- credentials
ZK = require_once('crypto_abc')
issuer = ZK.issuer_keygen()

-- keygen
sk1 = INT.random() -- signing key
ck1 = INT.random() -- credential key
PK1 = G2 * sk1     -- signature
   verifier

sk2 = INT.random()
ck2 = INT.random()
PK2 = G2 * sk2

-- issuer sign ZK credentials
Lambda1 = ZK.prepare_blind_sign(ck1*
   G1, ck1) -- credential request:
         p -> i
SigmaTilde1 = ZK.blind_sign(
   issuer.sign, Lambda1)    --
   issuer signs credential:  i -> p
Sigma1 = ZK.aggregate_creds(ck1, {
   SigmaTilde1})  -- credential
   sigma          p -> store
Lambda2 = ZK.prepare_blind_sign(ck2*
   G1, ck2)
SigmaTilde2 = ZK.blind_sign(
   issuer.sign, Lambda2)
Sigma2 = ZK.aggregate_creds(ck2, {
   SigmaTilde2})

-- sign

UID = ECP.hashtopoint(msg) -- the
   message's hash is the unique
   identifier

--------------
-- SETUP done
--------------

print "--------------------------"
print "first␣base␣signing␣session"
r = INT.random()
R = UID * r      -- session

-- add public keys to public session
    key
PM = (G2 * r) + PK1 + PK2

-- Session opener broadcasts:
-- 1. R   - base G1 point for
   signature session
-- 2. PM  - base G2 point for public
    multi-signature key
-- 3. msg - the message to be signed

-- proofs of valid signature
-- uses public session key as UID
Proof1,z1 = ZK.prove_cred_uid(
   issuer.verify, Sigma1, ck1, UID)
Proof2,z2 = ZK.prove_cred_uid(
   issuer.verify, Sigma2, ck2, UID)
-- each signer signs
S1 = UID * sk1
S2 = UID * sk2

-- generate the signature
-- each signer will communicate: UID
    * sk
SM = R + S1 + S2


-- print signature contents to
   screen
I.print({pub = PM, -- session public
    keys
                sign = SM,
                uid = UID,
                proofhash1 = sha256
                   ( ZEN.serialize(
                    Proof1 ) ),
```

```
                    proofhash2 = sha256
                        ( ZEN.serialize(
                         Proof2 ) ),
                    zeta1 = z1,
                    zeta2 = z2,
                    issuer =
                        issuer.verify
})

-- verify
assert( ZK.verify_cred_uid(
    issuer.verify, Proof1, z1, UID),
                    "first␣proof␣
                        verification␣
                        fails")
assert( ZK.verify_cred_uid(
    issuer.verify, Proof2, z2, UID),
                    "second␣proof␣
                        verification␣
                        fails")
assert( ECP2.miller(PM, UID)
                        == ECP2.miller(G2
                            , SM),
            "Signature␣doesn't␣validates
                ")
```

V Evaluation

This section lists results of evaluation benchmarks we have run using the Zencode scenario implementation of Multidarkroom on 4 target platforms:

- **X86 64bit** on a Intel i5 4th generation
- **ARM 64bit** on a Raspberry Pi 4 board
- **ARM 32bit** on a Raspberry Pi 0 board
- **WASM 32bit** on a build made with Emscripten

Our benchmarks provide lab measurements for the 3 main steps in the signature flow: for each of them is reported the time of execution (expressed in μs) and the size of output (expressed in bytes) in different conditions with a progressive number N of participants (5, 10, 50, 100 signatures).

Table 1: Execution timings on **X86 64bit** in μs

N	create	sign	verify
2	0.0188	0.0729	0.0405
10	0.0253	0.0730	0.0515
50	0.0410	0.1031	0.0996
100	0.0623	0.1373	0.1619

Table 2: Execution timings on **ARM 64bit** in μs

N	create	sign	verify
2	0.0605	0.2087	0.1049
10	0.0685	0.2273	0.1393
50	0.1200	0.3031	0.2978
100	0.1828	0.3930	0.4937

Table 3: Execution timings on **ARM 32bit** in μs

N	create	sign	verify
2	0.3333	1.0439	0.5753
10	0.4358	1.1458	0.7199
50	0.8473	1.5151	1.6056
100	0.1828	0.3930	2.4817

References

Dan Boneh, Sergey Gorbunov, Riad Wahby, Hoeteck Wee, and Zhenfei Zhang. BLS Signatures. Internet-Draft draft-irtf-cfrg-bls-signature-04, IETF Secretariat, September 2020. URL `http://www.ietf.org/internet-drafts/draft-irtf-cfrg-bls-signature-04.txt`.

Alberto Sonnino, Mustafa Al-Bassam, Shehar Bano, and George Danezis. Coconut: Threshold Issuance Selective Disclosure Credentials with Applications to Distributed Ledgers. *CoRR*, abs/1802.07344, 2018. URL `http://arxiv.org/abs/1802.07344`.

Lars Marten Luscuere. Materials passports: Optimising value recovery from materials. *Proceedings of the Institution of Civil Engineers - Waste and Resource Management*, 170(1):25–28, 2017. doi:10.1680/jwarm.16.00016. URL `https://doi.org/10.1680/jwarm.16.00016`.

Dyne.org. Reflow os. *constRuctive mEtabolic processes For materiaL flOWs in urban and peri-urban environments across Europe*, 2020. URL `https://reflowproject.eu/knowledge-hub/`.

Dan Boneh, Manu Drijvers, and Gregory Neven. Compact Multi-signatures for Smaller Blockchains. In *Advances in Cryptology – ASIACRYPT 2018*, volume 11273 of *Lecture Notes in Computer Science*, pages 435–464. Springer, 2018. doi:10.1007/978-3-030-03329-3_15.

Michael Scott. The apache milagro crypto library (version 2.2). 2017.

Neal Koblitz and Alfred J. Menezes. The random oracle model: a twenty-year retrospective. *Designs, Codes and Cryptography*, 77(2):587–610, Dec 2015. ISSN 1573-7586. doi:10.1007/s10623-015-0094-2. URL `https://doi.org/10.1007/s10623-015-0094-2`.

Dan Boneh. The decision diffie-hellman problem. Technical report, Berlin, Heidelberg, 1998.

Anna Lysyanskaya, Ronald L Rivest, Amit Sahai, and Stefan Wolf. Pseudonym systems. Technical report, 1999.

List of Figures