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

January 6, 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. Multidarkroom uses short and computationally efficient signatures and credentials whose verification takes the longest time to compute.

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. Some do not provide the necessary efficiency, re-randomization, or blind issuance properties necessary to implement the privacy preserving features necessary for the application to trustless distributed systems. Other schemes are prone to rogue-key attacks [Boneh et al., 2020] in the absence of authentication methods to grant that signatures are produced by legitimate key holders.

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

Multidarkroom uses 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 size grows linearly. Furthermure, after a one-time setup phase where the users collect and aggregate a threshold number of verification keys from the autorities, the attribute showing and verification O(1) in terms of both cryptographic computations and communication of cryptographic material, irrespective of the number of authorities.

Our evaluationof the Multiparty primities shows very promising results. Verifiction takes about 10ms, while signing a document is about 3 times faster.

Contributions: 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).

II Signature

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

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 $s_k = (x, y) \in \mathbf{Z}^2$ 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 m, then computes two more variables c and s defined as

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

where r and k are fresh randomly generated integers, G_1 is the base point of the elliptic curve BLS-383, and HS is an hard-encoded point of the same curve. 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 m, k, r, c and $ck_i \cdot G_1$ (called also γ). The Non-Iteractive Zero Knowlwdge proof (for short NIZK proof) π_s generated by the function `blind_sign` is computed as follows:

- **Randomization phase.** Three new nonces $w_m, w_k, w_r \in \mathbf{Z}$ are generated, each one related to the input variables m, 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
$$A_w = w_k \cdot G_1$$
$$B_w = w_k \cdot \gamma + w_m \cdot c$$
$$C_w = w_r \cdot G_1 + w_m \cdot HS$$
Then these variables are used as input of a function φ producing an integer $c_m = \varphi(\{A_w, B_w, C_w\})$;
- **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_m, w_k, w_r with m, k, r, i.e:
$$r_m = w_m - c_m m$$
$$r_k = w_k - c_m k$$
$$r_r = w_r - c_m r$$

So basically, the proof π_s contains the three response variables r_m, r_k, r_r and also the commitment value c_m, 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,

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

If the request is correct, then we will have that

$$\varphi(\{\widehat{A}_w, \widehat{B}_w, \widehat{C}_w\}) = \varphi(\{A_w, B_w, C_w\}) = c_m$$

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

$$\begin{aligned}\pi_s = \text{ NIZK}\{&(c_k, m, r, k) : \\ &\gamma = c_k \cdot G, \\ &c = r \cdot G_1 + m \cdot HS, \\ &s_m = (k \cdot G_1, k \cdot \gamma + m \cdot c), \\ &\phi(m) = 1\}\end{aligned}$$

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 $\sigma = (c, s) = (c, \tilde{b} - d(\tilde{a}))$, 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 H, 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 H) 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 H.

The new objects θ and ζ are derived as follows:

- as before the user hashes ck into m, 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

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

- 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

$$\pi_v = \text{NIZK}\{(m, r) : \kappa = \alpha + m \cdot \beta + r \cdot G_2 \quad \nu = r \cdot c' \quad \phi'(m) = 1\}$$

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

TODO: how to build π_v

IV Implementation

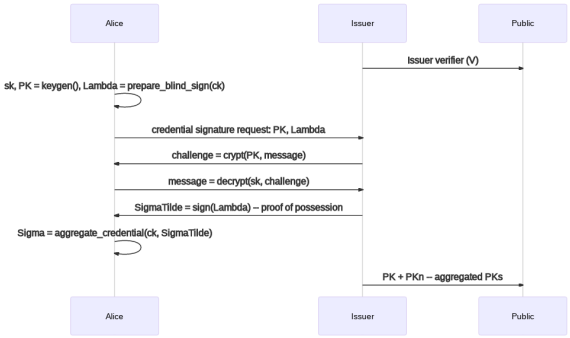
In this section we illustrate our implementation of Multidarkroom keygen, sign and verify operations outlining for each:

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

Figure 1: Keygen process sequence diagram




```
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 $prepareBlindSign$ 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')
issuer = ZK.issuer_keygen() -- setup
sk = INT.random() -- signing key
ck = INT.random() -- credential key
```

Figure 2: Signing process sequence diagram

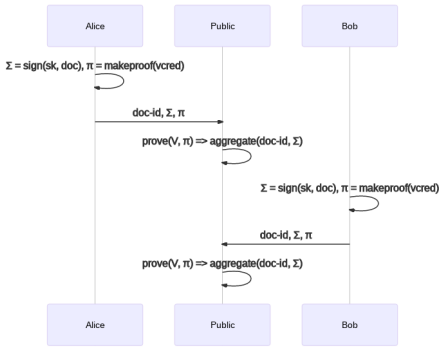
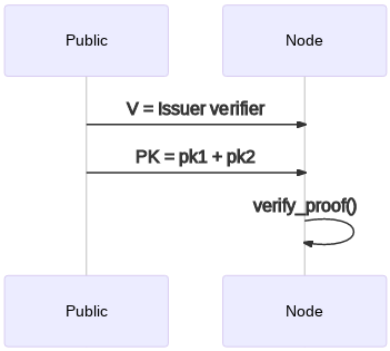


Figure 3: Verification process sequence diagram




```
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
---------

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

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

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`.

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.

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`.

List of Figures