REFLOW

ZERO KNOWLEDGE MULTI PARTY SIGNATURES WITH APPLICATION TO DISTRIBUTED AUTHENTICATION

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

April 30, 2021

ABSTRACT

Reflow is a novel signature scheme supporting unlinkable signatures by multiple parties authenticated by means of a zero-knowledge credential scheme. Reflow integrates with blockchains to ensure confidentiality, authenticity and availability even when credential issuing authorities are offline. We implement and evaluate a Reflow smart contract for Zenroom and present an application related to multiple anonymous signatures by authenticated parties and their non-interactive verification. Reflow 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 trust-less 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.

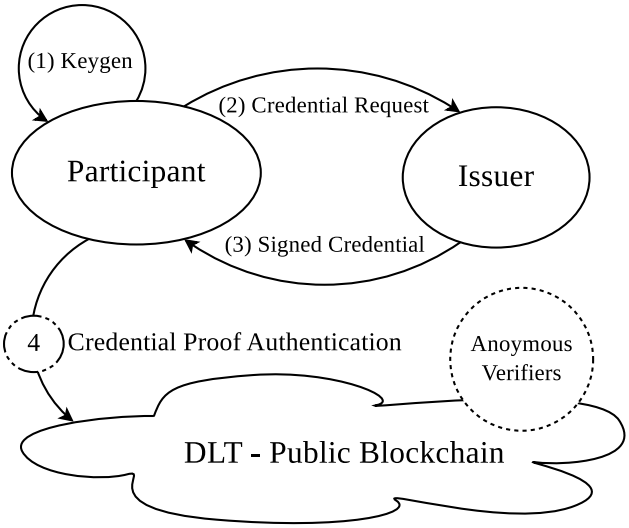
Reflow applies to the signature process a mechanism of credential issuance by one or more authorities for the generation of non-interactive and unlinkable proofs, resulting in short and computationally efficient signatures composed of exactly two group elements that are linked to each other. The size 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 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 [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.

Overview

Reflow 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.

Figure 1: Basic credential authentication



The participation to a signature will be governed by one or more issuers holding keys for the one-time setup of signature credentials. The steps outlined below are represented in figure 1:

① Participant generates keys
② Participant sends a credential request to Issuer
③ Issuer signs the credential request and sends it back
④ Participant can create anonymous credential proofs

Following this setup any participant will be able to produce a zero-knowledge proof of possession of the credential, which can be verified by anyone anonymously on the blockchain.

The base application of Multidarkdroom is obviously the collective signature of digital documents and the signed credential proof will be a requirement to participate to any signature process.

A Reflow signature process is best described in 3 main steps: session creation, signature and verification.

1. Anyone creates a session A session may be created by anyone, no credentials are required, but only information that should be public: the public keys of participants holding a credential to sign, the public verifier of the issuer who has signed the credentials and at last a document to be signed. The steps below are represented in figure 2 and illustrate how a signature session is created:

① Participants will publish their public keys, available to anyone
② Anyone may indict a signature session by selecting a document, the public keys of signing participants and the issuer
③ The signature session is then published without disclosing the identity of any participant

Figure 2: Session Creation

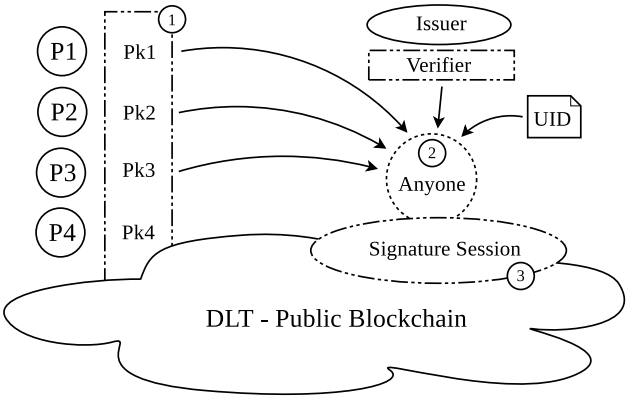
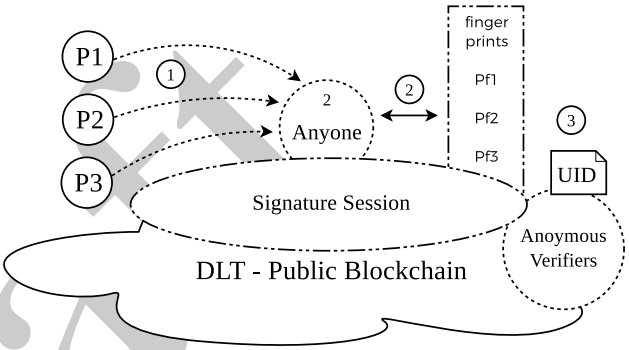


Figure 3: Sign and Verify



A Reflow signature session can then be published for verification and its existance may be confidentially communicated to the participants elected to sign it. The possession of the session will allow to disclose the identity of the participants, but only that of the Issuer and the document (or the hash of it) signed.

2. Participants sign the session Only elected participants that were initially chosen to sign the session may sign it, this is forced through credential authentication. Whenever they know about the existance of a session requesting their signature, they may chose to sign it. The steps below are illustrated in figure 3.

① Participants may be informed about the signature session and may create an anonymous signature to be added to the session
② Anyone may check that the anonymous signature is authentic and not a duplicate, then adds it to the session
③ Anyone may be informed about the signature session and be able to verify if the document is signed by all and only all participants

A delicate aspect of BLS signatures is avoiding double-signing: if a participant signs twice then the whole signature will never be valid. Relying on a stateless credential authentication alone does not avoid this case in Reflow, therefore we use a list of anonymous "fingerprints" of the

signature related to the document being signed. Each signature will produce a participant fingerprint saved in a list that is checked against duplication before adding a new signature. This procedure adds significant computational overhead for sessions with a large number of participants, but it can be switched off in a system that avoids double-signing in its own architecture.

Anyone verifies signatures Until the session will have collected all the signatures of participants, its verification will not be valid. It is also impossible to know if all participants have signed the session or how many are missing. Anyone can verify the state of the signature session in any moment just by having the document and the session, as illustrated in figure 3 along with the signature process.

Configurable features may be introduced in the Reflow signature flow that may or may not disclose more information, for instance who has indicted the signature session, what documents are linked to signatures and how many participants were called to sign: this depends from the implementation and the metadata it may add to signature sessions or the communication protocols adopted. In any case the basic signature and verification flow of Reflow requires that only one identity is really made public and is that of the issuer.

Applications

Moving further in envisioning the possibilities opened by Reflow is important to state the possibility to aggregate (sum) signatures into compact multi-signatures, a core feature of our BLS based signature scheme [Boneh et al., 2018a].

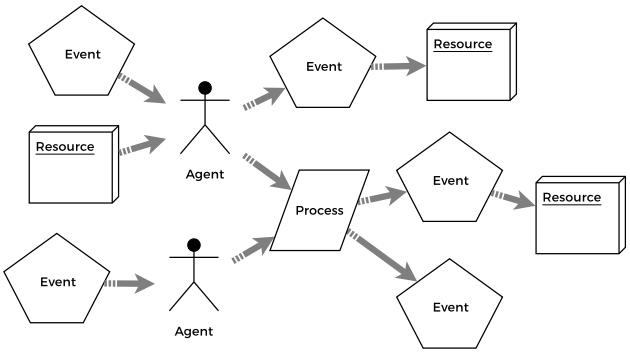
Need to Know. The base implementation exploiting this feature is that of a signature scheme for a single document split in separate sections to be signed by different participants: all the signatures can be later aggregated in a single one proving the whole document has been signed by all participants, without being disclosed to all of them in its entirety. This application helps to enforce the principles of need-to-know and least privilege to the access of information [Saltzer and Schroeder, 1975] and is useful for the realization of privacy-aware applications in various sectors, for instance for medical and risk mitigation analysis.

Disposable Identities A Disposable Identity is based on four properties common to other authentication and identification systems: verifiability, privacy, transparency, trustworthiness; then in addition introduces a fifth property: disposability. Disposability permits purpose-specific and context-driven authentication, to avoid linking the same identity across different authentication contexts [Goulden et al., 2021].

Reflow can be adopted by such an application to remove the need of context-free identifiers and implement authentication functions through a disposable identity whose traceability is bound to a context UID. Furthermore, being a signature scheme, Reflow can add the feature of context-free verifiable signatures (and multi-signatures) that are untraceable in public, but can be traced and even revoked in a known context by the means of context-specific signature fingerprints.

This scenario is relevant for the implementation of privacy-preserving public sector applications that allow authentication and signatures through disposable identity systems.

Figure 4: Valueflows



Material Passport. Drawing on feature of multiple signature aggregation, Reflow 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].

The provision of the information that forms the content of the material passport should be done by every actor in the supply chain and among the most important technical necessities for such an application are the confidentiality issues regarding access to information and the guarantees of the quality of information [Damen, 2012].

As an ideally simple and effective ontology we adopt ValueFlows to organize knowledge in a graph made of 4 main type of nodes:

① Events that are the mean of creation and transormation of Resources

② Agents capable of creating Events and Processes

③ Processes where more than one Agent can consume and create Events

We then consider Resources as material passports made of the track and trace of all nodes - Events, Agents and Processes - they descend from. The material passport is an authenticated graph structure: in figure 4 the Resources on the right side have an UID which is the aggregation of all UIDs of elements leading to their existance.

The integrity of the material passport can be verified by recalculating the UID aggregation and see it matches the signed one attached to the Resource. In case one or more

UIDs are wrong or missing, the Resource will not verify as valid. In brief is possible that:

① One or more Agents may interact to create material passports

② A material passport is the aggregation of all parent nodes

③ Anyone may verify the integrity and validity of a material passport

④ The verification of a material passport does not reveal the identity of Agents signing it

⑤ One may export and import a material passport as a graph query

Reflow's unlinkability of credentials and signatures satisfy the privacy requirement for the material passport, while the possibility to aggregate and link all the elements of its graph allow to group multiple signatures into a single compact one, without requiring any interaction with the previous signers. The material passport signature will then be the sum of all Agents, Processes and Events involved, created or consumed for it. The Reflow material passport is the authenticated, immutable and portable track record of all nodes connected in its graph: material passports can be signed on export and verified on import, which makes them reliably portable from one graph to another in a federated environment.

This paper makes three key contributions

- We describe the signature scheme underlying Reflow, including how key generation, signing and verification operate (Section II). The scheme is an application of the BLS signature scheme [Boneh et al., 2018b] fitted with features to grant the unlinkability of signatures and to secure it against rogue-key attacks.
- We describe the credential scheme underlying Reflow, 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 issuance that is re-randomizable.
- We implement a Zencode scenario of Reflow 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 VI).

Notations and assumptions

We will adopt the following notations:

- $\mathbb{F}_p$ is the prime finite field with p elements (i.e. of prime order p);
- E denotes the (additive) group of points of the curve BLS-383 [Scott, 2017] 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 the same of that of E;

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 $P_1, Q_1 \in \mathbb{G}_1$ and $P_2, Q_2 \in \mathbb{G}_2$, we have

$$e(P_1 + Q_1, P_2) = e(P_1, P_2) \cdot e(Q_1, P_2)$$
$$e(P_1, P_2 + Q_2) = e(P_1, P_2) \cdot e(P_1, Q_2)$$

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 $\mathbb{G}_2 = E$, and $\mathbb{G}_T \subset \mathbb{F}_{p^{12}}$ is the subgroup containing the n-th roots of unity, where n is the order of the groups E and E_T. Instead $e : E_T \times E \to \mathbb{G}_T$ is the *Miller pairing*, which in our work is encoded as the method `miller(ECP2 P, ECP Q)`.
To conclude, the credential scheme in section III uses non-interactive zero-knowledge proofs (for short NIZK proofs) to assert knowledge and relations over discrete logarithm values. They will be represented using the notation introduced by Camenisch and Stadler [1997] as

$$\text{NIZK}\{(x, y, \dots) : textstatementsaboutx, y, \dots\}$$

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 three 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 point $U \in E$, which in our scheme is done by the method `hashtopoint`; the related signature is then given by $\sigma = sk \cdot U$;
- **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, meaning that (with the assumption made on e) signature and verification are obtained in 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 in a suitable way. 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 participants.

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 participants involved in the session start their own signing phase during the session, 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 $\sigma_0 = r \cdot U$, then supposing k participants have already aggrgated their σ_i, obtaining 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 $P = R + \sum_{i=0}^{N} pk_i$, where $R = r \cdot G_2$, working 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. By recalling that $\sigma = r \cdot U + \sum_{i=1}^{N} \sigma_i$, $P = R + \sum_{i=1}^{N} pk_i$, by using the property of the pairing e we have

$$\begin{aligned} e(P, U) &= e(R + \sum_{i=1}^{N} pk_i, U) \\ &= e((r + \sum_{i=1}^{N} sk_i) \cdot G_2, U) \\ &= e(G_2, U)^{r + \sum_{i=1}^{N} sk_i} \\ &= e(G_2, (r + \sum_{i=1}^{N} sk_i) \cdot U) \\ &= e(G_2, r \cdot U + \sum_{i=1}^{N} \sigma_i) \\ &= e(G_2, \sigma) \end{aligned}$$

□

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 $\sigma_0 = r \cdot U$, 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 E_T;
2. the user i, with its respective keys (sk_i, PK_i) make a credential request on its secret attribute $ck_i \in \mathbb{Z}$ to the issuer, represented by 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 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, HS is an hard-encoded point on the curve E, and $\gamma = ck_i \cdot G_1$. 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. 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 h. 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 un-blind it using its secret credential key, obtaining $\sigma_{ck} = (c, s) = (c, \tilde{b} - ck_i(\tilde{a}))$, 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_{ck} into $\sigma'_{ck} = (c', s') = (r' \cdot c, r' \cdot s)$ 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, θ will be 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;
- ζ will be instead the elliptic curve point obtained as $h \cdot U \in E$.

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

$$\begin{aligned}A_w &= \alpha + w_h \cdot \beta + w_r \cdot G_2 \\ B_w &= w_r \cdot c' \\ C_w &= w_h \cdot U\end{aligned}$$

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

$$\begin{aligned}r_h &= w_h - hc \\ r_r &= w_r - rc\end{aligned}$$

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

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

3. checking either that

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

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

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

Actually the predicate ϕ in the definition of π_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 left-hand side 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 right-hand side 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 unforgeability, blindness, and unlinkability.

Reserves can be made about the maturity of pairing-based ellipcic curve cryptography despite various efforts to measure its security and design curve parameters that raise it, it is reasonable to consider this as a pioneering field of cryptography in contrast to well tested standards.

In addition to considerations on the maturity of EC, the future growth of quantum-computing technologies may be able to overcome the Discrete Logarithmic assumptions by qualitatively different computational means. Reflow then may be vulnerable to quantum-computing attacks, as well hard to patch, because the pairing-based design sits at its core with the adoption of ATE / Miller loop pairing of curves in twisted space, a practice that is not covered by research on quantum-proof algorithms and will eventually need more time to be addressed; however this is all speculative reasoning on what we can expect from the future.

The Reflow implementation we are presenting in this paper and that we have published as a Zenroom scenario ready to use is based on the BLS383 curve [Scott, 2017] that in the current implementation provided by the AMCL library has shown to pass all lab-tests regarding pairing properties, a positive result that is not shared with the slightly different BLS381-12 curve adopted by ETH2.0. Debating the choice of BLS381 is well beyond the scope of this paper, but is worth mentioning that our lab-tests have proved also the BLS461 curve to work in Reflow: it is based on a 461 bit prime and hence upgrades our implementation to 128 bit security [Barbulescu and Duquesne, 2019] against attacks looking for discrete logs on elliptic curves [Lim and Lee, 1997].

At last the complexity and flexibility of Reflow in its different applications, its optional use of fingerprint lists, multiple UIDs saved from aggregation and other features covering the different applications also represent a security risk in the technical integration phase. We believe that the adoption of Zenroom and the creation of a Zencode scenario addresses well this vulnerability by providing an easy to use integrated development environment (apiroom.net) and a test-bed for the design of different scenarios of application for Reflow that can be deployed correctly granting end-to-end encryption and data minimization according to privacy by design guidelines [Hoepman et al., 2019].

IV Implementation

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

- the communication sequence diagram
- the Zencode statements composing the sequence

Zencode is actual code executed inside the Zenroom VM, behind its implementation the algorithms follow closely the mathematical formulation explained in this article and can be reviewed in the free and open source code published at Zenroom.org: the Reflow Zencode scenario implementation is contained inside the Zenroom source code.

Credential Setup

We begin with the Credential Setup sequence which has to be executed only once to enable participants to produce signatures. This sequence is briefly illustrated with a diagram and it consists in the creation of keypair for both the Issuer, who will sign the participant credentials, and the participant who will request an issuer credential.

Listing 1: Issuer Keygen

```
Given I am 'The Issuer'
When I create the issuer key
Then print my 'keys'
```

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 communicated, as well public BLS keys that can be aggregated for signature verification.

Figure 5 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.

Figure 5: Keygen process sequence diagram

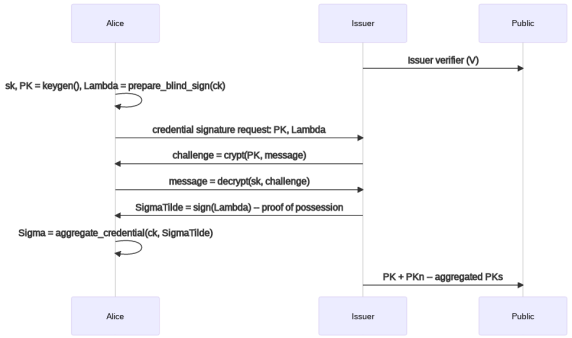
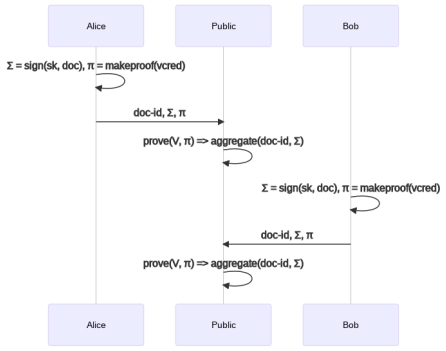


Figure 6: Signing process sequence diagram



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
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.

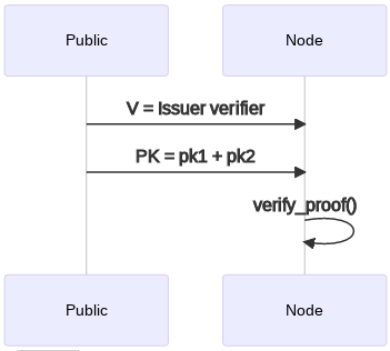
③ **When I create the credential signature**
will be executed by the Issuer after the proof-of-possession challenge is positive (exchange and confirmation of an encrypted message using BLS public keys) to sign the credential.

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:

Figure 7: Verification process sequence diagram




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

This section lists results of evaluation benchmarks we have run using the Zencode scenario implementation of Reflow 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 milliseconds) and the size of output (expressed in bytes) in different conditions with a progressive number σ_N of participants (5, 10, 50, 100 signatures).

INIZIO ANDREA

VI Evaluation

The goal of this section was to produce benchmarks of the implementation of the crypto flow, in realistic conditions of use, so in a way that is similar to how a software solution would use the software.

Our approach was opposite to testing single algorithms in a sandbox or a profiler, instead we tested Zenroom scripts, written in Zencode, that include also loading and parsing input data from streams or file system, as well as producing output as a deterministically formed JSON object.

Platforms

The three target platforms used for the benchmarks were:

- **X86-64** on Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz, running Ubuntu 18.04 (64 bit)
- **ARM 32bit** on Raspberry Pi 4 board, running Raspberry Pi OS (32bit, kernel version: 5.10)
- **ARM 32bit** on Raspberry Pi 0 board, running Raspberry Pi OS (32bit, kernel version: 5.10)

The the X86-64 machine runs on a Lenovo X250 laptop, with 8GB of RAM. The Raspberry 0 and 4 have been chosen as benchmarking platforms because of their similarities to, respectively, very low-cost IoT devices (5 USD) and very low cost mobile phones (sub 100 USD).

Builds

Being a self-contained application written in C, Zenroom can be built for several CPU architectures and operative systems, both as command line interface (CLI) application and as a library. The configurations We chose for this benchmark are:

- Two binary Zenroom CLI, compiled on the GCC toolchain in native 32bit ELF binaries, once for the X86-64 platform and once for the ARMv7-32bit. We refer to these builds as **Zenroom CLI**.
- One mixed library, transcompiled to WASM using the Emscripten toolchain, then built into an NPM package along with a JavaScript wrapper. At build time, the compiled WASM is converted to base64 and embedded in the JavaScript wrapper, which unpacks it at run-time. The library runs in browsers (using the native WASM support, currently present in Chromium/Chrome, Firefox and Safari), as well in node JS based application. We refer to this build as **Zenroom WASM**.

CLI and WASM use cases

The use case the CLI application is typically a server-side application or micro-service. The WASM library can run iin the browser, as the client side of a web application, in a server-side application based on node JS or a mobile application.

Zenroom may also be built as a native library for Android and iOS (on ARM 32bit, ARM 32bit or X86), but no benchmarks of the Reflow crypto flow have been performed using Zenroom as a native library. In other benchmarks, not published on this paper, we noted similar performances for the native CLI and the native library versions of Zenrooms: we may therefore assume (!!!! VA BENE???)

Testing

Benchmarking of Zenroom CLI and Zenroom WASM required using two different tools. Both tools performed chained execution of the Zenroom scripts and allowed for parametrized execution, where the amount of participants and the number of recursion cycles can be configured.

- The testing of Zenroom CLI was executed using a single, self-contained bash script. The script outputs the duration of the execution of each Zenroom scripts, along with the memory usage and size of output data, in a CSV formatted summary. The script also saves to files both the data and scripts for quality control reference.
- The testing of Zenroom WASM required building a JavaScript application, running in node JS. The application outputs the duration of the execution of each Zenroom scripts, along with the memory usage, it calculate averages and retursn output in a JSON formatted summary.

Benchmarks for the crypto flow were run on both the CLI and the WASM builds of Zenroom, for each platform, for a total of six different data collections.

Scripts description

Our benchmarks provide measurements for the all the steps in the signature flow: for each of them is reported the

time of execution (expressed in seconds) and the size of output (expressed in bytes) in different conditions with a progressive number of participants (5, 10, 50, 100, 1000 signatures).

The scripts used in the flow and for bench-marking can be divided in three categories:

- Scripts with **variable execution times** executed by anyone and marked with (A), include the creation of the Reflow seal (Session start), the aggregation of the signatures (Collect sign) and the verification of the signatures on Reflow seal. Their duration is proportional to the amount of participants and can vary greatly. These scripts are the most calculation intensive ones and expected to typically run server-side.
- Scripts executed by each participant, marked with (P), include participant's setup and signing, they're duration is not correlated with the amount of participants. These scripts are expected to typically run in the browser or on mobile devices.
- Scripts executed by the issuer, marked (I), include issuer's setup and signing, they're duration is not correlated with the amount of participants. These scripts are expected to typically run server-side.

Each scripts was executed 50 times, on each platform and each configuration, an the average execution time was extracted.

Findings

- The execution time of the scripts in the (A) group, grows linearly with the amount of participants.
- On a X86-64 machine, the benchmarks execute on Zenroom CLI have a comparable duration with their counterparts run on WASM. The execution times differ otherwise greatly between Zenroom CLI and Zenroom WASM, on the ARM 32bit based Raspberry Pi 0 and 4 machines. While investigating the cause of the differences is beyond the purpose of this paper, we speculate this as signaling different levels of optimization, for the of WASM interpreters built for X86-64 on one hand, and ARM 32bit on the other.
- The most numerically outstanding benchmark is the execution time of the issuer's keygen script on Raspberry Pi 0, when compared with the other platform. Once more, investigating the reason for this discrepancy is beyond the scope of this paper: we speculate ECP2 pairing performed in script may justify it.

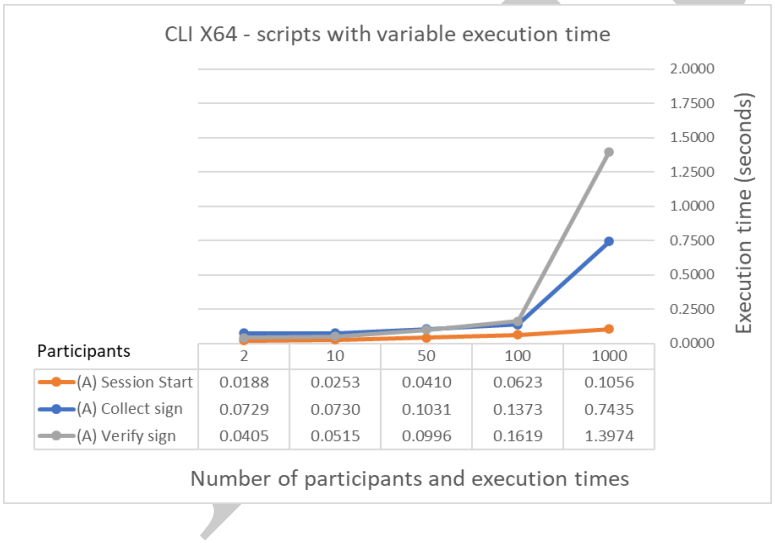
VII Benchmarks

Following are the benchmarks of the benchmarks for the (A) group of scripts (whose execution times change based on the amount of participants), running in CLI binaries, grouped by platform.

Zenroom CLI X86-64

Table 1: Execution timings on **CLI X64** in *seconds* per *participant*

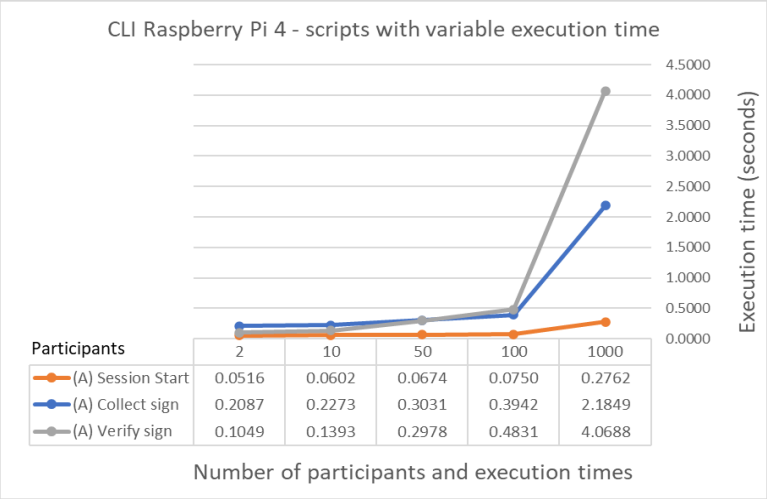
S/P	**2**	**10**	**50**	**100**	**1000**	**5000**
(A) Session start	0.0188	0.0253	0.0410	0.0623	0.1056	0.5039
(A) Collect sign	0.0729	0.0730	0.1031	0.1373	0.7435	3.6061
(A) Verify sign	0.0405	0.0515	0.0996	0.1619	1.3974	7.0460



Zenroom CLI Raspberry Pi 4

Table 2: Execution times on **CLI Raspberry 4** in *seconds* per *participant*

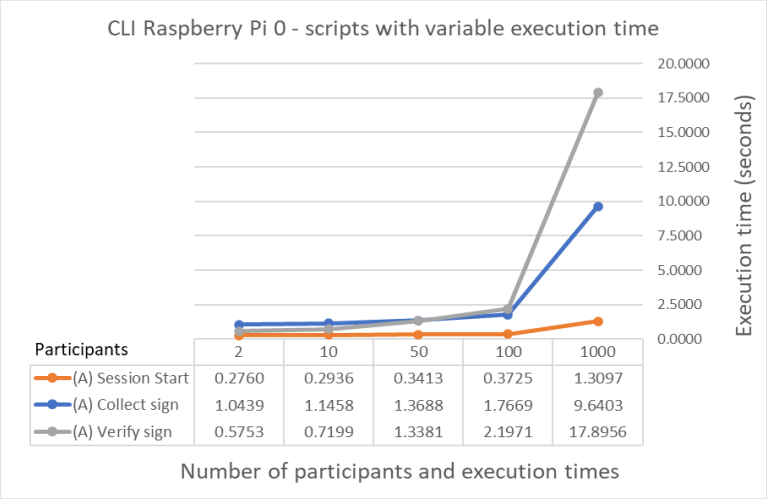
S/P	**2**	**10**	**50**	**100**	**1000**
(A) Session start	0.0516	0.0602	0.0674	0.0750	0.2762
(A) Collect sign	0.2087	0.2273	0.3031	0.3942	2.1849
(A) Verify sign	0.1049	0.1393	0.2978	0.4831	4.0688



Zenroom CLI Raspberry Pi 0

Table 3: Execution times for**Zenroom CLI Raspberry Pi 0** in *seconds* per *participant*

S/P	**2**	**10**	**50**	**100**	**1000**
(A) Session start	0.2760	0.2936	0.3413	0.3725	1.3097
(A) Collect sign	1.0439	1.1458	1.3688	1.7669	9.6403
(A) Verify sign	0.5753	0.7199	1.3381	2.1971	17.8956



Zenroom CLI - all platforms

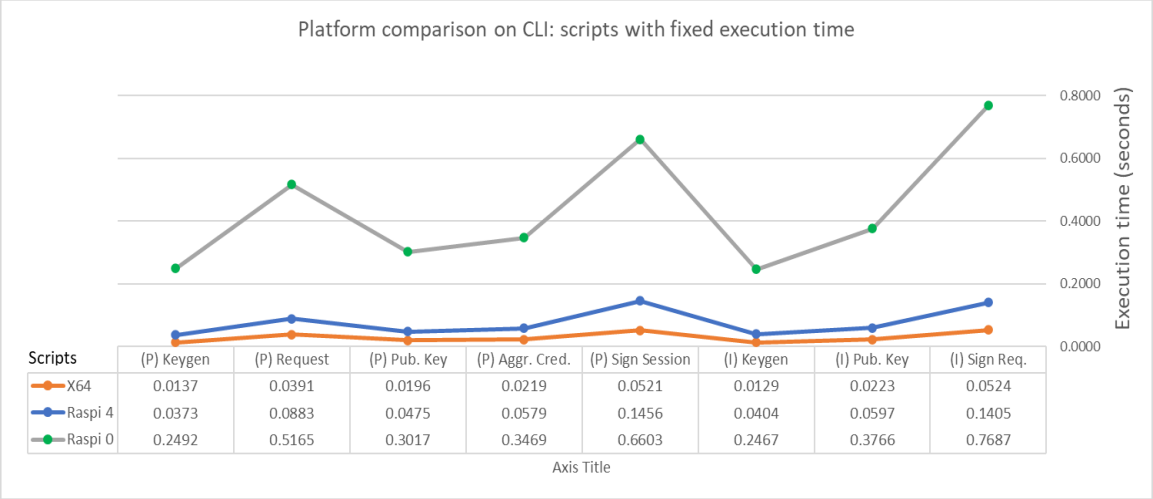
Following are the benchmarks of the benchmarks for the (P) and the (I) groups of scripts (whose execution times don't change based on the amount of participants), with a comparison of all the platforms, running in CLI binaries.

Table 4: Execution timings on **CLI** of scripts in $seconds$ per $platform$

$script/platf.$	**(P) Keygen**	**(P) Request**	**(P) Pub-Key**	**(P) Aggr. Cred.**	**(P) Sign Session**
X86-X64	0.0137	0.0391	0.0196	0.0219	0.0521
Raspberry Pi 4	0.0373	0.0883	0.0475	0.0579	0.1456
Raspberry Pi 0	0.2492	0.5165	0.3017	0.3469	0.6603

Table 5: Execution timings on **CLI** of scripts in $seconds$ per $platform$

$script/arch$	**(I) Keygen**	**(I) Pub-Key**	**(I) Sign Req.**
X64	0.0129	0.0223	0.0524
Raspberry Pi 4	0.0404	0.0597	0.1405
Raspberry Pi 0	0.2467	0.3766	0.7687

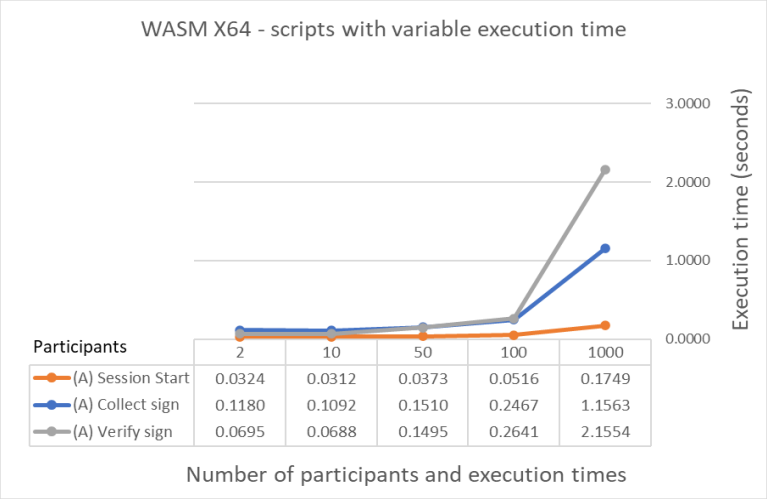


Following are the benchmarks of the benchmarks for the (A) group of scripts (whose execution times change based on the amount of participants), running in WASM libraries, grouped by platform.

Zenroom WASM X86-64

Table 6: Execution timings on **WASM X64** in $seconds$ per $participant$

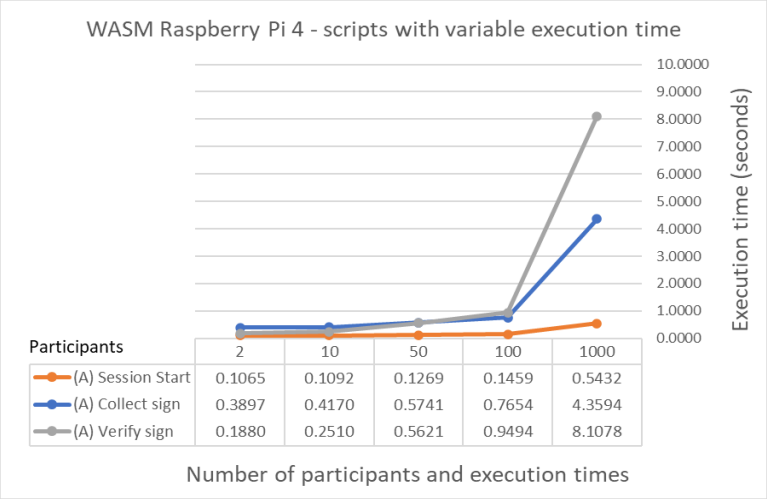
S/P	**2**	**10**	**50**	**100**	**1000**
(A) Session start	0.0324	0.0312	0.0373	0.0516	0.1749
(A) Collect sign	0.1180	0.1092	0.1510	0.2467	1.1563
(A) Verify sign	0.0695	0.0688	0.1495	0.2641	2.1554



Zenroom WASM Rasperry Pi 4

Table 7: Execution timings on **WASM Raspberry Pi 4** in *seconds* per *participant*

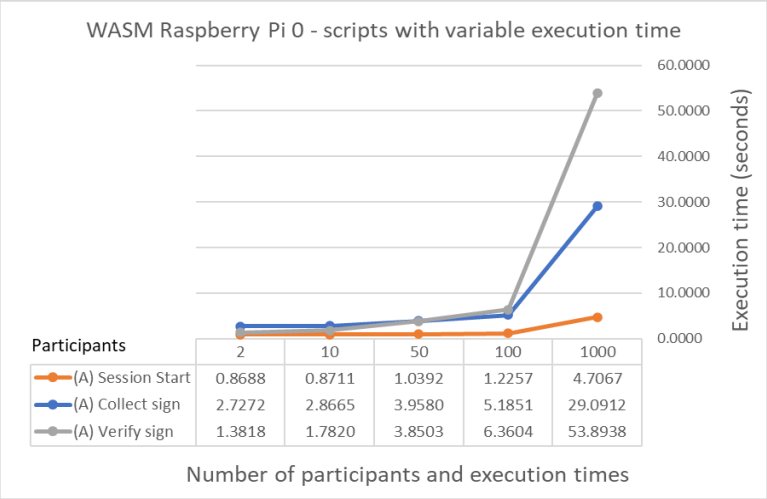
S/P	**2**	**10**	**50**	**100**	**1000**
(A) Session start	0.1065	0.1092	0.1269	0.1459	0.5432
(A) Collect sign	0.3897	0.4170	0.5741	0.7654	4.3594
(A) Verify sign	0.1880	0.2510	0.5621	0.9494	8.1078



Zenroom WASM Rasperry Pi 0

Table 8: Execution timings on **WASM Raspberry Pi 0** in *seconds* per *participant*

S/P	**2**	**10**	**50**	**100**	**1000**
(A) Session start	0.8688	0.8711	1.0392	1.2257	4.7067
(A) Collect sign	2.7272	2.8665	3.9580	5.1851	29.0912
(A) Verify sign	1.3818	1.7820	3.8503	6.3604	53.8938



Zenroom WASM - all platforms

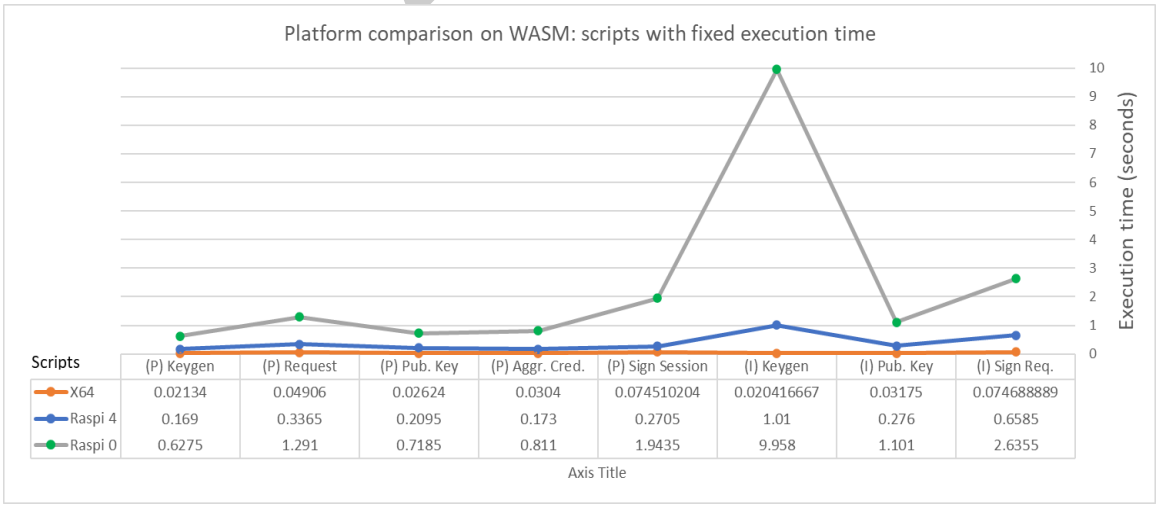
Following are the benchmarks of the benchmarks for the (P) and the (I) groups of scripts (whose executions don't change based on the amount of participants), with a comparison of all the platforms, running in CLI binaries.

Table 9: Execution times on **Zenroom WASM** of scripts in $seconds$ per $platform$

$script/platf.$	(P) Keygen	(P) Request	(P) Pub-Key	(P) Aggr. Cred.	(P) Sign Session
X86-X64	0.0213	0.0490	0.0262	0.0304	0.0745
Raspberry Pi 4	0.169	0.3365	0.2095	0.173	0.2705
Raspberry Pi 0	0.6275	1.291	0.7185	0.811	1.9435

Table 10: Execution times on **Zenroom WASM** of scripts in $seconds$ per $platform$

$script/platf$	(I) Keygen	(I) Pub-Key	(I) Sign Req.
X86-X64	0.0204	0.0317	0.0746
Raspberry Pi 4	1.01	0.276	0.6585
Raspberry Pi 0	9.958	1.101	2.6355



FINE ANDREA

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.

Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. Cryptology ePrint Archive, Report 2018/483, 2018a. https://eprint.iacr.org/2018/483.

Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

Lorna Goulden, Kai M. Hermsen, Jari Isohanni, Mirko Ross, and Jef Vanbockryck. The disposable identities privacy toolkit; enabling trust-by-design to build sustainable data driven value. 2021.

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/.

Maayke Aimée Damen. A resources passport for a circular economy. Master's thesis, 2012.

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, 2018b. doi:10.1007/978-3-030-03329-3_15.

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

Jan Camenisch and Markus Stadler. Proof systems for general statements about discrete logarithms. Technical report, 1997.

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.

Razvan Barbulescu and Sylvain Duquesne. Updating key size estimations for pairings. *Journal of Cryptology*, 32(4): 1298–1336, 2019.

Chae Hoon Lim and Pil Joong Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In *Annual International Cryptology Conference*, pages 249–263. Springer, 1997.

Jaap-Henk Hoepman, Shehar Bano, Alberto Sonnino, Eleonora Bassi, Marco Ciurcina, Juan Carlos De Martin, Selina Fenoglietto, Antonio Santangelo, Francisco Sacramento Gutierres, David Laniado, and Pablo Aragón. Privacy design strategies for the decode architecture. 2019.

List of Figures