Technical Overview: zkID for the EUDI Wallet

Jane Doe[1,2] and John Doe[1]

[1] Institute A, City, Country, jane@institute
[2] Institute B, City, Country, john@institute

Abstract. Main deliveries: 1. Technical report on zk component for the digital id wallet 2. A comparison with current works 3. Applying to EUDI.

Keywords: Anonymous credential · programmable zkp

1 Introduction

According to the Cryptographers' Feedback on the EU Digital Identity's ARF[1], an Anonymous Credential AC scheme, is a suitable cryptographic primitive to instantiate the new EU Digital Identity Wallet (EUDIW) which is an important step towards developing interoperable digital identities in Europe for the public and private sectors.

Informally speaking, an Anonymous Credential AC scheme allows:

- An Identity Provider IP to (possibly blindly[2]) sign a set of (eligible) attributes for a User U;
- The User U can show, only if they hold the signed attributes (a.k.a Unforgebality), usually through a Presentation, to a Relying Party RP such that:
  - The RP can verify that the set of attributes (signed by IP) that the User U holds satisfy some condition of their interest (a.k.a Correctness);
  - The RP cannot learn any *additional*[3] information beyond the fact that the condition is satisfied or information that can be inferred from the satisfaction of the condition (a.k.a Zero-Knowledge or Anonymity);
  - The immediate previous requirement also implies that the RP cannot link the various presentations by the same User U (a.k.a. Unlinkability);
- The IP can revoke all or a part of the signed attributes that it has issued to the User U, from upon which, the eligible attributes of the User U are updated, and subsequent presentations have to be based on the new and updated attributes (a.k.a Revocation);
- The User U cannot transfer its set of signed attributes to to another User U' (a.k.a Non-transferability).

In the aforementioned feedback document, BBS and BBS+[4] were promoted as the main candidate, besides that, there have been two independent work from Google and

[1] https://github.com/user-attachments/files/15904122/cryptographers-feedback.pdf

[2] i.e. the IP does not know the content that it signs, only its provenance is satisfied.

[3] We stress that the RP may have obtained some privacy sensitive information prior to this presentation.

[4] For BBS, thanks to prior work by the W3C, the Decentralized Identity Foundation, IETF/IRTF, ISO, and other standardization bodies, as well as the availability of open-source software libraries, the EC can develop a standard and reference implementation with only a modest effort. The feedback additionally recommend that the EUDI be designed following the principle of crypto-agility, meaning that its underlying technologies can be upgraded quickly in the future if the need arises.

Microsoft that attempted to offer candidate solutions. In this document, we attempt to offer a new candidate, called **zkID**.

In comparison, these approaches show the current trade-off: systems either reuse existing issuer infrastructure but pay high per-presentation costs, or they achieve fast online proofs at the price of large setups and pairing-based assumptions. Our construction, zkID, aims to combine issuer compatibility with reusable offline work, while remaining transparent and modular.

1.1 Related Work

Let us first outline a reference architecture that represents what an anonymous-credential system would ideally look like if it is to integrate smoothly with current infrastructures. In this model, the issuer is treated as fixed components that continue to use their existing public-key algorithms (such as RSA or ECDSA) and standard credential formats (e.g., JWT or mDL), since it's typically difficult to change once deployed. All additional logic is placed in the user's wallet and the verifier. The wallet is expected to operate in two stages: an offline Prepare step, which verifies the issuer's signature once using standard libraries, parses and normalizes credential attributes (for example, turning a date of birth into an integer age), and commits to those attributes using a binding and hiding commitment scheme (a cryptographic way to lock values so they can later be revealed or proven in restricted form); and an online Show step, which runs per presentation, where the wallet selects only the attributes or predicates required by a relying party's policy, proves them in zero knowledge against the stored commitments, and includes a fresh device signature over the session challenge to ensure the proof is tied to the holder's device. A further requirement is modularity: each major function—issuer signature verification, attribute commitment, predicate proofs, and device binding—should be defined as a separate module with a clear interface. This separation makes it possible to swap the underlying proof engine (for example, using a SNARK today or a post-quantum proof system in the future) without requiring changes to parts of the system that are costly or impractical to modify. The purpose of this modular view is to act as a comparison framework: it outlines how a deployment-friendly anonymous-credential stack could be structured, making it easier to compare proposals by the modules they cover, the constraints they address, and the trade-offs they make.

BBS-based anonymous credentials. [BBC+24] BBS-based anonymous credentials are recommended in public feedback for the EUDI wallet as a way to meet the program's requirement that presentations must not be tracked, linked, or correlated [BBC+24]. This work treats a credential as a constant-size signature on a vector of attributes in pairing-friendly groups, as introduced by Boneh–Boyen–Shacham and proven secure for BBS+ by Au–Susilo–Mu [BBS04, ASM06]. A holder then produces zero-knowledge proofs that reveal only the required attributes or predicates; each presentation is freshly generated so separate verifications cannot be linked. This matches our reference system view on the presentation side-privacy enforced at the holder with per-session, non-repeating outputs. Where these designs differ from our constraints is issuance. To use BBS/BBS+, issuers sign credentials with a pairing-based scheme rather than the RSA or ECDSA schemes used today [BBS04, ASM06]. To remain compatible with standardized curves such as P-256 while keeping public verifiability, a pairing-free, server-aided variant (often termed BBS#) allows the holder to prefetch small auxiliary data through an oblivious interaction with an issuer-side helper and later perform non-interactive presentations; the helper data scales linearly with the number of planned presentations [CAHLT25]. In both variants, device binding and revocation checks can be encoded as attributes or verified within the proof so that transcripts and status queries avoid stable identifiers.

Anonymous Credentials from ECDSA. [Fas24] This work considers environments where credential issuers already sign with ECDSA on standardized curves (such as P-256) and hash data with SHA-256. The main challenge is that proving correctness of an ECDSA signature in zero knowledge is costly with standard proof systems, because the arithmetic used in P-256 and the bit-level operations in SHA-256 do not align well with the fast polynomial techniques (such as number-theoretic transforms, a method that speeds up polynomial multiplication over special fields) that many modern ZK libraries rely on. To handle this, the authors introduce custom circuits for ECDSA and SHA-256, and use a layered protocol based on the sum-check technique and a lightweight encoding (Reed–Solomon code) to control proof size. An additional "consistency check" ensures that the same hidden signing key is used across both the signature and the hash logic. At presentation, the wallet produces a proof for the verifier and the device also signs a fresh challenge (this is the device-binding step: a live signature that ties the proof to the holder's device). In terms of the reference system view, issuer compatibility is preserved, selective disclosure is supported, and device binding is included; however, there is no reusable offline phase, so the full proof is generated at every presentation. The reported costs are about 60 ms to prove one ECDSA signature and about 1.2 s for a complete mDL presentation on mobile devices [Fas24, §5.3,§6.2], with larger proof sizes and higher verifier effort than systems based on succinct setup-dependent SNARKs.

Crescent Credentials. [PPZ24] This work considers environments where issuers continue using existing credential formats such as JWT or mDL and their current signing keys, so no issuer-side changes are required. Its workflow is split into a heavy one-time Prepare phase and a lightweight per-presentation Show phase. In Prepare, the wallet verifies the issuer's signature, parses the credential into attributes, and creates two reusable artifacts—that is, cryptographic objects the wallet reuses across presentations: (i) a Groth16 proof that these checks were done correctly, and (ii) a Pedersen vector commitment over the attributes, enabling selective disclosure. Both artifacts support re-randomization for unlinkability. In the Show phase, the wallet re-randomizes the prepared artifacts and attaches only the proofs required by the verifier's policy, such as proving an age threshold or linking two credentials to the same holder. Device binding can be added at this step by letting the secure element sign the verifier's challenge. In terms of the reference system view, Crescent realizes the two-phase design with reusable offline work and modular predicates, while leaving issuers unchanged. The trade-offs are significant: the Prepare phase is heavy (tens of seconds for JWTs and minutes for mDLs), the scheme depends on pairing-based Groth16 proofs with a large universal setup ($\approx$ 661 MB–1.1 GB [PPZ24, §4]), and the security model is classical only, without post-quantum protection. The Show step, however, runs with low latency-typically 22–41 ms with $\approx$1 KB proofs, or about 315 ms with device binding [PPZ24, §4].

1.2 Our zkID

Our construction works with standardized credentials (e.g., SD-JWT, mDL) and existing PKI (RSA/ECDSA), so issuers do not need to change their issuance pipelines. The zkID workflow follows the two-phase split in the reference view: a one-time Prepare phase and a per-presentation Show phase. In Prepare, the wallet verifies the issuer's signature, parses the credential into normalized messages, computes the associated hashes, and produces two reusable artifacts: (i) zero-knowledge proofs that issuer-side checks and parsing were done correctly, and (ii) Hyrax-style Pedersen vector commitments to a designated message column, supporting efficient proofs over multiple attributes. In Show, the wallet proves only the verifier's requested predicates and includes a fresh device-binding signature. To link Prepare and Show without revealing values, the verifier checks equality of commitments across both proofs; the wallet reuses the corresponding randomness for that session. The proving backend is transparent (no trusted setup). It checks the arithmetic constraints with a sum-check–style protocol and uses a small inner-product check to verify commitment openings. For device binding, we choose a curve whose scalar field matches the device's signature field (e.g., P-256), so the device signature can be verified directly inside the proof without emulation or field translation. In terms of the reference system view, issuer compatibility is preserved, the two-phase reuse is integrated into the workflow, predicates are modular, and there is no trusted setup. The trade-offs are that security currently relies on discrete-log assumptions (not post-quantum) and that commitment equality requires using the same curve across Prepare and Show; the modular interface leaves room to swap in lattice-based commitments when suitable.

2 Application to EUDI

Within the EUDI Architecture and Reference Framework [Eur23], the practical question is how to introduce zero-knowledge capabilities without disrupting established roles, formats, and certification paths. This section states how the construction fits that setting and what trade-offs it entails. Subsections are structured according to Topic G in the EUDI ARF discusssion thread.

Throughout this discussion, we continue to refer to EUDI's Wallet User as the "Prover", the Relying Party as the "Verifier", and the EUDI Attestation Authority (EAA) that acts as a PID/Attestation Prover as the "Issuer".

Issuance. The construction is designed to wrap existing credential encodings rather than replace them. It accommodates SD-JWT and ISO/IEC 18013-5 mDL so that wallets and relying parties retain current disclosure grammars and parsing logic. Issuers remain oblivious to the use of zkSNARKs; no changes to issuance pipelines or device secure elements are required, and Issuers also maintain exclusive control of their private keys. The proof layer is circuit-defined and therefore highly programmable, which allows our scheme to easily adapt to future Issuer-side migrations (for example, a change of signature scheme) by simply updating the Prover circuit and public parameters, rather than introducing new format-specific protocols. The approach interoperates with current public-key infrastructure based on ECDSA or RSA and does not prescribe a switch of algorithm or hardware.

Efficiency. Proving is split into two relations. A fixed relation captures Issuer-signature verification, credential parsing, and commitment preparation; it runs infrequently and is amortized per credential. A live, presentation-specific relation captures the disclosures and predicates for a single session; it runs per presentation. This separation aims to keep Prover time and memory costs within typical web and mobile budgets, and to bound latency where most critical: at the time of Prover-Verifier interaction. The proof system and commitment layer are modular, so improvements in either component can be adopted without redesigning the higher-level flow. In contrast to other designs, proofs of Issuer-signature verification and parsing are pre-computed offline to reduce work during live Prover-Verifier interaction.

Discussion. The present instantiation follows the Spartan line and relies on sumcheck and Hyrax-style Pedersen commitments under the Discrete Log assumption, rather than pairing-based assumptions; there is no universal trusted setup. We avoid pairing-friendly curves and the operational burden of a trusted setup ceremony across many Provers, Issuers, Verifiers, and other independent bodies. The construction is not currently post-quantum secure, but the modular structure leaves a path to replacing the commitment scheme layer with lattice-based alternatives as they mature. Some components have not yet been standardized; we note this is a shared condition across competing approaches that we call out explicitly. Regarding standardizations: sumcheck is a highly well-known protocol with information-theoretic security independent of cryptographic assumptions; Pedersen commitments have been used since 1991 [Ped92] and rely only on the discrete-log assumption, which standardized ECDSA signatures already rely on.

Summary for EUDI. The design aligns with Annex 2 format expectations (see Section 8), requires no changes to Issuers, supports current PKI deployments, and separates fixed from presentation-specific work to keep live presentation costs low. It avoids pairing-based assumptions and a universal setup, and leaves a path to future cryptographic upgrades without disrupting wallet or Issuer operations.

3 Security

In our security model, we assume that the Prover is malicious, and that each Verifier is semi-honest, meaning that if the Prover presents a valid proof that they own a credential with some property, the Verifier will grant access to any services for which the property suffices.

Verifier's side For security on the Verifier's side, our soundness analysis considers the probability that a malicious Prover without real ownership of a valid credential can generate a false proof of ownership.

Prover's side For security on the Prover's side, we guarantee that our proofs are zero-knowledge, so that a semi-honest and computationally-bounded Verifier cannot get any additional information about the Prover's credential beyond what is publically revealed in the proof. In particular, we do not consider the case where the Verifier is malicious during presentation, e.g. where a false Verifier pretends to be an authorized Verifier. The problem of Verifier identity lies outside the scope of this paper.

Furthermore, we assume that Verifiers can collude with each other, i.e. that Verifiers $V_1, \ldots, V_N$ that have received proofs $\{\pi_1\}, \ldots, \{\pi_N\}$ from a given Prover P can compute functions $f(\pi_1, \ldots, \pi_N)$. Therefore, we desire the **unlinkability property**: given $pi_1, \ldots, \pi_N$, the Verifiers should not be able to determine whether or not any two of these proofs came from the same Prover P. Note that this requires the Prover to re-randomize each presentation's proof; a static zero-knowledge proof of the same statement, while not revealing private credential information, would still look the same across presentations. In that case, it may be possible for the Verifier to de-anonymize a Prover by linking their "anonymous" activity across presentations and analyzing metadata, e.g. time of presentation. Fortunately, our scheme is unlinkable due to the re-randomization of proofs between each presentation. By the zero-knowledge property for each presentation, we can simulate the distribution of proofs without knowledge of the witness. To simulate an entire set of proofs received by distinct colluding Verifiers, we can independently simulate each proof.

Finally, as our scheme is currently presented in Section 5, we assume that Verifiers will not collude with Issuers even though they can see the Issuer public key. To bypass this assumption and prevent Issuer tracking in the case of malicious Verifiers that collude with Issuers, we propose here the maintenance of a trusted Merkle tree on trusted Issuer public keys. Then our Prover's circuit would prove knowledge of a valid Issuer signature from some key in the Merkle tree, and the public input/output would just be the Merkle root rather than any specific Issuer public key.

Both Prover and Verifier security When modelling the Issuer, we assume that the Issuer is trusted during issuance by both the Prover and Verifier, i.e. will not Issue false credentials or sell personal information that is necessarily to obtain about individuals to issue a credential.

3.1 Other considerations

Our scheme currently does not require any interaction from the Issuer for credential presentation beyond initial issuance.

However, our scheme does require the use of internet access (without, there will be risks with authorizing someone before their credential can be checked against the current state). In the case (as presented) where Issuer public key is a public input/output, we assume there is an online registry of trusted Issuer keys that the Verifier can check the proof against. This requires live internet access in the same way that credit card transactions do, in order to check the most current registry of public keys. Even in the case of a Merkle inclusion proof, where the Issuer key is also private, the Verifier would need to check that the public Merkle root matches the trusted root stored online. It is possible to store encrypted transactions/credential presentations to be checked later once internet access is restored, in the same way as offline credit card transactions do. However, there are necessary risks with this approach; it would be up to the specific vendor and/or service prover what levels of risk can be tolerated from delayed credential authentication. For example, some service provider (Verifier) may be fine only periodically downloading the current registry of trusted Issuer keys (and/or Merkle roots) and simply checking against their last downloaded version before granting access.

Finally, as mentioned previously, our scheme is not quantum resistant due to the use of Hyrax commitments. Again, we believe this is easily fixable with the introduction of modified Ajtai lattice-based commitments, which are post-quantum secure.

4 Preliminaries - WIP

Notation For $n \in \mathbb{N}$ we write $[n] = \{1, \ldots, n\}$. Bold letters denote vectors, e.g., $\mathbf{m} = (m_1, \ldots, m_n)$. Concatenation is written $\|$. The security parameter is λ; $\mathsf{negl}(\lambda)$ denotes a negligible function. For a (possibly randomized) algorithm Alg, we write $y \leftarrow \mathsf{Alg}(x)$ for its output on input x.

There are three roles:

- The *issuer* I signs credentials with a long-term key pair (SK_I, PK_I) (e.g., ECDSA P–256 or RSA).
- The *prover* P is the holder's wallet, which stores credentials and generates proofs.
- The *verifier* V is the relying party that checks proofs against a policy.

For device binding, the prover's secure element holds an additional signing key pair (SK_D, PK_D) used only to sign fresh per-session challenges.

Credentials A credential is a standardized signed object S (e.g., SD–JWT [FYC25] or mDL [fS21]). Parsing maps S into an ordered vector of attributes

$$\mathbf{m} = (m_1, \ldots, m_n).$$

Non-numeric fields (strings, dates) are encoded injectively into integers. The resulting integers are interpreted in a prime field $\mathbb{F} = \mathbb{F}_q$ chosen for the proof backend. For each attribute m_i we sample a salt $s_i \leftarrow \mathbb{F}$ and compute

$$h_i = \mathsf{H}(m_i \,\|\, s_i),$$

where H is instantiated as SHA–256. The issuer's signature is

$$\sigma_I = \mathsf{Sign}_{SK_I}(h_1, \ldots, h_n),$$

verified under PK_I.

Commitments and Proof Interface To support selective disclosure without revealing raw attributes, the wallet commits to $\mathbf{m}$ using Pedersen vector commitments. Let $\mathbb{G}$ be a cyclic group of prime order q with public generators $(g_1, \ldots, g_n, h)$ derived from a domain-separated seed. For randomness $r \leftarrow \mathbb{F}$, the commitment is

$$C = \prod_{i=1}^{n} g_i^{m_i} \cdot h^r \ \in \ \mathbb{G}.$$

Under discrete-logarithm hardness in $\mathbb{G}$, these commitments are computationally binding; they are also perfectly hiding. To avoid linkability, the wallet re-randomizes r across sessions. If it precomputes several reusable commitments, we index them $(C^{(j)}, r^{(j)})$; both offline and online proofs in a session reference the same $C^{(j)}$, allowing the verifier to link the phases without learning $\mathbf{m}$.

Credential use is captured by two relations:

- *Prepare (offline).* Once per credential, the wallet verifies σ_I under PK_I, parses S into $\mathbf{m}$, computes digests $\{h_i\}$, derives a commitment $C^{(j)}$, and produces a reusable proof
$$\pi^{(j)}_{\text{prep}} : \text{ “}S\text{ parses to }\mathbf{m}, \sigma_I\text{ verifies, and }C^{(j)}\text{ commits to }\mathbf{m}\text{”.}$$

- *Show (online).* For each presentation, the verifier sends a challenge ch. The device signs it as $\sigma_{ch} = \mathsf{Sign}_{SK_D}(ch)$. The wallet proves that all predicates in the verifier's policy hold with respect to $C^{(j)}$ and incorporates σ_{ch}:
$$\pi^{(j)}_{\text{show}} : \text{ “policy holds for }C^{(j)}\text{, and the session is bound via }\sigma_{ch}\text{”.}$$
The verifier checks $\pi^{(j)}_{\text{prep}}$, $\pi^{(j)}_{\text{show}}$, their consistency on $C^{(j)}$, and verifies σ_{ch} under PK_D.

This split amortizes heavy work (signature verification, parsing, commitment) offline, leaving online interaction to short proofs plus one device signature.

Predicates and Policies A *predicate* is a Boolean function $f(\mathbf{m}[S]) \in \{0, 1\}$ over a subvector indexed by $S \subseteq [n]$. Typical predicates include range checks ($m_i \geq 18$), equality or membership tests (e.g., m_i equals a country code), and cross-credential comparisons. A *policy* is a finite set of predicates chosen by the verifier. In each session, the wallet proves in zero knowledge that all predicates in the policy hold with respect to $C^{(j)}$, revealing only what the policy requires. Because predicates are modular, the proving backend can be swapped (e.g., from a SNARK to a post-quantum argument system) without changes to issuer infrastructure.

We assume the issuer is honest and operates standard PKI. Verifiers are semi-honest: they check proofs correctly but may collude to compare transcripts. Unlinkability relies on re-randomization of commitments, so the only stable value within a session is $C^{(j)}$, intentionally shared between *Prepare* and *Show.*

5 Our zkID

At a high-level, we propose a generic zkSNARK wrapper over an EUDI digital credential, which will either be issued in SD-JWT format or the mDL data format specified in standard ISO/IEC 18013-5). The backend proving system we use will be a combination of Spartan and Hyrax commitments, with modifications to be zero-knowledge.

There are two (2) key ideas to highlight within our proposed architecture:

- **Pre-processing batches of re-randomized proofs** of issuer-signature and credential-parsing, to re-use across each new presentation. We call this the `prepare` relation.
- **Committing to the credential disclosures with Hyrax commitments** [WTas+17], which allows us to re-use (or "link") witnesses across circuits "for free"

For the first item, we note that pre-processing proofs for the `prepare` relation is possible because the relation is independent of the presentation, including the choice of disclosures or predicate proofs. Further optimizations can likely be made to only parse the discosures of certain attributes within the credential if it is known that the Wallet User will rarely or never present certain disclosures to external verifiers.

The second items differs from the linking circuits approach that Google uses [Fas24]. Note that Google verifiably computes a hiding and binding MAC of the shared witnesses as a public output of the circuit, which the verifier checks consistency of in plain. Although this is only a few linear relations, it requires the prover to also commit to their portion of the key to prevent forging. We instead simply manually separate out the disclosures $m_1, \ldots, m_n$ into a designated column when committing to the witness, which is already needed to prove the circuit relation. Then, the verifier simply checks consistency of these commitments when verifying each circuit's proof.

We note that by using Hyrax commitments, our PoC is not post-quantum secure. In particular, post-quantum computers break the discrete-log assumption, which breaks the binding property of the polynomial commitment scheme. Thus, a malicious Prover could potentially make false proofs about their identity. In future work, we hope to incorporate modified Ajtai lattice-based commitments to ensure post-quantum security [HSS24].

Throughout the remainder of this section, we refer to the EUDI's Wallet User as the "Prover", the Relying Party as the "Verifier", and the EUDI Attestation Authority (EAA) as the "Issuer". Below, we briefly detail a high-level flow of the interaction between the Issuer, Prover, and Verifier.

5.1 Underlying ZK Circuit

In this section, we describe our high-level ZK circuit C underlying the knowledge the prover needs to present to the verifier. Throughout, we refer to the Issuer with variable I, Prover with variable P, and Verifier with variable V (e.g. in subscripts).

We will detail the ZK wrapper around the SD-JWT credential as an example, but the protocol is analogous for other credential formats. Throughout, we will refer to digests as "message hashes" or just "hashes", and disclosures as "messages".

We define a circuit C for proving ownership of an anonymous credential. We let our witness $w = S$ be the SD-JWT credential consisting of messages $\{m_i\}_{i=1}^N$, hash salts $\{s_i\}_{i=1}^N$, hashes $\{h_i\}_{i=1}^N$, and an Issuer signature $\sigma_I = \sigma(h_1, \ldots, h_N; SK_I)$. Without loss of generality, we assume that the Prover's public key PK_P is contained in message m_1 of the credential and indexable as $m_1[1]$. We let our instance $x = (PK_I, \{f_i\}_{i=1}^K, \{p_i\}_{i=1}^K)$ contain the Issuer's public key PK_I, functions f_i over the messages, output predicates p_i, and finally the nonce signature σ_{nonce} for proving device-binding. The f_i can be arbitrary statements we wish to prove about the messages. For example, one could define a function $f_i(m_1, \ldots, m_N) = m_1$ would output a predicate that is just the disclosure of message m_1.

Underlying ZK Circuit C for Verifiable Credential

We define circuit $C(x = (PK_I, \{f_i\}_{i=1}^K, \{p_i\}_{i=1}^K), w_C = (S))$ as follows:

1. Assert $\text{parse}_{\text{SD-JWT}}(S) = (\{m_i\}, \{s_i\}, \{h_i\}_i, \sigma_I)$ parsing of the SD-JWT into messages $\{m_i\}_{i=1}^N$, message salts $\{s_i\}_{i=1}^N$, hashes $\{h_i\}_{i=1}^N$ and Issuer signature σ_I.
2. Assert $h_i = \text{SHA256}(m_i, s_i) \quad \forall i \in [n]$, i.e. that messages hashes correspond to messages and salts
3. Assert $p_i = f_i(m_1, \ldots, m_n) \quad \forall i \in [n]$, i.e. correct evaluation of the predicates
4. Assert $\text{ECDSA.verify}(\sigma_I, PK_I) = 1$, i.e. the credential signature verifies under the Issuer public key
5. Assert $\text{ECDSA.verify}(\sigma_{\text{nonce}}, m_1[1]) = 1$, i.e. that the live nonce signature corresponds to the public key the credential was issued to

5.2 Pre-processing and linking proofs

The main speedups from our proving system will come from splitting our high-level circuit C above into two (2) circuits for different relations regarding the digital credential, namely a `prepare` and a `show` relation, analogous to Microsoft's Crescent Credentials [PPZ24]. This is advantageous because proofs of the `prepare` relation can be computed a-priori for any credential, as they do not depend on the claim being proved at presentation time. Pre-computing these proofs will save significant time per presentation, and reduce the performance bottleneck to that of proving the `show` relation.

One issue that arises is the need to ensure consistency of witnesses across these separate circuits, or what we call "linking proofs". At a high level, as opposed to Google's MAC approach [Fas24], the prover sends Hyrax commitments to the parts of the witness re-used across circuits, which ends up being just the raw messages $\{m_i\}_i$. The verifier can then check consistency of these witnesses across the circuits C_i by comparing the Hyrax commitments they receive as part of the proof. This approach gives us linking "for free", as the Prover alreadys needs to compute these Hyrax commitments as part of the proof.

We highlight that we are no longer splitting up circuits by their field operations (e.g. SHA256 attestations over a binary extension field and an ECDSA verifications over a prime field), but rather by pre-processing and per-presentation relations. In particular, the circuit for `prepare` will necessarily involve wrong-field arithmetic by including both the SHA256 hashes and the Issuer ECDSA signature verification. However, because of the ability to pre-compute proofs of the `prepare` relation, the more important thing becomes to choose curves that allow for i) efficient show relations and ii) linking the prepare and show relation. Since the verifier can only check equality of Hyrax Pedersen commitments defined over the same curve, we must use the same curve for proving both the `prepare` and `show` relations. Thus we choose a curve with a scalar field equivalent to the base field of the nonce signature σ_{nonce} for efficient signature verification. Because most Hardware Security Modules (HSMs) sign over the P256 curve, we choose the Tom256 (T256) curve for our backend, which has scalar field equivalent to the base field of P256.

We now detail each of the two (2) relations/circuits below.

5.2.1 The `prepare` relation:

The `prepare` relation checks the validity of issuer signature, parses the SD-JWT, and verifies all the message hashes, none of which depend on the specific presentation. Thus, the prover will periodically pre-compute and store a batch of re-randomized proofs of the prepare relation. These proofs will utilize Hyrax Pedersen vector commitments as introduced above in order to link the proofs of `prepare` relation to the `show`.

Circuit C_1 for the `prepare` relation

We define circuit $C(x = (PK_I), w_i = S, w = (\{m_i\}_{i=1}^N))$ as follows:

1. Assert $\text{parse}_{\text{SD-JWT}}(S) = (\{m_i\}, \{s_i\}, \{h_i\}_i, \sigma_I)$ parsing of the SD-JWT into messages $\{m_i\}_{i=1}^N$, message salts $\{s_i\}_{i=1}^N$, hashes $\{h_i\}_{i=1}^N$ and Issuer signature σ_I.
2. Assert $h_i = \text{SHA256}(m_i, s_i) \quad \forall i \in [n]$, i.e. that messages hashes correspond to messages and salts
3. Assert $\text{ECDSA.verify}(\sigma_I, PK_I) = 1$, i.e. the credential signature verifies under the Issuer public key

The backend proving system we will use for verifiably computing circuits is Spartan, coupled with a Hyrax-style Pedersen commitment scheme. We can express the circuit computation as some R1CS relation

$$(A \cdot Z) \circ (B \cdot Z) = (C \cdot Z),$$

where $\vec{Z} = (io, 1, \vec{w})$ and io are the public input/outputs. Spartan proves knowledge of a vector Z of length $n := |Z|$ that satisfies the R1CS instance.

To produce zkSNARK proofs for this circuit C_1, the prover will proceed in two phases:

1. `prepareCommit`: Separates out a column containing only message hashes $\{m_i\}_{i\in[N]}$ in Z and computes an initial Hyrax commitment $c^{(1)} = \{c_i^{(1)}\}_{i\in[\sqrt{n}]}$, which includes a Pedersen commitment to the messages column $c_1^{(1)} = com(m_1, \dots, m_N; r_1^{(1)}) = g_1^{m_1} \dots g_N^{m_N} g_{N+1}^{r_1^{(1)}}$ with initial randomness $r_1^{(1)}$.
2. `prepareBatch`:
   (a) Re-randomizes this initial Hyrax commitment to get a batch of commitments $c^{(j)} = \{c_i^{(j)}\}_{i\in[\sqrt{n}]}$, each of which contains a Pedersen commitment to the messages $c_1^{(j)} = com_1^{(1)} \cdot g_{N+1}^{r_1^{(j)} - r_1^{(1)}}$ for all $j \in [m]$, where our batch size m depends on the frequency of proof generation and demand for the credential
   (b) Continues the Spartan sumcheck IOP on each $c^{(j)}$ to produce a batch of proofs $\{\pi_{\texttt{prepare}}^{(j)}\}$ for $j \in [m]$ of the `prepare` relation.

The prover will run `prepareBatch` periodically to both generate re-randomized commitments $c^{(j)}$ and store the randomness for the message column commitment $r_1^{(j)}$ for linking purposes, as well as generate and store batches of issuer-signature proofs $\pi_{\texttt{prepare}}^{(j)}$ that can be used for each presentation.

5.2.2 The `show` relation:

At a high-level, our show relation will i) verifiably compute any functions f_i over the SD-JWT messages (such as disclosures, range checks, etc.), and ii) check that the credential belongs to the prover's device (also known as proof of "device-binding"). As part of device-binding, the prover will sign a verifier `nonce` outside of the circuit, as outlined in flow **??**. Let us denote this signature by $\sigma_P = \sigma(\text{nonce}; SK_P)$

Again, we will use T256 curve for our backend proving system so that the holder in-circuit signature verification can proceed naturally in the right field.

Circuit C_2 for the `show` relation

We define circuit $C_2(x = (\{f_i\}_{i=1}^K, \{p_i\}_{i=1}^K), w = \{m_i\}_{i=1}^N)$ as follows:

1. Assert $p_i = f_i(m_1, \dots, m_n) \quad \forall i \in [n]$, i.e. correct evaluation of the predicates
2. Assert $\text{ECDSA.verify}(\sigma_{\text{nonce}}, m_1[1]) = 1$, i.e. that the live nonce signature corresponds to the public key the credential was issued to

As part of computing proof $\pi^{(j)}_{\texttt{show}}$ for presentation $j \in [m]$, the Prover will once again separate out the messages into a separate column to compute a Hyrax commitment over the Tom256 curve. In particular, the Prover uses *the same* randomness $r_1^{(j)}$ used during the `prepareBatch` process to compute the Pedersen commitment to the messages column. The verifier will then check that the Pedersen commitment to the messages column for $\pi^{(j)}_{\texttt{show}}$ equals that of proof $\pi^{(j)}_{\texttt{prepare}}$ for circuit C_1.

5.3 Adding ZK to Spartan

Our construction uses Circom in the frontend to compile our computation into an R1CS (instance, witness) pair $(x = (\mathbb{F}, A, B, C, io, n, m), \vec{w})$, which we then feed into the Spartan IOP coupled with Hyrax-style Pedersen polynomial commitments.

Recall that our R1CS constraint looks like the following:

$$(A \cdot \vec{Z}) \circ (B \cdot \vec{Z}) - (C \cdot \vec{Z}) = 0$$

where our square matrices A, B, C have size n and $\vec{Z} = (\vec{w}, 1, io)$.

Recall that Spartan converts an R1CS constraint into the following zero-check:

$$\sum_{x \in \{0,1\}^{\log n}} \widetilde{eq}(x,\tau) \Bigg[\Bigg(\sum_{y \in \{0,1\}^{\log n}} \widetilde{A}(x,y)\, \widetilde{Z}(y) \Bigg) \Bigg(\sum_{y \in \{0,1\}^{\log n}} \widetilde{B}(x,y)\, \widetilde{Z}(y) \Bigg) - \Bigg(\sum_{y \in \{0,1\}^{\log n}} \widetilde{C}(x,y)\, \widetilde{Z}(y) \Bigg) \Bigg] = 0$$

for some random challenge $\tau \in \mathbb{F}$

There are two components in Spartan that we need to modify to be ZK. The first is making the sumchecks ZK. The second is to ensure that the opening $\widetilde{Z}$ using the commitment to $\vec{Z}$ does not leak information about our witness $\vec{w}$.

5.3.1 Adding ZK to sumcheck

The Spartan protocol consists of several sumchecks in parallel and operates over some field $\mathbb{F}$. There are various existing techniques to make sumcheck ZK. We employ one using similar methods as in Zhang et. al. [ZXZS19], which adds uniformly random pads to the sumcheck transcript.

In particular, suppose at each round i of the sumcheck protocol, the prover sends over $s_i(X) := \sum F(r_1, \dots, r_{i-1}, X, x_{i+1}, \dots, x_m)$ where r_i is the Verifier challenge sent for round i. Then instead of having the verifier check the sumcheck, the prover will prove in

ZK that the unpadded transcript satisfies the verifier's (linear) checks. To do this, the prover will need to commit to the uniformly random pads ahead of time. Then, as long as the Fiat-shamir challenges is generated from the transcript including these random pad commitments, the prover cannot simply lie about the pads to satisfy the sumcheck relation.

Adding ZK to sumcheck

1. Prover commits to pads $R_i(X) \overset{\$}{\leftarrow} \mathbb{F}_1[x]$ for all $i \in [\log n]$. These are linear polynomials, and can thus be represented by its two coefficients $R_i[0]$ and $R_i[1]$.

2. Instead of sending partial sums

$$s_i(X) := \sum_{(x_{i+1},\ldots,x_n)\in\{0,1\}^{n-i}} F(r_1,\ldots,r_{i-1},X,x_{i+1},\ldots,x_n)$$

   for each round of sumcheck, the Prover sends polys $s'_i(X) = s_i(X) + R_i(X)$, essentially a one-time-padded transcript.

3. It suffices to show the following linear relation in zero-knowledge,

$$\left[\ A \mid B \mid C\ \right] \begin{bmatrix} \vec{S} \\ \hline \vec{R} \\ \hline C \\ F(r) \end{bmatrix} = \vec{0}$$

   where

$$\vec{S} = \left[\, s'_1[0], s'_1[1], \ldots, s'_n[0], s'_n[1] \,\right]^{\mathsf{T}},$$

   is the column vector of sumcheck transcripts such that $s'_i = s'_i(X) = s'_i[0]X + s'_i[1]$, and

$$\vec{R} = \left[\, R_1[0], R_1[1], \ldots, R_n[0], R_n[1] \,\right]^{\mathsf{T}},$$

   is the column vector of random pads, and where matrices A, B, C are given by

$$A = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ -r_1 & -1 & 1 & 2 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & -r_2 & -1 & 1 & 2 & 0 & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & -r_{n-1} & -1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & r_n & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} -1 & -2 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ r_1 & 1 & -1 & -2 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & r_2 & 1 & -1 & -2 & 0 & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & r_{n-1} & 1 & -1 & -2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & -r_n & -1 \end{bmatrix}$$

$$C = \begin{bmatrix} -1 & 0 \\ 0 & 0 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \\ 0 & -1 \end{bmatrix}$$

4. The Prover computes a public random challenge α (e.g. hashing the transcript) and compresses the relation into a dot product

$$[\vec{u}] \left[\begin{array}{c} \vec{S} \\ \hline \vec{R} \\ \hline C \\ F(r) \end{array} \right] = \vec{0} \tag{1}$$

where $\vec{u} = [1, \alpha, \alpha^2, \dots, \alpha^n] \left[\begin{array}{c|c|c} A & B & C \end{array} \right]$ is a random linear combination of the rows.

5. The Prover and Verifier engage in a proof-of-dot product protocol to prove the relation above, such as an Inner Product Argument used in Bulletproofs [BBB+17]

5.3.2 Adding ZK to the opening of $\widetilde{Z}$

In order to add ZK to the opening of, we simply "append" Z with uniformly random pads. Specifically, we assign random evaluations $Z(x) \xleftarrow{\$} \mathbb{F}$ on any remaining point in the hypercube $x \in \{0,1\}^{\log n}$. Then, we see that $\widetilde{Z'}(x_1, \dots, x_{\log n}) = \widetilde{Z}(x_1, \dots, x_{\log n}) + eq(r,x)Z(x)$. Note that now $\widetilde{Z'}(x_1, \dots, x_{\log n})$ is distributed uniformly at random. If $n = |Z|$ is not already a power of 2, then we can simply fill at least one of the remaining evaluations on $\{0,1\}^{\log n}$ with a single random pad. If $n = 2^m$, we can add another dimension to the hypercube of evaluations of Z.

5.4 Cost analysis

The following section computes the Prover and Verifier costs of Spartan instantiated with Hyrax Pedersen commitments on R1CS instances $(x = (\mathbb{F}, A, B, C, io, n, m), \vec{w})$, where io denotes the vector of public inputs/outputs, $n = |\vec{w}| + io + 1$ is the dimension of our matrices, and m is the number of nonzero entries in our matrices A, B, C. We let $Z = (\vec{w}, io, 1)$. It is often reasonable to assume that our R1CS matrices are sparse, i.e. $m = O(n)$. However, we present the costs below independent of this assumption.

- **Prover time**: (1) $O(m)$ to generate sumcheck transcript, (2) $O(m)$ to evaluate MLEs of A, B, C, (3) $O(n)$ to commit to the MLE of Z (computing $\sqrt{n}$ MSMs of size $\sqrt{n}$) and opening the MLE of Z, for a total cost of $O(m)$.

- **Proof length**: (1) $O(\log n) \cdot |\mathbb{F}|$ length of the sumcheck transcript, (2) $O(\sqrt{n}) \cdot |\mathbb{G}|$ length commitment to MLE of Z, (3) $O(\log n) \cdot |\mathbb{G}|$ length of argument opening MLE of Z, for a total length of $O(\sqrt{n})$ group or field elements.

- **Verifier time**: (1) $O(\log n)$ to verify sumcheck transcript, (2) $O(m)$ to evaluate the MLEs of A, B, C (with sparse commitment scheme and memory checking), (3) $O(\sqrt{n})$ to open the MLE of Z, for a total of $O(m + \sqrt{n})$.

With the ZK modifications to Spartan, we can see the asymptotic costs remain the same, as follows:

- **Prover time**: additionally computes $O(\log n)$ constant-size commitments to the sumcheck transcript pads r_i, and $O(\log n)$ engages in new sumcheck relation IPA (or some other ZK dot product argument) for vector of length $O(\log n)$, which still gives $O(m)$ prover work.

- **Proof length**: sumcheck and openings are the same length but just padded, but added on $O(\log n)$ size $|\mathbb{G}|$ commitments, and a length $O(\log \log n)$ sumcheck relation IPA proof, which still gives a proof length of $O(\sqrt{n})$ group or field elements.

- **Verifier time**: no longer needs to do $O(\log n)$ (sumcheck), but still needs $O(m)$ (evaluating MLEs of A, B, C) + $O(\sqrt{n})$ (opening MLE of Z) + $O(\log n)$ for sumcheck relation IPA verification, which still gives $O(m + \sqrt{n})$ runtime.

5.5 Security analysis

The correctness follows immediately from the correctness of the Spartan SNARK and the fact that the Prover uses the same randomness for the Hyrax commitments across the `show` and `prepare` circuits for each presentation i.

The soundness of our protocol follows from the soundness of Spartan. In particular, we can extract the full witness credential from the `prepare` relation.

Intuitively, zero-knowledge follows from the hiding property of the commitment scheme as well as the zero knowledge property of the Spartan zkSNARK proving system; For proof i, simulator can randomly sample the linked commitment com_i both distributions to reuse across both proofs, both in the commitment itself and also in the IPA used to open the Hyrax commitment to $Z(r_1, \ldots, r_{\log n})$. We can show that this commitment is independent of the rest of the view of the Verifier, which consists of the following:

- Sumcheck polynomials $\{s'_i(X)\}_{i \in [\log n]}$ for each of the sumchecks in Spartan

- $\{r_i\}$ Fiat-Shamir challenges during the sumcheck

- Transcript from the IPA on the sumcheck relation in ZK

- $\{com(z_i)\}_{iin[\sqrt{n}]}$ Hyrax commitment to Z, which involves a Pedersen commitment to each of the columns of a $\sqrt{n} \times \sqrt{n}$ matrix representation of $\vec{Z}$

- Transcript from the IPA for opening $Z(r_1, \ldots, r_{\log n})$

- The claimed value of $Z(r_1, \ldots, r_{\log n})$

Since we appended random pads to $\vec{Z}$ in our ZK modifiction in Section 5.3.2, the distribution of $Z(r_1, \ldots, r_{\log n})$ is random and independent of Z, and therefore independent of $\{m_i\}_i$. Furthermore, $s'_i(X)$ have totally random pads on them and their distribution is independent of Z, and therefore independent of $\{m_i\}_i$. Assuming the hiding property of the Pedersen commitment schemes for messages sent during an IPA, we can also use the simulators for the IPAs without changing their joint distribution with the rest of the transcript.

Then, we can simply run the piece-wise simulators for each zkSNARK proof for circuits C_1 and C_2 to simulate the remainder of the view.

6 Experiments

7 Conclusion

References

[ASM06] Man Ho Au, Willy Susilo, and Yi Mu. Constant-size dynamic k-TAA. In Roberto De Prisco and Moti Yung, editors, *SCN 06: 5th International Conference on Security in Communication Networks*, volume 4116 of *Lecture Notes in Computer Science*, pages 111–125, Maiori, Italy, September 6–8, 2006.

[BBB$^+$17] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. Cryptology ePrint Archive, Paper 2017/1066, 2017.

[BBC$^+$24] Carsten Baum, Olivier Blazy, Jan Camenisch, Jaap-Henk Hoepman, Eysa Lee, Anja Lehmann, Anna Lysyanskaya, René Mayrhofer, Hart Montgomery, Ngoc Khanh Nguyen, et al. Cryptographers' feedback on the eu digital identity's arf. *Tech. Rep.*, 2024.

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In Matthew Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 41–55, Santa Barbara, CA, USA, August 15–19, 2004.

[CAHLT25] Rutchathon Chairattana-Apirom, Franklin Harding, Anna Lysyanskaya, and Stefano Tessaro. Server-aided anonymous credentials. Cryptology ePrint Archive, Paper 2025/513, 2025.

[Eur23] European Commission. The european digital identity wallet architecture and reference framework. Technical report, European Commission, 2023.

[Fas24] Matteo Frigo and abhi shelat. Anonymous credentials from ECDSA. Cryptology ePrint Archive, Paper 2024/2010, 2024.

[fS21] International Organization for Standardization. Iso/iec 18013-5:2021 personal identification — iso-compliant driving licence part 5: Mobile driving licence (mdl) application, 09 2021.

[FYC25] Daniel Fett, Kristina Yasuda, and Brian Campbell. Selective disclosure for JWTs (SD-JWT). Technical Report draft-ietf-oauth-selective-disclosure-jwt-22, IETF OAuth WG, May 2025. Internet-Draft.

[HSS24] Intak Hwang, Jinyeong Seo, and Yongsoo Song. Concretely efficient lattice-based polynomial commitment from standard assumptions. Cryptology ePrint Archive, Paper 2024/306, 2024.

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO'91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140, Santa Barbara, CA, USA, August 11–15, 1992.

[PPZ24] Christian Paquin, Guru-Vamsi Policharla, and Greg Zaverucha. Crescent: Stronger privacy for existing credentials. Cryptology ePrint Archive, Paper 2024/2013, 2024.

[WTas+17] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. Cryptology ePrint Archive, Paper 2017/1132, 2017.

[ZXZS19] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. Cryptology ePrint Archive, Paper 2019/1482, 2019.

8 Appendix: EUDI Annex 2 Requirements

This section is devoted to a review of the EUDI ARF's Annex 2, which covers high-level requirements for the EUDI Wallet. The full Annex can be found here.
The EUDI Annex 2 covers over fifty topics spanning cryptographic guarantees, privacy, trust infrastructure, wallet lifecycle management, and product-level usability. We cluster the requirements into four classes depending on their relationship to the zkID protocol itself:

Group A: Directly satisfied by zkID.
Requirements that are already implemented by the zkID protocol as described in Sections §5–§5.3, without additional assumptions or infrastructure. These are shown in Table A.

Group B: Satisfied with minor extensions.
Requirements that can be met by trivial modifications or configuration changes to zkID's proving circuits, predicates, or interface (for instance, adding a new predicate or exposing one more public input). No new trust anchor or cryptographic primitive is required. These appear in Table B.

Group C: Depend on external systems or operational flows.
Requirements that require the presence of registries, PKI governance, revocation lists, wallet attestation management, or other policy frameworks external to the proving layer. zkID integrates with these systems but does not replace them. These are summarized in Table C.

Group D: Product / UX / policy-facing requirements.
Requirements that concern wallet behaviour, user experience, accessibility, or legal presentation. These fall outside the scope of cryptographic protocol design but are compatible with zkID's guarantees. These are listed in Table D.

Each table records:

- the corresponding Annex 2 topic and its HLRs;
- a short informal description of the requirement;
- and the relevant subsection(s) in this document where we discuss how zkID meets or relates to the requirement.

The following four landscape tables provide the detailed mapping for each group:

Table A — Requirements directly implemented by zkID

Annex 2 ID	**Requirement**	**Where in zkID**
Topic 1 — Online Identification and Authentication (OIA)		
OIA_01	For OIA_01, see Interface in § 5.	see Interface in § 5.
OIA_02	For OIA_02, see Component prepare batches in § 5.2.1 and Component linking in § 5.2.	see component prepare batches in § 5.2.1 and component linking in § 5.2.
OIA_03a	For OIA_03a, see Component predicate in § 5.1 and § 5.2.2 and Components zkSNARK wrapper in § 5 and § 5.1.	see component predicate in § 5.1 and § 5.2.2 and zkSNARK wrapper in § 5 and § 5.1.
OIA_03b	For OIA_03b, see Prepare relation in § 5.2 and Show relation in § 5.2.	see Prepare relation in § 5.2 and Show relation in § 5.2.
OIA_03c	For OIA_03c, see Prepare relation in § 5.2 and Component predicate in § 5.1 and § 5.2.2.	see Prepare relation in § 5.2 and component predicate in § 5.1 and § 5.2.2.
OIA_04	For OIA_04, see Component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.	see component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.
OIA_05	For OIA_05, see Component predicate in § 5.1 and § 5.2.2 and Security notes in § 6.	see component predicate in § 5.1 and § 5.2.2 and Security notes in § 3.
OIA_06	For OIA_06, see Component predicate in § 5.1 and § 5.2.2.	see component predicate in § 5.1 and § 5.2.2.
OIA_07	For OIA_07, see Component prepare batches in § 5.2.1, Components zkSNARK wrapper in § 5 and § 5.1, and Component predicate in § 5.1 and § 5.2.2.	see component prepare batches in § 5.2.1, zkSNARK wrapper in § 5 and § 5.1, and component predicate in § 5.1 and § 5.2.2.
OIA_08	For OIA_08, see discussion in Security notes in § 6 and Components zkSNARK wrapper in § 5 and § 5.1.	see Security notes in § 3 and zkSNARK wrapper in § 5 and § 5.1.
OIA_09	For OIA_09, see discussion in Security notes in § 6 and Components zkSNARK wrapper in § 5 and § 5.1.	see Security notes in § 3 and zkSNARK wrapper in § 5 and § 5.1.
OIA_10	For OIA_10, see Component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.	see component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.
OIA_11	For OIA_11, see Component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.	see component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.
OIA_12	For OIA_12, see Prepare relation in § 5.2.1 and discussion in Security notes in § 6.	see Prepare relation in § 5.2.1 and Security notes in § 3.

continued on next page

Annex 2 ID	Requirement	Where in zkID
OIA_13	For OIA_13, see Prepare relation in § 5.2.1 and discussion in Security notes in § 6.	see Prepare relation in § 5.2.1 and Security notes in § 3.
OIA_14	For OIA_14, see Prepare relation in § 5.2.1 and discussion in Security notes in § 6.	see Prepare relation in § 5.2.1 and Security notes in § 3.
OIA_15	For OIA_15, see Prepare relation in § 5.2.1 and discussion in Security notes in § 6.	see Prepare relation in § 5.2.1 and Security notes in § 3.
OIA_16	For OIA_16, see Security notes in § 6 and Component predicate in § 5.1 and § 5.2.2.	see Security notes in § 3 and component predicate in § 5.1 and § 5.2.2.
Topic 10 — Issuance and Credential Handling (ISSU)		
ISSU_02	For ISSU_02, see components SD-JWT/mDL wrapper in § 5 and § 5.1 and Components zkSNARK wrapper in § 5 and § 5.1.	see SD-JWT/mDL wrapper in § 5 and § 5.1 and zkSNARK wrapper in § 5 and § 5.1.
ISSU_07	For ISSU_07, see components Prepare relation in § 5.2.1 and prepareCommit in § 5.2.1.	see Prepare relation in § 5.2.1 and `prepareCommit` in § 5.2.1.
ISSU_08	For ISSU_08, see components Prepare relation in § 5.2.1 and prepareCommit in § 5.2.1.	see Prepare relation in § 5.2.1 and `prepareCommit` in § 5.2.1.
ISSU_09	For ISSU_09, see components Prover's side discussion in § 6 and Security notes in § 6.	see Prover-side discussion in § 3 and Security notes in § 3.
ISSU_10	For ISSU_10, see components Prepare relation in § 5.2.1 and Prepare relation in § 5.2.	see Prepare relation in § 5.2.1 and Prepare relation in § 5.2.
ISSU_12	For ISSU_12, see components SD-JWT/mDL wrapper in § 5 and § 5.1 and Components zkSNARK wrapper in § 5 and § 5.1.	see SD-JWT/mDL wrapper in § 5 and § 5.1 and zkSNARK wrapper in § 5 and § 5.1.
ISSU_12a	For ISSU_12a, see components SD-JWT/mDL wrapper in § 5 and § 5.1 and Proof interface in § 5.2.	see SD-JWT/mDL wrapper in § 5 and § 5.1 and Proof interface in § 5.2.
ISSU_16	For ISSU_16, see components SD-JWT/mDL wrapper in § 5 and § 5.1 and Interface in § 5.	see SD-JWT/mDL wrapper in § 5 and § 5.1 and Interface in § 5.
ISSU_27	For ISSU_27, see components Show relation in § 5.2.2 and Show relation in § 5.2.	see Show relation in § 5.2.2 and Show relation in § 5.2.
ISSU_33	For ISSU_33, see components SD-JWT/mDL wrapper in § 5 and § 5.1 and Component commitment in § 5 and § 5.2.1.	see SD-JWT/mDL wrapper in § 5 and § 5.1 and component commitment in § 5 and § 5.2.1.
ISSU_33a	For ISSU_33a, see components SD-JWT/mDL wrapper in § 5 and § 5.1 and Component commitment in § 5 and § 5.2.1.	see SD-JWT/mDL wrapper in § 5 and § 5.1 and component commitment in § 5 and § 5.2.1.

continued on next page

Annex 2 ID	Requirement	Where in zkID
ISSU_33b	For ISSU_33b, see components prepareCommit in § 5.2.1 and SD-JWT/mDL wrapper in § 5 and § 5.1.	see `prepareCommit` in § 5.2.1 and SD-JWT/mDL wrapper in § 5 and § 5.1.
ISSU_35a	For ISSU_35a, see components prepareCommit in § 5.2.1 and Prepare relation in § 5.2.	see `prepareCommit` in § 5.2.1 and Prepare relation in § 5.2.
ISSU_37	For ISSU_37, see components Component prepare batches in § 5.2.1 and prepareBatch in § 5.2.1.	see component prepare batches in § 5.2.1 and `prepareBatch` in § 5.2.1.
ISSU_37a	For ISSU_37a, see components prepareBatch in § 5.2.1 and Proof interface in § 5.2.	see `prepareBatch` in § 5.2.1 and Proof interface in § 5.2.
ISSU_38	For ISSU_38, see components Show relation in § 5.2.2 and Show relation in § 5.2.	see Show relation in § 5.2.2 and Show relation in § 5.2.
ISSU_39	For ISSU_39, see components prepareBatch in § 5.2.1 and Prepare relation in § 5.2.	see `prepareBatch` in § 5.2.1 and Prepare relation in § 5.2.
ISSU_41a	For ISSU_41a, see components Show relation in § 5.2.2 and Show relation in § 5.2.	see Show relation in § 5.2.2 and Show relation in § 5.2.
ISSU_41b	For ISSU_41b, see components Component prepare batches in § 5.2.1 and Show relation in § 5.2.	see component prepare batches in § 5.2.1 and Show relation in § 5.2.
ISSU_41c	For ISSU_41c, see components Show relation in § 5.2.2 and Show relation in § 5.2.	see Show relation in § 5.2.2 and Show relation in § 5.2.
ISSU_44	For ISSU_44, see components Component prepare batches in § 5.2.1 and Show relation in § 5.2.	see component prepare batches in § 5.2.1 and Show relation in § 5.2.
ISSU_58	For ISSU_58, see components Show relation in § 5.2.2 and Show relation in § 5.2.	see Show relation in § 5.2.2 and Show relation in § 5.2.
ISSU_59	For ISSU_59, see components Show relation in § 5.2.2 and Show relation in § 5.2.	see Show relation in § 5.2.2 and Show relation in § 5.2.
ISSU_60	For ISSU_60, see components Component predicate in § 5.1 and § 5.2.2 and Proof interface in § 5.2.	see component predicate in § 5.1 and § 5.2.2 and Proof interface in § 5.2.
ISSU_61	For ISSU_61, see components Component predicate in § 5.1 and § 5.2.2 and Proof interface in § 5.2.	see component predicate in § 5.1 and § 5.2.2 and Proof interface in § 5.2.
ISSU_62	For ISSU_62, see components SD-JWT/mDL wrapper in § 5 and § 5.1 and Components zkSNARK wrapper in § 5 and § 5.1.	see SD-JWT/mDL wrapper in § 5 and § 5.1 and zkSNARK wrapper in § 5 and § 5.1.
ISSU_63	For ISSU_63, see components prepareCommit in § 5.2.1 and SD-JWT/mDL wrapper in § 5 and § 5.1.	see `prepareCommit` in § 5.2.1 and SD-JWT/mDL wrapper in § 5 and § 5.1.

continued on next page

Annex 2 ID	Requirement	Where in zkID
ISSU_64	For ISSU_64, see components Proof interface in § 5.2 and Proof interface in § 5.2.	see Proof interface in § 5.2.
Topic 23 — PID and (Q)EAA issuance		
Topic 23	23 PID issuance and (Q)EAA issuance. No HLRs, see Topic 10.	covered by Topic 10 components in § 5, § 5.1, § 5.2.1, § 5.2.2.
Topic 47 — Protocols and interfaces for PID and (Q)EAA issuance		
Topic 47	47 Protocols and interfaces for PID and (Q)EAA issuance and (non-)qualified. No HLRs, see Topic 10, 23.	covered by Topic 10 / Topic 23 components in § 5, § 5.1, § 5.2.1, § 5.2.2.
Topic 53 — Zero-Knowledge Proofs (ZKP)		
ZKP_01	For ZKP_01, see components Component predicate in § 5.1 and § 5.2.2 and Security notes in § 6.	see component predicate in § 5.1 and § 5.2.2 and Security notes in § 3.
ZKP_02	For ZKP_02, see components Prepare relation in § 5.2 and Component predicate in § 5.1 and § 5.2.2.	see Prepare relation in § 5.2 and component predicate in § 5.1 and § 5.2.2.
ZKP_03	For ZKP_03, see components Component commitment in § 5 and § 5.2.1 and Component linking in § 5.2.	see component commitment in § 5 and § 5.2.1 and component linking in § 5.2.
ZKP_04	For ZKP_04, see components Component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.	see component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.
ZKP_05	For ZKP_05, see components Prepare relation in § 5.2 and Show relation in § 5.2.	see Prepare relation in § 5.2 and Show relation in § 5.2.
ZKP_06	For ZKP_06, see components Components zkSNARK wrapper in § 5 and § 5.1 and SD-JWT/mDL wrapper in § 5 and § 5.1.	see zkSNARK wrapper in § 5 and § 5.1 and SD-JWT/mDL wrapper in § 5 and § 5.1.
ZKP_07	For ZKP_07, see components Prepare relation in § 5.2.1 and Component commitment in § 5 and § 5.2.1 in Security notes in § 6 and § 5.	see Prepare relation in § 5.2.1 and component commitment in § 5 and § 5.2.1 and Security notes in § 3.
ZKP_08	For ZKP_08, see components Backend modularity in § 5.3 and Security notes in § 6.	see backend modularity in § 5.3 and Security notes in § 3.
ZKP_09	For ZKP_09, see components Show relation in § 5.2.2 and Proof interface in § 5.2.	see Show relation in § 5.2.2 and Proof interface in § 5.2.

Table B — Requirements implementable by minor extension or modification of zkID components

Annex 2 ID	Requirement	Where in zkID
Topic 6 — Relying Party authentication and User approval		
RPA_01	For RPA_01, modify components Prepare relation in § 5.2.1 and Prover's side discussion in § 6.	modify components Prepare relation in § 5.2.1 and Prover's side discussion in § 3.
RPA_01a	For RPA_01a, modify components Show relation in § 5.2 and Proof interface in § 5.2.	modify components Show relation in § 5.2 and Proof interface in § 5.2.
RPA_02	For RPA_02, modify components Show relation in § 5.2 and Component predicate in § 5.1 and § 5.2.2.	modify components Show relation in § 5.2 and Component predicate in § 5.1 and § 5.2.2.
RPA_02a	For RPA_02a, modify components Proof interface in § 5.2 and Component linking in § 5.2.	modify components Proof interface in § 5.2 and Component linking in § 5.2.
RPA_03	For RPA_03, modify components Prepare relation in § 5.2.1 and Show relation in § 5.2.2.	modify components Prepare relation in § 5.2.1 and Show relation in § 5.2.2.
RPA_04	For RPA_04, modify components Prover's side discussion in § 6 and Security notes in § 6.	modify components Prover's side discussion in § 3 and Security notes in § 3.
RPA_05	For RPA_05, modify components Proof interface in § 5.2 and Component linking in § 5.2.	modify components Proof interface in § 5.2 and Component linking in § 5.2.
RPA_06	For RPA_06, modify components Component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.	modify components Component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.
RPA_06a	For RPA_06a, modify components Show relation in § 5.2 and Proof interface in § 5.2.	modify components Show relation in § 5.2 and Proof interface in § 5.2.
RPA_07	For RPA_07, modify components Show relation in § 5.2.2 and Component predicate in § 5.1 and § 5.2.2.	modify components Show relation in § 5.2.2 and Component predicate in § 5.1 and § 5.2.2.
RPA_07a	For RPA_07a, modify components Show relation in § 5.2 and Proof interface in § 5.2.	modify components Show relation in § 5.2 and Proof interface in § 5.2.
RPA_08	For RPA_08, modify components Show relation in § 5.2.2 and Show relation in § 5.2.	modify components Show relation in § 5.2.2 and Show relation in § 5.2.
RPA_09	For RPA_09, modify components Component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.	modify components Component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.

continued on next page

Annex 2 ID	Requirement	Where in zkID
RPA_10	For RPA_10, modify components Proof interface in § 5.2 and Proof interface in § 5.2.	modify components Proof interface in § 5.2 and Proof interface in § 5.2.
Topic 11 — Pseudonyms		
PA_01	For PA_01, modify components Component predicate in § 5.1 and § 5.2.2 and Prepare relation in § 5.2.	modify components Component predicate in § 5.1 and § 5.2.2 and Prepare relation in § 5.2.
PA_02	For PA_02, modify components Show relation in § 5.2 and Show relation in § 5.2.2.	modify components Show relation in § 5.2 and Show relation in § 5.2.2.
PA_03	For PA_03, modify components Proof interface in § 5.2 and Proof interface in § 5.2.	modify components Proof interface in § 5.2 and Proof interface in § 5.2.
PA_04	For PA_04, modify components Prepare relation in § 5.2 and Component predicate in § 5.1 and § 5.2.2.	modify components Prepare relation in § 5.2 and Component predicate in § 5.1 and § 5.2.2.
PA_05	For PA_05, modify components Proof interface in § 5.2 and Proof interface in § 5.2.	modify components Proof interface in § 5.2 and Proof interface in § 5.2.
PA_06	For PA_06, modify components Show relation in § 5.2 and Proof interface in § 5.2.	modify components Show relation in § 5.2 and Proof interface in § 5.2.
PA_07	For PA_07, modify components Proof interface in § 5.2 and Proof interface in § 5.2.	modify components Proof interface in § 5.2 and Proof interface in § 5.2.
PA_08	For PA_08, modify components Proof interface in § 5.2 and Proof interface in § 5.2.	modify components Proof interface in § 5.2 and Proof interface in § 5.2.
PA_08a	For PA_08a, modify components Security notes in § 6 and Proof interface in § 5.2.	modify components Security notes in § 3 and Proof interface in § 5.2.
PA_09	For PA_09, modify components Proof interface in § 5.2 and Proof interface in § 5.2.	modify components Proof interface in § 5.2 and Proof interface in § 5.2.
PA_10	For PA_10, modify components Show relation in § 5.2.2 and Security notes in § 6.	modify components Show relation in § 5.2.2 and Security notes in § 3.
PA_11	For PA_11, modify components Show relation in § 5.2.2 and Security notes in § 6.	modify components Show relation in § 5.2.2 and Security notes in § 3.
PA_12	For PA_12, modify components Show relation in § 5.2.2 and Show relation in § 5.2.2.	modify components Show relation in § 5.2.2 and Show relation in § 5.2.2.
PA_13	For PA_13, modify components Show relation in § 5.2.2 and Proof interface in § 5.2.	modify components Show relation in § 5.2.2 and Proof interface in § 5.2.

continued on next page

Annex 2 ID	Requirement	Where in zkID
PA_14	For PA_14, modify components Show relation in § 5.2.2 and Security notes in § 6.	modify components Show relation in § 5.2.2 and Security notes in § 3.
PA_15	For PA_15, modify components Prepare relation in § 5.2.1 and Security notes in § 6.	modify components Prepare relation in § 5.2.1 and Security notes in § 3.
PA_16	For PA_16, modify components Prepare relation in § 5.2.1 and Component predicate in § 5.1 and § 5.2.2.	modify components Prepare relation in § 5.2.1 and Component predicate in § 5.1 and § 5.2.2.
PA_17	For PA_17, modify components Component prepare batches in § 5.2.1 and Prepare relation in § 5.2.1.	modify components Component prepare batches in § 5.2.1 and Prepare relation in § 5.2.1.
PA_18	For PA_18, modify components prepareCommit in § 5.2.1 and Security notes in § 6.	modify components `prepareCommit` in § 5.2.1 and Security notes in § 3.
PA_19	For PA_19, modify components Proof interface in § 5.2 and Security notes in § 6.	modify components Proof interface in § 5.2 and Security notes in § 3.
Topic 17 — Identity matching		
...		
Topic 18 — Combined presentations of attributes		
ACP_01	For ACP_01, modify components Component predicate in § 5.1 and § 5.2.2 and Component linking in § 5.2.	modify components Component predicate in § 5.1 and § 5.2.2 and Component linking in § 5.2.
ACP_02	For ACP_02, modify components Component commitment in § 5 and § 5.2.1 and Show relation in § 5.2.2.	modify components Component commitment in § 5 and § 5.2.1 and Show relation in § 5.2.2.
ACP_03	For ACP_03, modify components Prepare relation in § 5.2 and Show relation in § 5.2.	modify components Prepare relation in § 5.2 and Show relation in § 5.2.
ACP_04	For ACP_04, modify components Proof interface in § 5.2 and Proof interface in § 5.2.	modify components Proof interface in § 5.2 and Proof interface in § 5.2.
ACP_05	For ACP_05, modify components Component commitment in § 5 and Component predicate in § 5.1 and § 5.2.2.	modify components Component commitment in § 5 and Component predicate in § 5.1 and § 5.2.2.
ACP_06	For ACP_06, modify components prepareCommit in § 5.2.1 and prepareBatch in § 5.2.1.	modify components `prepareCommit` in § 5.2.1 and `prepareBatch` in § 5.2.1.
ACP_07	For ACP_07, modify components Prepare relation in § 5.2.1 and Component prepare batches in § 5.2.1.	modify components Prepare relation in § 5.2.1 and Component prepare batches in § 5.2.1.

continued on next page

Annex 2 ID	Requirement	Where in zkID
Topic 20 — Strong User authentication for electronic payments		
SUA_01	For SUA_01, modify components Show relation in § 5.2.2 and Show relation in § 5.2.	modify components Show relation in § 5.2.2 and Show relation in § 5.2.
SUA_02	For SUA_02, modify components Component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.	modify components Component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.
SUA_03	For SUA_03, modify components Show relation in § 5.2 and Prepare relation in § 5.2.	modify components Show relation in § 5.2 and Prepare relation in § 5.2.
SUA_04	For SUA_04, modify components Show relation in § 5.2.2 and Component predicate in § 5.1 and § 5.2.2.	modify components Show relation in § 5.2.2 and Component predicate in § 5.1 and § 5.2.2.
SUA_05	For SUA_05, modify components Security notes in § 6 and Show relation in § 5.2.	modify components Security notes in § 3 and Show relation in § 5.2.
SUA_06	For SUA_06, modify components Component predicate in § 5.1 and § 5.2.2 and Proof interface in § 5.2.	modify components Component predicate in § 5.1 and § 5.2.2 and Proof interface in § 5.2.
Topic 43 — Embedded disclosure policies		
EDP_01	For EDP_01, modify components Component commitment in § 5 and § 5.2.1 and prepareCommit in § 5.2.1.	modify components Component commitment in § 5 and § 5.2.1 and `prepareCommit` in § 5.2.1.
EDP_02	For EDP_02, modify components Component predicate in § 5.1 and § 5.2.2 and Proof interface in § 5.2.	modify components Component predicate in § 5.1 and § 5.2.2 and Proof interface in § 5.2.
EDP_03	For EDP_03, modify components Prover's side discussion in § 6 and Security notes in § 6.	modify components Prover's side discussion in § 3 and Security notes in § 3.
EDP_05	For EDP_05, modify components Proof interface in § 5.2 and Proof interface in § 5.2.	modify components Proof interface in § 5.2 and Proof interface in § 5.2.
EDP_06	For EDP_06, modify components Component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.	modify components Component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.
EDP_07	For EDP_07, modify components Component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.	modify components Component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.
EDP_09	For EDP_09, modify components prepareCommit in § 5.2.1 and SD-JWT/mDL wrapper in § 5 and § 5.1.	modify components `prepareCommit` in § 5.2.1 and SD-JWT/mDL wrapper in § 5 and § 5.1.
EDP_10	For EDP_10, modify components prepareCommit in § 5.2.1 and Proof interface in § 5.2.	modify components `prepareCommit` in § 5.2.1 and Proof interface in § 5.2.

continued on next page

Annex 2 ID	Requirement	Where in zkID
EDP_11	For EDP_11, modify components Security notes in § 6 and Prepare relation in § 5.2.	modify components Security notes in § 3 and Prepare relation in § 5.2.
Topic 51 — PID or attestation deletion		
PAD_01	For PAD_01, modify components Proof interface in § 5.2 and Proof interface in § 5.2.	modify components Proof interface in § 5.2 and Proof interface in § 5.2.
PAD_02	For PAD_02, modify components prepareCommit in § 5.2.1 and SD-JWT/mDL wrapper in § 5 and § 5.1.	modify components `prepareCommit` in § 5.2.1 and SD-JWT/mDL wrapper in § 5 and § 5.1.
PAD_03	For PAD_03, modify components Component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.	modify components Component predicate in § 5.1 and § 5.2.2 and Show relation in § 5.2.
PAD_04	For PAD_04, modify components Show relation in § 5.2.2 and Show relation in § 5.2.	modify components Show relation in § 5.2.2 and Show relation in § 5.2.
PAD_05	For PAD_05, modify components Security notes in § 6 and Component commitment in § 5 and § 5.2.1.	modify components Security notes in § 3 and Component commitment in § 5 and § 5.2.1.
PAD_06	For PAD_06, modify components Component prepare batches in § 5.2.1 and Show relation in § 5.2.	modify components Component prepare batches in § 5.2.1 and Show relation in § 5.2.
Topic 52 — Relying Party intermediaries		
...		