# Anonymous Credentials from ECDSA

Matteo Frigo\* Google abhi shelat<sup>+</sup> Google

December 21, 2024

#### Abstract

Anonymous digital credentials allow a user to prove possession of an attribute that has been asserted by an identity issuer without revealing any extra information about themselves. For example, a user who has received a digital passport credential can prove their "age is > 18" without revealing any other attributes such as their name or date of birth.

Despite inherent value for privacy-preserving authentication, anonymous credential schemes have been difficult to deploy at scale. Part of the difficulty arises because schemes in the literature, such as BBS+ [CDL16], use new cryptographic assumptions that require system-wide changes to existing issuer infrastructure. In addition, issuers often require digital identity credentials to be *device-bound* by incorporating the device's secure element into the presentation flow. As a result, schemes like BBS+ require updates to the hardware secure elements and OS on every user's device.

In this paper, we propose a new anonymous credential scheme for the popular and legacy-deployed Elliptic Curve Digital Signature Algorithm (ECDSA) signature scheme. By adding efficient ZK arguments for statements about SHA256 and document parsing for ISO-standardized identity formats, our anonymous credential scheme is that first one that can be deployed *without* changing any issuer processes, *without* requiring changes to mobile devices, and *without* requiring non-standard cryptographic assumptions.

Producing ZK proofs about ECDSA signatures has been a bottleneck for other ZK proof systems because standardized curves such as P256 use finite fields which do not support efficient number theoretic transforms. We overcome this bottleneck by designing a ZK proof system around sumcheck and the Ligero argument system, by designing efficient methods for Reed-Solomon encoding over the required fields, and by designing specialized circuits for ECDSA.

Our proofs for ECDSA can be generated in 60ms. When incorporated into a fully standardized identity protocol such as the ISO MDOC standard, we can generate a zero-knowledge proof for the MDOC presentation flow in 1.2 seconds on mobile devices depending on the credential size. These advantages make our scheme a promising candidate for privacypreserving digital identity applications.

<sup>\*</sup>matteof@google.com

<sup>&</sup>lt;sup>+</sup>shelat@google.com

# Contents

| 1 | Introduction   | 3  |
|---|--|----|
|   | 1.1 Anonymous Credentials from ECDSA                     | 5  |
| 2 | Zero-Knowledge Argument System                           | 7  |
|   | 2.1 Verify Interactive Protocol (IP)                     | 8  |
|   | 2.2 Ligero Zero-Knowledge System                         | 11 |
|   | 2.3 ZK Protocol  | 12 |
|   | 2.4 Applying the Fiat-Shamir Transform                   | 13 |
| 3 | Optimization   | 14 |
|   | 3.1 Reed-Solomon encoding over prime fields              | 14 |
|   | 3.2 Reed-Solomon encoding over binary fields             | 15 |
|   | 3.3 Reduction of the Ligero proof size                   | 21 |
|   | 3.4 Input consistency for circuits over different Fields | 22 |
| 4 | Circuit design   | 24 |
| • | 4.1 ECDSA Signature Verification                         | 24 |
|   | 4.2 SHA-256  | 25 |
|   | 4.3 CBOR parsing   | 27 |
| 5 | Evaluation   | 33 |
|   | 5.1 Algebraic primitives                                 | 33 |
|   | 5.2 SHA256 hash verification                             | 35 |
|   | 5.2.1 Comparison with Ligero                             | 36 |
|   | 5.2.2 Comparison with Binius                             | 38 |
|   | 5.2.3 Comparison with other ZK systems                   | 39 |
|   | 5.3 ECDSA signature verification                         | 39 |
| 6 | Applications   | 40 |
|   | 6.1 A simple anonymous credential                        | 40 |
|   | 6.2 Anonymous Credentials for MDOC                       | 41 |
| A | Interpolation via convolution                            | 47 |
| B | The MDOC standard  | 48 |

## 1 Introduction

David Chaum [Cha85] introduced the idea of an *anonymous credential system*, or *pseudo-nym system* as a method to provide individuals full control over the dissemination of personal information. Such a system consists of users, issuers, and relying parties. Issuers can issue a set of digital credentials to a user; users can prove to a relying party that they possess credentials satisfying a certain property without revealing anything more than the fact that they own credentials that satisfy the property. Colluding relying parties are unable to link a specific user to a pair of interactions they participate in. Even a colluding issuer and relying party should not be able to link a specific user to a session.

After Chaum introduced this notion, early constructions of such schemes based on specific signature schemes were presented, but they all involved restrictions on usage. Lysyanskaya, Rivest, Sahai and Wolf [LRSW99] devised a scheme based on one-way functions and generic zero-knowledge protocols. They also proposed a specific construction based on a non-standard discretelogarithm assumption that also had restrictions on credential issuance and use to achieve the unlinkability property. Brands [Bra99] introduces a selective disclosure system based on certificates.

Camenisch and Lysyanskaya [CL01] introduced an RSA-based credential system that satisfied all of the multi-use and unlinkability properties desirable in a credential system, but their scheme still required several hundred exponentiation operations for a basic presentation and relied on a novel cryptographic assumption. They observed that one key problem for an anonymous credential scheme is to design a signature scheme for which there is also an efficient proof of knowledge of a signature. Later, Camenisch, Drijvers, and Lehmann presented the BBS+ credential scheme [CDL16] as an efficient method to prove knowledge of a signature and thus construct an anonymous credential system. Such a scheme has been proposed for standardization, however, it still relies on a new cryptographic assumption that requires pairing-friendly elliptic curves.

Kosba et al. [KZM<sup>+</sup>15] propose the use of general purpose zkSNARKS to build anonymous credentials based on *standardized* signature schemes. In one of their schemes, a credential consists of a 1KB message that is signed using RSA-PSS with a 2048-bit key and SHA-256 as the hash function. A user can prove one inequality between a value in the message and an external one (i.e., the basic age-over-X application). Their prover runs in 54–108s depending on parameters. Later, Delignat-Lavaud, Fournet, Kohlweiss, and Parno [DLFKP16] constructed a zkSNARK system for proving possession of an X509 certificate for RSA signatures, which requires parsing the ASN.1 format in the ZK circuit. Their system took 31s to produce just the RSA signature proof, and took between 61-160s for the entire anonymous credential presentation. In addition to the impractical time to present a credential, both of these proof systems also require a costly *trusted parameter setup*, and they also require using pairing-friendly curves like the BBS+ scheme.

Along a similar line, Rosenberg, White, Garmin and Miers [RWGM23] build

anonymous credentials using general ZK techniques applied to non-standardized signature and hash schemes. Their system gains an advantage by relying on a blockchain to establish *common knowledge*: an issuer must augment a public data structure every time they issue a credential to a user. The user refers to this public datastructure when they present a credential. As a result, their system reduces the credential presentation problem to a ZK proof of set membership. In addition, they use a non-standard hash function (Poseidon) to make substantially more efficient ZK proofs concerning set membership. Altogether, the addition of the blockhain and the use of Poseidon allows their proof system to run between 460–542ms. In an alternative formulation using (non-standard) Schnorr signatures over a (non-standard) pairing-friendly curve, their proof system runs in 130ms. Their prover requires up to 800mb of memory to produce a proof, and as before, a trusted parameter that is several 100s of megabytes and trusted by all issuers and relying parties in the system, and finally, a computational hardness assumption about bilinear pairings that has not been standardized.

**Impediments** While there have been attempts to deploy credential systems for digital identity applications, such efforts encounter three major roadblocks.

The first problem is pedantic yet significant. Organizations that issue credentials have embraced and deployed *standardized* digital signature hardware and software that has been blessed by organizations such as NIST. Specifically, for state or nation-level issuers, many issuers have deployed hardware security modules that only supports RSA or ECDSA signatures. Through informal interview, it appears that such organizations are unwilling or incapable of making the investments necessary to update their infrastructure to support more recent CL or BBS+ credential systems, or even pairing-friendly curves for that matter. Even the current anonymous credential schemes that are based on *standardized cryptography* are unsuitable: as discussed above, they all require minutes to produce proofs, they require large trusted setup parameters, and they also require accepting new cryptographic assumptions about curves with bilinear pairings.

A second problem is that credentials are often issued by multiple organizations (e.g., each state issues a driver's license), and relying parties need a system that is inter-operable among all issuers. The task of producing a *single* common trusted parameter via a ceremony involving *all* issuers seems highly impractical and unlikely. To date, all of the solutions based on standardized cryptograpgy in the literature require trusted parameters, and are thus challenging to deploy.

The final, and most critical impediment is the potential for *scaled abuse* of anonymous credentials. If credentials can be copied from one malicious user to another, then the system that verifies credentials may not be able to provide any guaranteed soundness. Unlike traditional credentials which can be identified and *revoked* when they are discovered to have been copied or forged, an anonymous credential does not leak any information other than the shared

attributes during a presentation protocol. As a result, it is more difficult or impossible to revoke such credentials just by observing the presentation flows. Note that this problem arises even in the setting of certificate systems that only offer selective disclosure properties without full anonymity.

To address this *copied credential* problem, issuers of digital identity credentials often require users to hold their credential in some form of trusted hardware. Modern mobile devices often contain a *secure element* which is a special hardware module that is purported to be tamper-resistant, side-channel resistant, and thus secure against most key extraction attacks. The identity protocols in the ISO MDOC standard, for example, specify an issuance and a presentation flow that validate the presence of the secure element on the receiving device, and the participation of the secure element during every presentation flow. Unfortunately, these secure elements are difficult to change in the field, and they currently only support legacy standardized signature schemes such as RSA and ECDSA. In particular, they do not support performing operations on pairing-friendly curves, lattices, or even well-deployed signature schemes such as EDDSA.

#### 1.1 Anonymous Credentials from ECDSA

To address all of the shortcomings noted above, namely (a) efficiency, (b) requirement to use standardized signature schemes and hash functions, and (c) no need for common knowledge or trusted setups, we present a performant anonymous credential that builds on ECDSA signatures.

Specifically, following the approach from prior works, we first construct an efficient proof of knowledge of an ECDSA signature and then develop the other tools necessary for an issuer to issue a set of credentials or attributes to a user. We then combine these into a protocol that allows a user to demonstrate posession of an arbitrary subset of those credentials to a relying party.

In contrast to prior work, our proposed proof of knowledge of an ECDSA signature runs in almost the same time as a BBS+ proof, and it only relies on the security of a collision-resistant hash function such as the standardized SHA256 function. The key to our approach is to combine two recent ZK proof techniques: an optimized verifiable computation protocol that is based on sumcheck, and the Ligero ZK proof system that is based on the IOP (or more precisely, IPCP) model. The use of sumcheck in recent ZK constructions has shown promise: it avoids having to compute an expensive Number Theoretic Transform (NTT) on the full witness for a given circuit. Instead, it is possible to run a sumcheck-based verification protocol in roughly *linear* time in the size of the theorem-verifying circuit. Moreover, verifiable computation protocols that are based on sumcheck are information-theoretically secure as they require no computational assumptions. Similarly, the Ligero proof system in the IPCP model requires the instantiation of a secure proof oracle, which can also be achieved using Merkle trees that only rely on cryptographic hash functions. In practice, we instantiate our protocols in a non-interactive manner, and thus rely on the random oracle model for the soundness and zero-knowledge properties.

**Key Results** One key bottleneck to proving knowledge of an ECDSA signature relates to the elliptic curves that have been standardized, and specifically to the properties of their respective finite fields. Standardized curves for ECDSA such as P256 are designed without regard to the NTT in that they do not have the required roots of unity needed for efficient NTT algorithms. As a result, prior work essentially *emulates* the arithmetic of the P256 field in *another* finite field that is optimized for ZK proofs. Even using very clever arithmetic tricks, this overhead amounts to 40–100x increase in arithmetic operations.

We overcome this bottleneck in a few ways. First, we design a specific NTT in a quadratic field extension of primes such as P256 which allows us to perform NTTs at roughly 4x the cost of doing so in an NTT-friendly prime of the same size<sup>1</sup>. Note that this extra cost makes it even more important to minimize the size of the witness.

We also develop a novel Reed-Solomon encoding approach which allows exploitation of the subfield structure of the field, yielding a constant-factor improvement in proof size over the standard NTT-based methods. Specifically, most projects define the Reed-Solomon code as a polynomial interpolation over the roots of unity. In contrast, we define the code as an interpolation over an arithmetic sequence of evaluation points in the base field. We present a method to perform the interpolation over these points using a linear convolution. Thus, when our witness values are over a small subfield or base field, even though the NTT itself may be carried out over a larger extension field, the resulting codeword remains in the subfield, yielding a smaller proof.

In some cases, we need to include zero-knowledge arguments about the pre-image of certain boolean-heavy functions such as SHA256. Inspired by the work of Diamond and Posen [DP23, DP24], we recognize the inherent speedup possible with using the extension field  $GF(2^k)$  to perform the sumcheck. In this case, the standard NTT algorithms no longer apply, and we instead use the family of additive FFTs [LCH14]. Specifically, we develop a new *bidirectional truncated* additive FFT to solve the problem of polynomial interpolation over cosets. As our benchmarks show, this operation provides almost an order of magnitude speedup in these basic algebraic operations.

The next component of our result arises from the specific circuits we have developed for verifying ECDSA signatures, verifying the pre-image of SHA-256, and for parsing nested document formats such as CBOR and JSON that are often specified in current identity credential standards such as ISO MDOC.

Our final contribution is a method to achieve the best performance when a credential verification task involves both ECDSA signatures and SHA-256 hashes. All prior ZK proof systems verify a single arithmetic circuit that is defined over one finite field **F**. However, as our benchmarks show, it is almost an order of magnitude faster to verify properties about SHA-256 preimages

<sup>&</sup>lt;sup>1</sup>We believe this insight can be used by other proof systems as well.

using a binary field instead of a prime field. A natural approach is to verify parts of a statement under a convenient circuit over  $\mathbb{F}_p$ , while verifying other constraints under a more convenient field  $GF(2^k)$ . A problem arises when a common witness value must be used in both parts if this circuit. A malicious prover may include inconsistent versions of this witness value in the two circuits. To address this problem, we introduce a new *witness consistency protocol* in which two witnesses that are committed in two different ZK proofs over different fields can be proven to be consistent with high probability. Our basic approach here is to include the evaluation of a random MAC on the witness value—a computation that is small and thus easy to simulate over any field in both  $\mathbb{F}$  and  $GF(2^k)$ . We measure the overhead this consistency check and show that after the SHA-256 pre-image verification exceeds a few blocks, our consistency method begins to be more efficient.

**Special notes about this application** Some recent ZK NARK systems produce very small proofs on the order of hundreds of bytes. These systems, however, require a trusted parameter setup ceremony. As noted above, in an anonymous credential system that involved many issuers, relying parties, and users, we consider such a setup ceremony as too difficult to organize, and thus focus on ZK solutions that do not require any trusted setup.

Additionally, while the system we develop is succinct in proof size, many recent ZK systems are also succinct in verifier-time. In the anonymous credential application, however, it is more important to conserve *prover time*, because the prover is often the user's mobile device. Therefore, the goal of our system is instead to reduce prover time and energy use.

## 2 Zero-Knowledge Argument System

We construct a zero-knowledge argument (ZKARG) system by composing the Ligero proof system with a public-coin verifiable computation (VC) protocol that is based on the sumcheck protocol. A VC protocol allows a prover to convince a verifier that C(x) = 0 in time and space that is smaller than |C|. Remarkably, such protocols have been implicit in the literature since the introduction of sumcheck [LFKN92].

Our approach to constructing a ZK argument follows the same pattern of Hyrax [WTas<sup>+</sup>18], which itself is an instantiation of the idea from Ben-Or et al. [BOGG<sup>+</sup>88]. Specifically, on input a circuit *C*, public input *x*, and witness *w*, the prover first sends a commitment com to *w* and commitment parameters. The prover and verifier then interactively produce a *committed* transcript *T* for a VC protocol on input (*C*, *x*); specifically, when the VC protocol instructs the prover to send its *i*<sup>th</sup> message *m<sub>i</sub>* to the verifier, it instead sends a commitment  $t'_i$  to that message; the verifier always responds to the prover's *i*<sup>th</sup> commitment with a random challenge as per the usual public-coin VC protocol and defers performing all of the prescribed verification steps. Finally, the prover

and verifier engage in a zero-knowledge proof for the theorem " $(t'_1, \ldots, t'_\ell)$  is a commitment to transcript *t* and *t* is an accepting VC transcript for (C, x)." In particular, the ZK proof for this theorem implicitly ensures that all of the verifier's VC steps hold with respect to the original transcript *t*.

The resulting composition enjoys efficiencies. Because VC produces a succinct proof with respect to *C*, the Ligero proof in our composed system is *smaller* than a standalone Ligero proof for C(x) = 0. As we show below, this significantly improves the performance of the overall system because the most costly component of Ligero, the NTT, is applied to a small transcript for C(x) = 0 instead of a full tableau for the computation of C(x). Note that the VC protocol is not a zero-knowledge protocol, but Ligero is, and in this case, the composition yields a protocol that satisfies the zero-knowledge property.

In the rest of this section, we explain each component in more detail, and provide formal descriptions of the complete protocol and its performance.

#### 2.1 Verify Interactive Protocol (IP)

We describe a verifiable computation protocol that is derived from a long line of work on VC that is based on the sumcheck protocol [GKR08, CMT12, Tha13, WJB<sup>+</sup>17, WTas<sup>+</sup>18].

**Sumcheck** The sumcheck interactive protocol [LFKN92] allows a Prover to efficiently convince a Verifier that a claimed value equals the sum of a polynomial evaluated at every point on the boolean hypercube. Here we quote a performance theorem taken from Thaler [Tha22] characterizing sumcheck.

**Theorem 2.1** ([Tha22, Proposition 4.1]). Let g be a v-variate polynomial of degree at most d in each variable, defined over a finite field  $\mathbb{F}$ . For any specified  $H \in \mathbb{F}$ , let  $\mathcal{L}_H$  be the language of polynomials g (given as an oracle) such that

$$H = \sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \dots \sum_{b_v \in \{0,1\}} g(b_1, \dots, b_v) .$$
<sup>(1)</sup>

The sum-check protocol is an interactive proof system for  $\mathcal{L}_H$  with completeness error  $\delta_c = 0$  and soundness error  $\delta_s \leq vd/|\mathbb{F}|$ .

Moreover, if deg(g) = O(1), then the protocol requires v rounds of communication, the total communication cost is O(v) field elements, the verifier time is O(v) + T(g), and the prover's time is O(M(g)) where T represents the cost of 1 evaluation of g and M represents the cost of evaluating g on the hypercube.

As a notational convenience, we often use capital letters to represent a vector of variables over  $\{0,1\}$ . For example, letting  $B = (b_1, ..., b_v)$ , we write:

$$H = \sum_{B \in \{0,1\}^v} g(B)$$

or even  $H = \sum_{B} g(B)$  as shorthand for Equation (1).

**Applying sumcheck for circuit verification** Thaler [Tha22] provides a comprehensive overview of the history and application of this protocol to the task of verifiable computing. In particular, as first shown in [GKR08], sumcheck can be applied to the task of verifying that C(x) = 0 where *C* a *layered circuit* starting with layer 0 as the output layer, and *d* as the input layer.

We present an optimized variation of this style of protocol that builds on observations by Cormode et al. [CMT12], Thaler [Tha13], and other follow-ups [WJB<sup>+</sup>17, WTas<sup>+</sup>18]. Our improvements consist of a simplification of the wiring predicate, a different approach to the layer-to-layer reduction, and different methods to compute the evaluation of the sparse wiring predicate.

In our formulation, layer *j* of the circuit computes an array  $V_j[0, ..., w_j - 1]$  of output wires given an array  $V_{j+1}[0, ..., w_{j+1} - 1]$  of input wires, which are the outputs of layer (j + 1). Instead of the separate addition and multiplication gates of [GKR08], we have only one kind of "gate" which computes the generic quadratic form

$$V_j[i] = \sum_{\ell,r} Q_j[i,\ell,r] \cdot V_{j+1}[\ell] \cdot V_{j+1}[r]$$
 ,

and specialize sumcheck for this circuit structure. We call *Q* the *quad* and  $Q_i[i, \ell, r]$  the *quad term*.

By convention, the input wire  $V_j[0] = 1$  for all layers, and thus the quad representation handles the classic add and multiplication gates in a uniform manner. For example, this quad form approach reduces the depth of the standard elliptic curve addition formula from 5 to 3. Also by convention, an input satisfies the circuit if all outputs  $V_0[i]$  of the output layer are 0.

The *arrays* defined above naturally specify multi-variate—and specifically multi-linear—polynomials. We use tilde-notation to denote the unique multi-linear extension for an array. For example,  $\tilde{V}_i(i)$  represents the unique multi-linear extension that interpolates the array  $V_i[i]$ .

Let  $\omega_j = \lceil \log_2(w_j) \rceil$ . At the beginning of the protocol, the verifier picks a random point  $r_0 \in \mathbb{F}^{\omega_0}$ . Because all outputs are expected to be 0, the verifier expects  $\tilde{V}_0(r_0) = 0$ . Our process begins with *two* claims  $(c_{0,0}, c_{0,1}) = (0, 0)$  on the values  $\tilde{V}_0(r_0)$ . In this first step, both claims are on values on the same evaluation point, but in later rounds, these claims will be on different points.

**Protocol 2.2.** Verify<sub>*P*,*V*</sub>(*C*, *x*) for theorem C(x) = 0 –

The depth-*d* circuit *C* is defined over field  $\mathbb{F}$ . Prover and Verifier hold input (C, x). Let  $(Q_0, \ldots, Q_{d-1})$  be the quad form for each layer of *C*, and define  $w_j$  to be the number of wires in layer *j* and  $\omega_j = \lceil \log_2(w_j) \rceil$ . Let  $V_j[i]$  represent the value of the *i*<sup>th</sup> output wire in layer *j*. Verifier rejects if any check fails.

- 1. Verifier samples  $r \leftarrow \mathbb{F}^{\omega_0}$  and sets  $r_{0,b} \leftarrow r$ ,  $c_{0,b} \leftarrow 0$  for  $b \in \{0,1\}$ .
- 2. For j = 0, 1, ..., d 1: // Prover is asserting claims  $c_{j,b} = \tilde{V}_j(r_{j,b}), b \in \{0, 1\}$ .

- (a) Verifier sends  $\alpha_i \leftarrow \mathbb{F}$  and sets  $Y = \alpha_i \cdot c_{i,0} + (1 \alpha_i)c_{i,1}$ .
- (b) Define the  $2\omega_i$ -variate polynomial

$$g_j(L,R) = \left[\alpha_j \cdot \tilde{Q}_j(r_{j,0},L,R) + (1-\alpha_j)\tilde{Q}_j(r_{j,1},L,R)\right]$$
$$\cdot \tilde{V}_{i+1}(L) \cdot \tilde{V}_{i+1}(R)$$

in 
$$L = L_0, \ldots, L_{\omega_i - 1}$$
 and  $R = R_0, \ldots, R_{\omega_i - 1}$ 

(c) Prover and Verifier apply sumcheck to the claim:

$$Y = \sum_{L,R \in \{0,1\}^{\omega_j}} g_j(L,R)$$

The sumcheck protocol produces two random points  $\ell$  and r, a value y, and a claim that  $y \stackrel{?}{=} g_i(\ell, r)$ .

- (d) Prover sends claims  $c_{j+1,0} = \tilde{V}_{j+1}(\ell)$  and  $c_{j+1,1} = \tilde{V}_{j+1}(r)$ .
- (e) Verifier checks:

$$y \stackrel{\prime}{=} \left[ \alpha \cdot \tilde{Q}_{j}(r_{j,0},\ell,r) + (1-\alpha)\tilde{Q}_{j}(r_{j,1},\ell,r) \right] \cdot c_{j+1,0} \cdot c_{j+1,1}$$

and continues with points  $(r_{j+1,0}, r_{j+1,1}) = (\ell, r)$  and claims  $c_{j+1,b}$ .

3. Verifier verifies the claims  $c_{d,b} \stackrel{?}{=} \tilde{V}_d(r_{d,b})$  by direct evaluation of  $\tilde{V}_d(r_{d,b})$ .

**Theorem 2.3.** Let C be a depth-d layered circuit over  $\mathbb{F}$  whose  $j^{th}$  layer has width  $w_j$ , let  $\omega_j = \lceil \log(w_j) \rceil$  and let  $Z = 2\sum_{i=1}^{d} \omega_i$ .

*Verify*<sub>P,V</sub>(C, x) *is a Z-round interactive protocol for the theorem* C(x) = 0 *that has completeness error 0, soundness error*  $O(Z/|\mathbb{F}|)$  *and total communication*  $\Theta(Z)$  *field elements.* 

Moreover, the verifier computation consists of sampling  $\Theta(Z)$  elements from  $\mathbb{F}$ , performing  $\Theta(Z)$  operations in  $\mathbb{F}$ , and performing one dot product between the input *x* and a vector with tensor form derived from the randomly sampled challenges.

*The Prover computation consists of* O(|C|) *field operations.* 

**Remarks** Achieving the stated runtime requires careful, but well-known optimizations of sumcheck for the Prover and Verifier, as well as an efficient method to compute the evaluation of the wiring predicate  $\tilde{Q}$  at each layer.

For convenience, the notation sz(C, x) represents this protocol's transcript length on instance (C, x).

It is clear that step 2e can be moved outside of the loop. Moreover, all of the checks that the verifier makes in Sumcheck in step 2(c) can also be moved

outside of the loop. In this formulation, the Verifier can be split into the interactive part that includes step 1, 2(a), and the random sampling steps in 2(c), and a second part that performs all of the checks in steps 2(c),2e, and (3). For convenience, we use the notation  $\text{check}_V(C, x)$  for the second part.

**SIMD optimization** It is well known that sumcheck can be used to exploit *regularity* in the wiring predicates. For example, when *C* validates multiple instances or copies of the same smaller circuit, it is possible to achieve verifier succinctness in time [CMT12, Tha13, WJB+17]. These extensions can be added to our formulation above by including a *copy* variable, but are omitted because our system does not require verifier succinctness.

#### 2.2 Ligero Zero-Knowledge System

Ligero [AHIV22] is a zero-knowledge interactive PCP (ZKIPCP) that represents an optimized instantiation of the "MPC-in-the-head" technique. The proof size is roughly the square-root in the size of the circuit, and the proof system only assumes the existence of a cryptographic hash function. Note that an IPCP is a special case of the an IOP in which the prover sends a proof oracle only in the first round and in the subsequent rounds it simply responds to the verifier's challenge a message instead of an oracle<sup>2</sup>.

Whereas earlier we considered a circuit *C* with input *x*. From this point in the paper, we further split the input into a pair (x, w) where *x* represents the *public input* that is known to both Prover and Verifier, and *w* is the *private witness* that is known only to the Prover.

Our ZK protocol uses Ligero to *wrap* the check<sub>V</sub> verifier from the previous section, i.e, it can be interpreted as composition of an efficient verifiable computation protocol with a generic zero-knowledge protocol.

For convenience, we identify some components of the Ligero protocol. The LigeroCommit(x, w) method allows a Prover to produce a short commitment com to inputs (x, w). Next, prover and verifier respectively run LigeroProver on inputs (C, x, w, com) and LigeroVerify on inputs (C, x, com), and finally, the verifier method outputs accept or reject.

Ames et al. [AHIV22] characterize the performance of Ligero as follows.

**Theorem 2.4** ([AHIV22, Cor 5.3]). *Fix parameters n, m, l, k, t, e such that e* < (n-k)/2 and n > 2k + e. Let  $C : \mathbb{F}^n \to \mathbb{F}$  be an arithmetic circuit of size s, where  $|\mathbb{F}| \ge l + n, m \cdot l > nl + s$  and k > l + t. Then the protocol Ligero( $C, \mathbb{F}$ ) is a non-interactive proof with:

- Adaptive knowledge error  $\epsilon_{FSK}(C, T, \lambda) = T \cdot \epsilon(C) + 3(T^2 + 1)/2^{\lambda}$  against *T*-query provers;
- Adaptive knowledge error  $\Omega(T \cdot \epsilon_{FSK})$  against  $T O(q(C) \log(|C|))$ -query quantum provers;

<sup>&</sup>lt;sup>2</sup>The treatment of subsequent Prover communication as messages instead of oracles consisting of a single point enables a simpler and more efficient compilation from IOP to protocol.

- Statistical zero knowledge  $z'(C, \lambda) = 4|C|/2^{\lambda/4}$
- Efficiency: V makes t queries of symbols of size  $\mathbb{F}^{4m+5\sigma}$ . The number of field operations is poly(|C|, n). The number of field elements communicated by P to V is  $\sigma \cdot n + \sigma (k + l 1) + \sigma (2k 1)$ .

where

$$\epsilon(C) = \epsilon_K(C) = \max\left(\frac{(n+2)}{|F|^{\sigma}}, (1-e/n)^t\right)$$
$$|C| = n + (k+\ell-1) + (2\cdot k - 1)$$

#### 2.3 ZK Protocol

We now describe our new ZK protocol.

**Protocol 2.5. ZK protocol** (P, V) for circuit C(x, w) = 0

The circuit *C* is defined over field  $\mathbb{F}$ . Prover holds input (C, (x, w)) and Verifier holds input (C, x).

- 1. *P*, *V* compute the size  $\ell \leftarrow sz(C)$  of the Verify transcript.
- 2. *P* computes the augmented witness  $w' \leftarrow (w, pad)$  where  $pad \leftarrow \mathbb{F}^{\ell}$ .
- 3. *P* computes com  $\leftarrow$  LigeroCommit(w') and sends com to *V*.
- 4. *P*, *V* begin running Verify<sub>*P*,*V*</sub>(*C*, (*x*, *w*)) with these changes:
  - Whenever Verify<sub>P</sub> instructs the prover to send the  $i_{\text{th}}$  message  $m_i \in \mathbb{F}$  to V, P sends  $t'_i \leftarrow m_i + \text{pad}_i$ .
  - When Verify<sub>V</sub> instructs the verifier to send the *i*<sup>th</sup> random challenge *r<sub>i</sub>*, *V* returns *r<sub>i</sub>*.

At the end, *P*, *V* hold an encrypted sumcheck transcript  $T' = ((t'_i, r_i))_i$  for instance (C, (x, w)).

- 5. Define the circuit C'(x, w', T') to first *decrypt* the transcript  $T = ((t_i = t'_i + pad_i, c_i))_{i \in [0,\ell]}$ , and then run check<sub>V</sub>(C, (x, w), T).
- 6. *P*, *V* run the protocol methods LigeroProve(*C*', (*x*, *w*'), *T*', com) and LigeroVerify(*C*', *x*, *T*', com) respectively; *V* outputs the result.

#### Remarks

• The prover in line 2 only commits to  $|(x, w)| + \ell$  elements. This size is often substantially smaller than |C|, which is what the Ligero prover must commit to if it is applied to *C* directly. This is an important performance advantage because the Ligero commitment involves the NTT over  $\mathbb{F}$ , which is the only non-linear time component in the proof system. This improvement is especially important when  $\mathbb{F}$  does not natively have suitable roots, as is the case for the ECDSA circuit.

• The LigeroProver is applied to the C' circuit which decrypts the transcript (via the one-time pad, i.e., using one field operation per element of T'), and then runs the check<sub>V</sub> circuit. This check circuit performs a small number of field operations and then computes one dot product with (x, w). Moreover, this dot product occurs between the hidden witness and a publicly computable challenge vector derived from the random challenges sent by the verifier. (This public vector also has a *tensorform* that is induced by the Verify protocol.) Altogether, the Ligero proof system is used to verify a small statement with respect to the original circuit *C*.

**Theorem 2.6.** If *C* is a depth-*d* and width-*w* circuit over  $\mathbb{F}$ , and if Ligero is a zeroknowledge protocol with adaptive knowledge soundness error  $\epsilon_{FSK}(|C|, T, \lambda)$  and statistical zero-knowledge  $z'(C, \lambda)$ , then protocol 2.5 is an honest-verifier zero-knowlege protocol with completeness and knowledge soundness error  $O(\eta(|C|, |\mathbb{F}|) + d \cdot \log(w) / |\mathbb{F}|)$ and zero-knowledge  $z'(C, \lambda)$ .

*Proof.* (Sketch) Extraction error follows from the union bound over the failure of the Ligero extractor and the information-theoretic soundness gap of the verify protocol from Theorem 2.3. The zero-knowledge property follows directly from Ligero simulator run on the augmented circuit C'.

### 2.4 Applying the Fiat-Shamir Transform

To construct a non-interactive protocol from our basic protocol, we apply the Fiat-Shamir transform, and argue soundness by applying the Round-by-round Soundness lemma presented by Canetti et al [CCH<sup>+</sup>18].

**Theorem 2.7** ([CCH<sup>+</sup>18]). Suppose that  $\Pi = (P, V)$  is a 2*r*-message public-coin interactive proof for a language L with perfect completeness, poly log(*n*) total bits of prover-to-verifier communication, and round-by-round soundness with a corresponding state function State. Let  $X_n$  denote the set of partial transcripts (including the input and all messages sent) and let  $Y_n$  denote the set of verifier messages when  $\Pi$  is executed on an input of length *n*. If a hash family  $H = \{H_n : X_n \to Y_n\}$  is  $R_{\text{State}}$ correlation intractable and evaluable in time  $\tilde{O}(n)$ , then there exists (Gen,  $\tilde{P}, \tilde{V}$ ) that constitute an adaptively sound publicly verifiable argument for L.

Canetti et al [CCH<sup>+</sup>18] provide an argument for why sumcheck is roundby-round sound with  $O(d/|\mathbb{F}|)$ . Ligero is also a round-by-round sound with parameter  $O(\eta(C')/|\mathbb{F}|)$ . Our composition is also round-by-round sound with similar parameters.

## 3 Optimization

#### 3.1 Reed-Solomon encoding over prime fields

The Ligero polynomial commitment specifically requires Reed-Solomon codes, as opposed to generic error-correcting codes. Ligero interprets the message (an array of field elements) as the evaluations of a low-degree polynomial at certain points, and it depends on the fact that the codeword consists of the evaluations of the same polynomial at certain other points. The choice of the points does not matter for soundness, but it does affect performance. In this section we describe the choices that we made for prime fields  $\mathbb{F}_{q_r}$  and we discuss binary fields in Section 3.2.

A natural set of evaluation points would be  $\omega^i$ , for a root of unity  $\omega$ , so that FFT-based methods can be used for interpolation. However, this choice leads to complicated interactions between the factorization of (q - 1), the length of the codeword, and the efficiency of the interpolation. In particular,  $\mathbb{F}_{P256}$  where P256 is the NIST prime P256 =  $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$  has no usable roots of unity.

To avoid this complexity, we make the following choices. A *message* consists of the evaluations of a polynomial p(x) of degree at most d at points  $x \in \{\mathcal{F}_q(i) : i \leq 0 \leq d\}$ , where  $\mathcal{F}_q(i) = i \mod q$  is the natural injection of integer i into the field  $\mathbb{F}_q$ . Similarly, a *codeword* of length m consists of the m evaluations  $\{p(\mathcal{F}_q(i)) : 0 \leq i < m\}$ . These definitions are well-formed for fields of sufficiently large characteristic, and the only remaining problem is how to compute the polynomial interpolation.

As in [CMT12, Section 2.1], our general strategy is to reduce polynomial interpolation to convolution, and then apply the most convenient convolution algorithm. Specifically, we use the following identity, which holds over the integers: For integer k > d,

$$p(k) = (-1)^d \cdot (k-d) \binom{k}{d} \cdot \sum_{0 \le i \le d} \left(\frac{1}{k-i}\right) \cdot (-1)^i \binom{d}{i} p(i) .$$
<sup>(2)</sup>

See Appendix A for a proof of Equation (2)<sup>3</sup>. Except for scaling by constants, Equation (2) is indeed a (linear) convolution: Multiply the input p(i) elementwise by the constants  $(-1)^i \binom{d}{i}$ , convolve with the kernel  $(k - i)^{-1}$ , and divide elementwise by the constants  $(-1)^d \cdot (k - d) \binom{k}{d}$  yielding p(k). Equation (2) also holds (mod q) when k < q.

Convolution over arbitrary fields can be computed in  $O(n \log n \log \log n)$  field operations by the Nussbaumer [Nus80] algorithm. (See also [Knu97, Exercise 4.6.4-59].) If the field is FFT-friendly, i.e., the order of its multiplicative group factors into small primes, then the convolution can be computed in  $O(n \log n)$  field operations via some form of FFT.

<sup>&</sup>lt;sup>3</sup>Alternatively, Equation (2) gives the explicit form of a special case of the matrix factorization  $V_0^{-1}V_1 = D_0CD_1$ , where  $V_i$  are Vandermonde, *C* is Cauchy, and  $D_i$  are diagonal [Gow92, Theorem 1]

In our case, the ECDSA signature is standardized over a small set of elliptic curves such as P256, P384, P521. Each curve is defined over a specific *base* finite field  $\mathbb{F}_q$  for prime q which is not FFT friendly. However, every odd prime q is of the form  $q = r \cdot 2^{\ell} - 1$ , and the quadratic extension  $\mathbb{F}_{q^2}$  contains roots of unity of order  $2^{\ell}$ , because the order of the multiplicative group is  $q^2 - 1 = (q+1)(q-1)$  and  $2^{\ell}$  divides  $(q+1) = r \cdot 2^{\ell}$ . While this fact is not useful in the trivial case  $\ell = 0$ , in the specific case of the ECDSA primes one can pick  $\ell$  large enough that the FFT over the extension field is viable. Zhang et al. [ZXZS19] make the same observation in the case q is a Mersenne prime  $q = 2^{\ell} - 1$ .

Thus, at least for the ECDSA primes, we compute the convolution in the base field via two FFTs in the quadratic extension, assuming precomputation of the transform of the convolution kernel. Although special-purpose algorithms exist for this case, roughly equivalent to the complex FFT algorithm of real inputs, or perhaps finite-field analogues of the discrete Hartley transform, we naively perform the convolution fully within the extension field, incurring a factor of two overhead over more specialized algorithms.

#### 3.2 Reed-Solomon encoding over binary fields

**Overview** In fields of characteristic 2, we have 1 + 1 = 0 and the choice of interpolation points in Section 3.1 does not apply. Instead, as in [LCH14, DP24], we make a different choice described in the following.

GF(2<sup>*k*</sup>) forms a vector space of dimension *k* over GF(2). Fix once and for all a basis  $\beta_i$  of this vector space, where  $0 \le i < k$ . (See Section 3.3 for a concrete choice of basis.) Writing  $j_i$  for the *i*-th bit of the binary representation of *j*, that is,  $j = \sum_{0 \le i < k} j_i 2^i$  and  $i_j \in \{0, 1\}$ , we inject integer *j* into a field element  $\mathcal{F}(j)$  by interpreting the bits of *j* as coordinates in terms of the basis:

$$\mathcal{F}(j) = \sum_{0 \le i < k} j_i \beta_i \,. \tag{3}$$

Then, as in Section 3.1, we postulate that the message to be encoded consists of the evaluations of a polynomial p(x) of degree at most d at the d + 1 points  $x \in \{\mathcal{F}(i) : 0 \le i \le d\}$ . Similarly, a codeword of length m consists of the m evaluations  $\{p(\mathcal{F}(i)) : 0 \le i < m\}$ .

The particular choice of the injection  $\mathcal{F}(j)$ , as opposed for example to  $\mathcal{F}(j) = \omega^j$  for some appropriate root of unity  $\omega$ , is motivated by the existence of the *additive FFT*, which solves the interpolation problem in  $O(m \log d)$  operations.

We use the additive FFT first presented by Lin, Chung, and Han [LCH14], which builds upon previous algorithms with higher asymptotic complexity. Lin et al. define a *novel polynomial basis* for polynomials as an alternative to the usual monomial basis  $x^i$ , and give an algorithm for evaluating a degree-(d-1) polynomial at all *d* points in a subspace, for  $d = 2^{\ell}$ , and for polynomials expressed in the novel basis. The algorithm can be executed backwards to reconstruct the coefficients in the novel basis given the evaluation points. See [LANHC16, LCK<sup>+</sup>18] for alternative presentations of the same idea.

While the algorithm in [LCH14] is recursive, Diamond and Posen [DP24, Algorithm 2] reformulate the algorithm as a triply-nested loop, resembling common implementations of the Cooley-Tukey FFT.<sup>4</sup> In addition, Diamond and Posen extend the algorithm to obtain  $2^{\ell+\mathcal{R}}$  evaluations of a polynomial of degree less than  $2^{\ell}$ , whereas [LCH14] evaluates  $2^{\ell}$  points at an arbitrary coset of a subspace.

For our purposes, we prefer to avoid restrictions such as *d* and *m* being powers of two, even though the additive FFT algorithms only work for  $2^{\ell}$  sizes. For the evaluation of a polynomial of arbitrary degree *d*, we use the triplynested loop formulation of Diamond and Posen with  $\mathcal{R} = 0$ , augmented with the original coset parameter from [LCH14]. We pad the array of coefficients with zeros to reach size  $2^{\ell}$ , and evaluate sufficiently many cosets to obtain the desired *m* evaluations, possibly discarding unneeded points.

The reconstruction of the polynomial coefficients in the novel basis given d + 1 evaluations is more challenging when d + 1 is not a power of two. We cannot simply pad the evaluations with  $2^{\ell} - (d + 1)$  zeroes and apply the inverse additive FFT, since this process would produce a polynomial of degree  $2^{\ell} - 1$  and not d. To solve this problem, we adapt the inverse *truncated FFT* of van der Hoeven [vdH04] to work with the additive FFT algorithm. The truncated FFT computes a subset of the outputs of the FFT, and van der Hoeven [vdH04] presents an elegant algorithm for inverting the process. While the truncated FFT literature is mostly concerned with saving field operations or avoiding storing implicit zeroes, we prefer to look at the algorithm as a bidirectional FFT, as follows. Consider a linear transformation

$$\begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} = A \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} ,$$

where  $X_i$  and  $Y_i$  are column vectors of  $n_i$  elements. For the concrete case where A is the Fourier matrix, or its additive variant, the forward FFT computes Y given X, and the inverse FFT computes X given Y. We call an algorithm that computes  $Y_1$  and  $X_0$  given  $Y_0$  and  $X_1$  a *bidirectional FFT*. Thus, a bidirectional FFT algorithm computes either the forward or inverse transform, as well as many combinations thereof.

**Details of the additive FFT** Following notation introduced by Diamond and Posen [DP24], let  $U_i$  be the span of the prefix  $\{\beta_k : 0 \le k < i\}$  of the basis. Define the *subspace vanishing polynomial*  $W_i(x) = \prod_{u \in U_i} (x - u)$  and its normalized variant  $\widehat{W}_i(x) = W_i(x)/W_i(\beta_i)$ .

Polynomials  $W_i$  are *linearized*:  $W_i(x + y) = W_i(x) + W_i(y)$ . See [vzGG96, Lemma 2.3], where this fact is stated without proof. See Mateer's thesis [Mat08, Theorem 15] for a proof. Linearization is crucial for the efficiency of the additive FFT, and it also allows for the efficient computation of  $W_i(x)$ : If  $x = \sum_k x_k \beta_k$  in terms of the basis, then  $W_i(x) = \sum_k x_k W_i(\beta_k)$ .

<sup>&</sup>lt;sup>4</sup>The Cooley-Tukey FFT requires a bit-reversal of either the input or the output, but the Diamond and Posen algorithm does not, so the two algorithms are not quite the same.

Moreover, we have  $W_{i+1}(x) = W_i(x)W_i(x + \beta_i)$  because  $U_{i+1} = U_i \cup (\beta_i + U_i)$ , and thus  $W_{i+1}(x) = W_i(x)(W_i(x) + W_i(\beta_i))$  holds by linearity. Together with the base case  $W_0(x) = x$ , these two properties allow for efficient computation of  $W_i(\beta_j)$  and  $\widehat{W}_i(\beta_j)$  for all necessary *i* and *j*. We precompute and store  $\widehat{W}_i(\beta_i)$  in an array  $\widehat{W}[i][j]$ .

The additive FFT [LCH14] evaluates a polynomial  $p(x) = \sum_{0 \le k < 2^i} c_k X_k(x)$  for all x is an coset  $U_i + \alpha$ , given the *coefficients*  $c_k$  of the polynomial's expansion in terms of the *novel polynomial basis*  $X_k(x)$ . Running the algorithm backwards (the inverse FFT) computes the coefficients from the evaluations. Lin et al. [LCH14, Equation 6] give an explicit formula for the basis  $X_k(x)$ , but the precise form of  $X_k(x)$  does not really matter for interpolation as long as  $X_k(x)$  is a polynomial of degree k. That is, one can start from the evaluations over  $U_i$ , compute the coefficients, and evaluate the polynomial over a coset  $U_i + \alpha$  to obtain new evaluations, and nowhere does one have to know the exact form of the basis.<sup>5</sup>

Algorithm 1 shows pseudo-code for our implementation of the additive FFT, largely lifted from [DP24, Algorithm 2], and of its inverse. We show procedure TWIDDLE, which computes the "twiddle factors," as presented in the literature, but in the actual implementation we compute all twiddle factors at the same time via the recurrence TWIDDLE( $i, u + 2^k$ ) =  $\widehat{W}[i][k]$  + TWIDDLE(i, u), which yields an algorithm for doubling the size of an array of twiddle factors in linear time.

**Details of the bidirectional FFT** Ignoring bit-reversal permutations, the computation graph of the additive FFT is the same as the one of the ordinary Cooley-Tukey FFT. (See Figure 1.) Given *n* input coefficients *C*[.] in the novel polynomial basis, a loop of elementary *butterfly* operations decomposes the input even  $C_e[.]$  and off  $C_o[.]$  arrays, to which the transform is applied recursively yielding the evaluation array *E*[.].

In the bidirectional FFT setup, the inputs consist of evaluations E[i] for a prefix  $0 \le i < k$  of E, and coefficients C[i] for a suffix  $k \le i < n$  of C. The goal is to compute the remaining coefficients and evaluations. To this end, we adapt the discussion from [vdH05, Section 6] to the case of additive FFT.

Consider the elementary *forward butterfly* operation described in procedure BUTTERFLY-FWD in Algorithm 1. Let (a, b) = (B[w], B[w+s]) before applying the butterfly operation, and (c, d) = (B[w], B[w+s]) after applying the operation. Then the butterfly computes

$$\begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} 1 & t \\ 1 & t+1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} .$$
(4)

<sup>&</sup>lt;sup>5</sup>The situation is somewhat analogous to Newton finite-difference methods of interpolation, where one starts with the evaluations and computes a bunch of additions and subtractions, obtaining evaluations at additional points. In one sense, some intermediate values can be interpreted as the expansion of the polynomial in terms of the Newton basis, but in another sense, one can view the method as an algorithm for interpolation without any mention of the Newton basis.

Algorithm 1 Additive FFT [DP24, Algorithm 2]

**procedure** FFT( $\ell$ ,  $\alpha$ , B[])  $\triangleright$  **Require**: Array B of size  $2^{\ell}$  containing the coefficients of the expansion  $p(x) = \sum_{0 \le k \le 2^{\ell}} B[k] X_k(x)$  in terms of the novel polynomial basis  $X_k(x)$ .  $\triangleright$  *Ensure*:  $B[k] \leftarrow p(\mathcal{F}(k+\alpha))$ , where  $\mathcal{F}(.)$  is defined in Equation (3).  $\triangleleft$ for all  $i : 0 \le i < \ell$  in *descending* order of i **do**  $s \leftarrow 2^i$ for all  $u: 0 \leq 2s \cdot u < 2^{\ell}$  do  $t \leftarrow \text{TWIDDLE}(i, 2s \cdot u + \alpha)$ for all  $v : 0 \le v < s$  do BUTTERFLY-FWD(B,  $2s \cdot u + v$ , s, t) **procedure** IFFT( $\ell$ ,  $\alpha$ , B[]) ▷ *Inverse of procedure* FFT.  $\triangleleft$ **for all**  $i : 0 \le i < \ell$  in *ascending* order of *i* **do**  $s \leftarrow 2^i$ for all  $u: 0 \leq 2s \cdot u < 2^{\ell}$  do  $t \leftarrow \text{TWIDDLE}(i, 2s \cdot u + \alpha)$ for all  $v : 0 \le v < s$  do BUTTERFLY-BWD $(B, 2s \cdot u + v, s, t)$ **procedure** TWIDDLE(*i*, *u*) Let  $u_k$  denote the *k*-th bit of *u*, i.e.,  $u = \sum_k u_k 2^k$ . **return**  $\sum_k u_k \cdot \widehat{W}[i][k]$ **procedure** BUTTERFLY-FWD(*B*[], *w*, *s*, *t*))  $B[w] \leftarrow B[w] + t \cdot B[w+s]$  $B[w+s] \leftarrow B[w+s] + B[w]$ **procedure** BUTTERFLY-BWD(*B*[], *w*, *s*, *t*))  $B[w+s] \leftarrow B[w+s] - B[w]$  $B[w] \leftarrow B[w] - t \cdot B[w+s]$ **procedure** BUTTERFLY-DIAG(*B*[], *w*, *s*, *t*))  $(B[w], B[w+s]) \leftarrow (B[w] - t \cdot B[w+s], B[w] + B[w+s])$ 



Figure 1: Flow graph of the ordinary additive FFT, which is the same as the one for the Cooley-Tukey FFT if one ignores bit-reversal permutations. The "butterflies" with two inputs and two outputs denote that the output is some  $2 \times 2$  linear transformation of the input. Lin and others [LCH14] draw a slightly different diagram factoring the linear transformation into triangular matrices, but for our purposes this level of detail is not important.

Similarly the *backward butterfly* operation in procedure BUTTERFLY-BWD computes the inverse transformation

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} t+1 & -t \\ -1 & 1 \end{bmatrix} \begin{bmatrix} c \\ d \end{bmatrix} .$$
 (5)

The key insight [vdH05, Equation (7)] of the bidirectional FFT algorithm is the *diagonal butterfly* (procedure BUTTERFLY-DIAG), which proceeds forward on *b* and backwards on *c*:

$$\begin{bmatrix} a \\ d \end{bmatrix} = \begin{bmatrix} -t & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} b \\ c \end{bmatrix} .$$
 (6)

The symmetric case of computing (b,c) from (a,d) is also possible, and viable for the ordinary Cooley-Tukey FFT [vdH05, Equation (8)]. However, the corresponding equation for the additive FFT

$$\begin{bmatrix} b \\ c \end{bmatrix} = (1+t)^{-1} \cdot \begin{bmatrix} -1 & 1 \\ 1 & t \end{bmatrix} \begin{bmatrix} a \\ d \end{bmatrix}$$
(7)

requires (1 + t) to be invertible, which is not guaranteed in our case. This is the reason why we chose the convention that a prefix of the evaluations are

#### Algorithm 2 Bidirectional FFT, adapted from [vdH05, Section 6]

| <b>procedure</b> BIDIRECTIONAL-FFT( $i, \alpha, k, B[]$ )  | _               |
|--|-----------------|
| $\triangleright$ <b>Require:</b> Array B of length $n = 2^i$ , whose first k elements are evaluations of |                 |
| a polynomial and the rest are coefficients of the same polynomial in the novel                           |                 |
| polynomial basis.  | $\triangleleft$ |
| <i>Ensure:</i> Overwrite B with an array whose first k elements are coefficients                         |                 |
| and the rest are evaluations.  | $\triangleleft$ |
| if $i > 0$ then  |                 |
| $i \leftarrow i - 1$   |                 |
| $s \leftarrow 2^i$   |                 |
| $t \leftarrow \texttt{TWIDDLE}(i, \alpha)$   |                 |
| if $k < s$ then  |                 |
| for all $v: k \le v < s$ do  |                 |
| BUTTERFLY-FWD $(B, v, s, t)$   |                 |
| BIDIRECTIONAL-FFT $(i, \alpha, k, B)$  |                 |
| for all $v: 0 \le v < k$ do  |                 |
| BUTTERFLY-DIAG $(B, v, s, t)$  |                 |
| $FFT(i, \alpha + s, B[s:])$  |                 |
| else   |                 |
| $\mathrm{IFFT}(i, \alpha, B)$  |                 |
| for all $v: k - s \le v < s$ do  |                 |
| BUTTERFLY-DIAG $(B, v, s, t)$  |                 |
| BIDIRECTIONAL-FFT $(i, \alpha + s, k - s, B[s :])$   |                 |
| for all $v: 0 \le v < k - s$ do  |                 |
|  |                 |

known, as opposed to a prefix of the coefficients. With this choice, we only need (6) but not (7).

Consider again Figure 1, and assume that  $k \ge n/2$ . Thus, the entire output of the top N/2-FFT block is known, the entire  $C_e[.]$  can be obtained from the inverse FFT of the known prefix of E[.]. Given  $C_e[.]$  and the known suffix C[i]for  $i \ge n - k$ , diagonal butterflies compute  $C_o[i]$  for  $i \ge k - n/2$ . The result is a bidirectional FFT problem of size n/2: a suffix of length n/2 - (k - n/2)of  $C_o$  and a prefix of length k - n/2 of the bottom half of E. This bidirectional subproblem is solved recursively, thus computing the missing elements of E and  $C_o$ . Given the entire  $C_e$  and  $C_o$ , the missing C's are computed via backward butterflies.

Symmetric considerations apply to the case k < n/2. See Algorithm 2 for complete pseudo-code of the bidirectional FFT algorithm.

Even though the FFT algorithm reduces a problem of size *n* to two subproblems of size n/2, the bidirectional FFT algorithm produces one bidirectional problem of size n/2 and another problem of size n/2 which is either a fully forward or a fully backward FFT. Thus, assuming an optimized implementation of the latter transform, the overhead of the recursion is minimal:  $O(\log n)$ 

recursive calls and  $O(\log n)$  computations of twiddle factors. In practice, the performance of the bidirectional FFT is essentially the same as that of the fully forward or fully backward FFT.<sup>6</sup>

**Details of Reed-Solomon encoding** Our original interpolation problem takes d + 1 evaluations of a degree-d polynomial  $p(\mathcal{F}(i))$  for  $0 \le i \le d$  as input, and must compute  $p(\mathcal{F}(i))$  for  $0 \le i < m$ , where d and m are arbitrary (i.e., not necessarily powers of 2). Let  $\ell$  be minimal such that  $d < 2^{\ell}$ . Run the bidirectional FFT algorithm with  $n = 2^{\ell}$  and k = d + 1, assuming that the first k coefficients are unknown and the remaining are zero. The bidirectional FFT produces the first missing (n - k) evaluations, as well as the full array of k coefficients. Then the forward FFT can be run on as many cosets as needed to produce m evaluations.

#### 3.3 Reduction of the Ligero proof size

Soundness requires using large fields such as  $GF(2^{128})$ . We implement  $GF(2^{128})$  as GF(2)[x]/(Q(x)) where  $Q(x) = x^{128} + x^7 + x^2 + x + 1$ . With this choice of Q(x), x is a generator of the multiplicative group of the field.

Almost all messages in our system consist of witness *bits* in  $\{0, 1\}$  that are packed into arrays of size less than 2<sup>14</sup>. Recalling Reed-Solomon encoding from Section 3.2, if the message is an array of elements in some subfield of GF(2<sup>128</sup>), and if the evaluation points are also elements in the same subfield, then the entire codeword consists of subfield elements and it can be represented compactly. In order to ensure that all evaluation points are in the subfield, it suffices to choose  $\beta_i$  as a basis of the subfield rather than the entire field.

Consequently, we choose  $GF(2^{16})$  as the subfield of  $GF(2^{128})$ , with  $g = x^{(2^{128}-1)/(2^{16}-1)}$  as the generator of  $GF(2^{16})$ , and  $\beta_i = g^i$  for  $0 \le i < 16$  as the basis of the subfield. This allows us to encode elements in our commitment scheme with 16-bits instead of 128. Whenever we have a choice, we inject witness bits and small integers into the subfield via Equation (3).

Because we do not represent subfields explicitly in the field representation, we have the problem of determining whether a general field element  $u \in$ GF(2)[*x*]/(*Q*(*x*)) belongs to the subfield, and if so, to determine *u*'s representation  $u = \sum_{0 \le i \le 16} u_i \beta_i$  in terms of the basis  $\beta_i$ .

We solve this problem via standard Gaussian elimination. We view elements of  $GF(2^{128})$  as row vectors of 128 elements of GF(2) with respect to the monomial basis  $x^j$ , a convention that allows two row vectors to be added via a single 128-bit xor instruction. We then form a matrix *B* whose *i*-th row consists of the row vector  $\beta_i$ . With these conventions, we need to solve the rectangular system xB = u for *x*, which we do via preliminary LU factorization that reduces *B* to row-echelon form, followed by standard solution of triangular systems. The solution costs O(n) row operations for n = 16, which is efficient

<sup>&</sup>lt;sup>6</sup>With some care, the recursion can be removed entirely, but we found the recursion-free algorithm needlessly confusing.

since row operations take a single wide xor instruction. In our system, the time spent on projections onto the subfield is negligible.

In our current implementation, the subfield optimization only reduces proof size, specifically during I/O time to serialize and deserialize field elements. We perform all arithmetic operations in the full field. Diamond and Posen [DP24] argue for a tower representation of the field which makes subfields explicit, and which can exploit subfield arithmetic when possible. We leave a comparison of the two strategies for future work.

#### **3.4** Input consistency for circuits over different Fields

The round-by-round soundness property requires that the sumcheck and verify protocols use a field of size at least  $2^{128}$ . Ligero commitments require a subfield of size at least  $O(\sqrt{|(x,w)|})$ , where (x,w) is the size of the input to the circuit *C*. Thus, it is usually most efficient to run sumcheck over a field extension of a sufficiently large base field. However, the ECDSA circuit imposes a different constraint: namely that efficiently verifying elliptic curve operations over curve *P*256 requires the circuit to be defined over a specific 256-bit field.

Consider the theorem statement (e, pk) which verifies that there exists a message *m* and signature values (r, s) such that e = SHA256(m), and (r, s) is a signature on *e* under public key *pk*. A single circuit that verifies this statement over  $\mathbb{F}_{p256}$  incurs the overhead of verifying the sha256 hash of *m* over a larger prime field than is necessary for soundness. Similarly, a circuit that verifies the full statement over  $\mathbb{F}_{2^{128}}$  incurs the overhead of simulating the arithmetic for the elliptic curve, which incurs  $\approx 100x$  overhead compared to the native field.

A natural approach for higher efficiency is to perform the ECDSA verification of instance (e, pk) in the optimal  $\mathbb{F}_{p256}$  field, and perform the SHA verification of the instance (e, m) over the optimal binary field. The concern is that a cheating prover may use 2 inconsistent values of e in the 2 different circuits in two different fields. In this section, we describe a technique to ensure consistency of inputs when the verification of the theorem occurs using *two* or more proof fields, or more generally, two or more proof systems.

Our solution to ensure input consistency is to augment each circuit in the two fields with a message authentication code (MAC) verification for the common input *e*. We use the information-theoretic  $MAC_{a,b}(x) = ax + b$  where the arithmetic is performed over  $\mathbb{F}_{2^{128}}$ . Note that in the ECDSA circuit, this requires simulating the arithmetic of  $\mathbb{F}_{2^{128}}$  over the prime field  $\mathbb{F}_{p^{256}}$ . For small inputs, this computation incurs little overhead. At a high level, each circuit is augmented to take as a public input  $m_e = MAC_k(e)$ , and as a private input, the key k = (a, b), and each circuit internally computes the MAC on *e* and compares it with the public input.

A secondary issue relates to how the key k is chosen. Allowing the verifier to select k may break the zero-knowledge property; allowing the prover to select k breaks the soundness, since the prover can intentionally select the a key that allows two different values of e to have the same MAC value. In our

scheme, this key is jointly sampled by the Prover and Verifier. Specifically, the Prover first commits to its share  $k_p = (a_p, b_p)$  as well as the secret input *e*. The verifier next randomly samples  $k_v$  and sends it to the Prover in the clear. The Prover then computes the key  $k = k_p + k_v$ , and then uses *k* to compute the mac on the shared input *e*, and then sends the MAC value  $m_e$  to the verifier. The verifier uses  $m_e$  as a public input to both circuit  $C_h$  and circuit  $C_{sig}$ . Both of these circuits include a check to verify that  $m_e = MAC(e)$ .

#### **Protocol 3.1. Two-fields ZK protocol** $(P_2, V_2)$

Let circuit  $C_1(x_1, w_1, w)$  be defined over field  $\mathbb{F}_1$  and  $C_2(x_2, w_2, w)$  defined over  $\mathbb{F}_2$ , and let  $w \in \{0, 1\}^c$  be the common input to both that is embedded into the respective  $\mathbb{F}_i$  in the natural way, and  $x_1, w_1 \in \mathbb{F}_1^*$  and  $x_2, w_2 \in \mathbb{F}_2^*$ . Both parties hold  $(x_1, x_2)$ , and the prover additionally holds  $(w_1, w_2, w)$ .

- Define circuits  $\hat{C}_i((x_i, m, k_v), (w_i, w, k_p))$  where  $k_p, k_v \in \{0, 1\}^{256}$  as follows:
  - Assert  $C_i(x_i, w_i, w) = 0$
  - Assert  $MAC_k(w) = m$  where  $k = k_p \oplus k_v$  with all computation performed in GF(2<sup>128</sup>).
- *P*, *V* run Protocol 2.5 on circuit  $\hat{C}_1$ ,  $\hat{C}_2$  in parallel, except with the following change to step 3:
  - 3. *P* samples  $k_p \leftarrow GF(2^{128})^2$  and then computes

 $\operatorname{com}_i \leftarrow \operatorname{LigeroCommit}((w_i, w, k_p, \operatorname{pad}_i), \mathbb{F}_i)$ 

and sends to V.

- (a) *V* samples  $k_v \leftarrow GF(2^{128})^2$  and sends to *P*.
- (b) *P* computes  $m \leftarrow MAC_{k_n \oplus k_n}(w)$  and sends to *V*.

**Theorem 3.2.** If  $C_1$  and  $C_2$  are depth-d and width-w circuits over fields  $\mathbb{F}_1$  and  $\mathbb{F}_2$ , and if MAC<sub>k</sub> is an information-theoretic message authentication code with statistical security  $\lambda_s$ , if Protocol 2.5 is a zero-knowledge protocol with adaptive knowledge soundness error  $\eta(|C_1|, |C_2|)$ , then protocol 3.1 is a zero-knowlege protocol with completeness and knowledge soundness error  $O(\eta(|C_1|, |C_2|) + \lambda_s)$ .

*Proof.* (Sketch) The modified protocol is essentially a parallel composition of Protocol 2.5. To construct a knowledge-soundness extractor algorithm for this composed protocol, run the extractor for  $C_1$  and for  $C_2$  sequentially. If either fails, then the composed protocol extractor fails. Suppose the first extractor outputs  $(x_1, w_1, w, \text{pad}_1)$  and the second outputs  $(x_2, w_2, w', \text{pad}_2)$ . If  $w \neq w_1$ , then the composed extractor fails. Otherwise, output  $(x_1, x_2, w_1, w_2)$  and halt.

By a standard union bound argument over the failure probability of the individual extractors, as well as the statistical soundness parameter of the MAC, the combined extractor succeeds with all but negigible probability. Furthermore, the running time of the composed extractor is essentially the sum of the running times for the individual extractors, and thus polynomial-time in the running time of a malicious prover oracle.

To argue zero-knowledge, the simulator randomly samples *m* and runs the piece-wise simulators for  $C_1$ ,  $C_2$ .

## 4 Circuit design

### 4.1 ECDSA Signature Verification

This section describes the construction of an arithmetic circuit that verifies the existence of an ECDSA signature under public key  $(pk_x, pk_y)$  for the hashed message e = H(m). Following the Sec-1 [Cer09] specification, which is identical to the ANSI X9.62.2005 standard, an ECDSA signature verification proceeds as follows:

Algorithm 3 ECDSA-Verify(Q, H(m), r, s) [Cer09]

- 1: Derive integer *e* from H(m).
- 2: Verify that  $r, s \in [1, q 1]$ .
- 3: Compute  $u_1 = es^{-1} \mod q$  and  $u_2 = rs^{-1} \mod q$ .
- 4: Compute  $R = (r_x, r_y) = G \cdot u_1 + Q \cdot u_2$  and verify  $R \neq id$ .
- 5: Verify that  $r = (r_x \mod q)$ .

Several of these steps occur over the integers or in the field  $\mathbb{F}_q$ , whereas our checks must be performed in the field  $\mathbb{F}_p$ . Towards this goal, the verification circuit defined in Alg 4 succeeds on an input *x* only if an easily computable tuple t(x) induces Alg. 3 to succeed. Furthermore, the circuit is almost perfectly complete, with only a handful of cases that do not pass our checks but do pass the official spec—this is acceptable in all applications that we consider, even when malicious signers are considered.

| <b>Algorithm 4</b> Verification Circuit $V(Q, H(m), r, s, r_y)$            |  |  |  |  |  |
|--|--|--|--|--|--|
| 1: Derive integer <i>e</i> from $H(m)$ .                                   |  |  |  |  |  |
| 2: Verify that $r, s \in [1, q - 1]$                                       |  |  |  |  |  |
| 3: Verify that $Q, R = (r, r_y) \in E$ and $R \neq id$ .                   |  |  |  |  |  |
| 4: Verify $id = G \cdot e + Q \cdot r - R \cdot s$                         |  |  |  |  |  |
| The main difference in the two precedures is that the later avoids compute |  |  |  |  |  |

The main difference in the two procedures is that the later avoids computations of inverses modulo q by performing a multi-scalar exponentiation in the group and performing additional checks to ensure that points are on the elliptic curve group. **Theorem 4.1.** If  $V(Q, H(m), r, s, r_y)$  succeeds where  $r, s \in [0, 2^{\lceil \log p \rceil} - 1]$  and  $r_y \in \mathbb{F}_p$ , then ECDSA(Q, H(m), r, s) succeeds.

**Verifying a multi-scalar multiplication** The most expensive step to verify in Alg. 4 is step 4, which verifies that  $G \cdot e + Q \cdot r - R \cdot s$  is equal to the identity element. By providing the intermediate values as witnesses, we produce a low-depth circuit that computes the left-hand side using a simple extension of the folklore *Shamir* trick. Specifically, the computation consists of a standard double-and-add loop over the bits of the exponents. In the *i*<sup>th</sup> iteration, the bits  $e_i, r_i, s_i$  are used to index a table that contains eight combinations of the G, Q, and R elements. The current accumulator is doubled and the result of the table lookup is added. As described, the input of the  $i + 1^{st}$  iteration would depend on the output of the *i*<sup>th</sup> iteration, resulting in a deep circuit. Instead, we provide as witnesses the outputs of each iteration, and compare the output of the basic double-and-add block to the input for the next block. To reduce the depth, the 7 non-zero table values are given as witnesses and verified by the circuit. A full accounting for the resulting circuit size for ECDSA verification is given in Table 1.

|                      | DEPTH   | QUADS           | TERMS            | INPUTS |
|----------------------|---------|-----------------|------------------|--------|
| Multi-exponentiation | 7<br>12 | 19,534<br>5,475 | 37,550<br>10 569 | 1,038  |
| Range check + lest   | 12      | 5,475           | 10,509           | 1,050  |
| Total                | 12      | 23,453          | 47,598           | 1,038  |

Table 1: Circuit size and depth for ECDSA verification. The circuit compiler exploits shared terms and thus the total is slightly less than the sum.

#### 4.2 SHA-256

The SHA-256 hash function consists of several iterations of a compression function which is comprised of the *Ch*, *Maj*,  $\Sigma_0$ , and  $\Sigma_1$  functions and finally addition modulo 2<sup>32</sup>. A large *n*-bit message *m* is first broken into 64-byte blocks; the basic block function is applied to the *i*<sup>th</sup> block to produce a compressed 32-byte output, which is then combined with the *i* + 1<sup>st</sup> block. The final message block must be at most 55 bytes, as it is padded with zeros and then augmented by an 8-byte encoding of *n*.

To verify a message of at most 64 \* k - 9 bytes, our circuit must evaluate k SHA-256 basic block functions. A basic block takes as input eight 32-bit state values, designated by letters A-H, 64 bytes of the message, and produces a

new state value. The block uses the Ch, Maj,  $\Sigma_0$ , and  $\Sigma_1$  functions:

$$Ch(E, F, G) = (E \land F) \oplus (E \land \neg G)$$
  

$$Maj(A, B, C) = (A \land B) \oplus (A \land C) \oplus (B \land C)$$
  

$$\Sigma_0(A) = (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22)$$
  

$$\Sigma_1(E) = (E \ggg 6) \oplus (E \ggg 11) \oplus (E \ggg 25)$$

The state values *B*, *C*, *D*, *F*, *G*, *H* are assigned the values of *A*, *B*, *C*, *E*, *F* respectively. The values *A*, *E* are updated as

$$A = W_i \boxplus K_i \boxplus H \boxplus Ch(E, F, G) \boxplus \Sigma_1(E) \boxplus \operatorname{Maj}(A, B, C) \boxplus \Sigma_0(A)$$
  
$$E = W_i \boxplus K_i \boxplus H \boxplus Ch(E, F, G) \boxplus \Sigma_1(E) \boxplus \operatorname{Maj}(A, B, C) \boxplus D$$

where  $\boxplus$  represents addition modulo  $2^{32}$ .

**Verifying Modular Addition** Verification of addition modulo  $2^{32}$  can be performed by some form of Boolean adder circuit, either ripple-carry or parallelprefix. In fact, verification of the addition  $A \boxplus B = C$  can be performed by a constant-depth circuit if all of A, B. and C are provided as inputs, by using the fact that the *i*-th carry bit can be inferred from the equation  $C[i] = A[i] \oplus B[i] \oplus \text{carry}[i]$ , and thus one only needs to verify that the carry has been propagated correctly, as opposed to actually propagating the carry.

At least in certain fields, however, performing the addition using the native field arithmetic yields a smaller circuit. We use two variants of this method in our circuits. Both variants exploit the fact that we compute sums modulo  $2^{32}$  of a small number *t* of terms T[i], where concretely  $t \leq 7$ .

If the characteristic of the field is greater than  $2^{32} \cdot t$ , then the addition of t terms in the field does not overflow. Let  $s = \sum_{0 \le i < t} T[i]$  in the field arithmetic, and let r be the claimed value of the sum modulo  $2^{32}$ , interpreted as a field element. Let z = s - r in the field. Then  $z = 0 \pmod{2^{32}}$  iff  $\prod_{0 \le i < t} (z - 2^{32}i) = 0$ , where the latter check occurs by means of a tree of multiplications in the field.

The previous verification method does not work on fields of small characteristic, such as  $GF(2^{128})$ . In this case, we apply the previous method to the multiplicative group of the field instead of the additive group. Let *g* be a generator of the multiplicative group, and assume that  $order(g) > 2^{32}t$ . Then  $g^s = \prod_{0 \le i < t} g^{T[i]}$  assumes one of *t* possible values  $g^{r+2^{32}i}$  for  $0 \le i < t$ , which can be compared against via a tree of multiplications in the field, as before.<sup>7</sup>

Table 2 reports the sizes of instantiations of our SHA256 circuits in different fields and using different *packing* parameters. To reduce the number of inputs, and thus the size of the commitment, we can pack several witnesses that are bits into a single witness, and then apply a simple circuit to extract the individual bits from the witness.

<sup>&</sup>lt;sup>7</sup>In the special case where the field has a (multiplicative) root of unity of order  $2^{32}$ , we can choose *g* to be such root, and obtain modulo- $2^{32}$  arithmetic "for free". This special case does not apply to the fields used by our circuits.

|                        | PACKING | DEPTH | QUADS  | TERMS   | INPUTS |
|------------------------|---------|-------|--------|---------|--------|
| $\mathbb{F}_{P256}$    | -       | 7     | 37,974 | 167,348 | 6,657  |
|                        | 2       | 9     | 65,690 | 215,504 | 3,585  |
|                        | 3       | 10    | 76,287 | 239,919 | 2,625  |
|                        | 4       | 11    | 85,732 | 273,556 | 2,049  |
| $\mathbb{F}_{2^{128}}$ | -       | 13    | 53,435 | 87,642  | 6,657  |
|                        | 2       | 14    | 65,727 | 150,991 | 3,585  |
|                        | 3       | 15    | 73,818 | 166,494 | 2,625  |
|                        | 4       | 16    | 79,607 | 177,959 | 2,049  |

Table 2: Circuit size and depth for 1 SHA-256 block over 2 different fields. The packing column indicates how many bits are packed into a witness (and then extracted in the circuit).

### 4.3 CBOR parsing

MDOC credentials are encoded in the Concise Binary Object Representation (CBOR) [BH20] format. For the purposes of this paper<sup>8</sup>, a CBOR datum is either an integer (signed or unsigned), a boolean, an array of bytes, an array of CBOR datums, or a map from datum to datum.

CBOR is intended to minimize both the size of the encoded document and the size of the parser. To this end, the first byte of the encoding of a datum consists of a three-bit *tag*, denoting the various types (integer, array, map, etc.), and a five-bit *count*. Small integers are encoded inline in the count. For larger integers, the count encodes the size of the integer, which is then stored in subsequent bytes of the document. If the tag indicates that the datum is an array or a map, the count similarly encodes the number of elements in the array or the number of pairs in the map, with the actual contents stored in subsequent bytes in the document.

Let in[i] be the *i*-th byte of input, where in[i] is itself an array of eight input wires each carrying a bit. In the context of our proof system, "parsing CBOR" does not mean converting in[] into an abstract syntax tree. Instead, the parsing circuit is expected to assert the truth of statements such as the following:

The top-level document is a map. The top-level map contains a pair whose key is the string valueDigests and whose value is another map. The valueDigests map contains a pair whose key is the string org.iso.18013.5.1 and whose value is another map. The org.iso.18013.5.1 map contains a pair whose key is the unsigned number 4 and whose value is a byte array of length 32, which a SHA256 hash of ...

To design circuits to make such assertions, our strategy is to provide as witnesses to the circuit the positions in the input of the relevant datums. While

<sup>&</sup>lt;sup>8</sup>CBOR supports a few more cases such as floating-point numbers, which we ignore.

leaf comparisons such as "the key at position i is the string valueDigests" are relatively straightforward, structural assertions "the pair at position i is an entry of the map that starts at position j" require the circuit to reconstruct the tree structure of the document. In the remainder of this section, we focus on techniques for computing such reconstruction.

In the "combinational" arithmetic circuit model, the whole input is given to a fixed-size circuit, and thus the size of documents are limited to N bytes for some constant N known at circuit-generation time. Similarly, because the model does not contain a stack to support arbitrarily nested arrays and maps, we restrict the parser to L nesting levels, for some constant L.

**Toy CBOR** To illustrate the main problems and solutions in CBOR parsing, we focus our discussion on parsing a context-free CBOR-like toy language defined below.

A *toy-CBOR* document consists of a sequence of *bytes* in[i] that are grouped into *tokens*. The token starting at position *i* is defined as follows.

- **Case**  $0x00 \le in[i] < 0x10$ . The token is Int(in[i]), that is, a four-bit integer. The next token starts at position i + 1.
- **Case** in[i] = 0x10. The token is Int(in[i+1]), that is, the eight-bit integer stored at in[i+1]. The next token starts at position i + 2.
- **Case**  $0x20 \le in[i] < 0x30$ . The token is Array(in[i]). The next token starts at position i + 1.
- **Case** in[i] = 0x30. The token is Array(in[i+1]). The next token starts at position i + 2.

#### Otherwise: Error.

In addition to the lexical structure of tokens, toy CBOR has a context-free syntactic structure which we informally define by giving a recursive-descent parser for it. Specifically, the parser reads the first token from the stream, consuming it. If the token is Int(k), the parser returns the integer k. Otherwise the token is Array(k), in which case the parser calls itself recursively k times, returning an array of length k containing the results of the recursive calls.

Full CBOR has both a richer token structure, including variable-length integers and strings, and a richer syntactic structure, including both arrays and maps, but the main difficulties already arise in our Toy version.

**Lexer** The first step is to identify token boundaries. Given input *in* of length N, Algorithm 5 produces an array *header* such that *header*[*i*] is true if and only if a token starts a position *i*. Thus, assertions of the form "an array starts at position *i*" must assert *header*[*i*] first.

We omit procedure DECODE in Algorithm 5 because it is complex but not insightful. One should think of it as combinational logic which, for each input

#### Algorithm 5 Sequential algorithm for tokenization.

1: **procedure** TOKENIZE(*N*, *in*[])  $\triangleright$  *Procedure* DECODE (not shown) produces an array length where length[i] 2: is the length in bytes of the token that starts at position in[i]. 3:  $(token, length) \leftarrow DECODE(N, in)$  $\triangleright$  Now set header  $[i] \leftarrow$  **true** if and only if a token actually starts a position *i*. 4: < 5: *header*[0]  $\leftarrow$  **true**  $slen[0] \leftarrow length[0]$ 6: 7: for i = 1 to N - 1 do  $header[i] \leftarrow ((slen[i-1]-1) = 0)$ 8: 9: if *header*[*i*] then  $slen[i] \leftarrow length[i]$ 10: else 11:  $slen[i] \leftarrow (slen[i-1]-1)$ 12: 13: **return** (token, header)

position *i*, decodes the bits of in[i] into a set of signals that indicate whether the token is an integer or an array and computes a field value length[i] that represents the length of the token so that the next token starts at position i + length[i]. For toy CBOR,  $length[i] \in \{1, 2\}$  can be computed solely from in[i], whereas the real CBOR requires peeking at in[i + 1] and possibly subsequent input bytes. Being actually a circuit and not a procedure, DECODE speculatively computes length[i] for all positions *i*, irrespective of whether or not a token actually starts at position *i*. In the actual implementation, DECODE must also cope with invalid tokens and output a validity status, and we must assert that header[i] implies that the token at position *i* is valid.

**Reducing the depth of lexing** Given that an upper bound to *N* is known at circuit-generation time, one could unroll the loop in Algorithm 5 and generate a circuit. However, such a circuit would be of depth O(N), which is inefficient for a sumcheck-based prover. We now use well-known parallel-prefix techniques to reduce the depth of the circuit.

If *header*[*i*] were always false in line 9 of Algorithm 5 (which is not the case), then the update equation for *slen* would be slen[i] = slen[i-1] + A[i] for A[i] = -1, which is an instance of a *prefix sum* or *scan*. Because  $slen[i] = \sum_{j \le i} A[j]$ , each slen[i] can be computed in depth  $O(\log n)$  simply by building an independent tree of adders for each *i*. Various algorithms, collectively called *parallel-prefix*, allow computation in depth  $O(\log n)$  of slen[i] for all *i* while maximizing reuse of common subexpressions. See Blelloch [Ble90] and references therein for details.

The full update equation

$$slen[i] = if header[i] then length[i] else slen[i-1] + A[i]$$
, (8)

again for A[i] = -1, has the form of a *segmented scan*. Logarithmic-depth circuits exist that can compute *slen*[.] given *header*[.], *length*[.], and A[.] = -1 as inputs. See [Ble90, Section 1.5] for a general technique to transform segmented scans into unsegmented scans.

The segmented scan technique would solve our circuit-depth problem, except that segmented scans require the segment marker *header*[.] to be provided as input to the circuit, whereas in our case, *header*[*i*] is computed as a function of slen[i-1] in line 8. To work around this problem, we provide booleans *header*[*i*] as witnesses, one per byte input position, and use Algorithm 6 as our starting point instead of Algorithm 5.

#### Algorithm 6 Algorithm for tokenization with witnesses

1: **procedure** TOKENIZEW(*N*, *in*[], *header*[])  $(token, length) \leftarrow DECODE(N, in)$ 2: 3:  $slen[0] \leftarrow length[0]$ for i = 1 to N - 1 do 4: if *header*[*i*] then 5:  $slen[i] \leftarrow length[i]$ 6: else 7:  $slen[i] \leftarrow (slen[i-1]-1)$ 8: **assert**(*header*[0]) 9: for i = 1 to N - 1 do 10: **assert**(*header*[i]  $\iff$  (*slen*[i-1]-1) = 0) 11: **return** (*token*, *header*) 12:

It is not immediately obvious that Algorithm 6 and Algorithm 5 are equivalent, because Algorithm 6 computes slen[.] from an untrusted witness header[.] and then it asserts that header[.] is correctly computed from slen[.], whereas Algorithm 5 interleaves the computation of slen[.] and header[.]. The correctness of Algorithm 6 can be proven by induction on the loop variable *i*, in essence mimicking the control flow of Algorithm 5. If the assertion in line 9 holds, then slen[0] and header[0] are both correct. If the state variables are correct for (i - 1) and the assertion in line 11 holds, then header[i] is correct, and thus slen[i] is correct as well because the update of slen[i] is the same for both algorithms.

Of the many concrete parallel-prefix algorithms, we use the Sklansky [Skl60] form (see Algorithm 7), which has  $O(n \log n)$  gates and depth lg *n*. The Brent-Kung form [BK82], among others, requires only O(n) gates, but it has twice the depth as the Sklansky algorithm. In the layered circuits required by sumcheck, values need to be propagated across each layer, and thus a lower depth is preferable even at the cost of extra gates.

**Reconstruction of tree structure** A toy-CBOR document ultimately represents a tree. One can view each token in the document as a tree node: A token Int(k) represents a leaf, and a token Array(k) represents an inner node with k children. On this tree, we define the *tree address* of a node as follows. The root

#### Algorithm 7 Sklansky [Skl60] parallel prefix algorithm

**procedure** SCAN-ADD(*i*<sub>0</sub>, *i*<sub>1</sub>, *B*[]) ▷ *In-place*, set *B*<sub>out</sub>[*i*<sub>0</sub>] = *B*<sub>in</sub>[*i*<sub>0</sub>] and *B*<sub>out</sub>[*i*] = *B*<sub>in</sub>[*i*] + *B*<sub>out</sub>[*i* − 1] for *i*<sub>0</sub> < *i* < *i*<sub>1</sub>. ⊲ **if** *i*<sub>1</sub> − *i*<sub>0</sub> > 1 **then**   $i_m \leftarrow i_0 + \lfloor (i_1 - i_0)/2 \rfloor$ SCAN-ADD(*i*<sub>0</sub>, *i*<sub>m</sub>, *B*) SCAN-ADD(*i*<sub>m</sub>, *i*<sub>1</sub>, *B*) **for all** *i* : *i*<sub>m</sub> ≤ *i* < *i*<sub>1</sub> **do**  $\lfloor B[i] \leftarrow B[i] + B[i_m - 1]$ 

#### Algorithm 8 Sequential algorithm for updating tree addresses.

 

 procedure UPDATE-COUNTERS(i, counters[][], token[], header[])

 ▷ Given the tree address counters[i] and the output (token, header) of the tokenizer, compute counters[i + 1].

 counters[i + 1] ← counters[i]

 if header[i] then

 Let l[i] ∈ {l ∈ Nat : counters[i][l] ≠ 0 ∧ ∀m > l : counters[i][m] = 0}

 counters[i + 1][l[i]] ← counters[i][l[i]] - 1

 if token[i] = Array(k) then

 | counters[i + 1][l[i] + 1] ← k

has depth l = 0, and its tree address is the array [1]. The tree address of a node at depth (l + 1) is  $[p[0], p[1], \ldots, (p[l] - 1), j]$ , where p is the tree address of the parent, and j is child number. We use peculiar conventions that facilitate computation of tree addresses in a circuit. First, note the use of (p[l] - 1) in the tree address of the child, as opposed to the perhaps more natural p[l]. Second, we number children in decreasing order left to right. The right-most child (the one parsed last) has j = 1, and the child number of a left sibling is one plus the child number of its right sibling. With these conventions, the last element in the tree address is always nonzero.

While the tree address is a variable-length array, in a circuit we represent it as an array of wires of fixed size *L*, where *L* is the maximum nesting depth supported by the circuit, implicitly padded with zeroes. Thus, in the actual circuit, all tree addresses are stored in an array *counters*[*i*][*l*] of wires, indexed by position and nesting depth.

Assuming we have the tree address of every position *i* that starts a token (i.e., where *header*[*i*] is true), we can make assertions such as "this node at depth (l + 1) is a child of that node at depth *l*", by comparing prefixes of the tree addresses up to *l*. Therefore, having the tree address available allows us to generate circuits that check assertions about the syntactic structure of the document.

The only remaining problem is now to design a circuit that computes the

|         | MSG (B) | DEPTH | QUADS     | TERMS     | INPUTS |
|---------|---------|-------|-----------|-----------|--------|
|         | 247     | 28    | 115,602   | 226,557   | 2,227  |
|         | 503     | 30    | 253,340   | 545,409   | 4,531  |
| T       | 1079    | 34    | 616,331   | 1,559,052 | 9,715  |
| IF P256 | 1591    | 34    | 895,257   | 2,702,352 | 14,323 |
|         | 2231    | 36    | 1,343,877 | 4,588,287 | 20,083 |
|         | 2551    | 36    | 1,526,867 | 5,651,883 | 22,963 |

Table 3: Circuit size and depth for CBOR parsing messages of varying size over a prime field. The message sizes correspond to messages that can be hashed with SHA256 using an integral number of blocks.

tree address. While we defined tree addresses for tokens, or equivalently, for positions *i* such that *header*[*i*] holds, in a circuit it is more convenient to compute counters[i] for every *i*. If *header*[*i*] is false, we define counters[i] = counters[i-1], which is well-defined as long as *header*[0] is true.

We proceed as per the case of the tokenizer by first giving a sequential algorithm for computing tree addresses, and then converting it into a circuit. Our starting point is Algorithm 8, which computes counters[i + 1] given counters[i]and the output of the tokenizer. If header[i] is false, the procedure sets  $counters[i] \leftarrow$ counters[i - 1], as we postulated. Otherwise, let l (explicitly called l[i] in the code) be the highest depth such that  $counters[i][l] \neq 0$ , which is the depth of the tree node by definition of tree address. Thus, counters[i + 1][l] gets the tree address counters[i][l] of its left sibling minus one, as per our conventions on the numbering of children. If the token at position i starts an array of length k, we extend the tree address by one depth  $counters[i + 1][l + 1] \leftarrow k$ , where, again by convention, k is the child number of the leftmost child in the array (the first array element).

As in the case of the tokenizer, one could in principle unroll the circuit for all *i*, obtaining a deep circuit that computes all *counters*[*i*] starting with counters[0] = [1, 0, 0, ...]. To reduce the depth, we provide l[i] as witness to break the dependencies, at which point the computation of counters[i][l] is reduced to *L* segmented scan operations, one per depth *l*, plus assertions that the witnesses are correct.

**Circuit sizes** Table 3 presents the cost of parsing CBOR documents of varying sizes corresponding to the message size that can be hashed using *n* SHA256 blocks. These sizes refer to a parser for the full CBOR language, not the toy CBOR that we use for illustration purposes. The table refers to one point in design space, but other tradeoffs are possible. For example, in our implementation, the input to the circuit consists of nine field elements per input byte. Eight field elements encode the input to be hashed, one element per bit, and the ninth field element encodes the *header* and l[i] witnesses in a packed format. One could trade input size for circuit size depending on application needs.

## 5 Evaluation

We implemented our ZK protocol in roughly 25,500 lines of C++ code of which roughly 8,600 lines correspond to unit tests or benchmarks. Our implementation uses only a single thread; our reported wall-clock timings thus reflect the overall computation performed.

In this section, we report micro-benchmarks and end-to-end benchmarks for several applications involving anonymous credential operations. We collected these performance numbers using the C++ benchmark package on both an x86\_64 machine from the Emerald Rapids family and a Pixel Pro 6 phone. The x86\_64 platform is a c4-highcpu-8 instance running in the central1-c availability zone in Google Cloud. This machine has four Intel Xeon PLAT-INUM 8581C CPU (cpu family 6, model 207) cores running at 2.30GHz with 16gb of RAM. The Pixel Pro 6 platform contains a Google Tensor processor consisting of 8 cores and 12 GB of RAM. The 8 cores consists of two 2.80 GHz Cortex-X1, two 2.25 GHz Cortex-A76 and four 1.80 GHz Cortex-A55. Our benchmarks are executed on the 2.80 Ghz Cortex-X1 processor. We use SHA256 to implement the collision-resistant hash function, and the random oracle, resulting in a 128-bit computational security parameter. For our Ligero commitment, we open 128 columns, which corresponds to roughly 86 bits of statistical security. These choices result in a proof size for our main application that fits within the Android framework's *Intent buffers* for interprocess communication. When collecting timings on the phone, benchmarks are run on the performance core and restricted to 1s runs to reduce measurement errors arising from thermal throttling.

We store prime-field elements in Montgomery form, and we implement field multiplication on top of the 64x64-bit scalar multiplier (that is, we don't use SIMD instructions). For  $GF(2^{128})$  we use 128-bit SIMD registers on both arm64 and x86\_64, and implement multiplication on top of the 64x64-bit carryless multiplier, which is implemented by x86\_64 and by the arm64 AES instruction set. We currently do not use AVX256, AVX512, or GFNI.

#### 5.1 Algebraic primitives

We begin by presenting low-level benchmarks for our algebraic primitives. The first benchmark in Table 4 shows the performance of our NTT implementation across different fields and processors. The  $\mathbb{F}_{p128}$  and  $\mathbb{F}_{p64^2}$  fields both have at least  $2^{32}$  roots of unity. As noted before,  $\mathbb{F}_{p256}$  does not have roots of unity, and therefore we must use a quadratic extension to introduce them for computing the NTT. As a result, we see roughly 4-5x penalty in performance in this field versus the 128-bit prime fields.

**Reed-Solomon Encoding** Table 5 reports on Reed-Solomon encoding benchmarks because this step is a bottleneck in the Ligero commitment. Each test in this section measures the time to encode one array of size *n* into a codeword of

|                | Time (in ms)         |                 |          |          |          |          |      |  |
|----------------|----------------------|-----------------|----------|----------|----------|----------|------|--|
|                |                      | 2 <sup>12</sup> | $2^{14}$ | $2^{16}$ | $2^{18}$ | $2^{20}$ | 222  |  |
|                | $GF(2^{128})$        | 0.042           | 0.195    | 0.883    | 5.17     | 26.7     | 134  |  |
| x86 64         | $\mathbb{F}_{p64}$   | 0.058           | 0.301    | 1.55     | 7.29     | 31.2     | 131  |  |
| X00_01         | $\mathbb{F}_{p64^2}$ | 0.175           | 0.860    | 3.89     | 17.2     | 77.0     | 353  |  |
|                | $\mathbb{F}_{p128}$  | 0.154           | 0.746    | 3.61     | 16.1     | 70.8     | 320  |  |
|                | $\mathbb{F}_{p256}$  | 1.41            | 6.70     | 32.7     | 146      | 660      | 2970 |  |
|                | $GF(2^{128})$        | 0.095           | 0.480    | 2.29     | 9.98     | 49.6     | 224  |  |
| Pixel 6 Pro    | $\mathbb{F}_{p64}$   | 0.086           | 0.404    | 2.12     | 9.19     | 42.8     | 201  |  |
| 1 1.001 0 1 10 | $\mathbb{F}_{p64^2}$ | 0.248           | 1.19     | 5.91     | 28.2     | 130      | 566  |  |
|                | $\mathbb{F}_{p128}$  | 0.231           | 1.12     | 5.53     | 26.8     | 123      | 538  |  |
|                | $\mathbb{F}_{p256}$  | 1.84            | 8.82     | 43.9     | 197      | 882      | 3870 |  |

Table 4: NTT benchmarks over different fields. The x86 processor is between 1–2x faster than the Pixel mobile device processor. In all cases, the GF(2<sup>128</sup>) field is the most performant sound field by a factor of at least 2x. The  $\mathbb{F}_{p64}$ , while not sound, is presented to show the overhead for the quadratic field extension.

size 4*n* (i.e., rate <sup>1</sup>/4). Recall from §3 that our Reed-Solomon encoding, or polynomial interpolation algorithm for prime fields relies on a convolution, which itself requires 2 NTTs and multiplication by pre-computed constants. In constrast, the encoding in GF(2<sup>128</sup>) field uses 1 bidirectional FFT and 4 FFTs of the same size as the input. An interesting measurement here reports a  $\approx$ 3x overhead for encoding over the quadratic single-precision field  $\mathbb{F}_{p64^2}$  compared to  $\mathbb{F}_{p64}$ .

|             |                      | Time (in ms) |                 |          |                 |                 |                 |
|-------------|----------------------|--------------|-----------------|----------|-----------------|-----------------|-----------------|
|             |                      | $2^{10}$     | 2 <sup>12</sup> | $2^{14}$ | 2 <sup>16</sup> | 2 <sup>18</sup> | 2 <sup>20</sup> |
|             | $GF(2^{128})$        | 0.037        | 0.175           | 0.794    | 3.75            | 21.5            | 113             |
| 286 61      | $\mathbb{F}_{p64}$   | 0.125        | 0.647           | 3.24     | 15.2            | 65.4            | 278             |
| X00_04      | $\mathbb{F}_{p64^2}$ | 0.366        | 1.83            | 8.40     | 36.3            | 166             | 757             |
|             | $\mathbb{F}_{p128}$  | 0.395        | 1.86            | 8.81     | 38.8            | 170             | 758             |
|             | $\mathbb{F}_{P256}$  | 3.15         | 14.7            | 71.2     | 316             | 1420            | 6360            |
|             | $GF(2^{128})$        | 0.085        | 0.408           | 2.10     | 9.81            | 45.5            | 219             |
| Pivol 6 Pro | $\mathbb{F}_{p64}$   | 0.194        | 0.983           | 5.05     | 22.2            | 101.4           | 453             |
| 1120110     | $\mathbb{F}_{p64^2}$ | 0.560        | 2.79            | 13.6     | 63.6            | 289             | 1250            |
|             | $\mathbb{F}_{p128}$  | 0.582        | 2.94            | 14.3     | 66.7            | 298             | 1280            |
|             | $\mathbb{F}_{P256}$  | 4.16         | 19.6            | 97.2     | 432             | 1910            | 8510            |

Table 5: Reed-Solomon encoding performance over several fields.

**Comparison to Linear-time codes** Golovnev *et. al* [GLS+21] and Xie, Zhang, and Song [XZS22] propose the use of linear-time encodable error-correcting codes to construct a polynomial commitment scheme. It is not immediately clear how to augment either to support ZK verification of linear and quadratic constraints (instead, both of those schemes were designed to answer the final query in a sumcheck-based verifiable computation protocol). Moreover, such codes have poor distance properties, and thus require many queries to the oracle, resulting in proof sizes that are an order of magnitude larger and thus impractical for our application. For example, the Brakedown scheme is described with distance 1/20 and rate 3/5, and therefore requires at least 6500 queries.

We also show that known linear-time codes are *slower* than Reed-Solomon implementation for the relevant parameter regime. In Table 6, we benchmark the Brakedown implementation of linear-time encodable codes over 128-bit fields. Because that scheme requires at least 6500 queries to the codeword, their benchmark begins at  $2^{14}$ . For our parameter regimes, the coefficient for their linear-time encoding scheme seems to be larger than our quasi-linear Reed-Solomon encoding in GF( $2^{128}$ ), as our implementation begins at +5x faster and remains 3x faster at  $n = 2^{20}$ .

|           |                       |          | Time (in ms) |              |              |              | Queries         |              |
|-----------|-----------------------|----------|--------------|--------------|--------------|--------------|-----------------|--------------|
|           |                       | $2^{10}$ | $2^{12}$     | $2^{14}$     | $2^{16}$     | $2^{18}$     | 2 <sup>20</sup> | needed       |
| Brakedown | x86_64<br>Pixel 6 Pro | -        | -            | 5.00<br>6.09 | 19.7<br>25.3 | 77.7<br>98.6 | 360<br>466      | 6593<br>6593 |

Table 6: Encoding time for Linear-time encodable codes over a 128-bit field. These codes require at least 6500 queries in an IOP, and thus, we benchmark them starting at  $n = 2^{14}$ . These codes are slightly slower than our RS  $\mathbb{F}_{p64}$  code and substantially slower than our GF( $2^{128}$ ) code.

#### 5.2 SHA256 hash verification

Many ZK proof systems publish a benchmark for proving knowledge of a SHA256 pre-image of a value *e*. We design a circuit parameterized by the number of block operations required to hash a message. Our SHA256<sub>*n*</sub>(*e*, *m*) circuit verifies that the private message *m* can be hashed using at most *n* SHA256 block operations (i.e., is at most n \* 64 - 9 bytes long) and hashes to the public value *e*. This circuit corresponds to the one described in §4.2 with bit-packing 2. Our timing results for 1–32 blocks appear in Table 7. Table 8 reports commitment parameters required by our proof for one block of SHA256.

|             |                   | (     | OurZK Protocol, Time (ms) |      |       |     |     |  |
|-------------|-------------------|-------|---------------------------|------|-------|-----|-----|--|
|             |                   | n = 1 | n = 1 2 4 8 16            |      |       |     |     |  |
| v86 61      | Verify transcript | 6.07  | 12.1                      | 24.3 | 49.8  | 106 | 228 |  |
| X00_04      | Total ZK Prover   | 10.7  | 19.8                      | 35.2 | 67.9  | 140 | 286 |  |
| Pivol Pro 6 | Verify transcript | 11.3  | 23.5                      | 48.9 | 100.1 | 206 | 436 |  |
| 1 1201 100  | Total ZK Prover   | 19.2  | 37.0                      | 68.9 | 132   | 257 | 517 |  |

Table 7: Timing benchmark for our ZK proof of knowledge of an *n*-block pre-image to the SHA256 function. The circuit used is described in §4.2 over  $GF(2^{128})$ . The sumcheck row reports the time required to create the verify transcript for the circuit. The Total row reports the total prover time (including sumcheck) to produce the proof.

| SHA256 <sub>1</sub> | w     | Witness/row | Encoded row size | Total rows |
|---------------------|-------|-------------|------------------|------------|
| $\mathbb{F}_p$      | 4,001 | 681         | 2 <sup>13</sup>  | 11         |
| $GF(2^{128})$       | 4,228 | 681         | $2^{13}$         | 12         |

Table 8: Commitment size parameters for 1 block of our SHA256-preimage proof in different fields.

#### 5.2.1 Comparison with Ligero

A natural alternative to our system is to use Ligero to verify the same function. As mentioned in the introduction, Ligero requires committing to an internal tableau of witnesses and committing 3 elements for every quadratic constraint. As a result, the size of the Ligero commitment for a given circuit *C* is O(|C|), and in practice, many times larger than the corresponding verify transcript for verifying *C*. In this section, we provide concrete evidence for this claim with respect to the SHA256 circuit.

One issue is that the quad-form circuits that we generate for our protocol are optimized to reduce the number of *quads* whereas Ligero performs better with a circuit that optimizes for the total number of quadratic constraints. Thus, in order to provide a fair comparison, we used the circom system to generate an R1CS instance for our family of SHA256<sub>n</sub> circuits described above. Recall that an R1CS instance consists of a witness tableau *w* and a list of constraints in the form of  $A \cdot B - C = 0$  where A, B, C are linear combinations of a witness vector *w*. Smaller R1CS instances naturally require fewer multiplications to verify. We converted this R1CS into Ligero constraints following the description in their paper. Namely, for each A, B, C, we introduce new Ligero witnesses  $\alpha_i, \beta_i, \kappa_i$  respectively that are constrained to be equal to the linear combination of the original witness *w*. We then introduce a new witness value  $\gamma_i = \alpha_i \cdot \beta_i$ , and then add a linear constraint that verifies  $\gamma_i - \kappa_i = 0$ . In total, we introduce 4 new witnesses, 4 linear constraints and 1 quadratic constraint

for each R1CS constraint. Many R1CS constraints have an empty *C* term, i.e., C = 0; in this case, a simple optimization removes the extra witness value and the fourth linear constraint.

Altogether, we form a Ligero commitment using all of the base R1CS witnesses plus the extra witnesses needed by Ligero to check R1CS. Table 9 reports the both of these figures. In addition, each quadratic constraint on a witness requires adding a *copy* of the two terms and their product as witnesses. All of these witnesses are partitioned into rows of size w, with the additional constraint that the copies of the three witnesses for a quadratic constraint need to be on separate rows at the same index, and can only include half as many per row. Achieving ZK also requires reserving spots in each row for a random pad, and adding 2 additional random rows. A number of additional constraints govern the size of w. For each Ligero commitment, we loop over all feasible parameters, and chose a parameter set that minimizes the size of the commitment. These optimal choices are also presented in Table 9. Notice, as per Table 2, directly transforming our circuit into Ligero for n = 1 requires at least 167,348 quadratic constraints (roughly 3 times larger than the R1CS).

|        |                          | $SHA256_n(\cdot)$ |          |          |
|--------|--------------------------|-------------------|----------|----------|
|        |                          | n = 1             | 2        | 4        |
| D1CC   | Witnesses                | 29,853            | 60,381   | 121,437  |
| KIC5   | Constraints              | 29,725            | 60,053   | 120,709  |
|        | Total Witnesses          | 129,500           | 261,484  | 525,452  |
| Ligero | Quadratic constraints    | 29,725            | 60,053   | 120,709  |
| Ligero | Witness/row for <i>w</i> | 3,149             | 6,426    | 6,426    |
|        | Quadratic copies/row     | 1,511             | 3,149    | 3,149    |
|        | Encoded Row size         | $2^{15}$          | $2^{16}$ | $2^{16}$ |
|        | Number of rows           | 104               | 103      | 201      |

Table 9: Size of the Ligero instance for n blocks of SHA256 as derived from an optimal R1CS expression of SHA256. To select the row size used by Ligero, we test various power of 2 for the row to find the one which minimizes proof size.

Finally, we benchmark our implementation of Ligero on these instances and report timings in Table 10. Note that the most expensive component of Ligero is the commitment stage. For soundness, Ligero must use a 128-bit field size, which can often be implemented using a degree-2 or degree-4 field extension of a smaller field. We have not fully optimized our underlying NTT for Ligero, and therefore, we report benchmarks using a 64-bit prime. While this parameter setting is not as sound as our own implementation, it demonstrates that our system is at least *roughly 20x faster* for n = 1, 2, or 4 block messages, thus supporting our introductory claim. To be sure, our numbers in Table 10 roughly match those presented by Wang, Hazay, and Venkitasubramaniam in the optimized Ligetron [WHV24, Fig 6(a)] implementation. That system shows

|             |                           | Time (ms)  |            |              |
|-------------|---------------------------|------------|------------|--------------|
|             |                           | n = 1      | 2          | 4            |
| x86_64      | Commit<br>Total ZK Prover | 222<br>273 | 384<br>493 | 736<br>939   |
|             | Overhead wrt this work    | 26x        | 25x        | 27x          |
|             | [WHV24]                   | 250        | 450        | 750          |
| Pixel 6 Pro | Commit<br>Total ZK Prover | 294<br>380 | 536<br>717 | 1022<br>1370 |
|             | Overhead wrt this work    | 20x        | 19x        | 20x          |

roughly 250ms, 450ms, 750ms for n = 1, 2, 4 blocks respectively; however their benchmark uses 4 threads on a 4-core machine with a 50-bit prime, and higher statistical security parameter, so the comparison is not exact.

Table 10: Ligero ZK Prover time of our implementation for verifying the preimage of an *n*-block message under SHA256. The first line reports the time to Commit to the witness tableau. This benchmark was run using the  $\mathbb{F}_{p64}$  field, which is not sufficient for soundness, but yet represents a bound on Ligero performance. Ligero is essentially 20x slower on x86\_64 and Pixel platforms due to the larger commitment that is required.

The  $\approx 20x$  performance improvement of our system over pure-Ligero can be partly explained by inspecting the witness size. As per table 8, for a single SHA256 block, the *full* witness in our proof system contains roughly 9x fewer rows (11 vs 104) with a block size that is 4x smaller, roughly reflecting a 36x reduction in work to produce the commitment; however, our results use a 128bit field and thus incur a roughly 2x overhead with respect to  $\mathbb{F}_{p64}$  used in Table 10.

#### 5.2.2 Comparison with Binius

Diamond and Posen [DP23, DP24] implement a commitment scheme and a verifiable computation protocol (i.e., the protocol is not zero-knowledge) in the Binius [bin] system. They provide a SHA256 benchmark that verifies 32 independent applications of the SHA compression function; their x86\_64 implementation also exploits native gfni instructions which our system does not. On x86\_64, their benchmark, when run on a single core, runs in 498ms (versus our verify which runs in 228ms), and on Pixel, their benchmark runs in 904ms (versus our 436ms). Although their benchmark numbers are larger than ours for a similar problem, the performance gap closes as *n* increases; their system incorporates ideas that may also apply and speedup our implementation in future work.

#### 5.2.3 Comparison with other ZK systems

The STARK system is a ZK argument system that requires no trusted setup parameters and also produces a smaller proof than other proof systems including Ligero. However, the cost of the smaller proof, as established by many published benchmarks in the literature [GLS<sup>+</sup>21] show that the STARK prover time is larger than the Ligero prover. In future work, we aim to benchmark the same SHA256 for Stark.

We do not benchmark the proof systems which require a trusted parameter setup because they do not satisfy our deployment requirements. Nonetheless, results from prior work establish that these systems tend to have prover algorithms which require more work. In addition to the standard NTT on the size of the circuit, they also require multi-scalar multiplication routines over elliptic curve groups, which are generally costly.

#### 5.3 ECDSA signature verification

Consider the ZK instance (e, pk) and corresponding protocol that verifies that there exist (r, s) such that ecdsaVerify(pk, e, r, s) = 1. As described in §4.1, we construct a circuit that requires 1085 witness values to complete the verification in low-depth. Table 11 reports the basic ZK benchmarks for this problem. This benchmark demonstrates that contrary to folklore, a zero knowledge proof of possession of a signature for ECDSA—which requires roughly 60ms—is not a practical bottleneck to building a credential system, and is within a small factor of performance for a basic BBS+ proof of possesion of a signature as we show below.

|             |                              | Т     | ime (ms | 5)    |
|-------------|------------------------------|-------|---------|-------|
|             |                              | n = 1 | 2       | 3     |
| x86_64      | Ligero commitment            | 38.7  | 51.0    | 60.8  |
|             | Generating Verify transcript | 13.5  | 26.5    | 38.6  |
|             | Total ZK Prover              | 58.8  | 87.0    | 110   |
|             | Total Verifier               | 6.09  | 11.0    | 14.5  |
|             | Ligero commitment            | 51.0  | 67.2    | 80.0  |
| Pixel Pro 6 | Generating Verify transcript | 20.3  | 40.1    | 58.4  |
|             | Total ZK Prover              | 80.5  | 120.0   | 152.0 |
|             | Total Verifier               | 8.50  | 16.2    | 21.2  |

Table 11: Timing benchmark for verification of an ECDSA signature. The circuit used here is described in §4.1 over field  $\mathbb{F}_{p256}$ . The Total Prover time consists of the Ligero commitment time, the time to produce the Verify transcript, and other small steps.

**Comparison to the BBS+ scheme** To compare our results with BBS+, which provides the fastest zero-knowledge proof of possesion of a signature in the literature, we benchmarked the pairing\_crypto package [Pai24] configured to use the BLS12-381 curve and SHA256 with signatures that included 100 attributes and presentations that revealed 3 attributes. The criterion package produced benchmark results. On x86\_64, the BBS+ proof generation step required 10.2 ms and the verify step required 8 ms; our ECDSA proof generation method is within  $\approx$  6x the performance. However, the BBS microbenchmark does not account for additional tasks needed to perform a credential show operation, such as verification that the certificate has not expired.

## 6 Applications

In this section, we apply our anonymous credential system to applications for digital identity.

#### 6.1 A simple anonymous credential

To illustrate how the components from our system can be used in an anonymous credential system, we designed a toy document format for credentials that is based on the machine readable zone (MRZ) format used in passports.



Figure 2: A simple document format for a 183-byte credential.

In Fig 2, we describe a 183-byte credential that allows for at least 60 standard attributes to be encoded, in addition to a device public key for devicebounded checks, and time values to indicate the validity period. Notably, this format requires only 3 SHA256 blocks to hash, and it requires no parsing to determine the byte-location of an attribute. For this credential, we develop a simple benchmark to prove that the credential holder is older than 18 (which amounts to checking the value of byte 74 in the format). See Algorithm 9 for the specific theorem being proven.

The benchmark in table 12 shows that this toy format achieves performance that is within a small factor of the time required to verify on BBS+ credential

#### Algorithm 9 Toy Credential Verification

**Require:** x = (pk, a, id, tr, now) and  $w = (MSO, pk_{dx}, pk_{dy})$ 1: Verify the following constraints:  $e_1 = SHA256(MSO[0:183])$  a = MSO[id]  $(pk_{dx}, pk_{dy}) = MSO[96:160]$   $t_{start} = MSO[48:56]$   $t_{end} = MSO[56:64]$   $t_{start} < t_{now} < t_{end}$ true =  $p256.verify((r_1, s_1), e_1, PK_{II})$ true =  $p256.verify((r_2, s_2), H(tr||hdr), (pk_{dx}, pk_{dy}))$ 

(without incorporating the time to format the BBS+ attributes or check credential validity).

|             | DEPTH            | QUADS       | TERMS   | INPUTS    |
|-------------|------------------|-------------|---------|-----------|
| Toy Circuit | 12               | 276,465     | 818,533 | 10,123    |
|             |                  | <b></b> ( ) | D (     | . (11)    |
|             | Prover Time (ms) |             | Proof   | size (kb) |
| x86         | 332              |             | 291     |           |
|             |                  |             |         |           |

| Table 12: Toy credential benchmark over $\mathbb{F}_{n'}$ |
|---|
|---|

### 6.2 Anonymous Credentials for MDOC

ISO standard 18013-5 [18021] specifies a digital identity format that is widely used in mobile driver licenses and national identity formats. For example, Arizona, California, Colorado, Georgia, Maryland and New Mexico issue driver licenses in the MDOC format to certain mobile devices.

The MDOC format supports a form of *selective disclosure* in which the holder of an MDOC can reveal a subset of the attributes that are asserted in the credential. This disclosure format is implemented by including salted hashes of the attribute values in the credential format that is signed by the issuer. The salted-hash technique has two well-known privacy flaws. If Alice uses the same MDOC credential to assert identities to relying parties  $P_1$  and  $P_2$ , it is possible for  $P_1$  and  $P_2$  to combine their views of the presentation protocol in order to track the user across sessions. For example, the expiration times, or device key values can be used to track a user or link a user between different relying parties. A simple counter-measure is to require the user to use *different* MDOCs for each relying party—this measure comes at a cost for both the issuer, who now must issue 1000s of mdocs to each user, and the user, who must maintain state to ensure the unique property. A more serious flaw enables the issuer of the MDOC to collaborate with the relying party in order to identity users across sessions. This flaw has delayed the deployment of MDOC for some applications.

**ZK for MDOC** A solution to the issuer-RP collusion problem for the MDOC format is to modify the presentation protocol so that the user instead produces a zero-knowlege proof that "their mdoc verifies with respect to the requested attributes." Specifically, we aim to convince the verifier of the theorem statement shown in Algorithm 10 while leaking no extra information:

Algorithm 10 MDOC Verification

| <b>Require:</b> | $x = (PK_{II}, attr, Z, tr, time)$ and $w$ consists of a 2231 byte string MSO,             |
|-----------------|--|
|                 | a length len, a hash $e_1$ , a hash $h_2$ , an index $X$ , a signature pair $(r_1, s_1)$ , |
|                 | a parsing witnesses $c$ , a 32-byte nonce, a pair $(pk_{dx}, pk_{dy})$ , a pair of         |
|                 | strings $t_{start}$ , $t_{end}$ , a header hdr and a signature ( $r_2$ , $s_2$ )           |
| 1: Verify       | the following relations:   |
|                 | $e_1 = SHA256(MSO[0:len])$   |
|                 | $h_2 = MSO[valueDigests][org.iso.18013.5.1][X]$  |
|                 | $h_2 = SHA256(nonce, attr, Z)$   |
|                 | $(pk_dx, pk_dy) = MSO[deviceKeyInfo][deviceKey][-2, -3]$                                   |
|                 | $t_{start} = MSO[validityInfo][validFrom]$   |
|                 | $t_{end} = MSO[validityInfo][validUntil]$  |
|                 | $t_{start} < t_{now} < t_{end}$  |
|                 | $true = p256.verify((r_1, s_1), e_1, PK_{II})$   |
|                 | $true = p256.verify((r_2, s_2), H(tr  hdr), (pk_{dx}, pk_{dy}))$                           |

Importantly, the public values for this theorem are ( $PK_{II}$ , attr, Z, tr, time) these values are shared between the prover (the user with the mdoc) and the verifier (the relying party). Here,  $PK_{II}$  is the public key of the identity-issuer, Z is an attribute value to disclose, e.g, like the "age\_over\_18" boolean, and attris the name of this attribute (as written in the mso), tr is the liveness transcript, and  $t_{now}$  is used to verify that the mdoc has not expired. The rest of the values in the statement are hidden, in the sense that the verifier is convinced that such values exist, but does not learn them through the interaction.

Per Table 13, on an X86 processor, our benchmark prover runs in 1.15s and our verifier runs in 0.52s, and on the Pixel 6 Pro, 1.17s and 0.68s respectively. This benchmark supports an mdoc that is 2231 bytes long, and the most expen-

|             | Prover (ms) | Verifier (ms) |
|-------------|-------------|---------------|
| x86_64      | 1150        | 520           |
| Pixel 6 Pro | 1170        | 680           |

Table 13: Prover time for ZK MDOC presentation of the age-over-18 attribute for an MSO of at most 2231 bytes. The circuit corresponds to Alg. 10.

sive portion of the proof concerns the SHA256 hash of the MSO. Our benchmark implements the two-field optimization, with the SHA256 circuit verified in  $GF(2^{128})$ .

**Extensions** This basic application can be conveniently extended to support more privacy features. For example, it is possible to also hide the identity of the issuer, and instead prove that the issuer is a member of a certified list of issuers. It is possible to support the disclosure of multiple attributes instead of 1 at little cost. Finally, it is possible to support the revocation of MDOC credentials in zero-knowledge by using standard ZK list-membership techniques. In future work, we intend to include benchmarks for these extensions.

Acknowledgements John Kuszmaul implemented the Reed-Solomon encoding algorithm discussed in Section 3.1 while he was an intern in our group. We thank Sergeui Alleko for code review and helpful discussions. We thank Quan Nguyen, Cory Barker, Justin Brickell, Bogdan Brinzarea Iamandi, Gareth Oliver, and David Zeuthen for helpful comments and discussions. We also thank Justin Thaler, Muthu Venkitasubramaniam, Carmit Hazay, Jonathan Katz, and Mark Moir for helpful comments and references to related work. We are especially grateful to Shiv Venkataraman, Sunita Verma and Joshy Joseph for sponsoring this project.

## References

- [18021] ISO/IEC FDIS 18013-5. Personal identification—iso-compliant driving licence—part 5: Mobile driving licence (mdl) application, 2021.
- [AHIV22] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. *eprint/2022/1608 and CCS'17*, 2022.
- [BH20] Carsten Bormann and Paul E. Hoffman. Concise Binary Object Representation (CBOR). RFC 8949, December 2020.
- [bin] Binius implementation. gitlab.com/IrreducibleOSS/binius. git.

- [BK82] Brent and Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31(3):260–264, 1982.
- [Ble90] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.
- [BOGG<sup>+</sup>88] Michael Ben-Or, Oded Goldreich, Shafi Goldwasser, Johan Håstad, Joe Kilian, Silvio Micali, and Phillip Rogaway. Everything provable is provable in zero-knowledge. In *CRYPTO'88*, 1988.
- [Bra99] Stefan Brands. *Rethinking Public Key Infrastructure and Digital Certificates*. PhD thesis, Eindhoven Institute of Technology, 1999.
- [CCH<sup>+</sup>18] Ran Canetti, Yilei Chen, Justin Holmgren, Alex Lombardi, Guy N. Rothblum, Ron D. Rothblum, and Daniel Wichs. Fiat-shamir from simpler assumptions. In *eprint/2018/1004*, 2018.
- [CDL16] Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation using the strong Diffie Hellman assumption revisited. In *Cryptology ePrint Archive, Paper 2016/663*, 2016.
- [Cer09] Certicom. Sec 1: Elliptic curve cryptography, v2.0. Standards document, 2009.
- [Cha85] David Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 10(28):1030–1044, 1985.
- [CL01] Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *Eurocrypt'2001*, 2001.
- [CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, pages 90–112, 2012.
- [DLFKP16] Antoine Delignat-Lavaud, Cedric Fournet, Markulf Kohlweiss, and Bryan Parno. Cinderella: Turning shabby x.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *Oakland IEEE S&P 2016*, 2016.
- [DP23] Benjamin E. Diamond and Jim Posen. Succinct arguments over towers of binary fields. Cryptology ePrint Archive, Paper 2023/1784, 2023.
- [DP24] Benjamin E. Diamond and Jim Posen. Polylogarithmic proofs for multilinears over binary towers. Cryptology ePrint Archive, Paper 2024/504, 2024.

- [GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In *STOC'08*, 2008.
- [GLS<sup>+</sup>21] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic snarks for r1cs. In *eprint* 2021/1043, 2021.
- [Gow92] Roderick Gow. Cauchy's matrix, the Vandermonde matrix and polynomial interpolation. *Bulletin of the Irish Mathematical Society*, 28:45–52, March 1992.
- [Knu97] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.
- [KZM<sup>+</sup>15] Ahmed Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, T-H. Hubert Chan, Charalampos Papamanthou, Rafael Pass, abhi shelat, and Elaine Shi. CØCØ: A framework for building composable zero-knowledge proofs. eprint/2015/1093, 2015.
- [LANHC16] Sian-Jheng Lin, Tareq Y. Al-Naffouri, Yunghsiang S. Han, and Wei-Ho Chung. Novel polynomial basis with fast Fourier transform and its application to reed–solomon erasure codes. *IEEE Transactions on Information Theory*, 62(11):6284–6299, 2016.
- [LCH14] Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang S. Han. Novel polynomial basis and its application to reed-solomon erasure codes. In 2014 IEEE 55th Annual Symposium on Foundations of Computer Science, pages 316–325, 2014.
- [LCK<sup>+</sup>18] Wen-Ding Li, Ming-Shing Chen, Po-Chun Kuo, Chen-Mou Cheng, and Bo-Yin Yang. Frobenius additive fast Fourier transform, 2018.
- [LFKN92] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. J. ACM, 39:859– 868, October 1992.
- [LRSW99] Anna Lysyanskaya, Ron Rivest, Amit Sahai, and Stefan Wolf. Pseudonym systems. In *Selected Areas in Cryptography*, 1999.
- [Mat08] Todd Mateer. *Fast Fourier transform algorithms with applications*. PhD thesis, Clemson University, 2008.
- [Nus80] Henri J. Nussbaumer. Fast polynomial transform algorithms for digital convolution. IEEE Transactions on Acoustics, Speech, and Signal Processing, 28:205–215, 1980.

- [Pai24] Pairing crypto bbs implementation. https://github.com/ mattrglobal/pairing\_crypto, 2024.
- [RWGM23] Michael Rosenberg, Jacob White, Christina Garman, and Ian Miers. zk-creds: Flexible anonymous credentials from zksnarks and existing identity infrastructure. In *Oakland IEEE S&P*, 2023.
- [Skl60] Jack Sklansky. Conditional-sum addition logic. *IRE Trans. Electron. Comput.*, 9:226–231, 1960.
- [Tha13] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO'13*, 2013.
- [Tha22] Justin Thaler. Proofs, arguments, and zero-knowledge. Manuscript, 2022.
- [vdH04] Joris van der Hoeven. The truncated fourier transform and applications. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, page 290–296, New York, NY, USA, 2004. Association for Computing Machinery.
- [vdH05] Joris van der Hoeven. Notes on the truncated Fourier transform, 2005.
- [vzGG96] Joachim von zur Gathen and Jürgen Gerhard. Arithmetic and factorization of polynomial over f2 (extended abstract). In Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation, ISSAC '96, pages 1–9, New York, NY, USA, 1996. Association for Computing Machinery.
- [WHV24] Ruihan Wang, Carmit Hazay, and Muthuramakrishnan Venkitasubramaniam. Ligetron: Lightweight scalable end-to-end zeroknowledge proofs post-quantum zk-snarks on a browser. In *Oakland IEEE S&P'24*, 2024.
- [WJB<sup>+</sup>17] Riad S Wahby, Ye Ji, Andrew J Blumberg, abhi shelat, Justin Thaler, Michael Walfish, and Thomas Wies. Full accounting for verifiable outsourcing. In ACM CCS'2017, 2017.
- [WTas<sup>+</sup>18] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zksnarks without trusted setup. In *eprint*/2017/1132 and Oakland S&P'2018, 2018.
- [XZS22] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In *eprint/2022/1010*, 2022.
- [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.

## A Interpolation via convolution

Let p(k) be a polynomial of degree at most d. Given the evaluations p(i) at integer points  $0 \le i \le d$ , we want to compute p(k) for all integers k in some range. Our next goal is to express this polynomial interpolation as a single convolution operation.

**Lemma A.1.** (*Lagrange interpolation.*) Let p(k) be a polynomial of degree at most d over a field. For all k we have

$$p(k) = \sum_{0 \le i \le d} \binom{k}{i} \binom{d-k}{d-i} p(i) .$$
(9)

*Proof.* Let q(k) be the right-hand side of Equation (9). We want to prove that p = q.

The binomial coefficient  $\binom{x}{r}$  is a polynomial in *x* of degree at most *r*. Thus q(k) is a polynomial of degree at most *d*. We now prove that p(k) = q(k) for  $0 \le k \le d$ , and therefore that the two polynomials are the same.

Assume  $0 \le k \le d$ . If i = k, the *i*-th term of the sum is p(k). If i > k, we have  $\binom{k}{i} = 0$ , and thus the *i*-th term of the sum is 0. If i < k, we have  $\binom{d-k}{d-i} = 0$ , and thus *i*-th term of the sum is also 0. Thus q(k) = p(k) as desired.

**Lemma A.2.** (Lagrange interpolation via convolution.) Let p(k) be a polynomial of degree at most d over a field. For k > d, we have

$$p(k) = (-1)^d \cdot (k-d) \binom{k}{d} \cdot \sum_{0 \le i \le d} \left(\frac{1}{k-i}\right) \cdot (-1)^i \binom{d}{i} p(i) .$$
<sup>(2)</sup>

*Proof.* The proof is a straightforward manipulation of Equation (9) to separate the terms that depend on k from those that depend on i and on k - i.

We use the well-known binomial negation of the upper index

$$\binom{-r}{s} = (-1)^s \binom{r+s-1}{s}$$

to make the upper index non-negative in  $\binom{d-k}{d-i}$ , as follows.

$$\binom{k}{i}\binom{d-k}{d-i} = (-1)^{d-i}\binom{k}{i}\binom{k-i-1}{d-i}.$$

Expanding the binomial coefficients into factorials and simplifying, we have

$$\begin{split} (-1)^{d-i} \binom{k}{i} \binom{k-i-1}{d-i} &= (-1)^{d-i} \frac{k!}{i!(k-i)!} \cdot \frac{(k-i-1)!}{(d-i)!(k-d-1)!} \\ &= \frac{(-1)^{d-i}}{k-i} \cdot \frac{k!}{i!(d-i)!(k-d-1)!} \\ &= \frac{(-1)^{d-i}}{k-i} \cdot \frac{1}{d!} \cdot \binom{d}{i} \cdot (k-d) \cdot d! \cdot \binom{k}{d} \\ &= \frac{(-1)^{d-i}}{k-i} \cdot \binom{d}{i} \cdot (k-d) \cdot \binom{k}{d} \\ &= (-1)^d \cdot \frac{1}{k-i} \cdot (-1)^i \binom{d}{i} \cdot (k-d) \binom{k}{d} , \end{split}$$

from which Equation (2) follows.

**Remark A.3.** The equality

$$(k-d)\binom{k}{d} = k\binom{k-1}{d}$$

offers a way to compute  $\binom{k}{d}$  in linear time for k ranging over an interval.

## **B** The MDOC standard

MDOCs are specified in ISO 18013-5. Here we summarize the relevant portion that concerns a binary string which can be parsed into a hierarchical MSO[] object of the form:

```
{"version": "1.0",
"digestAlgorithm": "SHA-256",
"docType": "org.iso.18013.5.1.mDL",
"valueDigests": {
   "org.iso.18013.5.1":
       {13: h'B628...69D2',
       11: h'6F94...5BD4',
       . . .
14: h'8CFE...604C',
 . . .
   "org.iso.18013.5.1.aamva": {
       15: h'1034...402A',
      . . .
       8: h'7AC6...8485'}
   },
"deviceKeyInfo": {
       "deviceKey": {1: 2, -1: 1,
```

```
-2: h'7B8F...F321',

-3: h'859E...772C'}

},

"validityInfo": {

    "signed": 0("2023-10-11T13:18:15Z"),

    "validFrom": 0("2023-10-11T13:18:15Z"),

    "validUntil": 0("2023-11-10T13:18:15Z")

}
```

In our context, an mdoc credential consists of a binary serialization of this MSO object that is signed by an IdentityIssuer (II), whose public signing key  $PK_II$  is well-known; denote the signature  $(r_1, s_1)$ .

Consider the string MSO[valueDigests] [org.iso.18013.5.1] [14] which represents the SHA-256 hash of a tuple consisting of the strings abbreviated as (*id*, *nonce*, *attrid*, *attrvalue*):

```
"digestID": 14,
"random": h'8c082af3d5d78b98bcc00bd26fae5130',
"elementIdentifier": "age_over_18",
"elementValue": true
```

A user can selectively disclose the age\_over\_18 attribute id in the mdoc credential by providing a signature of the MSO, as well as a full preimage (consisting of those 4 strings) to the given ....[14] hash included in the MSO. This essentially asserts that the property "age\_over\_18" was included in the bundle that the IdentityIssuer signed, while hiding the other attributes that were signed. What prevents users from breaking soundness by copying credentials?

Notice that MSO[deviceKeyInfo][deviceKey][-2] and [-3] denote a public signing key  $PK_d$ , which corresponds to a secret key that is maintained in the secure element of the user's device.

When a user attempts to show an mdoc, in addition to  $(r_1, s_1)$  and the preimages of the attributes, the user's device also computes a signature  $(r_2, s_2)$  on a transcript of the identity request under the  $PK_d$  signing key. Assuming that when the mdoc was issued, the IdentityIssuer verified that the  $SK_d$  was securely stored in the device, then this second signature establishes device-bound soundness.