Xiezhi: Toward Succinct Proofs of Solvency

Youwei Deng

A Thesis in
The Concordia Institute for Information Systems Engineering (CIISE)

Presented in Partial Fulfillment of the Requirements
For the Degree of
Master of Applied Science
(Information and Systems Security)
at
Concordia University
Montréal, Québec, Canada

September 2024

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Youwei Deng**

Entitled: **Xiezhi: Toward Succinct Proofs of Solvency**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information and Systems Security)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Walter Lucia

_____ External Examiner
Kaiwen Zhang (ETS)

_____ Examiner
Amr Youssef

_____ Examiner
M. Mannan

_____ Examiner
Carol Fung

_____ Supervisor
Jeremy Clark

Approved by _____
Zachary Patterson, Graduate Program Director (CIISE)

01 Sept 2023 _____
Mourad Debbabi, Dean (GCS)

Abstract

Name: **Youwei Deng**

Title: **Xiezhi: Toward Succinct Proofs of Solvency**

Hello. No more than 250 words.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Diam donec adipiscing tristique risus nec feugiat in fermentum posuere. Et netus et malesuada fames ac turpis. Nullam non nisi est sit. Felis eget velit aliquet sagittis id. Mauris commodo quis imperdiet massa tincidunt. Tellus molestie nunc non blandit massa enim nec. Facilisis mauris sit amet massa. Et molestie ac feugiat sed. Metus vulputate eu scelerisque felis imperdiet proin.

Acknowledgments

Hello.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Diam donec adipiscing tristique risus nec feugiat in fermentum posuere. Et netus et malesuada fames ac turpis. Nullam non nisi est sit. Felis eget velit aliquet sagittis id. Mauris commodo quis imperdiet massa tincidunt. Tellus molestie nunc non blandit massa enim nec. Facilisis mauris sit amet massa. Et molestie ac feugiat sed. Metus vulputate eu scelerisque felis imperdiet proin.

Contents

List of Figures

List of Tables

0.1 Introduction

0.1.1 Motivation

When the Bitcoin exchange Mt. Gox was declared bankrupt in 2014, a curious fact was reported in the *New York Times*—the missing 744K BTC "had gone unnoticed for years." This led the Bitcoin community to propose that exchanges undergo regular financial audits, with proposals ranging from traditional audits conducted by specialists to completely disintermediated "crypto-audits" done directly by the exchange to its users using cryptography. Academics quickly showed these can be done in strict zero-knowledge [**provisions**], and generated a stream of research papers that continues to improve efficiency [**bulletproofs**; **zeroledge**; **dapol**; **spp**; **notus**; **izpr**] and examine the correctness of deployed proofs [**broken**]. Despite these efforts, exchanges are not legally required to use a proof of solvency in jurisdictions today, with some exchanges opting to do them anyways and many not. Meanwhile, many other exchanges have failed in similar ways to Mt. Gox (whether by incompetence or fraud), including higher profile cases like QuadrigaCX and FTX. We argue that proofs of solvency are not perfect but do provide meaningful barriers (or friction) to fraud and incompetence. As academics, we believe we should continue refining these proofs toward practical implementation.

0.1.2 Contributions

A proof of solvency (or proof of reserves) is a zero-knowledge proof conducted by centralized cryptocurrency exchange (or more generally, any custodian of cryptocurrencies) to offer evidence that the exchange owns enough cryptocurrency to settle each of its users balances. The zero-knowledge component protects the exchange's proprietary information such as: number of users, balances of individual users, total balance of all users, which cryptocurrency addresses belong to the exchange, and total amount of cryptocurrency owned by the exchange. The proof itself is broken into sub-components: (π_{keys}) a proof of knowledge of private signing keys associated with public cryptocurrency addresses (hidden in a freely-composable anonymity set of addresses not belonging to the exchange); (π_{assets}) a summation of these assets into the total assets; (π_{user}) an individualized proof given to each user asserting their balance as used in the overall proof; ($\pi_{\mathsf{liabilities}}$) a summation of these individual liabilities into the total liabilities; and (π_{solvency}) a demonstration that the subtraction of total liabilities from the total assets is at least 0.

In our paper, we examine the extent to which these sub-components can be made succinct. In particular, we are interested in constant-sized arguments[1] and constant-time verification. This is possible for general arithmetic circuits in the polynomial interactive oracle proof (Poly-IOP) model using protocols like Plonk and its variants. In reference to the "towards" in the title of our paper, we are not able to make each sub-

[1]We abuse terminology and generally do not distinguish between 'proofs' and 'arguments,' using the term 'proofs' for both. Proofs provide soundness against unbounded malicious provers, while arguments provide zero knowledge against unbounded malicious verifiers. Xiezhi is a hybrid.

component fully succinct, however we make progress as follows: (π_{assets},π_{user},π_{solvency}) are constant in size and time (once all public inputs have been interpolated into polynomials by the verifier); (π_{keys}) is linear in the number of addresses in the anonymity set (but is pre-computation that can be re-used when proofs are issued each day); ($\pi_{\mathsf{liabilities}}$) is linear in the number of bits used to represent each account balance (*e.g.,* 32 bits) and is independent of the number of users (technically there is an upper-bound, but it is beyond the reasonable size of the largest exchange).

Our contributions can be summarized as:

- Xiezhi:[2] A mostly succinct protocol that covers every step of the proof, where each sub-component of the proof works with each other sub-component.
- A novel technique for mapping knowledge of private keys of common blockchains, such as Bitcoin and Ethereum, from their group (`secp256k1`) into a pairing-friendly group (`bls12-381`) used for succinct arguments.
- Practical adjustments to the protocol to account for concrete parameters, such as the maximum root of unity in `bls12-381`.
- Proof of concept implementation of Xiezhi with performance experimentation.

0.1.3 Limitations

We do not argue that a proof of solvency is a silver bullet that prevents all fraud and insolvency. However we do believe it can meaningfully raise the bar for incompetence

[2]Folklore creature revered in ancient Chinese culture for its ability to distinguish truth from deceit.

and fraud within an exchange, while providing a trail of records useful during a financial audit or enforcement action. Limitations of Xiezhi include:

- Our protocol relies on a trusted setup. However the setup is universal (shareable with other zk-SNARK systems) and is secure with one honest participant in a decentralized computation of it [**tau**].

- π_{keys} assumes the public key (as opposed to only its hash) associated with every address in an anonymity set of keys is known. In Bitcoin and Ethereum, this (typically) corresponds to the address having originated at least one transaction.

- π_{keys} assumes funds are controlled by a single public key. We do not explore how to handle multisig, wallets, Gnosis Safe, *etc.* However we can support tokens (*e.g.,* ERC20, ERC721,*etc.*) given the mapping between balances and public keys is on-chain, and we can support such assets split across layer two solutions or EVM-chains.

Limitations of proofs of solvency in general (not specific to Xiezhi) include:

- Proofs of solvency rely on human behaviour. It needs to be unpredictable to the exchange which users will check and report inconsistencies.

- Proofs of solvency are a detection mechanism, not a prevention mechanism. Proofs do not help prevent hacks or exit scams. However they greatly complicate cover-ups by the exchange after fraud or a loss of funds has occured.

- Collusion between parties to pool assets can enable an insolvent exchange to pass a proof of solvency. However the exchange is forced in this case to proactively seek and implement collusion, which raises the chances of discovery. It also goes to motive for fraud (many bankrupt exchanges argue they were just incompetent and it is difficult to distinguish).

- Trusted execution environments (TEEs) can reduce trust amongst colluders by sharing the ability to prove ownership without sharing the key itself. However we can adapt π_{keys} (the Σ-protocol inside it) for complete knowledge [**completeknowledge**].

- Demonstrably holding cryptocurrencies in an address does not mean the money is unencumbered. For example, it might be in use off-chain as collateral for a loan.

0.2 Preliminaries

0.2.1 Accounting Terminology

Our terminology follows the accounting and auditing literature. A balance sheet consists of liabilities (value owed to others) and assets (value owned). When total asset value is the same or more than total liabilities, the firm is called solvent. The amount by which the assets exceed the liabilities is called capital or equity (depending on context). Some literature prefers the term 'proof of reserves' to 'proof of solvency.' This terminology is also sound, although it is not always defined correctly: technically

a reserve is an asset that cancels out a corresponding liability, both in amount and in type. A firm can be solvent but not have full reserves. Most banks operate this way with cash liabilities, some assets in cash, but most assets in loans and other investments.[3] Cryptographic protocols in the literature generally assume both liabilities and assets are the same cryptocurrency so exchanges are expected to be both solvent and have full reserves.

0.2.2 Related Work

Proofs of solvency began as a discussion on Bitcointalk, where a protocol was developed for an exchange to announce a commitment to a total liability, and offer a Merkle-tree proof to each user that their balance was reflected in this total liability amount. Provisions, a research paper, used homomorphic commitments and Σ-protocols to add zero-knowledge, plus it added a proof of assets that could be used with the proof of liabilities to prove overall solvency [**provisions**]. As Provisions relies heavily on range proofs for liabilities, Bulletproofs can reduce the proof size of Provisions by 300x [**bulletproofs**]. Our protocol uses a polynomial-based range proof [**rangeproof**] to further reduce proof size and verifier time.

Outside of Provisions, Bulletproofs, and Xiezhi, the vast majority of work on proofs

[3]Banks in the United States have reserve requirements while banks elsewhere, *e.g.,* Canada, have capital requirements. Capital requirements speak to solvency, while reserve requirements speak to reserve ratios. Banks that underwent the 2008 financial crisis were more robust when they had capital requirements, as opposed to reserve requirements.

[4]V. Buterin, "Having a safe CEX: proof of solvency and beyond," vitalik.ca, 2022

[5]`https://summa.gitbook.io/summa`

[6]`https://www.proven.tools/`

[7]`https://minaprotocol.com/`

of solvency have not attempted an end-to-end proof, focusing instead on just the liabilities or just the assets. Why? We hypothesize that the biggest impediment is that Bitcoin and Ethereum assets are controlled by `secp256k1` private keys (see Table 1). Outside of Bulletproofs (based on inner-product arguments that do not require bilinear pairings and thus, can be implemented in `secp256k1`), most other approaches to succinctness require a specific cryptographic setting that is not `secp256k1` (*i.e.,* RSA for accumulators, pairing-based cryptography for zk-SNARKs, and lattices for zk-STARKs). If one only considers liabilities, then this problem does not have to be dealt with.

Circuit-based solutions are feasible but expensive for the prover—the authors of IZPR report about 500K constraints needed per key and proving times in the order of days for an anonymity set of 6000 keys [**izpr**]. By contrast, Xiezhi is a few minutes of work for the prover for 6000 keys. As this part is not-succinct (it is based on Σ-protocols), the trade-off is that the verifier has to do a few minutes of work as well. In both cases, IZPR and Xiezhi, this step does not need to be repeated often, only when the exchange wants to introduce new keys holding its assets. It is also important to recognize IZPR can let the exchange add keys it has not used yet to π_{keys}, further reducing how often this proof needs to be redone. This is a desirable property we are not able to easily achieve in Xiezhi (in short, it is due to our use of selector polynomials instead of lookup arguments but future work could explore blending the best properties of Xiezhi and IZPR).

0.2.3 Cryptographic Background

Xiezhi is mostly a zero-knowledge argument that represents vectors of data as univariate polynomials, which are committed to and provided to the verifier. Properties of the vectors are demonstrated by manipulating the polynomials and opening points of the polynomials—a model called a polynomial interactive oracle proof (Poly-IOP). However before encoding the data into polynomials, the exchange will prove ownership of signing keys with a Σ-protocol.

We assume our protocol works in a finite field of prime order. For simplicity, we use $\mathcal{P}$ and $\mathcal{V}$ to denote the prover and the verifier in an interactive proof system respectively. We use $g_{\mathfrak{g}}$ to denote a generator in a group $\mathbb{G}_{\mathfrak{g}}$ while $h_{\mathfrak{g}}$ is another generator, and no one knows the relative discrete logarithm of these two generators. We use g_s and h_s to denote the two generators in $\mathbb{G}_s$, the `secp256k1` group; and we use g_b and h_b to denote the two generators in $\mathbb{G}_b$, the `bls12-381` group. If we need to distinguish which group an input to the pairing is from, we use the notations $[x]_1 := g_1^x, [x]_2 := g_2^x$, otherwise elements are assumed to be in the first group (including g_b and h_b). We use $e([x]_1, [x]_2)$ to denote a non-degenerate bilinear pairing. We use $\mathcal{H}$ to denote a collision-resilient hash function modelled as a random oracle. We use p.p.t to represent probabilistic algorithms run in polynomial time. For vectors, we use an overhead bar, like $\overline{v}$, to denote a vector and brackets to denote the elements in this vector, e.g., $\overline{v} = \langle v_1, v_2, \ldots \rangle$. We also use a vector at the top right of a variable to indicate this variable belongs to this vector for readability, e.g., $x^{(\overline{v})}$ means x is an element of $\overline{v}$. We may use these two denotations interchangeably.

When we say the summation between two vectors $\overline{a}, \overline{b}$, the result is a new vector $\overline{c}$ where each element is the summation of the elements from $\overline{a}$ and $\overline{b}$ at the same row.

We refer the reader to the appendix for the following cryptographic primitives: discrete logarithm assumption (Section .1.1), Pedersen commitments (Section .1.2), Σ-protocols (conjunction and disjunction) (Section .1.3), KZG polynomial commitments (Section .1.5), and roots of unity (Section .1.7). We review two additional primitives here: polynomial-based range proofs and batched KZG commitments with zero-knowledge extension.

Range proof

A range proof enables $\mathcal{P}$ to convince $\mathcal{V}$ a value x is in the range $[0, 2^k)$ without revealing x. Zero-knowledge range proofs (ZKRPs) have three typical approaches: square decomposition, n-ary decomposition, and hash chains [**zkrp**]. We use the polynomial-based range proof from Boneh *et al.* [**rangeproof**].

1. *Given input x, decompose x to a vector of binary digits $\overline{z} = \langle z_1, z_2, \ldots, z_k \rangle$, so that $x = \sum_{i=0}^{k-1} 2^i \cdot z_i$*

2. *Construct a vector $\overline{x} = \langle x_1, x_2, \ldots, x_k \rangle$ such that*

$$x_1 = x$$

$$x_k = z_k$$

$$x_i = 2x_{i+1} + z_i, i \in [1, k-1]$$

3. *Interpolate a polynomial f from $\overline{x}$ over a finite field H of order n with elements $\omega^0, \omega^1, \omega^2, \ldots$ (see Appendix .1.7)*

4. *Prove the following polynomials are vanishing in H*

$$w_1 := [f(X) - x] \cdot \frac{X^n - 1}{X - \omega^0}$$
$$w_2 := f(X) \cdot [f(X) - 1] \cdot \frac{X^n - 1}{X - \omega^{n-1}}$$
$$w_3 := [f(X) - 2 \cdot f(X\omega)] \cdot [f(X) - 2 \cdot f(X\omega) - 1] \cdot (X - \omega^{n-1})$$

   (a) $\mathcal{P}$ *sends the commitment to* $f(X)$

   (b) $\mathcal{V}$ *sends a random challenge* γ

   (c) $\mathcal{P}$ *sends the commitment to* $q(X) = w/(X^n - 1)$, *such that*

$$w = w_1 + \gamma \cdot w_2 + \gamma^2 \cdot w_3$$

   (d) $\mathcal{V}$ *sends a random evaluation point* $\zeta \in \mathbb{F}$

   (e) $\mathcal{P}$ *replies with* $f(\zeta), f(\zeta\omega), q(\zeta)$

   (f) $\mathcal{V}$ *checks*

      i. $w(\zeta) = q(\zeta) \cdot (\zeta^n - 1)$

      ii. $f(\zeta), f(\zeta\omega), q(\zeta)$ *are the correct evaluations*

Batched opening with zero-knowledge extension ($\mathsf{KZG.open}_{zk}$)

To efficiently prove several polynomials are vanishing at several points, there are some batched KZG opening schemes such as the schemes in [**plonk**; **bdfg**; **fflonk**]. Although a PCS does not reveal the committed polynomial directly, it leaks the information of the opening evaluation point. In our scenario, each evaluation is privacy-sensitive so we want to hide the polynomial evaluations. We use the opening scheme from [**plonk**] and the protocol from [**rangeproof**] with a zero-knowledge extension to explain how to prove the range-proof polynomials are vanishing efficiently and in zero-knowledge.

Assume $\mathcal{P}$ is given x and $\mathcal{V}$ is given $[x]_1$, $\mathcal{P}$ wants to prove x is in $[0, 2^k)$:

1. *$\mathcal{P}$ interpolates the polynomial f using the above range proof*
2. *$\mathcal{P}$ generates two random numbers ω', ω'' in $\mathbb{F} \setminus H$ and another two random numbers α, β in $\mathbb{F}$*
3. *$\mathcal{P}$ interpolates f' at two more points ω', ω'' such that*

$$f'(\omega') = \alpha$$

$$f'(\omega'') = \beta$$

4. *$\mathcal{P}$ computes w_1, w_2, and w_3 in the above range proof and sends the commitment to f', $\mathcal{C}_{f'}$*

5. $\mathcal{V}$ *sends a random challenge* $\gamma \in \mathbb{F}$

6. $\mathcal{P}$ *sends the commitment to* $q_w := w/(X^n - 1)$ *where*

$$w := w_1 + \gamma \cdot w_2 + \gamma^2 \cdot w_3$$

7. $\mathcal{V}$ *sends a random evaluation point* $\zeta \in \mathbb{F} \setminus H$

8. $\mathcal{P}$ *sends the evaluations* $f(\zeta), f(\zeta\omega), q_w(\zeta)$

9. $\mathcal{P}$ *sends the commitments to* $q_1(X), q_2(X)$*, where*

$$q_1(X) := \frac{f(X) - f(\zeta)}{X - \zeta} + \gamma \cdot \frac{q_w(X) - q_w(\zeta)}{X - \zeta}$$

$$q_2(X) := \frac{f(X) - f(\zeta\omega)}{X - \zeta\omega}$$

10. $\mathcal{V}$ *chooses random* $r \in \mathbb{F}$

11. $\mathcal{V}$ *accepts the proof if and only if*

    (a) $w_1(\zeta) + \gamma \cdot w_2(\zeta) + \gamma^2 \cdot w_3(\zeta) = q_w(\zeta) \cdot (\zeta^n - 1)$

    (b) $e(F + \zeta \cdot \mathcal{C}_{q_1} + r\zeta \cdot \mathcal{C}_{q_2}, [1]_2) = e(\mathcal{C}_{q_1} + r \cdot \mathcal{C}_{q_2}, [x]_2)$*, where*

$$F := \mathcal{C}_f - [f(\zeta)]_1 + \gamma \cdot (\mathcal{C}_{q_w} - [q_w(\zeta)]_1)da + r \cdot (\mathcal{C}_f - [f(\zeta\omega)]_1)$$

We prove the protocol has zero knowledge.

Let $\mathcal{S}$ generate $\{\omega_i', \omega_i'', \alpha_i, \beta_i\}, i \in [0, m]$ and interpolate $\{f_i\}$ such that

$$f_i(\omega_i') = \alpha_i$$

$$f_i(\omega_i'') = \beta_i$$

$$f_i(\omega^i) = 0$$

where $i \in [0, m]$ and ω is the root of unity in H. Note $\mathcal{V}$ can interact with $\mathcal{S}$ to execute the protocol and accept the proof from $\mathcal{S}$. Given $\{\omega_i', \omega_i'', \alpha_i, \beta_i\}_{i \in [0,m]}$ are chosen uniformly at random each time, $\mathcal{V}$ cannot distinguish between the transcript from $\mathcal{S}$ and the proof from $\mathcal{P}$. Therefore, the protocol has zero knowledge. We will use $\mathsf{KZG.open}_{zk}$ to denote this opening technique.

0.3 Proof of Assets (PoA)

We will begin on the asset-side of the proof of solvency. To ease the writing, we will take the example of an exchange proving solvency for ETH on Ethereum. The PoA is broken into two steps: π_{keys} and π_{assets}. π_{keys} takes the following public (exchange-chosen) input: a set of Ethereum public keys which consists of its own public keys hidden amongst a larger set of keys belonging to others (*i.e.,* an anonymity set which can scale to the full set of public keys on Ethereum); private input: set of private signing keys (in `secp256k1`); and outputs 'the keys that belong to it.' The exact format of the output can vary; in our case, the input keys are indexed and the output

is a binary 'selector' vector (in `bls12-381`) that records a 0 if the exchange is not claiming to know the secret key and a 1 if the exchange can prove knowledge of the secret key. The main alternative format would be a flat list of the known public keys which can serve as a set for set-membership proofs, which can be performed succinctly with lookup arguments (see IZPR for this approach [**izpr**] where it is called 'bootstrapping'). Presently, π_{keys} with such an output format is only known to be realizable by general zk-SNARK circuits, whereas we provide direct proof technique for a selector vector output format.

Once the π_{keys} output is provided, π_{assets} takes it as input, along with the current balance of ETH in each address in the anonymity set (publicly known on Ethereum's blockchain). The output is a commitment to the total assets across keys and an argument the total is computed correctly.

0.3.1 The π_{keys} proof

Before presenting our π_{keys} proof, we quickly discuss a few approaches that helped us develop it. A highly relevant Σ-protocol from the literature, COPZ, proves that two commitments in two different groups (*e.g.,* `secp256k1` and `bls12-381`) commit to the same value [**chase22**]. The paper cites proof of assets as a use-case but does not work out a protocol. COPZ allows an exchange to 'map' private keys from `secp256k1` to `bls12-381`. Two places this mapping could occur would be at the very start or the very end of the assets proof. At the end, it might look like this: an existing proof of assets protocol (*e.g.,* Provisions [**provisions**]) could be run to create

a commitment to the total assets in `secp256k1`, then COPZ can be used to prove the same commitment in `bls12-381`, and finally this can be 'glued' to a succinct proof of liabilities in `bls12-381`. However this does not leverage the fact that `bls12-381` might help make the assets proof succinct.

Alternatively, the exchange can map all their keys at the start. There is a roadblock: the exchange can only map keys they know the secret key for and the exchange cannot reveal which keys they know and which they do not. Assume there is a protocol that would allow the exchange to output a sparse vector of `bls12-381` private key values (sorted by index of known ETH public keys) containing the key value when they know it, and recording a 0 if they do not. We designed such a protocol only to realize that the key values in `bls12-381` are actually never used, we only use the fact that knowledge of them is proven (which is covered by the ability to produce the value in `bls12-381`) and the fact that unclaimed keys are zeroed-out.

This leads to our key observation: we do not need to map *values* from `secp256k1` to `bls12-381`, we just have to map the *success or failure* of a ZKP in `secp256k1` to `bls12-381`. This can be accomplished by composing (conjunction and disjunction of) Σ-protocols. The prover (exchange) will choose set of Ethereum public keys as its anonymity set of size κ (containing its actual keys) $\{\mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_\kappa\}$ and publish them in an ordered (indexed) way. It will also create a binary 'selector' array A_{keys} with a 1 in the same index of every key it is claiming to know and a 0 in the index of the keys it does not know (or does not want to claim for whatever reason). This vector is interpolated into the evaluation points of a polynomial $P_{\mathsf{keys}}(X)$

and committed to $C_{\mathsf{keys}} = \mathbf{C}_{\mathtt{bls}}(P_{\mathsf{keys}}(X))$ using the KZG polynomial commitment scheme [**kzg**]. For each index i, the prover shows the evaluation of $P_{\mathsf{keys}}(X_i)$ but instead of providing the evaluation value ($A_{\mathsf{keys},i}$) in plaintext, it provides a Pedersen commitment to it $\mathbf{C}_{\mathtt{bls}}(A_{\mathsf{keys},i})$ (a mild modification of the KZG showing protocol detailed in Section .1.6). It then shows the value is correct with the following Σ-protocol:

$$\text{ZKPoK}\{(sk_i, A_{\mathsf{keys},i}) : \mathbf{C}_{\mathtt{bls}}(A_{\mathsf{keys},i}) = \mathbf{C}_{\mathtt{bls}}(0) \vee [\mathbf{C}_{\mathtt{bls}}(A_{\mathsf{keys},i}) = \mathbf{C}_{\mathtt{bls}}(1) \wedge \mathsf{pk}_i = g^{\mathsf{sk}_i}]\}$$

In plain English, the prover either: (1) puts a 0 in the selector vector; or (2a) puts a 1 in the selection vector *and* (2b) knows the private key of the given public key. (1) and (2b) are a PoK of a representation for Pedersen commitments in `bls12-381` while (2b) is a Schnorr PoK of a discrete logarithm in `secp256k1`—both well studied Σ-protocols [**damgard10**; **sigma**]. The fact that (1) and (2b) are in `bls12-381` while (2c) is in `secp256k1` is not problematic because the disjunction ($\vee$) and conjunction ($\wedge$) operations for composing Σ-protocols are based only on how challenge values are constructed and both groups (`secp256k1` and `bls12-381`) can encode a large t-bit challenge (*e.g.*, $t = 254$) into their exponent groups.

As this protocol is repeated for each key, it is not succinct and will be $O(\kappa)$ in proof size and verifier time. However once the selector array is proven correct, the exchange can re-use it every time it does a proof of solvency until it updates its keys.

The full details are provided in Protocol 1.

0.3.2 The π_{assets} argument

The π_{keys} protocol proves that $\mathcal{C}_\phi$ is a commitment to a selector polynomial $\phi(X)$ (in `bls12-381`) which marks the public keys owned by the exchange. At a given time (block number), the balances of every public key included in the anonymity set will be encoded in polynomial $\delta(X)$. The product of $\phi(X) \cdot \delta(X)$ will preserve the balance values owned by the exchange and zero-out the balance values not claimed by the exchange. The final step is produce a summation over the values in $\phi(X) \cdot \delta(X)$. The exchange will put $\phi(X) \cdot \delta(X)$ in accumulator form $\sigma(X)$ and prove its correctness. In this from, the total assets will sit at the head (first index) of $\sigma(X)$, which is $\sigma(\omega^0)$. The full details are provided in Protocol 2.

0.4 Proof of Liabilities

0.4.1 The $\pi_{\mathsf{liabilities}}$ argument

The exchange will commit to every user balance and produce a commitment of the total amount across all balances. Since the exchange is free to make-up additional users and include them, we want to make sure this does not help an insolvent exchange in any way. To do this, we force all balances to be zero or positive numbers. For a finite field, this means small integers that have no chance of exceeding the group order (modular reduction) when added together. In practice, we can limit ourselves to an

even smaller range that is sufficient to capture what a balance in ETH (or fractions of ETH) might look like. These balances are expressed in binary and we use range proof from Section 0.2.3.

However when we turn to implement this in practice, we encounter a roadblock. If μ balances are placed as k-bit numbers side-by-side in a vector, we need a vector of size $\mu \cdot k$. If we want to optimize polynomial interpolation, we encode our array at x-coordinates that correspond to the roots of unity of the exponent group (see Appendix .1.7) and for `bls12-381`, we can only efficiently encode data vectors of length up to $2^{32} = 4,294,967,296$.[8] Consider an exchange with $\mu = 1,000,000$ accounts, only 12 bits are left to capture account balances, say, as between 0.01 and 40.96 ETH ($30 to $150K USD at time of writing). Exchanges could have more than 1 million accounts, the largest could be more than $150K USD, and an exchange could have a long tale of accounts with balances less than $30 such that rounding them all up to $30 creates a solvency issue. Clearly $k = 2^{32}$ is not large enough for directly encoding liabilities (as binary numbers) into a single polynomial.

To deal with this issue, there are three main alternatives. (1) The exchange can encode points at arbitrary x-coordinates and use general (Laplacian) interpolation, (2) the exchange can break down what it is proving into chunks but this will require one succinct proof per chunk, or (3) the range proof could be adapted for decomposition into something larger than bits (*e.g.,* bytes or 32-bit words). The latter may be feasible with lookup arguments, but we do not pursue modifying the range

[8] The exponent group in `bls12-381` has 2-adicity of 32.

proof [**rangeproof**] in this work. Instead we opt for (2). Specifically we will produce a polynomial argument for the first bit of every account, for the second bit of every account, *etc.* This means $\pi_{\mathsf{liabilities}}$ will be linear in proof size and verifier work but it is linear in the bit-precision of each account (k) and is in fact constant (succinct) in terms of the number of users on the exchange. For example, we will later show if accounts are captured with a precision of 32-bits, the proof size will be under 10KB and verifier time will be under 8ms independent of the number of users on the exchange (see Figure 3).

The protocol creates k polynomials—the first polynomial p_1 for the first bit of each of the μ accounts, the second polynomial for the second bit of every account, *etc.* It conducts a range proof 'vertically' (across $\{p_1(i), p_2(i), \ldots, p_k(i)\}$ for $\mathfrak{bal}_i$) for each account (for all i). It then converts the bits into integers 'vertically' (across $p_0(i) = \sum_{j=0}^{k-1} 2^j \cdot p_j(i)$) for each account, creating a polynomial p_0 of each user's balance. Last it sums up all elements 'horizontally' ($\sum_{i=0}^{\mu-1} p_0(\omega^i)$) in p_0 to produce the total liabilities. The bit decomposition is argued with the range proof and the summation of balances is argued with a sum-check. The full protocol is given in Protocol 3.

0.4.2 The π_{users} argument

The π_{users} argument is conducted between the exchange and the user, so the user can check that their balance is correctly encoded into the polynomials used in $\pi_{\mathsf{liabilities}}$. If two users have the same balance, a malicious exchange might include only one of the

balances and open up the same balance for each user. Unless if the users compared their proofs, they would not catch the exchange (*cf.* [**broken**]). This attack appears in other cryptographic protocols where users need to check things, the main one being cryptographic voting schemes. It has been studied under general definitions as a 'clash attack' [**clash**]. The solution is to label each balance with a unique user identifier [**provisions**]. Labeling can be done with an additional polynomial of labels under the assumption that a user id and a balance needs to be at the same index. A user id can be the hash of the user's account name or email address. The full protocol is given in Protocol 4.

0.4.3 The π_{solvency} argument

The final step of the proof is prove the total assets exceed the total liabilities. At the end of π_{assets}, total assets contained in the polynomial evaluation point $\sigma(\omega^0)$; while at the end of $\pi_{\mathsf{liabilities}}$, the total liabilities are contained in $p_1(\omega^0)$. Assuming assets exceed liabilities by some amount, this amount can be added to the liability-side to provide a difference of exactly zero. The full argument is given in Protocol 5.

0.5 Security Analysis

We adapt the security definition of a zero-knowledge proof of solvency from Provisions [**provisions**]. In Appendix .2.1, we recount the definitions and offer a proof sketch for the the main theorem:

Xiezhi ($\pi_{\mathsf{solvency}} \leftarrow \langle \pi_{\mathsf{keys}}, \pi_{\mathsf{liabilities}}, \pi_{\mathsf{assets}}, \pi_{\mathsf{users}} \rangle$) is a privacy-preserving proof of solvency with respect to Definition .2.1.

0.5.1 Claims

A Σ-protocol for relation ZKPoK$\{(sk_i, s_i) : [\mathsf{pk}_i = g^{\mathsf{sk}_i} \wedge p_i = \mathbf{C}_{\mathtt{bls}}(1, r_i)] \vee p_i = \mathbf{C}_{\mathtt{bls}}(0, r_i)\}$ exists which is complete, has special soundness, and is honest verifier zero-knowledge (HVZK).

Proof. To demonstrate completeness, consult Protocol 1 (the inner framed protocol). To demonstrate special soundness, let two accepting conversations between $\mathcal{P}$ and $\mathcal{V}$

$$(t_1, t_2, t_3, e, e_1, e_2, z_1, z_2, z_3), (t_1, t_2, t_3, e', e'_1, e'_2, z'_1, z'_2, z'_3) \text{ with } e \neq e'$$

be given. It is obivious we can compute sk_i, s_i such that the relation ZKPoK(sk_i, s_i) exists. Thus the Σ-protocol for the relation ZKPoK has special soundness.
Honest verifier zero-knowledge (HVZK) is clear: given e, z_1, z_2, z_3 at random and choose e_1, e_2 such that $e = e_1 \oplus e_2$, we can make a simulated conversation between the honest verifier and prover using the faking proof tricks. Since e, z_1, z_2, z_3 can be chosen freely, the simuated conversation is not distinguishable from the real one.

Since the probability that s is equal to zero or one is exactly the same as the real $\mathcal{P}$ does, $\mathcal{V}$ cannot distinguish the proof produced by $\mathcal{S}$ from the one generated by $\mathcal{P}$. □

A Σ-protocol for relation ZKPoK$\{(sk_i, s_i) : [\mathsf{pk}_i = g^{\mathsf{sk}_i} \wedge p_i = \mathbf{C}_{\mathtt{bls}}(1, r_i)] \vee p_i = \mathbf{C}_{\mathtt{bls}}(0, r_i)\}$ exists which is a non-interactive zero knowledge proof (NIZKP).

Proof. Given the relation can proven with a "standard" Σ-protocol (per Thereom 0.5.1), we can use the well-known Fiat-Shamir heuristic to compile it to a NIZKP in the random oracle model. We do not repeat the proof for this (see [**damgard10**; **sigma**]) but stress that strong Fiat-Shamir [**weakfs**] needs to be used here and in the Poly-IOP components of Xiezhi, or practical attacks could be leveraged against the system (*cf.* [**weakfsattacks**]). □

0.6 Performance Evaluation

0.6.1 Theoretical Performance

In this section, we anaylze the performance of Xiezhi, and compare the performance of our work with other prior schemes. Our analysis ignores the relatively trivial cost like Fast Fourier Transform (FFT) and focuses on the heavy work such as multi-scalar multiplication (MSM) and group operations. Our analysis also ignores the differences of the implementations and assumes each protocol is executed in a single thread.

Proof of Assets

We use κ to denote the size of the anonymity set and we assume κ is the power of two for simplicity. The performance analysis of π_{keys} and π_{users} are performed as follows:

- π_{keys}: For each public key in the anonymity set, $\mathcal{P}$ opens an evaluation of the KZG commitment and generates the corresponding proof of the Σ-protocol in Protocol 1. When opening an evaluation of a KZG commitment, one MSM for the witness and one MSM for the blinding polynomial are involved. The number of scalar multiplications of the Σ-protocol is constant. Thus, the overhead proving time of π_{keys} is $O(\kappa^2)$. In terms of verifier's work, $\mathcal{V}$ does not need to perform MSM to verify the proofs; $\mathcal{V}$ computes scalar multiplications for constant times instead. To verify the committed values are correct, $\mathcal{V}$ manipulates the batched KZG scheme rather than verifying each proof one by one, which means $O(1)$ verifying time and proof size for each public key. Therefore, the overhead verifying time and proof size of π_{keys} is $O(\kappa)$.

- π_{assets}: $\mathcal{P}$ constructs the accumulator and commits to constant number of polynomials. According to Schwartz-Zippel lemma, $\mathcal{P}$ only needs to open one point of each polynomial. Thus, the proving time is $O(\kappa)$ and the proof size is $O(1)$ because the number of polynomial commitments is constant. While it takes constant time for $\mathcal{V}$ to verify the proof of the PCS, $\mathcal{V}$ needs to process the inputs, i.e., interpolates the balances and commits to the balance polynomial, which means one MSM involved. Therefore, the overhead verifier's work of π_{assets} is $O(\kappa)$.

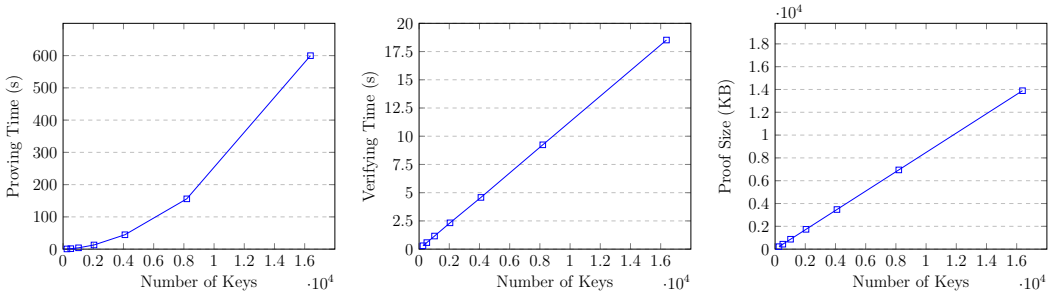
Proof of Liability

We use μ to denote the number of users and k to denote the allowed size of the range proof. Recall that $\mathcal{P}$ needs to construct several polynomials for the range proof. Although the number of the range-proof polynomials is arbitrary, 2^{64} is sufficient for almost all the exchanges' requirements in the real world, which means there are less than or equal to 64 polynomials for the range proof. Therefore, we can treat the number of range-proof polynomials as a constant, but we still use k to indicate the performance is related to the range proof. $\mathcal{P}$ also needs to compute the accumulative polynomial to prove the total liablity is correct, which can be done in linear time. Different from π_{keys}, $\mathcal{P}$ only opens each polynomial at one random evaluation point. Thus, the proving time is $O(\mu)$.

The verifier's work is broken into π_{users} and $\pi_{\mathsf{liabilities}}$:

- π_{users}: Each user verifies his balance is the evaluation of the polynomial p_0 and his user identifier is the evaluation of the polynomial u, and the two evaluation points should be the same. The user checks two proofs of KZG commitment, so the proof size and the verifying time for customers are both $O(1)$.

- $\pi_{\mathsf{liabilities}}$: Auditor verifies the constraints among polynomials $\{p_i\}$ are correct and the committed total liablity is the evaluation of aggregated p_0 at ω^0. The verification of constraints requires two steps: 1. validating each opening evaluation is correct; 2. the evaluations of $\{w_1, w_2, w_3, v_1, \ldots, v_{m-1}\}$ are zero. The first step can be done in constant time because the number of polynomials is



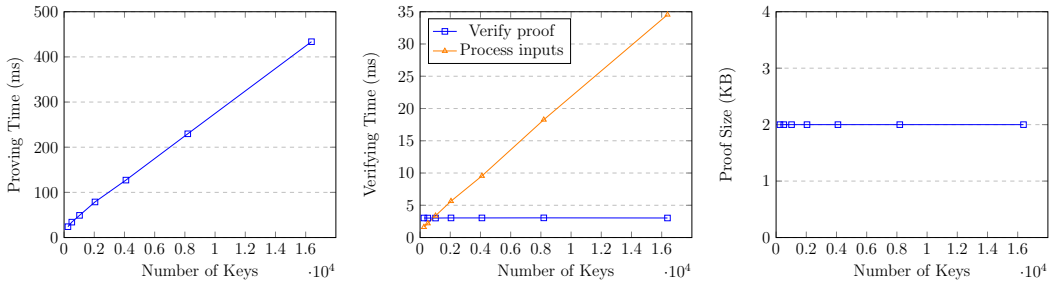
(a) (b) (c)

Figure 1: Performance of π_{keys}. Subfigure (a) illustrates the number of keys and the proving time; Subfigure (b) and (c) indicate the verifying time and the proof size are linear in the number of keys.

related to the range proof rather than the number of users. The second step takes several scalar multiplications and group additions but still in constant times. Recall that the proof of KZG commitment consists of one witness, one evaluation, and the corresponding evaluation point, which are unrelated to the degree of the polynomial. That means the verifying time and the proof size for auditors are both $O(k)$.

Comparison

In Table 2, we compare this work with other prior PoA schemes. Both IZPR and this work utilize bootstrapping, but the bootstrapping of IZPR will be introduced in their following paper. We only analyze the performance of the bootstrapping for this work. In Table 3, we compare this work with prior PoL schemes.



(a) (b) (c)

Figure 2: Performance of π_{assets}. Subfigure (a) and (b) suggest the proving time and the verifying time are linear in the number of keys; Subfigure (c) shows the proof size is constant, 2KB, based on our implementation.

0.6.2 Implementation and Benchmark Methodology

To evaluate the performance of Xiezhi, we implemented our protocols in Rust based on the popular library, arkworks[9]. Our implementation is publicly accessible on GitHub[10]. We chose the pairing-friendly elliptic curve `bls12-381` for the KZG commitment which has 128 bit security.

Our experiments were conducted on a personal computer with i9-13900KF and 32GB of memory. The experimental data including balances and `secp256k1` key pairs are randomly generated locally for simplicity. Since there is no range-proof for PoA, we tested the PoA with balances randomly distributed in $[1, 2^{64})$ to simulate the real distrbution of assets, and for PoL, we tested the program with balances randomly distributed in $[1, 2^8)$, $[1, 2^{16})$, $[1, 2^{32})$, and $[1, 2^{64})$. We simulated $2^8, 2^9 \ldots, 2^{14}$ and $2^{10}, 2^{11}, \ldots, 2^{20}$ users for PoA and PoL respectively. Simulating different number of users for PoA and PoL is because π_{keys} was time-consuming for larger number of users. For each protocol, we ran the test for ten times with the same experimental data. Our

[9] `https://github.com/arkworks-rs`

[10] GitHub: link removed for anonymity. Can supply code via the program chairs as necessary.

figures are interpolated from the average performance of ten times while discarding the maximum and the minimum samples.

0.6.3 Experimental Evaluation

Figure 2, 3, and 4 reflect the performance of Xiezhi in single thread with i9-13900KF. We discarded the samples of the maximum and the minimum to calculate the average for each point. The Subfigure (a) of Figure 2 suggests it takes around 600 seconds to generate the proofs for 16,384 keys with i9-13900KF for π_{keys}, and the proof size is 13,893KB. There are around 2^{28} unique Ethereum addresses reported by Ethercan.io in May 2024 [11], which means it requires 9,830,400 computing instances to generate the proofs for all the keys in 600 seconds if the exchange wants the maximum anonymous set. It seems impractical for the exchange to depoly such number of servers in the real world. However, recall the exchange only needs to perform π_{keys} once. The exchange can sacrifice the proving time to reduce the number of servers. Moreover, the proving time can be reduced significantly if manipulating more efficient KZG opening schemes. See Section 0.6.4 for more detailed optimizations.

Figure 3 shows the proving time and the verifying time are linear in the number of the keys. In our experiments, it takes 433.66 milliseconds to generate the proof and 37.57 milliseconds to verify the proof for 16,384 keys. This suggests the proving time is less than 2 hours if the anonymous set is the whole addresses on Ethereum without any other optimizations! Since the proof size of a KZG commitment is unrelated to

[11] `https://etherscan.io/chart/address`

the degree of the polynomial (the number of keys), the proof size of π_{assets} is 2KB based on our implementation.

Figure 4 illustrates the performance of our PoL with different number of users and allowed ranges for balance. Our experiments show the proving time grows linearly with the number of users while the verifying time and the proof size are constant. From the test result of Binance's PoL, it needs 1.5 days to generate the proof for 100 million accounts with 100 servers [12], but our PoL requires less than 10 minutes with the same number of servers! This indicates our protocol is practical to handle the real-world applications.

0.6.4 Optimization

Due to the additively homomorphic property of KZG commitment, the prover's work in our protocols can be easily splitted to arbitrary number of servers, and there is no need to provide extra proofs to aggregate the proofs from different servers (provers). That means the proving time will decrease with the growing number of servers. This is the most direct method to make the protocols more efficient. Another way to improve the performance without adding more servers is utilizing more efficient KZG opening schemes. Recall the heaviest work of π_{keys} is proving each committed point is correct, and the opening scheme we demonstrated from Plonk requires $t \cdot d$ scalar multiplications for prover, where t is the number of the opening points and d is the degree bound of the polynomial. The work in BDFG20 [**bdfg**] can reduce this

[12] `https://github.com/binance/zkmerkle-proof-of-solvency/?tab=readme-ov-file`

complexity to $2n$ scalar multiplications, which means the dominating complexity will become $O(n)$ rather than $O(n^2)$. The aggreation slightly increases the verifier's work but the extra cost is trivial because of the succinctness of KZG commitment scheme. These optimizations can be applied to both of our PoA and PoL. Moreover, the proof length for multiple points of the KZG commitment will also be decreased to $O(1)$ if BDFG20 is intergated, but the total proof length is still $O(n)$ because of the proof of the Σ-protocol.

0.7 Open Research Challenges

We presented Xiezhi as a Poly-IOP solvency argument. The efficiency and succinctness of Xiezhi might be further improved using advances in other poly-iop systems: lookup arguments and multivariate polynomials (and corresponding commitment schemes). Techniques from recursive SNARKs might enable solvency proofs that are repeated (say every day) to reduce prover time by ignoring ETH addresses and user balances that did not change across the day, while still allowing a verifier to have full confidence in the history of the exchange if they only check the most recent proof. If blockchains like Ethereum add low-gas cost support for `bls12-381`, a topic of discussion (EIP-2537[13]), verifying proofs of solvency could move on-chain. If an exchange fails to provide a smart contract with a proof of solvency in a timely fashion, the smart contract could be called to trigger penalties or other actions.

Standardization work could also be useful for proofs of solvency. Of the exchanges

[13] `https://eips.ethereum.org/EIPS/eip-2537`

that opt into providing proofs of solvency, the exact protocol can vary from other exchanges, and it is not always deployed correctly (at least initially) [**broken**]. Having the community coalesce behind a proof template, working out every detail, with audited and formally verified reference code could be helpful to exchanges. Xiezhi is a good start as it is a complete end-to-end proof system, and is competitive in prover time, verifier time and proof size with other sub-components introduced in the literature.

Scope. Blah blah blah.

Contributions. Our primary contributions are as follows.

1. Blah blah blah.

2. Blah blah blah.

3. Blah blah blah.

Σ-Protocols	The first proofs of solvency are based on Σ-Protocols which work on standard elliptic curves like `secp256k1` but are not succinct (linear proof space, linear verifier time, heavy constants).	Provisions [**provisions**]
Inner-product arguments	Protocols like bulletproofs work on standard elliptic curves like `secp256k1` and can reduce some sub-routines (e.g., range arguments) to constant space and logarithmic verifier time.	Bulletproofs [**bulletproofs**]
Liabilities only	As it is the asset-side of solvency that ties the protocol to standard elliptic curves like `secp256k1`, proving only the liability side can be done in any cryptographic setting.	ZeroLedge [**zeroledge**], DAPOL+ [**dapol**], SSVT-based [**spp**], Notus [**notus**], SafeCex[4]
Publish assets	A trivial proof of assets is one that is not zero-knowledge. An exchange could reveal all its addresses and prove ownership by signing a proof-specific message from each address.	Summa[5]
Circuit-level	A general zk-snark can implement any arithmetic circuit, including `secp256k1` operations, which offers a proof of constant size and constant verifier time.	IZPR [**izpr**], Proven.tools[6]
Custom blockchain	If new blockchains are deployed, they could use digital signatures over pairing-friendly curves.	Mina[7]
Mapping between groups	If `secp256k1` values can be mapped to a pairing-friendly group, Poly-IOP arguments can potentially reduce the rest of the proof to constant size and constant verifier time.	COPZ [**chase22**], Xiezhi

Table 1: How to deal with the fact that Bitcoin and Ethereum use `secp256k1` digital signatures when trying to make a succinct proof of solvency.

1. $\mathcal{P}$ publishes an anonymity set of public keys: $\langle \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{pk}_3, \ldots, \mathsf{pk}_n \rangle$.
2. $\mathcal{P}$ constructs a selector vector: $\overline{s} = \langle s_1, s_2, s_3, \ldots, s_n \rangle$ where $s_i = 0$ unless $\mathcal{P}$ can prove knowledge of sk_i given $\mathsf{pk}_i = g_s^{\mathsf{sk}_i}$, then $s_i = 1$.
3. $\mathcal{P}$ publishes individual commitments to $\overline{s}$: $\overline{p} = \langle \mathbf{C}_{\mathtt{bls}}(s_0), \mathbf{C}_{\mathtt{bls}}(s_1), \mathbf{C}_{\mathtt{bls}}(s_2), \ldots, \mathbf{C}_{\mathtt{bls}}(s_i) \rangle$ where $p_i = \mathbf{C}_{\mathtt{bls}}(s_i, r_i) = g_b^{s_i} h_b^{r_i}$. (To save a step, the value of r_i will be drawn from step 5(b)).
4. For each i, $\mathcal{P}$ and $\mathcal{V}$ run ZKPoK$\{(sk_i, s_i) : [\mathsf{pk}_i = g^{\mathsf{sk}_i} \wedge p_i = \mathbf{C}_{\mathtt{bls}}(1, r_i)] \vee p_i = \mathbf{C}_{\mathtt{bls}}(0, r_i)\}$ as follows:

   (a) Case 1: $s_i = 1$ ($\mathcal{P}$ claims knowledge of sk_i)
   - $\mathcal{P}$ selects $e_2 \xleftarrow{\$} \{0,1\}^t; z_3, \beta \xleftarrow{\$} \mathbb{Z}_b; \alpha \xleftarrow{\$} \mathbb{Z}_s$
   - $\mathcal{P}$ publishes $t_1 = g_s^{\alpha}$
   - $\mathcal{P}$ publishes $t_2 = h_b^{\beta}$
   - $\mathcal{P}$ publishes $t_3 = g_b^{-e_2} h_b^{z_3 - r_i e_2}$
   - $\mathcal{V}$ publishes t-bit challenge $e \xleftarrow{\$} \{0,1\}^t$ (or $\mathcal{P}$ via Fiat-Shamir)
   - $\mathcal{P}$ computes $e_1 = e \oplus e_2$ and publishes e_1 and e_2
   - $\mathcal{P}$ publishes $z_1 = e_1 \mathsf{sk}_i + \alpha$
   - $\mathcal{P}$ publishes $z_2 = e_1 r_i + \beta$
   - $\mathcal{P}$ publishes z_3

   (b) Case 2: $s_i = 0$ ($\mathcal{P}$ does not claim knowledge of sk_i)
   - $\mathcal{P}$ selects $e_1 \xleftarrow{\$} \{0,1\}^t; z_1 \xleftarrow{\$} \mathbb{Z}_s; z_2, \alpha \xleftarrow{\$} \mathbb{Z}_b$
   - $\mathcal{P}$ publishes $t_1 = g_s^{z_1} / \mathsf{pk}_i^{e_1}$
   - $\mathcal{P}$ publishes $t_2 = g_b^{e_1} h_b^{z_2 - r_i e_1}$
   - $\mathcal{P}$ publishes $t_3 = h_b^{\alpha}$
   - $\mathcal{V}$ publishes t-bit challenge $e \xleftarrow{\$} \{0,1\}^t$ (or $\mathcal{P}$ via Fiat-Shamir)
   - $\mathcal{P}$ computes $e_2 = e \oplus e_1$ and publishes e_1 and e_2
   - $\mathcal{P}$ publishes z_1
   - $\mathcal{P}$ publishes z_2
   - $\mathcal{P}$ publishes $z_3 = e_2 r_i + \alpha$

   (c) $\mathcal{V}$ checks
   - $e = e_1 \oplus e_2$
   - $g_s^{z_1} = \mathsf{pk}_i^{e_1} t_1$
   - $g_b^{e_1} h_b^{z_2} = p_i^{e_1} t_2$
   - $h_b^{z_3} = p_i^{e_2} t_3$

5. $\mathcal{P}$ interpolates a polynomial from $\overline{s}$, denoted by $\phi(X)$ and proves to $\mathcal{V}$ that it encodes the same s_i values that $\mathcal{V}$ just checked: $\phi(X_i) = s_i = \mathsf{Open}(p_i)$
   (a) $\mathcal{P}$ computes the KZG polynomial commitment to ϕ. Specifically, $\mathcal{P}$ constructs a random polynomial $\hat{\phi}(X)$ and publishes the polynomial commitment $\mathcal{C}_\phi$ such that $\mathcal{C}_\phi = g_b^{\phi(\tau)} h_b^{\hat{\phi}(\tau)}$
   (b) For each i, $\mathcal{P}$ opens the committed evaluations (see Section .1.6) of ϕ: $g_b^{\phi(i)} h_b^{\hat{\phi}(i)}$. This value matches $p_i = g_b^{s_i} h_b^{r_i}$ except the r_i term will be different. However by doing these steps (5a-b) before step 3 above, $\mathcal{P}$ can program $r_i = \hat{\phi}(i)$ exactly (otherwise a Σ-protocol can be used to show equivalence). $\mathcal{P}$ publishes the witness for each evaluation, $\{w_i\}$

1. $\mathcal{P}$ takes as input the selector vector of keys its shown ownership of, $\phi(X)$, from π_{keys}
2. $\mathcal{P}$ queries the balance of each public key in the set he chose, denoted by $\overline{b} = \langle \mathfrak{bal}_1, \mathfrak{bal}_2, \ldots, \mathfrak{bal}_n \rangle$
3. $\mathcal{P}$ interpolates a polynomial $\delta(X)$ from $\overline{b}$
4. $\mathcal{P}$ constructs an accumulative polynomial $\sigma(X)$ such that:
   (a) $\sigma(X) - \sigma(X\omega) = \delta(X) \cdot \phi(X), X \neq \omega^{n-1}$
   (b) $\sigma(\omega^{n-1}) = \delta(\omega^{n-1}) \cdot \phi(\omega^{n-1})$
5. $\mathcal{P}$ runs $\mathsf{KZG.open}_{zk}(\sigma, \delta, \phi)$
6. $\mathcal{V}$ accepts the proof in and only if
   (a) The construction of δ is valid
   (b) $[\sigma(X) - \sigma(X\omega) - \delta(X) \cdot \phi(X)] \cdot (X - \omega^{n-1}) = 0$
   (c) $[\sigma(X) - \delta(X) \cdot \phi(X)] \cdot \frac{X^n - 1}{X - \omega^{n-1}} = 0$

Protocol 2: The π_{assets} proof demonstrates that the balances associated with each key in the anonymity set are included, the subset not owned by the exchange (per selector vector from π_{keys}) are zero-ed out, and remaining balances are totalled correctly in $\sigma(1)$.

π_{keys}			
Scheme	**Proving time**	**Verifying time**	**Proof size**
Xiezhi (**Ours**)	$O(\kappa^2)$	$O(\kappa)$	$O(\kappa)$

π_{assets}				
Scheme	**Proving time**	**Verifying time**		**Proof size**
		π_{input}	π_{proof}	
Provisions[**provisions**]	$O(\kappa)$	N/A	$O(\kappa)$	$O(\kappa)$
Bulletproofs[**bulletproofs**]	$O(\kappa)$	N/A	$O(\kappa)$	$O(\log \kappa)$
IZPR[**izpr**]	$O(t \log t)$	$O(\kappa)$	$O(1)$	$O(1)$
Xiezhi (**Ours**)	$O(\kappa)$	$O(\kappa)$	$O(1)$	$O(1)$

Table 2: Comparison of this work with prior PoA schemes. π_{input} is the verifier processes the public inputs before validating the proof; π_{proof} is the verifier verifies the proof sent by the prover. Notation: κ is the number of keys that the exchange wants to prove. For IZPR[**izpr**], t is the throughput of the blockchain (number of addresses which have changed since the last proof).

$\mathcal{P}$'s private input: user balances, $\{\mathtt{bal}_1, \mathtt{bal}_2, \ldots, \mathtt{bal}_\mu\}$

1. $\mathcal{P}$ computes the binary decomposition (from most significant bit to least significant bit) of each balance, $\{z_j^{(\mathtt{bal}_i)}\}_{i\in[\mu], j\in[k]}$, such that $z_j^{(\mathtt{bal}_i)} \in \{0,1\}$ and $\mathtt{bal}_i = \sum_{j=k}^{0} 2^j \cdot z_j^{(\mathtt{bal}_i)}$
2. $\mathcal{P}$ puts the bits into accumulator form where $\chi_k^{(\mathtt{bal}_i)} = z_k^{(\mathtt{bal}_i)}$ and $\chi_i^{(\mathtt{bal}_i)} = 2\chi_{i+1}^{(\mathtt{bal}_i)} + z_i^{(\mathtt{bal}_i)}$. (Remark: visualized as a matrix, each row is a balance where the k-th column is the least significant bit and, moving right-to-left, each bit is folded in until it accumulates to $\mathtt{bal}_j$ in the first column.)

$$\begin{bmatrix} \chi_1^{(\mathtt{bal}_1)} & \chi_2^{(\mathtt{bal}_1)} & \chi_3^{(\mathtt{bal}_1)} & \cdots & \chi_k^{(\mathtt{bal}_1)} \\ \chi_1^{(\mathtt{bal}_2)} & \chi_2^{(\mathtt{bal}_2)} & \chi_3^{(\mathtt{bal}_2)} & \cdots & \chi_k^{(\mathtt{bal}_2)} \\ \chi_1^{(\mathtt{bal}_3)} & \chi_2^{(\mathtt{bal}_3)} & \chi_3^{(\mathtt{bal}_3)} & \cdots & \chi_k^{(\mathtt{bal}_3)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \chi_1^{(\mathtt{bal}_\mu)} & \chi_2^{(\mathtt{bal}_\mu)} & \chi_3^{(\mathtt{bal}_\mu)} & \cdots & \chi_k^{(\mathtt{bal}_\mu)} \end{bmatrix}$$

3. Due to the concrete parameters of `bls12-381`, $\mathcal{P}$ will work column-by-column (proof size and verifier time will be linear in k which is the bit-precision of each account). Let column j be vector $\overline{p}_j = \{\chi_j^{(\mathtt{bal}_1)}, \chi_j^{(\mathtt{bal}_2)}, \ldots, \chi_j^{(\mathtt{bal}_\mu)}\}$. The following constraints apply (for $i \in [\mu], j \in [k]$): $\overline{p}_1[i] = \mathtt{bal}_i$; $\overline{p}_j[i] - 2 \cdot \overline{p}_{j+1}[i] \in \{0,1\}$; and $\overline{p}_k[i] \in \{0,1\}$. $\overline{p}_1$ contains $\{\mathtt{bal}_1, \mathtt{bal}_2, \ldots, \mathtt{bal}_\mu\}$.
4. $\mathcal{P}$ builds an additive accumulator ν for $\overline{p}_1$ where $\nu_k = \mathtt{bal}_k = \overline{p}_1[k]$ and $\nu_i = \nu_{i+1} + \mathtt{bal}_i, i \in [1, \mu]$. Remark: ν_1 will contain the total liability balances.
5. $\mathcal{P}$ interpolates polynomials for $\overline{p}_j \to p_j(X)$ and publishes commitments to each using KZG in extended zero-knowledge form (see Section 0.2.3).
6. $\mathcal{P}$ shows the following polynomials are vanishing for all $x \in H$ where $H = \{\omega^0, \omega^1, \ldots, \omega^{k-1}\}$

$$\begin{aligned} w_1 :=& [p_0(X) - p_0(X\omega) - p_1(X)] \cdot (X - \omega^{\mu-1}) \\ w_2 :=& [p_0(X) - p_1(X)] \cdot \frac{X^\mu - 1}{X - \omega^{\mu-1}} \\ w_3 :=& p_m(X) \cdot [1 - p_m(X)] \\ v_1 :=& [p_1(X) - 2p_2(X)] \cdot [1 - (p_1(X) - 2p_2(X))] \\ v_2 :=& [p_2(X) - 2p_3(X)] \cdot [1 - (p_2(X) - 2p_3(X))] \\ & \vdots \\ v_{k-1} :=& [p_{k-1}(X) - 2p_k(X)] \cdot [1 - (p_{k-1}(X) - 2p_k(X))] \end{aligned}$$

   w_1 and w_2 prove the accumulative vector is correct, w_3 and $\{v_i\}$ prove each liability is greater than or equal to 0 (range proof). To complete the proof, $\mathcal{P}$ will run $\mathsf{KZG.open}_{zk}(p_0, p_1, p_2, \ldots, p_k)$ at a random evaluation point ζ and $\mathsf{KZG.open}_{zk}(p_0)$ at $\zeta\omega$

$\mathcal{V}$ protocol:

1. Verify each evaluation is valid
2. Verify $\{w_1, w_2, w_3\}$ and $\{v_i\}$ are vanishing in H

Protocol 3: The $\pi_{\mathsf{liabilities}}$ proof demonstrates that each liabilities is either zero or a positive number, and that the balances are totalled correctly in $\pi_1(\omega^0)$.

$\mathcal{P}$ private input: user identifiers, $\{\mathfrak{uid}_1, \mathfrak{uid}_2, \ldots, \mathfrak{uid}_\mu\}$, and user balances, $\{\mathfrak{bal}_1, \mathfrak{bal}_2, \ldots, \mathfrak{bal}_\mu\}$

1. $\mathcal{P}$ interpolates the identifier polynomial $u(X)$ such that $u(X_i) = \mathfrak{uid}_i$ for i from 1 to μ.
2. $\mathcal{P}$ publishes commitment to $u(X)$.
3. For check from user i, $\mathcal{P}$ tells the user they at index i and opens $u(i)$ and $p_1(i)$ (from $\pi_{\mathsf{liabilities}}$ above).

Customer's input: the commitments to the identifier polynomial $u(X)$ and the balance polynomial $p_1(X)$ and the witnesses at the corresponding points of $u(i)$ and $p_1(i)$

1. $\mathcal{V}$ verifies that the two indexes i are identical.
2. $\mathcal{V}$ verifies their user identifier is the evaluation of u at the given point i.
3. $\mathcal{V}$ verifies their balance is the evaluation of p_1 at the given point i.

Protocol 4: The π_{users} proof demonstrates that to each user who checks that their balance is recorded correctly under a unique identifier for them (to mitigate clash attacks).

1. $\mathcal{P}$ computes equality eq as the total assets minus the total liabilities.
2. $\mathcal{P}$ publishes commitment to polynomial $\phi(X)$ where $\psi(\omega^0) = \mathsf{eq}$.
3. $\mathcal{P}$ generates a range proof for eq in $\phi(X)$ to demonstrate it is a non-negative integer.
4. $\mathcal{P}$ opens $\sigma(\omega^0) - p_1(\omega^0) - \psi(\omega^0)$ to the value 0.

Protocol 5: The π_{solvency} proof demonstrates that the total assets exceed the total liabilities by a non-negative integer (called the equity).

Scheme	**Proving time**	**Verifying time**		**Proof size**	
		π_{users}	$\pi_{\mathsf{liabilities}}$	π_{users}	$\pi_{\mathsf{liabilities}}$
Provisions[**provisions**]	$O(\mu)$	$O(1)$	$O(\mu)$	$O(1)$	$O(\mu)$
DAPOL+[**dapol**]	$O(\mu \log \mu)$	$O(\log \mu)$	$O(1)$	$O(\log \mu)$	$O(1)$
SPPPOL[**spp**]	$O(\log_\lambda \mu)$	$O(\log_\lambda \mu)$	$O(1)$	$O(\log_\lambda \mu)$	$O(1)$
Notus[**notus**]	$O(\mu \log \mu)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Xiezhi (**Ours**)	$O(k \cdot \mu)$	$O(1)$	$O(k)$	$O(1)$	$O(k)$

Table 3: Comparison of this work with prior PoL schemes. Notation: μ is the number of users, k is the number of bits of the range proof. For SPPPOL [**spp**], λ is the arity of the Verkle Tree it uses.

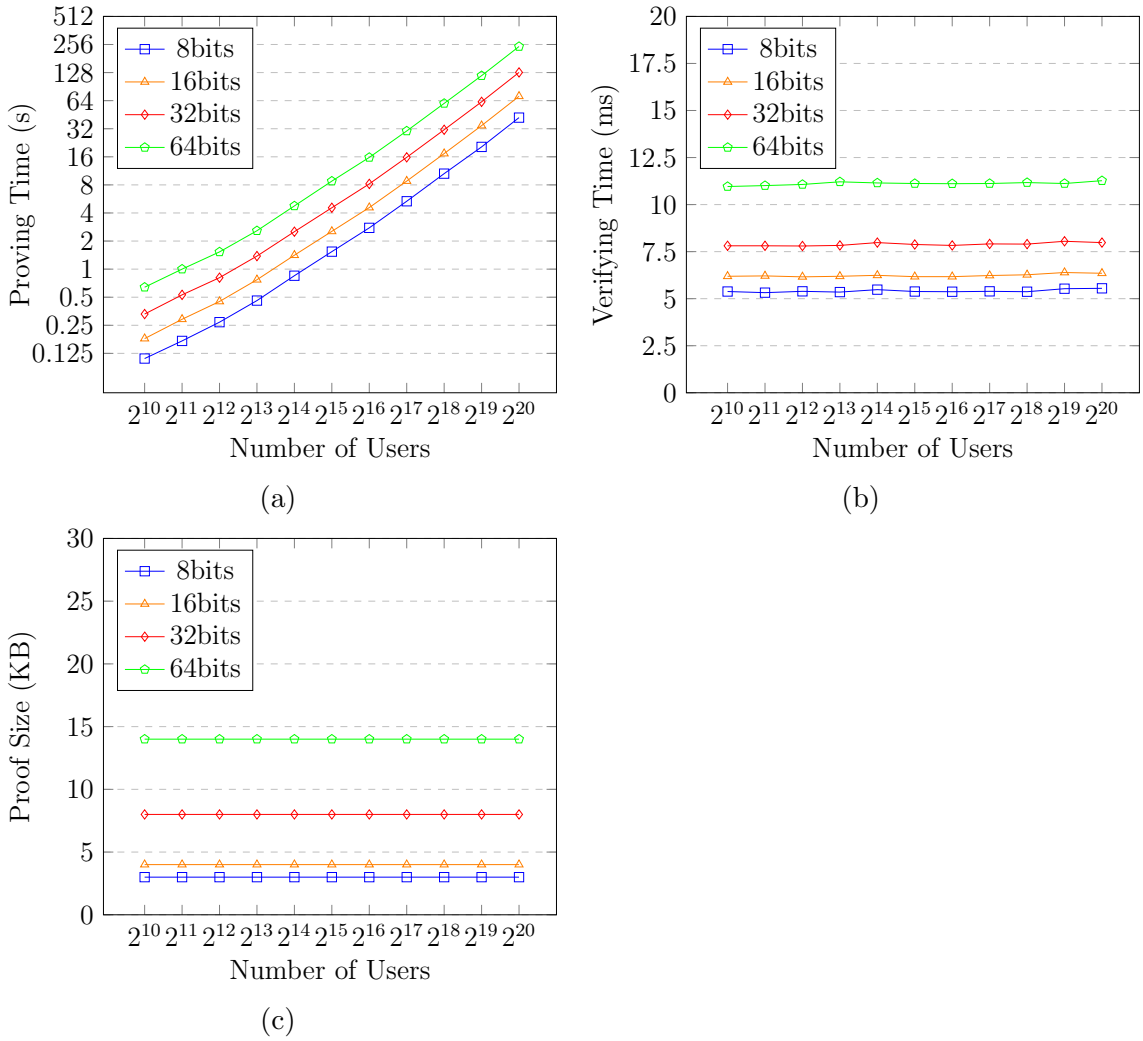


Figure 3: Performance of PoL by different number of bits for the range proof. Subfigure (a) illustrates the proving time linearly relates to the number of users, and it is also linear to the number of bits for the same number of users; Subfigure (b) and (c) show the verifying time and the proof size are unrelated to the number of users, but relate to the number of bits. Unlike the proving time, the verifying time and the proof size do not linearly increase following the number of bits.

.1 Cryptographic Primitives

.1.1 Discrete Logarithm Assumption:

The discrete logarithm problem describes, given a triplet $(\mathbb{G}_p, p, g_p)$ and an element $y \in \mathbb{G}_p$, it is infeasible to find an x such that $y = g_p^x$ in polynomial time.

.1.2 Pedersen Commitment:

Pedersen's commitment scheme [**pedersen**] enables $\mathcal{P}$ to *commit* to a value x without revealing it. Pedersen commitment provides perfectly hiding and computational binding based on the discrete logarithm assumption. Additionally, Pedersen commitments are *additively homomorphic*: given two commitments C_1 and C_2, the summation of their secrets x_1 and x_2 is the secret of $C_1 + C_2$. We use $\mathbf{C}(x)$ to denote a Pedersen commitment to the value x. Particularly, we use $\mathbf{C}_{\texttt{secp}}$ to denote a Pedersen commitment in secp256k1 and $\mathbf{C}_{\texttt{bls}}$ to denote a Pedersen commitment in BLS12-381.

.1.3 $\Sigma-$protocols:

The $\Sigma-$protocol is a three-move interactive proof system. We define the $\Sigma-$protocol similarly to [**damgard10**].

Definition 1. *Let R be a binary relation between the statement x and the witness w. Given common input x to $\mathcal{P}$ and $\mathcal{V}$, and private input (x, w) such that $(x, w) \in R$ to $\mathcal{P}$, they run the following protocol:*

1. *$\mathcal{P}$ computes a message m from (x, w) and sends m.*
2. *$\mathcal{V}$ sends a random challenge c.*
3. *$\mathcal{P}$ replies with z.*

At the end of the protocol $\mathcal{V}$ has the data (x, m, c, z). He decides to output **acc** *or* **rej***; such that*

- ***Completeness*:** *A $\Sigma-$protocol is complete if $\mathcal{P}$ follows the protocol to generate the message (m, c, z), $\mathcal{V}$ always accepts.*
- ***Special soundness*:** *A $\Sigma-$protocol is special sound if there exists a p.p.t extractor $\mathcal{E}$, given any input x and any two accepting $(m, c, z), (m, c', z')$ where $c \neq c'$, $\mathcal{E}$ can compute w where $(x, w) \in R$.*
- ***Honest verifier zero-knowledge*:** *A $\Sigma-$protocol is honest verifier zero-knowledge if there exists a p.p.t simulator $\mathcal{S}$, such that the transcript produced by $\mathcal{S}$ is indistinguishable from the messages between $\mathcal{P}$ and $\mathcal{V}$.*

A commonly well-known way to convert a $\Sigma-$protocol into non-interactive is using the Fiat-Shamir transform [**fs**]. But we still use the standard interactive $\Sigma-$protocol to demonstrate our work for comprehension.

.1.4 OR Proof:

The OR proof allows $\mathcal{P}$ to convince $\mathcal{V}$ that given two inputs x_1, x_2, he knows w such that $(x_1, w) \in R_1$ or $(x_2, w) \in R_2$, but $\mathcal{V}$ cannot learn which one $\mathcal{P}$ knows. We use the same definition as [**damgard10**].

Definition 2. *The OR proof is a $\Sigma-$protocol that given two inputs x_1, x_2, $\mathcal{P}$* and *$\mathcal{V}$ run the following protocol:*

1. *$\mathcal{P}$ computes the message m_1 using (x_1, w) as input.*
   $\mathcal{P}$ randomly generates c_2 as the challenge for x_2 and runs the simulator $\mathcal{S}(x_2, c_2)$ to produce (m_2, z_2).
2. *$\mathcal{P}$ sends m_1 and m_2.*
3. *$\mathcal{V}$ sends a master challenge c.*
4. *$\mathcal{P}$ computes $c_1 = c \oplus c_2$ and z_1 on inputs (x_1, c_1, m_1, w).*
   $\mathcal{P}$ sends (c_1, c_2, z_1, z_2).

At the end of the protocol $\mathcal{V}$ verifies $c = c_1 \oplus c_2$ and both (m_1, c_1, z_2) and (m_2, c_2, z_2) are valid to output **acc** *or* **rej***; such that*

- ***Completeness:*** *The case of c_2 is always accepted by $\mathcal{V}$ as the definition of a simulator; on the other side, the case of c_1 has no difference from the standard $\Sigma-$protocol. Therefore, the OR proof is complete.*
- ***Soundness:*** *Let $\mathcal{P}$ execute the protocol twice. Two transcripts*

  $$(x_1, x_2, c, c_1, c_2, z_1, z_2), (x_1, x_2, c', c_1', c_2', z_1', z_2'), c \neq c'$$

  are given. Then an extractor $\mathcal{E}$ can be constructed by combining two pairs of transcripts $(m_1, c_1, z_1), (m_1', c_1', z_1')$ and $(m_2, c_2, z_2), (m_2', c_2', z_2')$ to compute x_1 and x_2 respectively.
- ***Honest verifier zero-knowledge****: Given a master challenge c, choose c_1 or c_2 randomly and the other will be determined. Let the simulator run twice: $\mathcal{S}(x_1, c_1), \mathcal{S}(x_2, c_2)$.*

We use $x = x_1 \vee x_2$ to denote x is x_1 or x_2.

.1.5 Polynomial Commitment Scheme:

A polynomial commitment scheme (PCS) allows $\mathcal{P}$ to commit to a polynomial to convince $\mathcal{V}$ that claimed evaluations are of the committed polynomial [**kzg**]. We define PCS based on [**kzg**; **plonk**; **bdfg**].

Definition 3. *A polynomial commitment scheme consists of three moves:* **gen**, **com**, *and* **open** *such that*

1. **gen**(d) *is an algorithm that given a random number* $\tau \in \mathbb{F}$ *and a positive integer* d*, outputs a structured reference string (SRS)* **srs** *such that*

$$\mathbf{srs} = ([1]_1, [\tau]_1, \ldots, [\tau^{d-1}]_1, [1]_2, [\tau]_2)$$

2. **com**$(f,$ **srs**$)$ *outputs a commitment* $\mathcal{C}$ *to* f*, where* f *is a polynomial over* $\mathbb{F}$ *of degree smaller than* d*.*

3. **open** *is a protocol that* $\mathcal{P}$ *is given input* f*, and* $\mathcal{P}$ *and* $\mathcal{V}$ are both given

   - **srs**
   - $\mathcal{C}$ *- the commitment to* f
   - a *- an evaluation point of* f
   - b *- the claimed evaluation of* $f(a)$

   They run the protocol as follows:

   (a) $\mathcal{P}$ *reveals the proof* π *for the pair* (a, b) *where*

   $$\pi = [\frac{f(x) - b}{x - a}]_1$$

   (b) $\mathcal{V}$ *outputs* **acc** *if and only if*

   $$e(\mathcal{C} - [b]_1, [1]_2) = e(\pi, [\tau - a]_2)$$

- ***Completeness****: It is clear that* π *exists if and only if* $f(a) = b$*, which means* $\mathcal{V}$ *always accepts the proof if* $\mathcal{P}$ *follows the protocol.*

- ***Knowledge soundness in the algebraic group model****: For any algebraic adversary* $\mathcal{A}$ *in an interactive protocol of PCS, there exists a p.p.t extractor* $\mathcal{E}$ *given access to* $\mathcal{A}$*'s messages during the protocol, and* $\mathcal{A}$ *can win the following game with negligible probability:*

  1. *Given the inputs that* $\mathcal{P}$ *can access,* $\mathcal{A}$ *outputs* $\mathcal{C}$*.*
  2. $\mathcal{E}$ *outputs* $f \in \mathbb{F}_{<d}[X]$ *from* $\mathcal{A}$*'s output.*
  3. $\mathcal{A}$ *generates* π *and the evaluation* b' *at the evaluation point* a*.*
  4. $\mathcal{A}$ *wins if*
     - $\mathcal{V}$ *accepts the proof at the end of the protocol.*
     - $b' \neq b$*.*

Our work uses PCS with the root of unity. We use ω to denote the root of unity. We use $[f]_1$ and $\mathcal{C}_f$ to denote a commitment to f interchangeably.

.1.6 Open KZG with Committed Value

By Definition 3, to prove b is the evaluation of $f(a)$, $\mathcal{P}$ reveals the pair (a, b) to let $\mathcal{V}$ validate the proof through pairings. However, this leaks the evaluation at the point a. π_{keys} uses the variaint of this opening techinque. Here we describe the new opening scheme.

openToCommit *is a protocol that $\mathcal{P}$ is given input f and b. $\mathcal{P}$ and $\mathcal{V}$* are both given

- **srs**
- $\mathcal{C}$
- a - *an evaluation point of f*

They run the protocol as follows:

1. *$\mathcal{P}$ reveals the proof π for the pair (a, b) where*

$$\pi = [\frac{f(x) - b}{x - a}]_1$$

2. *Instead of sending (a, b) directly, $\mathcal{P}$ sends $(a, [b]_1)$*
3. *$\mathcal{V}$ outputs* **acc** *if and only if*

$$e(\mathcal{C} - [b]_1, [1]_2) = e(\pi, [\tau - a]_2)$$

This does not violate the soundness of the original KZG commitment scheme. Note $[b]_1$ is a Pedersen commitment. Recall the computational binding property of Pedersen commitment, that means it is infeasible for $\mathcal{P}$ to compute a $[b']_1$ such that $f(a) \neq b', [b']_1 = [b]_1$ based on discrete logarithm assumption. Since only π_{keys} requires opening KZG evaluation with committed value, we implicitly refer to this variaint opening scheme in the context of π_{keys}.

.1.7 Roots of Unity

We use the approach of encoding data vectors into polynomials, committing to them using a polynomial commitment scheme (PCS), and forming zero knowledge arguments—a model called a polynomial-based interactive oracle proof (Poly-IOP). The zk-snark system Plonk popularized Poly-IOPs and has many extensions and optimizations. A one-dimensional vector of data is encoded into a univariate polynomial using 1 of 3 methods (all 3 are used at different steps of Plonk): (1) into the coefficients of the polynomial, (2) as roots of the polynomial, and (3) as the y-coordinates ($\mathsf{data}_i = P(x_i)$) of points on the polynomial. Plonk mostly relies on (3) and an interpolation algorithm is used to find the corresponding coefficients of the polynomial, which is needed for the PCS. General interpolation algorithms are $O(n^2)$ work for n evaluation points but this can be reduced to $O(n \log n)$ with an optimization.

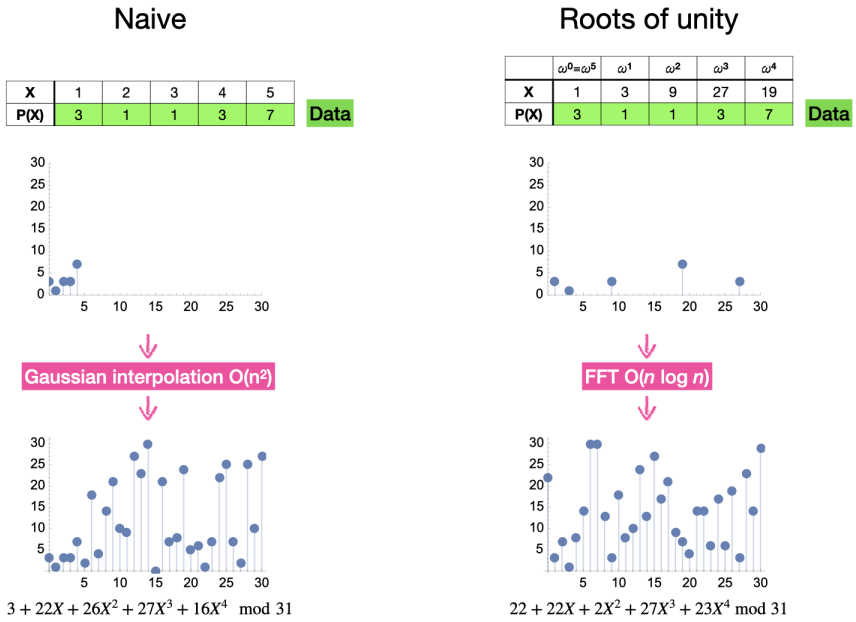


Figure 4: Small number ($\mathbb{Z}_{31}$) example of encoding a vector of integers $\langle 3, 1, 1, 3, 7\rangle$ into (a) the first 5 points of a polynomial, and (b) into 5th roots of unity ($\omega = 3$).

The optimization enables interpolation via the fast Fourier transform (FFT). It concerns how to choose the x-coordinates, which will serve as the index for accessing the data: evaluating $P(X)$ at x_i will reveal data_i. First note, x-coordinates are from the exponent group (Z_q) and the choices exceed what is feasible to use (2^{255} values in `bls12-381`). Any subset can be used and interpolated. The optimization is to chose them with a mathematical structure. Specifically, instead an additive sequence (e.g., $0, 1, 2, 3, \ldots$), we use a multiplicative sequence $1, \omega, \omega \cdot \omega, \omega \cdot \omega \cdot \omega, \ldots$ or equivalently: $\omega^0, \omega^1, \omega^2, \ldots, \omega^{\kappa-1}$. Further, the sequence is closed under multiplication which means that next index after $\omega^{\kappa-1}$ wraps back to the first index: $\omega^{k-1} \cdot \omega = \omega^\kappa = \omega^0 = 1$ (this property is also useful in proving relationships between data in the vector and its neighbouring values).

For terminology, we say ω is a generator with multiplicative order κ in Z_q. This implies $\omega^\kappa = 1$. Rearranging, $\omega = \sqrt[\kappa]{1}$. Thus we can equivalently describe ω as a κ-th root of 1. Finally, as 1 is the unity element in Z_q, ω is commonly called a κ-th root of unity.

For practical purposes, κ represents the length of the longest vector of data we can use in our protocol. Where does κ come from? Different elements of Z_q will have different multiplicative orders but every order must be a divisor of $q - 1$. Thus κ is the largest divisor of the exact value of q used in an elliptic curve standard. The value of q in `bls12-381` has $\kappa = 2^{32}$ (for terminology, this called a 2-adicity of 32).