Summary
This paper addresses a key vulnerability in Ethereum's Proof-of-Stake (PoS) protocol, namely the vulnerability of block proposers against Denial-of-Service (DoS) attacks. The attack is made possible through deanonymization, where adversaries get validator IP addresses, after which they check for matches and attack upcoming proposers. To counter this, Ethereum has proposed the Whisk protocol, a Single Secret Leader Election (SSLE) protocol that uses zero-knowledge proofs to hide proposer identities. Whisk is shuffle-based, meaning it resists adversarial tracking by iteratively shuffling a subset of a list of trackers, where each validator has a single tracker. Each shuffle in Whisk relies on a shuffle-based zero-knowledge proof called Curdleproofs, which uses Inner Product Arguments (IPAs) to validate the correctness of the shuffle. However, Curdleproofs is constrained to shuffling subsets of sizes that are only a power of two due to the recursive nature of the IPA.
For that reason, we propose a modified protocol, CAAUrdleproofs, which lifts this restriction by integrating ideas from Springproofs. CAAUrdleproofs introduces a new folding scheme that allows the use of arbitrary shuffle sizes, enabling more flexibility when reducing or increasing the size. The experiments show that CAAUrdleproofs maintains a similar performance to Curdleproofs for power of two shuffle sizes but outperforms it when shuffle sizes are not a power of two, especially when the size is slightly above a power of two. This paper also proposes reducing the shuffle size from 128 down to 80, which would result in a notable decrease in the block overhead created by the protocol. The overhead would decrease from 16.656 KB to 12.048 KB, resulting in annual savings of approximately 12.11 GB.
This paper also provides a security analysis of the shuffle mechanism with different shuffle sizes and under various amounts of adversarial influence. The results validate that smaller shuffle sizes can still maintain a secure shuffle within the total amount of shuffles available in a round of the Whisk protocol. The paper concludes by mentioning that CAAUrdleproofs is an efficient improvement to Curdleproofs. It suggests future development in the direction of post-quantum security, protocol refinement, or exploring the use of a Weighted Inner Product Argument (WIPA).

Zero-Knowledge Shuffle Improvement in Ethereum Single Secret Leader Election

Anders Malta Jakobsen*, Oliver Holmgaard†

Abstract—Ethereum is one of the leading Proof-of-Stake blockchains. However, it is still vulnerable to attacks. One such attack is the de-anonymization attack by Heimbach et al., where an adversary can obtain validator IP addresses and then perform a Denial-of-Service attack on them. To try and combat this attack, Ethereum has proposed the use of the Whisk protocol. Whisk is a Single Secret Leader Election protocol that uses a zero-knowledge proof called Curdleproofs that uses Inner Product Arguments to prove the validity of a shuffle of validators. This paper improves upon Curdleproofs' Inner Product Arguments by introducing CAAUrdleproofs, a modified version of Curdleproofs incorporating ideas from Springproofs to address the limitations of Curdleproofs regarding shuffle size. We show that CAAUrdleproofs has similar proving and verifying times to Curdleproofs when the shuffle size is a power of two. We also demonstrate that CAAUrdleproofs has a performance advantage for any shuffle size that is not a power of two and that this advantage increases as the shuffle size decreases below a power of two. After performing experiments, we also suggest a new shuffle size, which is smaller than the current one used in Curdleproofs, resulting in a more negligible block overhead than the one created by the current Curdleproofs protocol. All this is done while still preserving the anonymity of validators.

Index Terms—Ethereum, Proof of Shuffle, Distributed Systems, Inner Product Arguments, Zero-Knowledge Proof, Single Secret Leader Election

1 Introduction

Ethereum is a decentralized blockchain platform that enables developers to build and deploy smart contracts and decentralized applications. It is the second-largest blockchain platform by market capitalization, boasting a large and active developer community. Currently working as a Proof-of-Stake protocol, block proposal opportunities are allocated to validators, which can be created by community members willing to stake their ether cryptocurrency. However, previous work by Heimbach et al., also confirmed by our previous study, shows that adversaries can gather validator IP addresses [1, 2]. These can be used to perform a Denial-of-Service (DoS) attack on the validators, thereby threatening the liveness of the blockchain [2, 3].

In response to the potential threat, Ethereum has proposed a protocol, Whisk, which hides the identities of block proposers, making the DoS attack harder to perform [4]. Whisk is a Single Secret Leader Election (SSLE) protocol [5], where validators each publish a private tracker, which is used for proposer selection instead. When proposing a block, the validator will then prove the ownership of the tracker. To ensure that adversaries are unable to trace the tracker to specific validators, each block proposer shuffles the list of validator trackers while adding randomness to the trackers.

Making sure that this has been done correctly is essential to the protocol. Hence, Whisk uses a proof protocol called Curdleproofs, which is a Zero-Knowledge Proof of Shuffle [6]. Therefore, the block proposer constructs such a proof and adds it to the block, after which other validators can verify the proof.

Whisk introduces a block size overhead to the blockchain. Also, additional work is required for both provers and verifiers.

In this paper, we delve into the structure of Curdleproofs to identify areas where the protocol can be optimized. Specifically, we work with the concept of Inner Product Arguments (IPAs) and how they generally only work for vector sizes that are powers of two.

Our protocol, CAAUrdleproofs, aims to improve on the rigid nature of Curdleproofs. Following this, we also provide arguments for the conditions under which CAAUrdleproofs remains secure.

Working with this led to the following contributions:

- We have successfully modified Curdleproofs, using the Springproofs framework [7], to allow flexibility when choosing the shuffle size.
- We have implemented CAAUrdleproofs and run experiments on both protocols, showing that CAAUrdleproofs has the potential to be faster and smaller compared to Curdleproofs.
- We have experimentally demonstrated that CAAUrdleproofs remains secure even when reducing the size of shuffled elements.

2 Background

In this section, we provide the necessary background information on Ethereum and a specific attack it is vulnerable to, the Whisk protocol [4], and the Curdleproofs protocol [6] used in Whisk.

- *All authors are affiliated with the Dept. of Computer Science, Aalborg University, Aalborg, Denmark*
- *E-mails: *amja23, †oholmg20 @student.aau.dk*

The notation used throughout this paper can be seen in Table 1.

Since this work is based on the existing Curdleproofs protocol [6], it inherits the same security assumptions. Our work, therefore, runs as a public coin protocol in any cryptographic group where Decisional Diffie-Hellman (DDH) is hard [8]. DDH is defined as follows.

Definition 1 (DDH). *Given a finite, multiplicative cyclic group $\mathbb{G}$ of prime order p, the decisional Diffie-Hellman problem is defined as follows: Given $(g^a, g^b, g^c) \in \mathbb{G}$, where g is a generator of $\mathbb{G}$ and $a, b, c \in \mathbb{Z}_p$, decide whether $c = ab$.*

2.1 Zero-Knowledge Proofs

Before explaining the protocol, we must mention that Curdleproofs, and hence also Whisk, is a Zero-Knowledge Proof (ZKP) system. It is a system that allows a prover to convince a verifier that they know a secret without revealing the secret itself. Within the context of Ethereum, it could convince someone that a transaction is valid without revealing information about the transaction, such as its value. Whisk uses Curdleproofs to prove the validity of a shuffle.

Definition 2 (Zero-Knowledge Argument of Knowledge). *An argument $(Setup, P, V)$ is a zero-knowledge argument of knowledge of a relation $\mathbb{R}$ if it satisfies completeness, knowledge-soundness and is honest-verifier zero-knowledge.*

Definitions for knowledge-soundness, completeness, and Honest-Verifier Zero-Knowledge (HVZK) can be found in Appendix A.

Additionally, two of the three proofs that comprise Curdleproofs are proven using Inner Product Arguments (IPAs). These are also ZKPs and will be the focus of this paper. Hence, we define IPAs.

Definition 3 (Inner Product Argument). *The argument takes as input two binding vector commitments $C = \mathbf{c} \times \mathbf{g} \in \mathbb{G}$ and $D = \mathbf{d} \times \mathbf{g}' \in \mathbb{G}$ to the vectors $\mathbf{c}, \mathbf{d} \in \mathbb{Z}_p^n$ and $z \in \mathbb{Z}_p$. The goal is to prove that $z = \mathbf{c} \times \mathbf{d}$. The argument has logarithmic communication by halving the dimensions of $\mathbf{c}$ and $\mathbf{d}$ in each iteration.*

2.2 Whisk

Ethereum utilizes a Proof of Stake (PoS) consensus mechanism, enabling users to validate transactions and create new blocks by staking their ether (ETH) tokens. The PoS protocol operates in epochs of 32 slots, where each slot is 12 seconds long. In each slot, a proposer is chosen to propose a block, thereby allowing the network to reach a consensus on the state of the blockchain.

The proposer Denial-of-Service (DoS) attack is a type of attack that targets the block proposers, making them unable to propose blocks. An adversary can use the proposer DoS attack to prevent a proposer from receiving rewards from proposing a block, increasing the adversarial reward [9]. The proposer DoS is made possible by an attack on the Ethereum network, discovered by Heimbach et al. [1], which deanonymizes validators and obtains their IP addresses. In our preliminary work [2], we show that the attack is still possible to perform on the Ethereum network. As a response to the proposer DoS attack, Ethereum proposed a new protocol called Whisk [4] to mitigate the attack.

Whisk is a Zero-Knowledge (ZK) Single Secret Leader Election (SSLE) system that uses a ZK argument called Curdleproofs [6] to verify the correctness of a shuffle with size ℓ without revealing the input or output [5]. Whisk works by selecting a list of 16,384 validator trackers of the form (rG, krG), where k is a secret known by the validator, r some randomness, and G is a cryptographic generator. Then, the following 8,192 block proposers shuffle the trackers over 8,192 slots ($\sim$1 day). Then, 8,192 proposers are selected from the shuffled list to propose blocks for the next 8,192 slots while a new list is being shuffled. This way, a new list of proposers is created every day. After each shuffle, Whisk uses a ZKP to prove that the shuffle is correct. As the specific shuffle is hidden to prevent adversarial tracking, this is done to ensure that the trackers are shuffled according to protocol specifications. Whenever a proposer is chosen, they can prove that they are the correct proposer for the slot without revealing their identity. Therefore, Whisk mitigates the proposer DoS attack because the identities of the upcoming proposers are now hidden.

Curdleproofs is a ZKP system used by Whisk that allows a prover to prove knowledge of a shuffle without revealing the specific order in which the elements were shuffled. It takes as input the pre-shuffled sets $\mathbf{R}$ and $\mathbf{S}$, the shuffled sets $\mathbf{T}$ and $\mathbf{U}$, and a commitment M to the permutation σ. The protocol then constructs the proof using three different ZKPs, with one of them relying on two more ZKPs. The overview can be seen in Figure 1.

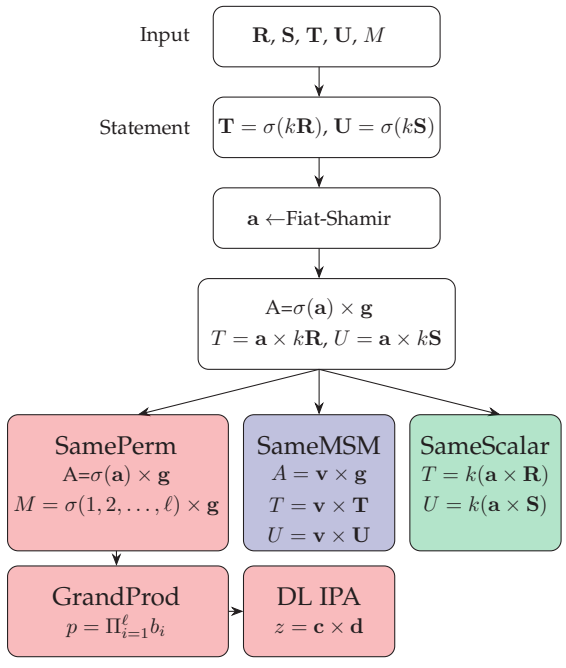


Fig. 1: Overall structure of the Curdleproofs protocol. Modified figure from [6].

The first proof is the Same Permutation (SamePerm) proof. The prover first constructs a commitment to the permutation, $\sigma()$, by saying $M = \sigma(1, 2, \ldots, \ell) \times \mathbf{g}$, where ℓ is the number of shuffled trackers, and $\mathbf{g}$ is a vector of cryptographic generators. Then, using the Fiat-Shamir trans-

Symbol	Description
$\mathbb{G}$	Cyclic, additive, group of prime order p
$\mathbb{Z}_p$	Ring of integers modulo p
$\mathbb{G}^n,\ \mathbb{Z}_p^n$	Vector spaces of dimension n over $\mathbb{G}$ and $\mathbb{Z}_p$
$\mathbb{Z}_p^*$	Multiplicative group $\mathbb{Z}_p \setminus \{0\}$
$H \in \mathbb{G}$	Generator of $\mathbb{G}$
$\ell \in \mathbb{Z}$	Number of shuffled ciphertexts in the Whisk protocol
$\gamma \in \mathbb{Z}_p^{\lceil \log n \rceil}$	Uniformly distributed challenges
$\mathbf{a} \in \mathbb{F}^n$	Vector $\mathbf{a} = (a_1, \ldots, a_n) \in \mathbb{F}^n$
$\mathbf{A} \in \mathbb{F}^{n \times m}$	Matrix with n rows and m columns
$\mathbf{b} = c \cdot \mathbf{a} \in \mathbb{Z}_p^n$	The vector where $b_i = c\,a_i$, with scalar $c \in \mathbb{Z}_p$ and $\mathbf{a} \in \mathbb{Z}_p^n$
$\mathbf{a} \times \mathbf{b} = \sum_{i=1}^n a_i \cdot b_i$	Inner product of $\mathbf{a}, \mathbf{b} \in \mathbb{F}^n$
$\mathbf{g} = (g_1, \ldots, g_n) \in \mathbb{G}^n, \mathbf{g}' = (g'_1, \ldots, g'_n) \in \mathbb{G}^n$	Vectors of generators (for Pedersen commitments)
$A = a \times G = \sum_{i=1}^n a_i \cdot G_i$	Binding (but not hiding) commitment to $a \in \mathbb{Z}_p^n \in$
$\mathbf{r}_A \in \mathbb{Z}^n$	Blinding factors, e.g. $A = \mathbf{a} \times \mathbf{g} + \mathbf{r}_A \times \mathbf{g}$ is a Pedersen commitment to $\mathbf{a}$
$\mathbf{a} \parallel \mathbf{b} \in \mathbb{Z}_p^{n+m}$	Concatenation: if $\mathbf{a} \in \mathbb{Z}_p^n$, $\mathbf{b} \in \mathbb{Z}_p^m$, then $\mathbf{a} \parallel \mathbf{b} \in \mathbb{Z}_p^{n+m}$
$\mathbf{a}_{[:k]} = (a_1, \ldots, a_k) \in \mathbb{F}^k,\ \mathbf{a}_{[k:]} = (a_{k+1}, \ldots, a_n) \in \mathbb{F}^{n-k}$	Slices of vectors (Python notation)
$\{\phi; w \mid \textit{properties satisfying } \phi, w\}$	Relation using the specified public input ϕ and private witness w

TABLE 1: Notation used throughout the paper.

formation, a challenge, $\mathbf{a}$, from public inputs is constructed, and a new commitment is made from that, $A = \sigma(\mathbf{a}) \times \mathbf{g}$. The SamePerm proof consists of convincing the verifier that the same permutation was used for constructing the commitments A and M. To do this, the two commitments are used to construct a polynomial equation. Then Neff's trick [10] is used, which observes that two polynomials are equal iff. their roots are the same up to permutation.

In order to show this, the protocol makes use of a Grand Product (GrandProd) argument. To prove that argument, Curdleproofs compiles it down to a Discrete-Logarithm Inner Product Argument (DL IPA) by expressing the multiplications of the grand product as individual equations. The proof of the DL IPA then stems from the protocol originally proposed by Bootle et al. [6, 11]

Hence, the SamePerm proof is done if the prover can prove the DL IPA.

The second proof is a Same Multiscalar Multiplication (SameMSM) argument. The prover has proven the existence of the permutation. Now, the goal of the SameMSM argument is to prove that the output ciphertext set was constructed with the same permutation, σ, here called multiscalar $\mathbf{v}$[1], committed to in commitment A. Note, therefore, that commitment A in SamePerm and SameMSM is the same commitment, where $\mathbf{v} = \sigma(\mathbf{a})$. As the multiscalar is a vector, this argument is an IPA by nature, contrary to the SamePerm argument.

The third proof is a Same Scalar argument. To mask the ciphertexts, each prover, besides permuting the set, multiplies all ciphertexts by a scalar, k. This is for randomization purposes, making it harder for adversaries to track the ciphertexts [4]. Also, all validators are still able to open their commitments if they are chosen as block proposers, even after several randomizations. Therefore, the goal of the Same Scalar argument is to prove the existence of the scalar, k, such that the commitment of the permuted set is equal to the commitment of the pre-permuted set multiplied by k.

In Chapter 6 of Curdleproofs [6] they explain that the proof has size $(18 + 10\log(\ell + 4))\mathbb{G} + 7\mathbb{F}$, where $\mathbb{G}$ is a cryptographic group point, and $\mathbb{F}$ is a field element.

1. Denoted as $\mathbf{c}$ in the Curdleproofs paper but changed for readability

2.3 Problem Definition

The current proposal of Curdleproofs only works when the shuffle size of Whisk is set to a power of two. The reason is that the underlying proofs, DL IPA in SamePerm and SameMSM, need to fold recursively down to 1 by halving the size in every round. With the current shuffling size of 128, being able to choose the size more flexibly could lead to both performance and size gains. The problem we study in this article is, therefore, how to extend Curdleproofs to ℓ values that are not a power of two.

3 Related Work

3.1 Single Secret Leader Election

An SSLE is a protocol where a group of participants randomly elects only one leader from the group. The identity of the leader is kept secret from all other participants so only the leader themselves knows that they have been chosen. The elected leader can then later publicly prove that they have been elected [5].

Leading research on SSLE includes proposals for post-quantum secure protocols based on Learning With Errors and Ring Learning With Errors [12]. This work also constructs a new concept called re-randomizable commitment (RRC) for easier work with such protocols. RRC is based on the commit-and-shuffle approach also used in Whisk.

One of the use cases of SSLE is to enhance the security of PoS blockchains by providing the proposer with added privacy.

One PoS blockchain that utilizes an SSLE is Polkadot, which uses Safrole as its SSLE protocol [13]. Safrole is the production version of the research protocol Sassafras [14]. In this, validators each produce several tickets, some of which are winning, depending on some threshold. A Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (ZK-SNARK) is then used to prove that a ticket is winning, after which the winning tickets are published to the chain. A randomization algorithm will then pick proposers from all the winning tickets for all the slots two epochs later.

3.2 Shuffling Algorithms

The Håstad square shuffle [15] is one of the proposed methods for shuffling, which can be integrated into a shuffling SSLE, such as Whisk. The Håstad square shuffle is a shuffling algorithm that shuffles a vector with n items with a shuffle size of $\sqrt{n}$. The algorithm works by rearranging the vector into a $\sqrt{n} \times \sqrt{n}$ square matrix. It then works in time steps, starting at 1. For each odd step, each column and its elements are shuffled independently. For each even step, each row and its elements are shuffled independently as well. Håstad shows that at least three time steps are needed for the shuffle to be secure. The Håstad shuffle is more rigid than the shuffling algorithm used in Curdleproofs [16] because of the fixed size of the shuffle being $\sqrt{n}$.

The Feistel shuffle [17] is a previously used shuffle method in the Whisk protocol [4]. It takes n number of validator trackers and arranges them in a $k \times k$ matrix. In each round, the i-th proposer selects the i-th row of the created matrix and shuffles it in the form $F(x, y) = (y, x + y^3 \bmod k)$. The Feistel shuffle was later replaced by the shuffle proposed by Larsen et al. [16]. Ethereum mentioned that the reason for this is that the shuffle by Larsen et al. provides a simpler protocol [4].

3.3 Bulletproofs

A big inspiration for the Curdleproofs protocol is Bulletproofs [18]. Bulletproofs is a type of range proof that uses IPAs to prove that a committed value is within a specific range without revealing the value itself. Bulletproofs is not a ZKP system in itself, but with the help of Fiat Shamir [18], it can be used to create a ZKP. Bulletproofs has also undergone a few iterations and improvements to increase speed and reduce the size of the proof since its introduction in Curdleproofs.

One of these is Bulletproofs+ [19], which uses a Weighted Innner Product Argument (WIPA) instead of the standard IPA to achieve a better performance. Bulletproofs+ is also a ZKP by itself, unlike the original Bulletproofs. Attempting to modify Curdleproofs with the WIPA introduces complications that necessitate larger modifications and is therefore not suitable. This can be seen in Appendix C

A third version of the Bulletproofs protocol is Bulletproofs++ [20], which uses a new type of argument called the norm argument to achieve a better performance. The increase in performance comes from the prover only needing to commit to a single vector rather than two. Therefore, with the two vectors, x and y of a standard IPA, they need to assume $x = y$ for their protocol to work. Then, along with the norm being weighted, which raises the same complications as with Bulletproofs+, this makes it unsuitable for Curdleproofs.

4 Approach

As explained in section 2, Curdleproofs makes use of three different proofs. This work focuses on improving the underlying IPAs, with a particular interest in the protocol's running time and proof size. The following outlines our approach to modifying the IPAs, with a focus on the DL IPA.

4.1 Springproofs

The Springproofs protocol [7] can be used very effectively in solving the problem stated in section 2.3. The theory of Springproofs provides support for IPAs to use vectors of arbitrary length. Using the findings of Springproofs means Curdleproofs could be used with shuffle sizes other than powers of two. As such, they could lower the shuffle size from the current 128 to a significantly smaller size, provided it remains secure.

One of the most notable findings in Springproofs is the usage of their so-called scheme function. This function is used to ensure that the IPA eventually will fold down to a vector of size 1. In a general IPA, Curdleproofs included, if the size of the vectors were not a power of two, the argument would not recurse down to size 1, as they work by halving the vectors every recursive round.

The core concept of the Springproofs scheme function is to split the vectors into sets, T and S, before each recursive round of the protocol. Then, the fold for that round is only done on one of the two sets, T, before the other set, S, is appended again at the end of the recursive round.

Springproofs present different scheme functions and prove some of them to be optimal. One of these optimal functions is an optimized version of their *pre-compression method*, which splits the vectors as seen in Listing 1. The computation is for finding the set, T.

```
input: n, where n > 0                                          1
                                                               2
{n} ← n                                                        3
N ← 2^⌈log n⌉−1                                                4
i_h ← ⌊(2N − n)/2⌋ + 1                                         5
i_t = ⌊n/2⌋                                                    6
if n ≠ N: #Not power of 2                                      7
    {T} ← (i_h : i_t) ∪ (N + 1 : n)                            8
else if n = N: #Power of 2                                     9
    {T} ← (1 : n) #Meaning S is empty                          10
{S} ← {n} − {T}                                                11
```

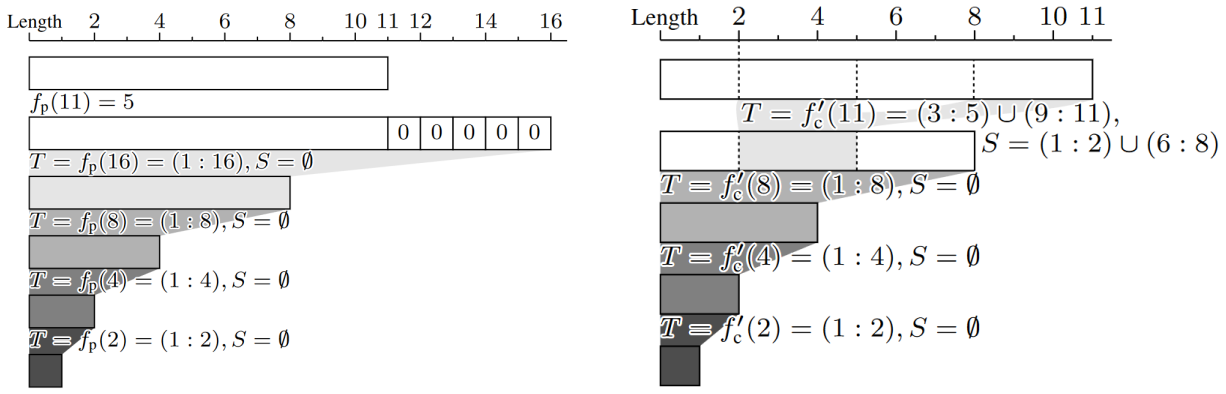
Listing 1: Scheme function f used in CAAUrdleproofs

This can also visually be seen in Figure 2(b), which is Figure 1 of the Springproofs paper [7]. Figure 2(a) is a scheme function that pads the vector to the next power of two before running an IPA. If one wanted to run current IPAs on vectors that are not a power of two, this is the easiest way to achieve that. However, this defeats the attempt to lower the proof size, as it would now correspond to running an IPA on the size of the next power of two.

It is worth mentioning that using the folding method, as shown in Figure 2(b), results in the second recursive round being a size corresponding to a power of two. Therefore, the rest of the protocol runs as a general IPA without the actual need for splitting the vectors, which can also be seen in Listing 1.

4.2 CAAUrdleproofs

With the idea from Springproofs in mind, we have modified the IPA of Curdleproofs We refer to this modified protocol



(a) Padding Method

(b) Optimal Pre-Compression Method

Fig. 2: Folding visualization as seen in the Springproofs paper. Source: [7]

as CAAUrdleproof. For generality and readability, we show the split of vectors happening every round.

Prover computation

First of all, we have the prover computation, where the proof is constructed. The construction can be seen in Listing 2.

First, we have step 1, which is the setup phase. It is implemented the same way as in Curdleproofs. In line 2 the prover gets the cryptographic generators, $\mathbf{G}$, $\mathbf{G}'$ and H which are going to be used for commitment constructions. To ensure zero-knowledge, two blinding vectors for each commitment are constructed on lines 3–4. These are also given the properties, $(\mathbf{r}_C \times \mathbf{d} + \mathbf{r}_D \times \mathbf{c}) = 0$ and $\mathbf{r}_C \times \mathbf{r}_D = 0$, ensuring the completeness of the protocol.

After this, commitments to the blinding vectors are constructed as B_C and B_D on lines 5–6. These will eventually be used for verification by the verifier.

From the public input, hash values α, β are then computed on line 7. These are used to ensure the soundness of the protocol.

On lines 8–10, the two vectors are then blinded and multiplied by the α hash to ensure the zero-knowledge and soundness, as well as $H = \beta H$.

Now, the recursive proof construction, and step 2, begins. As explained, at the start of the recursive round, line 13, we find the split of the vectors on line 14, with $f(n)$ being the scheme function from Listing 1. Then, on line 15 we find half the length of the T set, as it is the set we are doing the recursive folding round on. Equally, on lines 16–19, we split our witness vectors and the group vectors using T and S.

After this, the prover constructs cross-commitment elements on lines 20–23 that are computed on the T set. These are added to the proof on line 24, which eventually is available to the verifier. They are also used to construct a hash value, γ_j, in the next step on line 25.

This value is used on lines 26–29 for completing the folding of $\mathbf{c}, \mathbf{d}, \mathbf{G}, \mathbf{G}'$. We do the fold as in the original Curdleproofs protocol while also appending the elements of S back onto the vectors. The figure shows a concatenation, but it is important to know that the vectors are appended together as shown in Figure 2(b).

```
1  Step 1: #Setup phase
2  (G, G', H) ←parse(crs_dl_inner)
3  r_C, r_D ←$ F^n #Vector blinders
4    where (r_C × d + r_D × c) = 0 and r_C × r_D = 0
5  B_C ← r_C × G #Blinder commitments
6  B_D ← r_D × G'
7  α, β ←Hash(C, D, z, B_C, B_C) #FS challenges
8  c ← r_C + αc #Blinded vectors
9  d ← r_D + αd
10 H ← βH
11 Step 2: #Recursive protocol
12 m ← ⌈log n⌉
13 while 1 ≤ j ≤ m :
14     T, S ← f(n) #Scheme function
15     n ← |T|/2
16     c ← c_T , cS ← c_S #Vector splitting
17     d ← d_T , dS ← d_S
18     G ← G_T , GS ← G_S
19     G' ← G'_T , GS' ← G'_T
20     L_{C,j} ← c_[:n] × G_[n:] + (c_[:n] × d_[n:])H #Cross–comm
21     L_{D,j} ← d_[n:] × G'_[:n]
22     R_{C,j} ← c_[n:] × G_[:n] + (c_[n:] × d_[:n])H
23     R_{D,j} ← d_[:n] × G'_[n:]
24     π_j ← (L_{C,j}, L_{D,j}, R_{C,j}, R_{D,j}) #Proof elements
25     γ_j ← Hash(π_j) #Folding challenges
26     c ← cS‖c_[:n] + γ_j^{-1} c_[n:] #Next round vectors
27     d ← dS‖d_[:n] + γ_j d_[n:]
28     G ← GS‖G_[:n] + γ_j G_[n:]
29     G' ← GS'‖G'_[:n] + γ_j^{-1} G'_[n:]
30     n ← |c|
31 Step 3: #Final proof element
32 c ← c_1
33 d ← d_1
34
35 return (B_C, B_D, π, c, d) # Elements for verifier
```

Listing 2: Prover computation for CAAU-IPA in CAAUrdleproofs

At the end of the recursive round, on line 30, n is

updated to the length of the concatenated vectors before starting a new round.

The result of this is a proof, π, constructed in $\lceil \log n \rceil$ rounds, but with the proof size being smaller than if the shuffle size was a power of 2.

In step 3, lines 31–35, the folded vectors of size 1 are added to the proof as values as well as the commitments to the blinding values, B_C and B_D. The proof, folded vectors, and updated commitment are saved for the verifier to use for verification.

The now constructed proof is then supposed to be added to the block in the chain at the given time slot [4].

Verifier computation

Having the proof on the blockchain allows for each validator to verify whether it is a valid proof asynchronously. Again, the initially proposed verification protocol has been modified according to Springproofs, as shown in Listing 3.

```
1  Step 1: #Setup phase
2  (G, G', H) ←parse(crs_{dl_inner})
3  (C, D, z) ←parse(φ_{dl_inner}) #Public input
4  (B_C, B_D, π, c, d) ←parse(π_{dl_inner}) #From prover
5  α, β ←Hash(C, D, z, B_C, B_D) #FS challenges
6  H ← βH
7  C ← B_C + αC + (α²z)H #Blinded commitments
8  D ← B_D + αD
9
10 Step 2: #Recursive round
11 m ← ⌈log n⌉
12 for 1 ≤ j ≤ m
13     T, S ← f(n) #Scheme function
14     n ← |T|/2
15     G = G_T , GS = G_S #Vector splitting
16     G' = G'_T , GS' = G'_T
17     (L_{C,j}, L_{D,j}, R_{C,j}, R_{D,j}) ←parse(π_j) #Proof elem
18     γ_j ←Hash(π_j) #Folding challenges
19     C ← γ_j L_{C,j} + C + γ_j^{-1} R_{C,j} #Update comms
20     D ← γ_j L_{D,j} + D + γ_j^{-1} R_{D,j}
21     G ← GS‖G_{[:n]} + γ_j G_{[n:]} #Next round vectors
22     G' ← GS'‖G'_{[:n]} + γ_j^{-1} G'_{[n:]}
23     n ← |G|
24
25 Step 3: #Final check
26 Check C = c × G_1 + cdH #Initial ?= Folded
27 Check D = d × G'_1
28 return 1 if both checks pass, else return 0
```

Listing 3: Verifier computation for CAAU-IPA in CAAUrdleproofs

Many of the changes to the verifier protocol are equivalent to the ones made to the prover protocol.

In step 1, the verifier sets up to run the protocol.

The verifier, on line 2, gets the same cryptographic generators used by the prover, $\mathbf{G}$, $\mathbf{G}'$ and H, from the common reference string. Then, from the public input ϕ, line 3, the verifier gets hold of the original commitments, C and D, as well as the result of the inner product between c and d. From the prover's proof, on line 4, the verifier gets the blinding commitments B_C and B_D, the proof elements π, and the folded vector values c and d. With this, the verifier can compute the same α and β challenges as the prover in line 5, as well as computing the same H generator on line 6.

Now, the verifier updates the commitments C and D on lines 7–8. The reason is that on lines 7–8 in Listing 2, the witness vectors are updated to be both blinded and multiplied by the challenge, α. Those modifications mean that the commitments C and D need to be commitments to the modified witnesses instead.

The setup phase is now complete, and the verifier then executes the recursive protocol, as shown in step 2. First, the vectors are divided into two sets, T and S, on line 13, as in Listing 2. After this, the group vectors are in lines 14–16 split according to those sets, along with updating n to be half the size of T.

The verifier then, on line 17, retrieves from the proof the cross-product commitment update values for the given round, $L_{C,j}, L_{D,j}, R_{C,j}, R_{D,j}$. These are used for constructing a new commitment, lines 19–20, according to the fold made at round i.

By fetching the cross-commitments of the round, the verifier can compute a challenge γ_j, line 18, made from the same public inputs as the prover.

The corresponding left and right side cross-products are then, in lines 19–20, also multiplied by said challenge, γ_j, γ_j^{-1}, respectively. By this time, the C and D commitments are a commitment to the original commitments, along with the folded commitment.

$\mathbf{G}, \mathbf{G}'$ are on lines 21–22 updated as in Listing 2 before the protocol on line 23 updates n to be the length of the newly constructed vectors.

As in the prover protocol, this is then repeated for $\lceil \log n \rceil$ rounds, after which the vectors have length 1.

At the end of the protocol, in step 3, the verifier now does its final check. From the prover, line 4, it has received the folded down c and d vectors. It, therefore, constructs commitments with those elements. So, it constructs $c \times G_1 + cdH$ on line 26, which is the structure of the C commitment, as well as $d \times G'_1$ on line 27, which is the structure of the D commitment. The verifier now checks if these commitments match the commitments that were constructed in the recursive part of the protocol. If so, the verifier accepts the proof.

Theorem 1. *CAAUrdleproofs is a zero-knowledge argument of knowledge when* $|\ell| \geq 8$.

4.3 Shuffle security

The shuffle method proposed by Larsen et al. [16] that is used in Curdleproofs is based on the idea of shuffling a list of proposers over a set of slots. A formal definition of the shuffle is given in Figure 3.

Here the set $(c_1, \ldots, c_n)$ is a set of ciphertexts that are shuffled over T slots. In each slot t, a subset of the ciphertexts $i_1, \ldots, i_\ell$ is chosen randomly, shuffled and added back to the list of ciphertexts. The shuffler then re-encrypts the ciphertexts and publishes them. This process is repeated for T slots, and the shuffle is complete. During the T shuffles, some shuffles may be adversarial. An adversary can choose to do anything with its shuffle, including not shuffling. Hence, an adversarial shuffle can be seen as no shuffling

$\Pi(c_1, \ldots, c_n)$

For $t \in [T]$ **:**

S_t picks random $\{i_1, \ldots, i_\ell\} \subset [n]$
S_t computes $(\tilde{c}_{i_1}, \ldots, \tilde{c}_{i_\ell}) \leftarrow \text{Shuffle}(c_{i_1}, \ldots, c_{i_\ell})$
S_t publishes $(\tilde{c}_{i_1}, \ldots, \tilde{c}_{i_\ell})$

Fig. 3: Distributed shuffling protocol. Source: [16]

being done. Therefore, the number of honest shuffles that happen during the shuffle process is $T_H = T - \beta$, where β is the number of adversarial shuffles.

The adversary can also track ciphertexts. For instance, if the adversary owns some of the ciphertexts. Those tracked ciphertexts are denoted by α, which is $\leq n$.

The shuffle is secure if none of the following two events occur. The first event is a short backtracking, where an adversary can find the original ciphertexts from the shuffled ciphertexts. Since the subsets of ciphertexts are chosen randomly in each shuffle, if there are enough adversarial shufflers in a row at the end of the process, then a short backtracking is possible.

The second event that can occur is related to the fact that every shuffle distributes the probability of a specific ciphertext being in a particular slot. So, if a shuffle contains many ciphertexts with a larger-than-average chance of containing a specific ciphertext, then that would imply that there is a higher chance of that ciphertext being in that slot.

It is theoretically possible to find a number of shuffles, given a shuffle size and a number of adversarial shufflers, which guarantees that the shuffle is secure. For any $0 < \delta < 1/3$, if $T \geq 20n/\ell \ln(n/\delta) + \beta$ and $\ell \geq 256 \ln^2(n/\delta)(1-\alpha/n)^{-2}$. If T and ℓ are chosen such that the above two conditions are met, then the protocol is an (ϵ, δ)-secure (T, n, ℓ)-shuffle in the presence of a (α, β)-adversary where $\epsilon = 2/(n - \alpha)$.

This formula is the lowest theoretically proven bound for T and ℓ. Plotting numbers relevant to Whisk will show that this theoretical bound is too large to use for argumentation of security. It is, however, possible to find lower secure values for T and ℓ, but this has to be done experimentally.

4.4 Implementation

Implementing the above-explained CAAUrdleproofs protocol required some optimizations made by Curdleproofs to have the code run as fast as possible. These are explained in the following with a focus on how CAAUrdleproofs differentiates itself from Curdleproofs. Both our implementation of CAAUrdleproofs and the experiment involving the security of the shuffle are publicly available on GitHub [2]. The implementation of CAAUrdleproofs is a fork of and builds directly on the already existing Curdleproofs code.

2. https://github.com/AAU-Dat/curdleproofsplus/tree/SIPA

4.4.1 CAAUrdleproofs

The protocol in Curdleproofs [6] introduces a lot of multiscalar multiplications. As such, CAAUrdleproofs also introduces these multiplications, allowing for checking calculations of the form:

$$C \stackrel{?}{=} \mathbf{x} \times (\mathbf{g} \| \mathbf{h} \| G_T \| G_U \| H \| \mathbf{R} \| \mathbf{S} \| \mathbf{T} \| \mathbf{U}) \tag{1}$$

As explained by Curdleproofs, the verifier computation can be significantly optimized by checking the multiscalar multiplications as a single check at the end of the protocol instead.

CAAUrdleproofs differs on this topic regarding the IPAs SamePerm and SameMSM. In each recursive round, both the folded vectors and the commitments are multiplied by verification scalars, γ_j. To keep track of which elements of the vectors are multiplied by each γ_j, a function called `get_verification_scalars_bitstring` is used. The output of this function is a list of length ℓ, each element with a list corresponding to the rounds in which γ_j was multiplied to the element. Curdleproofs' implementation is simpler than CAAUrdleproofs' in this case. As Curdleproofs only works on powers of two, it is always the right half of the vectors in each round that are multiplied by the challenge.

The multiplication of challenges on each element is not as easily trackable in the CAAUrdleproofs protocol. Here, it is necessary to simulate a run through the recursive protocol. However, this should not have a significant impact on performance, as it is run over vectors of small integers and never actually requires any multiplications. It is used as a measuring tool.

The protocol used in the implementation is illustrated in Listing 4. A list, `ActivePos`, on line 32, keeps track of the original index placement and its position after each fold. By doing this, we can run the recursion and find the correct challenges for each index while still knowing what the original index was. A bit matrix, $b_{i,j}$, is constructed as in Curdleproofs, such that the vector, $\mathbf{s}$, is made in the same way for both protocols.

The vector, $\mathbf{u}$, seen on line 3, is used for optimization in the grand product argument rather than $\mathbf{G}'$, and the `AccumulateCheck` function, on lines 21 and 23, is used for the multiscalar multiplication optimization. For a thorough explanation of these, we refer to Curdleproofs [6].

In Curdleproofs, both the SamePerm and SameMSM proof are recursive IPAs. So, the modifications and optimization used on the SamePerm argument are also used on the SameMSM argument. The modifications include the split into set T and S before recursion and the construction of the bit matrix, $b_{i,j}$, to keep track of multiplications on individual elements.

It is also worth noting that the concatenation of T and S in the recursive phase, lines 26–29 in Listing 2, is handled effectively in the code. Instead of concatenating, the computation uses pointers to the original vector, so it never practically concatenates.

The code also uses the fact that the used scheme function will always end up with vectors being a power of two after the first round. So, after the first round of recursion, we use the original algorithm code from Curdleproofs to run the rest of the protocol.

Step 1: #Setup phase
$(\mathbf{G}, H) \leftarrow \text{parse}(crs_{dl_{inner}})$
$(C, D, z, \mathbf{u}) \leftarrow \text{parse}(\phi_{dl_{inner}})$
$(B_C, B_D, \pi, c, d) \leftarrow \text{parse}(\pi_{dl_{inner}})$
$\alpha, \beta \leftarrow \text{Hash}(C, D, z, B_C, B_D)$ #FS challenges

Step 2: #Recursive phase
$m \leftarrow \lceil \log n \rceil$
for $1 \leq j \leq m$
  $T, S \leftarrow f(n)$ #Scheme function
  $n \leftarrow \frac{|T|}{2}$
  $(L_{C,j}, L_{D,j}, R_{C,j}, R_{D,j}) \leftarrow \text{parse}(\pi_j)$ #Proof elem
  $\gamma_j \leftarrow \text{Hash}(\pi_j)$ #Folding challenges
  $n \leftarrow n + \text{len}(S)$

Step 3: # Accumulated checking phase
CP: $\gamma \leftarrow (\gamma_m, ..., \gamma_1)$ #Construction difference
CAAUP: $\gamma \leftarrow (\gamma_1, ..., \gamma_m)$
Compute **s***: see below for difference*

$\text{AccumulateCheck}(\gamma \times \mathbf{L}_C + (B_C + \alpha C + (\alpha^2 z)H)$
  $+\gamma^{-1} \times \mathbf{R}_C \stackrel{?}{=} (c\mathbf{s} \| cd\beta) \times (\mathbf{G} \| H))$
$\text{AccumulateCheck}(\gamma \times \mathbf{L}_D + (B_D + \alpha D)$
  $+\gamma^{-1} \times \mathbf{R}_D \stackrel{?}{=} d(\mathbf{s}' \circ \mathbf{u}) \times \mathbf{G})$
return 1

s-step Curdleproofs:
for $1 \leq j \leq n$: Simulate halving each **round**
  $s_i = \sum_{j=1}^{m} \delta_j^{b_{i,j}}$, $b_{i,j} \in \{0, 1\}$ s.t. $i = \sum_{j=1}^{m} b_{i,j} 2^j$
  $s'_i = \sum_{j=1}^{m} \delta_j^{-b_{i,j}}$
s-step CAAUrdleproofs:
$ActivePos \leftarrow [(i, i), i = 1, \ldots, n]$ #Pos after round
for $1 \leq j \leq m$:
  $h \leftarrow \frac{2^{\lceil \log n \rceil}}{2}$
  $f \leftarrow n - h$
  $nf \leftarrow h - f$
  $fs \leftarrow \frac{nf}{2}$
  for (i, k) **in** $ActivePos$:
    if $k \geq h$: #Elem has challenge j
      $b_{i,j} \leftarrow 1$
      $newPos = k - h - fs$
    else: #Elem has no challenge j
      $b_{i,j} \leftarrow 0$
      $newPos = k$
    $nextActivePos.push((i, newPos))$
  $ActivePos \leftarrow nextActivePos$ #New positions
  $n \leftarrow h$
for $1 \leq j \leq n$: #Same as Curdleproofs
  $s_i = \sum_{j=1}^{m} \delta_j^{b_{i,j}}$
  $s'_i = \sum_{j=1}^{m} \delta_j^{-b_{i,j}}$

Listing 4: Optimized verifier computation for CAAU-IPA in CAAUrdleproofs

4.4.2 Shuffle Security

As mentioned in section 4.3, the theoretically proven bound on the necessary number of shuffles to ensure security is too high. Hence, as also done in [16], we implement an experiment to find the bounds where the shuffle is secure. The goal of the experimental code is to find the number of honest shuffles required for security.

We inherit the terminology introduced by Larsen et al. [16] and interpret each ciphertext as a cup that can contain water. Each cup contains an amount of water between 0 and 1.

An experiment run starts with the first cup being full and the rest being empty. As mentioned, α cups are tracked by an adversary; the first $n - \alpha$ cups are called active cups, while the last α cups are tracked. So, at each shuffle, the shuffler randomly picks ℓ ciphertexts and shuffles them. Meanwhile, an average of the water between the active cups of the ℓ-shuffle is found. All active cups are given this amount of water.

Now, after each shuffle, if any cup has more than $2/(n - \alpha)$ water, its position can be predicted by the adversary. Hence, the shuffle is insecure [16]. If a position can be predicted, another round of shuffling is performed. This method is used until no cup exceeds the threshold, after which the shuffle is deemed secure.

The experiment indicates the number of rounds before the shuffle was secure.

By repeating this experiment for several runs, one can experimentally say when a shuffle with given parameters is secure.

4.4.3 Size reduction

If we reduce the shuffle size used in Whisk and still prove it secure, then we expect to see a reduction in the size overhead on the blockchain.

We first set our focus on Curdleproofs, as this is the protocol we have modified directly. As mentioned in section 2.2, the size of Curdleproofs is $(18 + 10\log(\ell + 4))\mathbb{G} + 7\mathbb{F}$. The dependence on the $\log$ stems from the number of recursive rounds that take place in the SamePerm and SameMSM proofs. The addition of four elements in the $\log$ stems from the protocol needing those as blinders. Hence, at a proof of size 128, ℓ is 124. In the proof of theorem 1, see Appendix B, we show that CAAUrdleproofs is $\mathcal{O}(\log n)$, which is the same as Curdleproofs. However, as discussed in section 4.2, CAAUrdleproofs' IPA proofs use $\lceil \log n \rceil$ recursive rounds. This means that the size of CAAUrdleproofs must be $(18 + 10\lceil \log(\ell + 4) \rceil)\mathbb{G} + 7\mathbb{F}$.

CAAUrdleproofs, therefore, has the same proof size as Curdleproofs.

The CAAUrdleproofs modification can still reduce the overall block size overhead. By using the overhead calculation described by Whisk on CAAUrdleproofs, it measures a block overhead of 16.656 KB when the shuffle size is 128 [4]. Note that this is the same size as Curdleproofs, as the shuffle size is a power of 2. The provided calculation of the block overhead is provided as the following, where $\mathbb{G} = 48$ bytes and $\mathbb{F} = 32$ bytes[3]:

- List of shuffled trackers ($\ell \cdot 96 \Rightarrow$ eg. $124 \cdot 96 = 11,904$ bytes).

3. As noted in the code on the Curdleproofs GitHub repository: https://github.com/asn-d6/curdleproofs/blob/main/src/whisk.rs. Accessed: 26/05/2025

- Shuffle proof ($(18+10\lceil\log(\ell+4)\rceil)\mathbb{G}+7\mathbb{F} \Rightarrow$ eg. $(18+10\lceil\log(124+4)\rceil)\cdot 48+7\cdot 32 = 4,448$ bytes).
- A fresh tracker (two BLS G1 points $\Rightarrow 48\cdot 2 = 96$ bytes).
- A new commitment $com(k)$ to the proposer's tracker (one BLS G1 point $\Rightarrow 48$ bytes).
- A Discrete Logarithm Equivalence Proof on the ownership of the elected proposer's commitment (two G1 points, two Fr scalars $\Rightarrow 2\cdot 48+2\cdot 32 = 160$ bytes).

The majority of the block overhead comes from the list of shuffled trackers. Hence, as the list size is heavily dependent on ℓ, using CAAUrdleproofs could majorly decrease the block overhead by allowing ℓ to be more flexibly chosen as a smaller size than 128.

5 Experimental Protocol

In this section, we will describe how our experiments are run and what we want to measure. We also discuss which parameters we can adjust in the various experiments we have.

The experiments are run on a virtual machine hosted on Strato CLAAUDIA at Aalborg University. The machine uses an Intel Xeon Cascadelake processor, CPU family 6, model 85. It features 16 virtual CPUs (VCPUs) with 3.092 GHz and 64 GB of RAM. One VCPU does not correspond to one physical CPU but rather to a hardware thread within the used CPU. Additionally, the VCPUs are over-provisioned, which means that the computation time may be shared with other virtual machines on the same server[4]. The virtual machine is running Ubuntu Server 24.

5.1 CAAUrdleproof

In this experiment, we measure the time to run the CAAUrdleproofs protocol. The results will be compared to those of Curdleproofs, which we re-run on our hardware. As Curdleproofs already has a Rust benchmark implemented, we will be using that same benchmark for both protocols. The parameter that we want to change between benchmark runs is the shuffle size, ℓ.

In CAAUrdleproofs, we will test the protocol with $\ell = \{8, 9, \ldots, 256\}$.

Since Curdleproofs is unable to run benchmarks, unless the shuffle size is a power of two, those benchmarks will be run on values $\ell = 2^N$, where $N = \{3, 4, 5, 6, 7, 8\}$.

5.2 Shuffle security

In this experiment, we run the shuffle protocol with varying shuffle sizes and varying numbers of adversarial tracked ciphertexts. The purpose of this experiment is to find the lowest possible shuffle size that is still secure against adversarial tracking. We, therefore, run the experiment with shuffle sizes, ℓ, between 32 and 512. For the number of adversarial tracked ciphertexts, we use the values $\alpha = \{1/2, 1/3, 1/4\}$

Because Curdleproofs is meant to be used in an Ethereum setting, a maximum of 8,192 shuffles is available. However, we will keep running the shuffling experiments until they are deemed secure. Additionally, the experiments are conducted with a set of 16,384 ciphertexts. Both of these numbers come from the Ethereum Whisk proposal [4]. Every experiment is run 1,000 times to avoid statistical uncertainty.

4. As explained in an earlier documentation of Strato: https://github.com/aau-claaudia/Documentation/blob/0d40577ed757c5e9640109f5aac5a7f0a36b7f85/docs/guides/performance/performance.md. Accessed: 10/06/2025

6 Results

6.1 Proving and Verifying Times

After running the experiment comparing Curdleproofs and CAAUrdleproofs across different shuffle sizes, we obtained the results shown in Figure 4.

As mentioned in section 5.1, CAAUrdleproofs was run with a shuffle size $\ell = \{8, 9, \ldots, 256\}$. Curdleproofs was only run with a shuffle size $\ell = 2^N$, where $N = \{3, 4, 5, 6, 7, 8\}$, as it is only able to run in powers of two. Hence, the results for Curdleproofs show that the shuffle size, ℓ, instantly increases to the next power of two because it theoretically would have to pad the input set until it reaches the next power of two.

From the results, we can see that CAAUrdleproofs and Curdleproofs have similar proving and verifying times when ℓ is a power of two. However, when ℓ is not a power of two, CAAUrdleproofs is faster. When ℓ is below a power of two, we observe that the performance advantage of CAAUrdleproofs over Curdleproofs increases as ℓ decreases.

The results for the verifying time also show that the verifying time jumps up quite significantly the first four times it reaches above a power of two. However, this is not the case, at least not as aggressively, when increasing ℓ from 128. We find, however, that the bump is smaller the higher ℓ is.

In addition to the proving and verifying times, the time used on shuffling is also lower for any ℓ that is not a power of two; see Appendix D. However, that was to be expected since CAAUrdleproofs uses the same shuffling algorithm as Curdleproofs but does not have to add additional padding to the non-power of two input sizes.

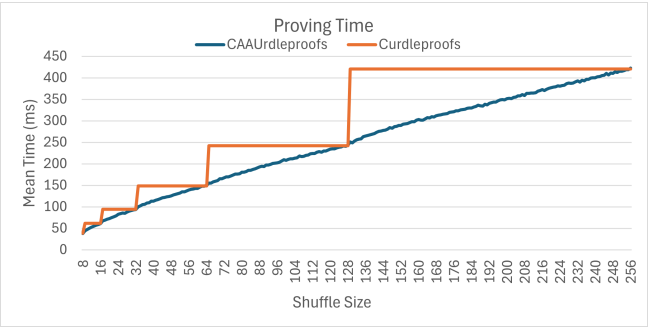
6.2 Shuffle security

The results of the shuffle security experiment are shown in Figure 5.

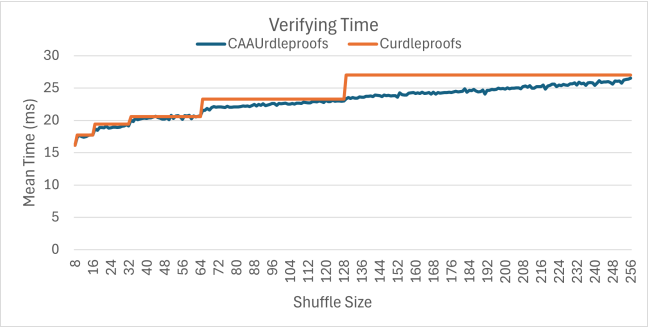
Figure 5 shows the mean of the 1,000 runs of each shuffle size ℓ and the standard deviation.

We can see that the bigger the shuffle size ℓ is, the less honest shuffles are necessary to make the shuffle secure. In Ethereum, each shuffling phase is limited to 8,192 shuffles, meaning that the maximum number of honest shuffles that can be used is 8,192. Therefore, the results of the experiment find $T_H = T - \beta$. T_H is how many of the T shuffles available during the shuffling phase are needed to be honest. The rest is, therefore, the maximum number of dishonest shuffles allowed, β. We also see that the bigger the shuffle size, the narrower the standard deviation gets.

From the results of the experiment, with $\alpha = 8,192$, we can see that the number of honest shuffles necessary to make the shuffle secure sharply decreases until the size of $\ell = 64$, and then it starts to level out. We can see that with a size of $\ell = 75$, we need about $1/3$ of the shuffles to be honest to

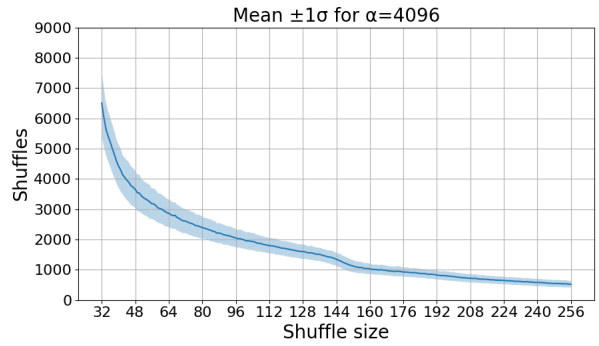


(a) Proving Time

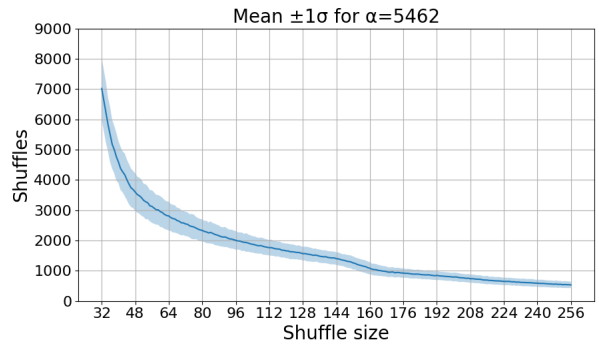


(b) Verifying Time

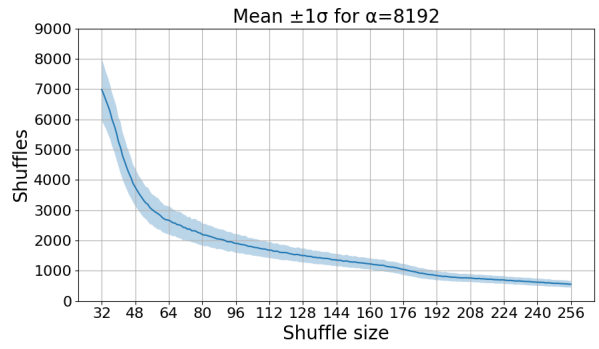
Fig. 4: The timed results compared between CAAUrdleProofs and Curdleproofs



(a)

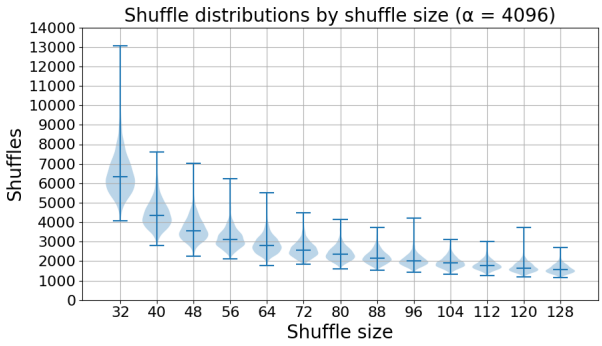


(b)

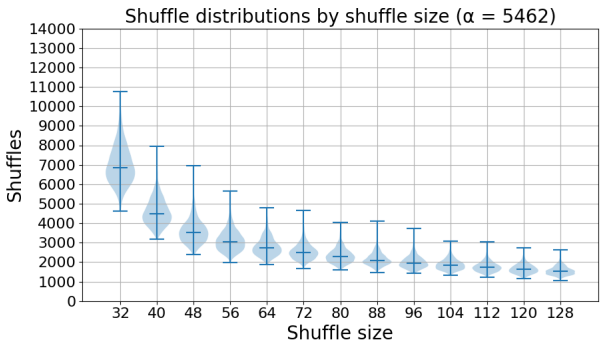


(c)

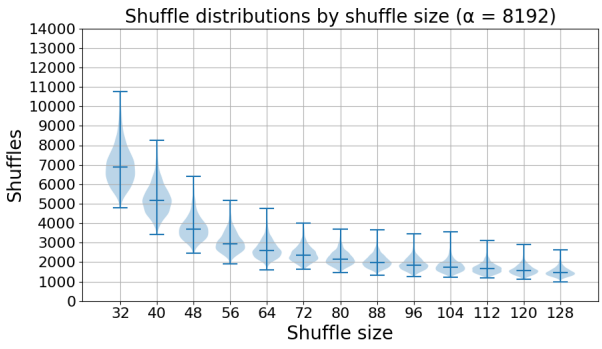
Fig. 5: Results of shuffle security experiment showing mean amount of honest shuffles necessary with one standard deviation



(a)



(b)



(c)

Fig. 6: Results of shuffle security experiment showing spread of nessecary shuffles needed for shuffle to be secure

make the shuffle secure. Likewise, we see that, at $\ell = 108$, we need about $1/4$ of the shuffles to be honest to make the shuffle secure.

In general, all three of the experiments, despite the difference in α, show the same trend. They all level out,

but the higher α is, the lower the leveling occurs, and the later it happens as well. There are two things, however, that are different between the experiments. At $\alpha = 4,096$, we see that with $\ell = 32$, the mean number of honest shuffles necessary to make the shuffle secure is ~ 500 lower than the two other α values. As ℓ increases, the mean number of honest shuffles necessary to make the shuffle secure becomes similar to the other α values. Another thing that differs between the experiments is that they all have a sudden dip in higher ℓ values in the experiment. Here, we observe a trend that the lower the α is, the earlier the dip occurs.

The results in Figure 6 show that for all three α values, the spread of the necessary honest shuffles tightens the larger the shuffle size ℓ gets. Like the results in Figure 5, Figure 6 also shows that the bigger a shuffle size ℓ, the less honest shuffles, on average, are necessary to make the shuffle secure.

We can also see that the widest point of the violin plot is below the mean, meaning the outliers are a lot more significant above the mean than below it.

It is worth noting that there is a spike in the distribution of the necessary honest shuffles at $\ell = 32$ for $\alpha = 4,096$. This spike is not present for the other two α values and is likely due to the probabilistic nature of the shuffling method.

Another notable aspect is that in the Ethereum setting, the maximum number of shuffles available is 8,192. Therefore, in the cases where more than the 8,192 shuffles were necessary to make the shuffle secure, the shuffle would not have been secure within the Ethereum setting. Hence, it is also possible to see the experiment as running 1,000 days worth of each shuffle size ℓ and then seeing how many of those days would have been secure. We found that the first size of ℓ that could have been secure for the entire duration of the experiment would be $\ell = 42$ for $\alpha = 8,192$ and $\ell = 40$ for $\alpha = 5,462$ and $\alpha = 4,096$.

7 Discussion

In this section, we will discuss the results of the experiments in section 6 and how they relate to the CAAUrdleproofs protocol. We will also discuss some of the limitations of the CAAUrdleproofs protocol and how it compares to Curdleproofs.

7.1 CAAUrdleproofs in comparison to Curdleproofs

As mentioned in section 6.1, the proving and verifying times between the two protocols are close to identical when ℓ is a power of two. This is because the added computation is negligible compared to the other computations present in the original Curdleproofs protocol.

On the prover, there is the addition of the scheme function from Springproofs. However, as seen in Listing 1, the scheme function only performs integer calculations based on n and hence should have a negligible impact compared to the cryptographic group computations. Additionally, as mentioned in section 4.4, the vector is never practically split in two; instead, it uses pointers. Therefore, we avoid having to add new variables to memory in every round.

Also, we mentioned in section 4.4 that every round after the first runs the same code as Curdleproofs. Thus, only the first round should be able to introduce some computational overhead. However, as mentioned before, the overhead should be negligible.

The same kind of explanation can be used to describe the same scenario at powers of two on the verifier side. Looking at Listing 4, we see that the only difference in computation between CAAUrdleproofs and Curdleproofs stems from the calculation of s. Comparing line 29 and lines 33–47, it becomes clear that both ways work in $\mathcal{O}(n \log n)$, as they do m computations for each of the n elements. CAAUrdleproofs does, however, require additional integer variables for splitting, as well as an array to keep track of the active positions of the vector elements during recursion. Nevertheless, Figure 4(b) shows that this does not have a big, if any, impact on the running time.

However, as mentioned in section 6.1, when ℓ is just above a power of two, we observe some more aggressively increasing verification times. We expect this to be a result of m being set to $\lceil \log n \rceil$ in line 8 of Listing 4. For, when ℓ is 65, computations for an additional round are added, compared to when ℓ is 64. This explains why the running time flattens when ℓ is increased. The pattern shows that the bump has a decreasing impact on the running time as ℓ increases. In theory, the extra recursive round should introduce a constant amount of work for the verifier. Therefore, we believe the bump to be an artifact of memory optimizations. For instance, pre-fetching, where the memory system can optimize access if it suspects some values in memory are going to be reused[5]. As ℓ increases, the memory system will have more data to predict and optimize memory access.

7.2 Shuffle Security

When looking at the results of the shuffle security experiment in Figure 5 and Figure 6, we can see that when taking into account the standard deviation, the shuffle can still be secure with an ℓ as low as 32 within the 8192 shuffles available. Even when taking into account the worst-case scenario from our experiment, the shuffle will still be secure with an ℓ as low as 42 within the 8192 shuffles available with an α of 8192.

We do not recommend using an ℓ lower than 80, as in this case, the worst-case scenario requires a little under half of the available shuffles to be honest, in order to be secure. As seen in Figure 5, the protocol would also only need a third of the 8192 shuffles to be honest to get within the standard deviation. Lowering the shuffle size to 80 would still lead to a reduction of the proving time of 62.69 ms, which is 74.25% of the current Curdleproofs time, and a reduction in the verifying time of 0.89 ms, which is 96.11% of the current Curdleproofs time. It would also reduce the block overhead size from 16.656 KB to 12.048 KB. The reduced size is only 72.33% of the current size for Curdleproofs, which would result in saving $\sim$ 12.11 GB of space on the blockchain each year. Some other factors to consider when determining the number of honest shuffles required

5. https://doc.rust-lang.org/core/arch/aarc.h64/fn._prefetch.html — Accessed: 29/05/2025

to secure the shuffle are that there are additional elements that can impact the blockchain's security. One such element is the known attacks that exploit the ability to control a large number of validators. Attacks like the $\geq 50\%$ stake attack and the 33% finality attack [3] take advantage of controlling a large number of validators in order to affect the blockchain system negatively. Because of attacks like these, which rely on controlling a large number of validators, we recommend that when evaluating how many honest shuffles are necessary to make the shuffle secure, one should also consider how many honest validators are required to secure the blockchain.

Another thing to keep in mind is that within the Ethereum system, not every validator is owned by a different person. Some nodes contain multiple validators, and this means that during the shuffling phase when selecting the 16,384 possible proposers, there is a chance that a single node controls multiple of the chosen validators. Likewise, it is also possible during the selection of the shufflers.

From the results in Figure 5, we see that the mean starts higher and ends lower for the experiments with a higher α. One reason for this could be the relationship between the number of adversarial tracked cups and the threshold required before the shuffle is secure. Since the threshold is $2/(n-\alpha)$, the higher α is, the higher the threshold for the amount of water allowed in any cup, see section 4.4. Therefore, the higher α is, the harder it is to get the water divided into the honest cups. The reason is that the distribution only happens in honest cups. More adversarial cups means less honest cups to distribute the water into. Hence, there potentially is a higher amount of water in the chosen cups after a shuffle when α is higher.

8 Conclusion

After examining the ZK SSLE protocol, Whisk, and the Curdleproofs protocol, we found that there was still room for improvement in the Curdleproofs protocol. We identified the strict requirement of the shuffle size being a power of two as a limitation, and we aimed to remove this limitation to reduce the block overhead related to the protocol.

To achieve this, we drew inspiration from Springproofs, which allows IPAs to be of any size. By combining the Curdleproofs protocol with the flexibility of Springproofs, we made the CAAUrdleproofs protocol. The implementation of the CAAUrdleproofs protocol is a modified version of the Curdleproofs protocol that allows for any shuffle size.

Through our experiments, we found that the CAAUrdleproofs protocol has similar proving and verifying times to the Curdleproofs protocol when the shuffle size is a power of two. However, for any shuffle size that is not a power of two, the CAAUrdleproofs protocol has a performance advantage. An advantage that increases the more below a power of two the shuffle size is.

Since CAAUrdleproofs enables the use of any shuffle size, it can be used to reduce the block overhead related to the protocol without compromising the security of the protocol.

We have shown the security through an experiment inspired by [16]. Here, we found that the shuffle size can be reduced to 80 and remain secure, also considering the domain in which the protocol is intended to operate. Using this, we see a block size overhead of 72.33% compared to that of Curdleproofs.

Hence, we have shown CAAUrdleproofs to be an optimized modification of Curdleproofs, as it allows for more flexibility in the choice of shuffle size. The optimization is based on reducing the size of the block overhead and achieving faster proving and verifying times.

9 Future work

In this section, we will focus on areas where the Whisk protocol still has room for improvement.

The main modification from Curdleproofs to CAAUrdleproofs is the added flexibility in choosing the shuffle size for Whisk. Hence, a topic for future improvements could be proof structure modifications. The goal of this is to improve the protocol in all cases, including those where the shuffle size is a power of two, for which Curdleproofs and CAAUrdleproofs yield similar results. As shown in Appendix C, we attempted to achieve this using WIPAs instead of IPAs. However, there was not enough time to follow through, as it seemed that significant structural changes were needed for this change to be possible.

Besides trying to make the proof faster and reduce the block overhead, there are also calls for making the protocol more secure. Specifically, work has already begun trying to make Curdleproofs post-quantum secure [21]. In this work, they make use of the isogeny-based protocol Commutative Supersingular Isogeny Diffie-Hellman (CSIDH) [22]. Isogeny-based cryptography is based on maps between elliptic curves. Using isogenies, a hard problem arises, namely the Group Action Inverse Problem (GAIP).

Definition 4 (Group Action Inverse Problem (GAIP))**.** *Given a curve E, with $End(E) = O$, find an ideal $a \subset O$ such that $E = [a]E_0$*

This problem bears some resemblance to the discrete logarithm problem. Hence, using this problem, an almost one-to-one conversion using post-quantum cryptography can be done on Whisk, as shown by Sanso [21]. Currently, however, there does not exist a Non-Interactive Zero-Knowledge (NIZK) proof of shuffle based on isogenies.

When using Whisk in the Ethereum blockchain, a list of upcoming proposers is still chosen and published some time before they are needed for duty. However, because upcoming proposers are published as trackers that can only be opened and proven by the chosen validator, attacks such as DoS attacks are significantly harder to perform accurately. Though, the first part of the proposer DoS attack involves de-anonymizing validators, as demonstrated by Heimbach et al. and confirmed by our research [1, 2]. Even if the blockchain is using Whisk, it is still possible for an adversary to gather and de-anonymize validator IP addresses only by running a node on the network. A sustainable solution for this, therefore, needs to be found. However, Ethereum is a system that encourages transparency, so a possible solution should take this into account.

10 Acknowledgements

We want to express our sincere gratitude to Daniele Dell'Aglio and Michele Albano for their supervision and guidance throughout this thesis.

We also want to thank the authors of Springproofs [7] for providing their code, which we used as a reference when building CAAUrdleproofs in this thesis. We also thank the authors of Larsen et al. [16] for their great help in providing answers to our questions regarding their paper on shuffle security.

We also acknowledge the usage of AI tools such as ChatGPT, GitHub Copilot, and Grammarly. These have been used for clarification and implementation purposes.

REFERENCES

[1] L. Heimbach, Y. Vonlanthen, J. Villacis, L. Kiffer, and R. Wattenhofer, *Deanonymizing ethereum validators: The p2p network has a privacy issue*, 2024. arXiv: 2409.04366 [cs.CR].

[2] A. M. Jakobsen and O. Holmgaard, *"denial of validator anonymity: A de-anonymization attack on ethereum", computer science 9 project*, 2025.

[3] ethereum.org, "Ethereum proof-of-stake attack and defense," 2024, Accessed: 22-10-2024.

[4] G. Kadianakis, "Whisk: A practical shuffle-based ssle protocol for ethereum," 2024, Accessed: 16-05-2025.

[5] D. Boneh, S. Eskandarian, L. Hanzlik, and N. Greco, "Single secret leader election," in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, ser. AFT '20, New York, NY, USA: Association for Computing Machinery, 2020, pp. 12–24, ISBN: 9781450381390. DOI: 10.1145/3419614.3423258.

[6] T. E. F. C. R. Team, "Curdleproofs," Accessed: 24-04-2025.

[7] J. Zhang, M. Su, X. Liu, and G. Wang, "Springproofs: Efficient inner product arguments for vectors of arbitrary length," in *2024 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2024, pp. 3147–3164.

[8] D. Boneh, "The decision diffie-hellman problem," in *Algorithmic Number Theory*, J. P. Buhler, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 48–63, ISBN: 978-3-540-69113-6.

[9] ethereum.org, "Secret leader election," 2024, Accessed: 22-10-2024.

[10] C. A. Neff, "A verifiable secret shuffle and its application to e-voting," in *Proceedings of the 8th ACM Conference on Computer and Communications Security*, ser. CCS '01, Philadelphia, PA, USA: Association for Computing Machinery, 2001, pp. 116–125, ISBN: 1581133855. DOI: 10.1145/501983.502000.

[11] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit, *Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting*, Cryptology ePrint Archive, Paper 2016/263, 2016.

[12] D. Boneh, A. Partap, and L. Rotem, *Post-quantum single secret leader election (SSLE) from publicly re-randomizable commitments*, Cryptology ePrint Archive, Paper 2023/1241, 2023.

[13] P. W. Foundation, "Safrole," Accessed: 16-05-2025.

[14] P. W. Foundation, "Sassafras," Accessed: 23-05-2025.

[15] J. Håstad, "The square lattice shuffle," *Random Structures and Algorithms*, vol. 29, no. 4, pp. 466–474, 2006.

[16] K. G. Larsen, M. Obremski, and M. Simkin, *Distributed shuffling in adversarial environments*, Cryptology ePrint Archive, Paper 2022/560, 2022.

[17] T. E. F. C. R. Team, "Privacy analysis of whisk," Accessed: 15-05-2025.

[18] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *2018 IEEE symposium on security and privacy (SP)*, IEEE, 2018, pp. 315–334.

[19] H. Chung, K. Han, C. Ju, M. Kim, and J. H. Seo, "Bulletproofs+: Shorter proofs for a privacy-enhanced distributed ledger," *Ieee Access*, vol. 10, pp. 42 081–42 096, 2022.

[20] L. Eagen, S. Kanjalkar, T. Ruffing, and J. Nick, "Bulletproofs++: Next generation confidential transactions via reciprocal set membership arguments," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2024, pp. 249–279.

[21] A. Sanso, "Towards practical post quantum single secret leader election (ssle) - part 1," 2022, Accessed: 02-06-2025.

[22] W. Castryck, T. Lange, C. Martindale, L. Panny, and J. Renes, "Csidh: An efficient post-quantum commutative group action," in *Advances in Cryptology – ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part III*, Brisbane, QLD, Australia: Springer-Verlag, 2018, pp. 395–427, ISBN: 978-3-030-03331-6. DOI: 10.1007/978-3-030-03332-3_15.

APPENDIX A
DEFINITIONS OF ZERO-KNOWLEDGE ARGUMENT OF KNOWLEDGE

The following definitions are the exact ones also used by Springproofs [7].

Definition 5 (Honest-Verifier Zero-Knowledge)**.** *An interactive argument $(Setup, P, V)$ for a relation R is honest-verifier zero-knowledge, if there exists a probabilistic polynomial time simulator, such that there exists a negligible function $\epsilon(\lambda)$, for all adversaries A_1 and A_2*

$$\left| \Pr\left[\begin{array}{l|l} (\mathbb{x}, \mathbb{w}) \in \mathcal{R} \wedge & \sigma \leftarrow \mathit{Setup}(1^\lambda) \\ \mathcal{A}_1(tr) = 1 & (\mathbb{x}, \mathbb{w}, \rho) \leftarrow \mathcal{A}_2(\sigma) \\ & tr \leftarrow \mathit{SIM}(x, \rho) \end{array} \right] - \Pr\left[\begin{array}{l|l} (\mathbb{x}, \mathbb{w}) \in \mathcal{R} \wedge & \sigma \leftarrow \mathit{Setup}(1^\lambda) \\ \mathcal{A}_1(tr) = 1 & (\mathbb{x}, \mathbb{w}, \rho) \leftarrow \mathcal{A}_2(\sigma) \\ & tr \leftarrow \langle \mathcal{P}(\sigma, \mathbb{x}, \mathbb{w}), \mathcal{V}_\rho(\sigma, \mathbb{x}) \rangle \end{array} \right] \right| \leq \epsilon(\lambda)$$

Definition 6 (Non-Interactive Knowledge-Soundness)**.** *A non-interactive random oracle argument $(Setup, P, V)$ for relation R is knowledge sound, if there exists an efficient knowledge extractor $\mathcal{E}$ and a positive polynomial z, such that for any statement $\mathbb{x} \in \{0,1\}^\lambda$ and prover P^* with at most Q queries to the random oracle RO,*

$$\Pr\left[(\mathbb{x}, \mathbb{w}') \in \mathcal{R} \;\middle|\; \begin{array}{l} \sigma \leftarrow \mathit{Setup}(1^\lambda) \\ \mathbb{w}' \leftarrow \mathcal{E}_{P^*}(\mathbb{x}) \end{array} \right] \geq \frac{\epsilon(P^*, \mathbb{x}) - \kappa(|\mathbb{x}|, Q)}{z(|\mathbb{x}|)},$$

where $\epsilon(P^*, \mathbb{x}) := \Pr[\langle P^*_{RO}(\sigma, \mathbb{x}), V_{RO}(\sigma, \mathbb{x}) \rangle = 1]$, $\kappa(\lambda, Q) \in [0, 1]$ is the *knowledge error* and is negligible. ε has a black-box oracle access to the prover P^* and can manipulate the random oracle RO for A arbitrarily.

Definition 7 (Completeness)**.** *An argument $(Setup, P, V)$ is complete, if for any statement $\mathbb{x} \in L$ and witness $\mathbb{w}$ such that $(\mathbb{x}, \mathbb{w}) \in R$, there exists a negligible function $\mu(\lambda)$, such that*

$$\Pr\left[\langle P(\sigma, \mathbb{x}, \mathbb{w}), V(\sigma, \mathbb{x}) \rangle = 1 | \sigma \leftarrow \mathit{Setup}(1^\lambda) \right] \geq 1 - \mu(\lambda)$$

APPENDIX B
PROOF OF THEOREM 1

Proof. The following proof is divided into two separate proofs. First, we prove HVZK.

After this, both knowledge-soundness and completeness are proven in the same proof.

Proof of HVZK: CAAUrdleproofs' modification is Curdleproofs' IPAs on which the Springproofs protocol has been applied. So, to help show that it is HVZK, we refer to Theorem 5 of Springproofs [7].

Theorem 2 (Springproofs Theorem 5)**.** *Suppose IPA_k is a HVZK IPA which reduces a relation $R_{zk,k}$ into a relation $R_{zk,k/2}$, and the blinding factors in the two relations distribute independently. Given a scheme function f, if the $SIPA_{IPA}(f)$ is terminative for any lengths n of the witness vector, and there exists a polynomial $poly(\lambda)$ such that the number of rounds $m < poly(\lambda)$, then $SIPA_{IPA}(f)$ is HVZK when $n \geq 2$.*

Given this theorem, we interpret IPA_k as Curdleproofs' DL IPA.

In Theorem 5.3.1 of the Curdleproofs paper, they prove their IPA to be ZK [6]. They do this with the help of a simulator and show that the prover's and simulator's responses are distributed identically. This approach aligns with the definition of HVZK from Definition 2; therefore, the Curdleproofs IPA is HVZK.

We also know that the IPA is a folding argument, which reduces the size of the argument by half after each iteration. In this reduction, Curdleproofs also proved in Theorem 5.3.1 that the values $B_C, B_D, L_{C,j}, L_{D,j}, R_{C,j}, R_{D,j}$ are blinded and identically distributed.

The scheme function used in CAAUrdleproofs, as seen in Figure 2(b), is shown by Springproofs to be a variant of their pre-compression method [7]. Springproofs shows this function to be optimal in the number of folding steps; hence, it must also terminate. Specifically, the pre-compression is shown to run in $\lceil \log n \rceil$ folding rounds, satisfying the existence of the polynomial mentioned in Theorem 5.

Curdleproofs show their argument to be ZK in the random oracle model provided $|\mathbf{G}| \geq 8$ [6]. Therefore, following Theorem 1, CAAUrdleproofs must be HVZK when $n \geq 8$

Proof of knowledge-soundness and completeness: For soundness and completeness, we refer to Theorem 3 of Springproofs [7].

Theorem 3 (Springproofs Theorem 3)**.** *Given a terminative $SIPA(f)$, if the number of compression steps in $SIPA(f)$ is $\mathcal{O}(\log n)$, then $SIPA(f)$ is a complete and computational knowledge sound argument of relation (1). Moreover, the Fiat-Shamir transformation of $SIPA(f)$ is a non-interactive random oracle argument having completeness and computational knowledge soundness as well.*

Here, relation (1) is

$$\{(\mathbf{g}, \mathbf{h} \in \mathbb{G}^n, u, P \in \mathbb{G}; \mathbf{a}, \mathbf{b} \in \mathbb{F}_p^n) : P = \mathbf{g}^{\mathbf{a}}\mathbf{h}^{\mathbf{b}}u^{\langle \mathbf{a},\mathbf{b} \rangle}\} \quad (2)$$

, or analogously for an additive cryptographic group:

$$\{(\mathbf{g}, \mathbf{h} \in \mathbb{G}^n, u, P \in \mathbb{G}; \mathbf{a}, \mathbf{b} \in \mathbb{F}_p^n) : \quad (3)$$
$$P = \mathbf{a} \times \mathbf{g} + \mathbf{b} \times \mathbf{h} + \langle \mathbf{a}, \mathbf{b} \rangle u\} \quad (4)$$

Relating that to Curdleproofs, they mention that they discuss the IPA for the relation:

$$\left\{ (C, D, z;\, \mathbf{c}, \mathbf{d}) \;\middle|\; \begin{array}{l} C = \mathbf{c} \times \mathbf{G}, \\ D = \mathbf{d} \times \mathbf{G}', \\ z = \mathbf{c} \times \mathbf{d} \end{array} \right\} \quad (5)$$

However, taking inspiration from Bulletproofs, which also happens to be the IPA used in Springproofs, they include a commitment to the inner product, z, in commitment C [18]. So, before the addition of blinding values and challenges, the relation they want to prove is:

$$\left\{ \left(\begin{array}{r} \mathbf{G}, \mathbf{G}' \in \mathbb{G}^n, \\ C, D \in \mathbb{G}, \\ z \in \mathbb{F}_p \end{array} \;\middle|\; \mathbf{c}, \mathbf{d} \in \mathbb{F}_p^n \right) \;\middle|\; \begin{array}{l} C = \mathbf{c} \times \mathbf{G} + zH, \\ D = \mathbf{d} \times \mathbf{G}', \\ z = \mathbf{c} \times \mathbf{d} \end{array} \right\} \quad (6)$$

We can now take a look at Springproofs' P commitment in comparison to Curdleproofs' C and D commitments. If we add together Curdleproofs' two commitments, we get:

$$C + D = \mathbf{c} \times \mathbf{G} + \mathbf{d} \times \mathbf{G}' + zH \quad (7)$$

, which is the same commitment as in Equation 4.

Therefore, using Curdleproofs' DL IPA and the pre-compression scheme function, we can instantiate SIPA(f), equivalent to CAAUrdleproofs, as a terminative SIPA(f), with $\mathcal{O}(\log n)$ compression steps. Hence, SIPA(f) is a complete and computational knowledge sound argument of relation (1). We have just shown that Curdleproofs' IPA proves the same relation, so the properties hold for our SIPA(f) as well. Furthermore, Curdleproofs uses the Fiat-Shamir transformation for its verifier challenges. So, the SIPA(f), analogously CAAUrdleproofs, is a non-interactive random oracle argument having completeness and computational knowledge soundness as well.

Now, we switch our focus to another argument, namely the SameMSM argument. As with the DL IPA, SameMSM is also an IPA. Hence, to work in CAAUrdleproofs, it also needs the optimizations used in the DL IPA.

Therefore, the SameMSM argument also uses the Springproofs scheme function. Also, it uses the new computation of $\mathbf{s}$, see Listing 4, used for coupling the correct challenges to each element in the vector. Furthermore, Curdleproofs blinds the argument in the same way as the DL IPA. Hence, our argumentation of HVZK, knowledge-soundness, and completeness follows from the above explanation of DL IPA.

From this, we can conclude that CAAUrdleproofs is a zero-knowledge argument of knowledge when shuffle size $|\ell| \geq 8$. □

APPENDIX C
CURDLEPROOFS WEIGHTED INNER PRODUCT ARGUMENT MODIFICATION ATTEMPT

In the following, we will show our unsuccessful attempt in trying to convert the DL IPA into a WIPA.

Before this, it should be mentioned that we have made code for the IPA in the Curdleproofs repository, which works with Bulletproofs+' Weighted Inner Product Argument. Both the prover and verifier work. The problem is connecting it to the rest of the Curdleproofs protocol.

First, the DL IPA stems from the grand product argument that it has been converted from. Hence, it makes sense to modify the structure of this.

The grand product argument compiles the grand product p down to an equation that can be expressed as an inner product [6]. The equation is:

$$p\beta^\ell - 1 = \sum_{i=1}^{\ell} c_i(\beta^i b_i - \beta^{i-1}) \tag{8}$$

This can be seen as the inner product $z = \mathbf{c} \times \mathbf{d}$, where $z = p\beta^\ell - 1$ and $d_i = (\beta^i b_i - \beta^{i-1})$, $i \in [1, \ell]$.

We will now follow the exact conversion from grand product to inner product as in Curdleproofs. This includes four steps; *separate*, *compress*, *rearrange*, and *compile*. But instead of converting from the grand product $p = \Pi_{i=1}^{\ell} b_i$, we try the conversion with $p = \Pi_{i=1}^{\ell} b_i^{y_i}$. Here, y is a Vandermonde vector of challenges, e.g., $y = \{y^0, y^1, \ldots, y^{\ell-1}\}$

C.1 Separate

The grand product $p = \Pi_{i=1}^{\ell} b_i^{y_i}$ has $\ell - 1$ multiplication, which we separate into $\ell + 1$ checks

$$c_1 = 1^{y_0} \wedge c_{i+1} = b_i^{y_i} c_i,\ i \in [1, \ell) \wedge p = b_\ell^{y_\ell} c_\ell \tag{9}$$

As explained by Curdleproofs, the final check $p = b_\ell^{y_\ell} c_\ell$ enforces the grand product $p = \Pi_{i=1}^{\ell} b_i^{y_i}$.

C.2 Compress

All equations are combined into a single polynomial to ensure that they hold

$$0 = (1 - c_1) + (b_1^{y_1} c_1 - c_2)X + (b_2^{y_2} c_2 - c_3)X^2 + \tag{10}$$

$$\cdots + ((b_{\ell-1}^{y_{\ell-1}} c_{\ell-1} - c_\ell))X^{\ell-1} + (b_\ell^{y_\ell} c_\ell - p)X^\ell \tag{11}$$

C.3 Rearrange & Compile

Now, this next step is where the equations start to create problems. In Curdleproofs, they rearrange the $\mathbf{c}$ terms. For example, if the shuffle size was 3, the equation would become:

$$c_1(Xb_1 - 1) + c_2(X^2 b_2 - X) + c_3(X^3 b_2 - X^2) \tag{12}$$

Or equivalently, by using the values of each equation in $\mathbf{c}$:

$$1(Xb_1 - 1) + b_1(X^2 b_2 - X) + (b_1 b_2)(X^3 b_3 - X^2) \tag{13}$$

This can also be stated as:

$$\sum_{i=1}^{\ell} c_i(X^i b_i - X^{i-1}) \tag{14}$$

Simplifying this equation, it becomes:

$$b_1 b_2 b_3 X^3 - 1 = pX^\ell - 1 \tag{15}$$

By the Schwartz-Zippel Lemma, as explained by Curdleproofs, the following inner product holds with overwhelming probability if at a random point β:

$$p\beta^\ell - 1 = \sum_{i=1}^{\ell} c_i(\beta^i b_i - \beta^{i-1}) \tag{16}$$

Or just $z = \mathbf{c} \times \mathbf{d}$.

Now, we will do the same thing with our equations, which include the weights $\mathbf{y}$. Keep in mind that for the WIPA to work, we need the following structure:

$$\sum_{i=1}^{\ell} c_i \cdot d_i \cdot y_i \tag{17}$$

Hence, in the following conversion, that structure is our goal.

Following the compression from the previous section, we will verify whether it retains the structure of Equation 17 after rearrangement and compilation. Again, we use size 3 for the example. As in the Curdleproofs case, we rearrange the $\mathbf{c}$ terms to be:

$$c_1(Xb_1^{y_1} - 1) + c_2(X^2 b_2^{y_2} - X) + c_3(X^3 b_3^{y_3} - X^2) \tag{18}$$

Again, we insert the values of $\mathbf{c}$:

$$1(Xb_1^{y_1} - 1) + b_1^{y_1}(X^2 b_2^{y_2} - X) + (b_1^{y_1} b_2^{y_2})(X^3 b_3^{y_3} - X^2) \tag{19}$$

This simplifies down to:

$$b_1^{y_1} b_2^{y_2} b_3^{y_3} X^3 - 1 = pX^\ell - 1 \tag{20}$$

Unfortunately, this does not align with the structure we set as our goal in Equation 17.

Instead, we can work directly from the given structure shown in Equation 17 and see what we get. We now use $p = \Pi_{i=1}^{\ell} b_i$:

$$\sum_{i=1}^{\ell} c_i(\beta^i b_i - \beta^{i-1}) y_i \tag{21}$$

Here, a problem arises. If we still look at the example with shuffle size 3:

$$1(Xb_1 - 1)y_1 + b_1(X^2 b_2 - X)y_2 + (b_1 b_2)(X^3 b_3 - X^2)y_3 \tag{22}$$

We are not able to simplify this equation. As a result of this, the verifier would need to know the values of $\mathbf{c}$ and $\mathbf{d}$ to compute z, which breaks ZK. Hence, the conversion is not useful.

Therefore, significant protocol changes are needed to implement the WIPA in Curdleproofs.

APPENDIX D
SHUFFLING RESULTS

Here, we present the results of the shuffling times given different shuffling size values. The results can be seen in Figure 7.

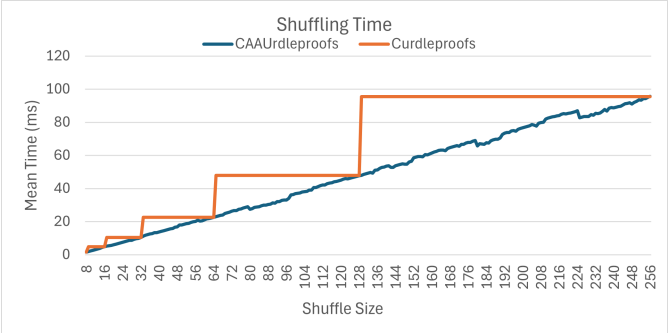


Fig. 7: The shuffling times at each benchmark