Zero-Knowledge Shuffle Improvement in Ethereum Single Secret Leader Election

Anders Malta Jakobsen*, Oliver Holmgaard†

Abstract—This is the abstract [1].

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

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 and has a large and active developer community.

2 Background

In this section, we provide the necessary background information on Ethereum and a specific attack it is vulnerable to, the Whisk protocol [2], and the Curdleproofs protocol [3] used in Whisk.

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

Since this work is based on the existing Curdleproofs protocol [3], 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 [4]. 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 be the ability to convince someone that a transaction is valid without revealing information about the transaction such as the value of it. Whisk uses Curdleproofs to prove the validity of a shuffle.

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

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.

Also, two of three proofs that make up Curdleproofs are Inner Product Arguments (IPAs). These are also ZKPs, and will be the focus of this paper. Hence, we provide a definition on 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 uses a Proof of Stake (PoS) consensus mechanism, which allows users to validate transactions and create new blocks by staking their Ether (ETH) tokens. The PoS protocol works in epochs of 32 slots, where slots are 12 seconds long. In each slot a proposer is chosen to propose a block thereby allowing the network to reach 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, gotten from proposing a block, and increase their own rewards [5]. As a response to the proposer DoS attack, Ethereum proposed a new protocol called Whisk [2] as an attempt to mitigate the attack. An attack on the Ethereum network that was discovered by Heimbach et al. [6] is the deanonymization attack on validators. In our preliminary work [7], we show that the attack is still possible to perform on the Ethereum network, and using the attack, a proposer DoS can be performed.

Whisk is a Zero-Knowledge (ZK) Single Secret Leader Election (SSLE) system that uses a ZK argument called Curdleproofs [3] to verify the correctness of a shuffle with size ℓ without revealing the input or output [8]. Whisk works by selecting a list of 16,384 validator trackers and shuffles

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 $\mathbb{G}$
$\gamma \in \mathbb{Z}_p^{\lceil \log n \rceil}$	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 phi and private witness w

TABLE 1: Notation used throughout the paper.

them over 8,192 slots (∼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. This is so that the proposer can prove that they are the correct proposer for the slot without revealing their identity, thereby mitigating the proposer DoS attack because of the identity of the upcoming proposers being hidden now.

Curdleproofs is a ZKP system that allows a prover to prove knowledge of a shuffle without revealing how it shuffled the elements. It does so by using three different ZKPs, with one of them relying on two more ZKPs. The overview can be seen in Figure 1.

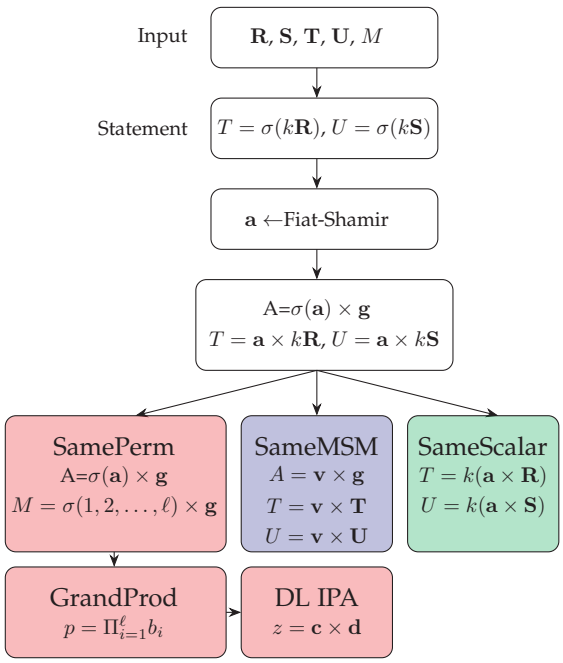


Fig. 1: Overall structure of the Curdleproofs protocol. Modified figure from [3].

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 transformation, 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 [9] 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 each multiplication of the grand product as its own equation. The proof of the DL IPA then stems from the protocol originally proposed by Bootle et al. [3, 10]

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 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 [2]. 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 [3] they explain that the proof has size $18 + 10\log(\ell + 4)\mathbb{G}$, $7\mathbb{F}$, where $\mathbb{G}$ is a crypto-

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

graphic group point, and $\mathbb{F}$ is a field element.

2.3 Problem definition

The current proposal of Curdleproofs only works when the shuffle size of Whisk is set to a power of 2. 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 being 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 2.

3 Related Work

3.1 Single Secret Leader Election

A 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 know that they have been chosen. The elected leader can then later publicly prove that they have been elected [8].

Leading research on SSLE includes proposals for post-quantum secure protocols based on Learning With Errors and Ring Learning With Errors [11]. 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 make PoS blockchains more secure due to the added privacy that the proposer has.

One PoS blockchain that uses an SSLE is Polkadot which uses Safrole as their SSLE protocol [12]. Safrole is the production version of the research protocol Sassafras [13]. In this, validators each produce a number of 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, from all the winning tickets, proposers for all the slots two epochs later.

3.2 Shuffling algorithms

The Håstad square shuffle [14] is one of the proposed ways of shuffling, which could be integrated in 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 re-arranging 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 [15] because of the fixed size of the shuffle being $\sqrt{n}$.

The Feistel shuffle [16] is a previously used shuffle method in the Whisk protocol [2]. It takes n number of validator trackers and arranges them in a $k \times k$ matrix. 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. [15]. Ethereum mentioned the reason for this to be that the shuffle by Larsen et al. provides a simpler protocol [2].

3.3 Bulletproofs

A big inspiration for the Curdleproofs protocol is bulletproofs [17]. Bulletproofs is a type of range proof that uses inner product arguments to prove that a committed value is within a certain range without revealing the value itself. Bulletproofs is in itself not a zero-knowledge proof system, but with the help of Fiat Shamir [17] it can be used to create a zero-knowledge proof. Bulletproofs also has had a few iterations and improvements to increase the speed and reduce the size of the proof since it was used in curdleproofs.

One of these is Bulletproofs+ [18] which uses a weighted inner product argument instead of the standard inner product argument to achieve a better performance. Bulletproofs+ is also a zero-knowledge proof by itself unlike the original bulletproofs. Trying to modify Curdleproofs with the weighted inner product argument introduces complications that would need larger modifications and is therefore not suitable. This can be seen in Appendix C

A third version of the Bulletproofs protocol is Bulletproofs++ [19] which uses a new type of argument called the norm argument to achieve a better performance. This 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 IPA, especially the running time and proof size of the protocol are of interest. The following is our approach to, how we modified the IPA.

4.1 Springproofs

The Springproofs protocol [20] 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 on shuffle sizes other than powers of two. As such, they could lower the shuffle size from the current 128 to a size significantly lower, given it is still secure. Seeing the proof size of Curdleproofs being dependent on ℓ means that this modification would greatly help in lowering it.

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 recursive 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, 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.

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

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 [20]. In Figure 2(a) is a scheme function which simply pads the vector to the next power of two before running an IPA. If one wanted to run current IPAss on vector that are not a power of two, this would generally be the easiest way to achieve that. Though, this defeats the attempt of lowering the proof size, as it would now correspond to running an IPA on the size of the next power of two.

It is notable to mention that using the folding as shown in Figure 2(b) results in the second recursive round being a size corresponding to a power of two. This means that the rest of the protocol will run 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 made a modification to the IPA of Curdleproofs We call this modified protocol, 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 done exactly 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 of 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).

At last, 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. This is what is returned 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 [2].

Verifier computation

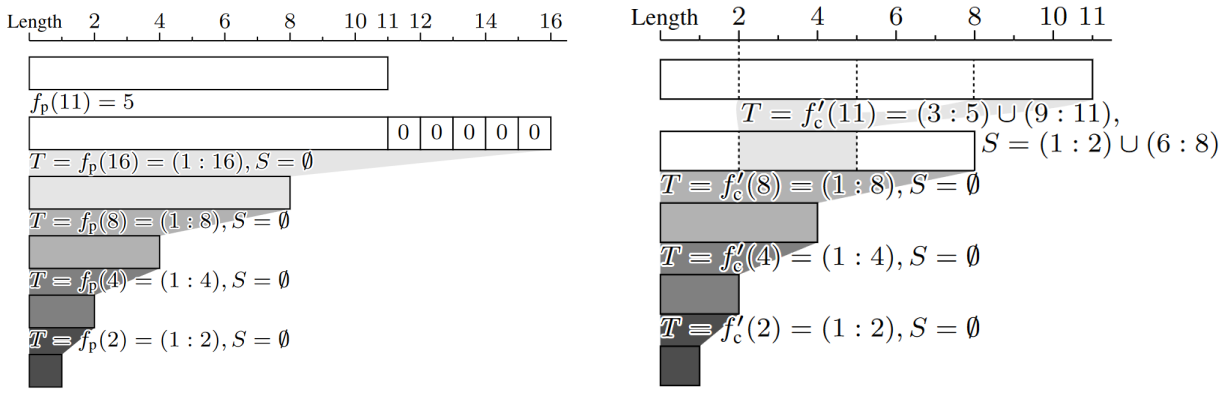
Having the proof on the blockchain allows for each validator to asynchronously verify whether it is a valid proof. Again, the originally proposed verifying protocol has been modified according to Springproofs, which is seen in Listing 3.

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

At 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 is able to 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 being 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



(a) Padding Method

(b) Optimal Pre-Compression Method

Fig. 2: Folding visualization as seen in the Springproofs paper. Source: [20]

```
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 ← len(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 CAAUr-dleproofs

```
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 + (α^2 z)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 ← len(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 CAAUr-dleproofs

that the commitments C and D need to be commitments to the modified witnesses instead.

The setup phase is now done, and the verifier has to also run a recursive protocol, which is shown in step 2. First, the vectors are on line 13 divided into the two sets, $|T|, |S|$, as in Listing 2. After this, the group vectors are in lines 14–16 split in 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 is able to compute a challenge γ_j, line 18, made from the same public inputs as the prover.

The corresponding left and right side cross-product 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 retreived 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. [15] that was 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 [15].

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

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: [15]

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 shufflers may be adversarial. This means that whenever the shuffling process is taking place, a part of the 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 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 cups. For instance, if some of the cups are owned by the adversary. Those tracked cups 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 possibility of a certain ciphertext to be in a certain slot. So, if a shuffle contains a lot of ciphertexts with a larger than average chance of containing a certain 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 the shuffle size, and a number of adversarial shufflers, to guarantee 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 introduced some optimizations required 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.

4.4.1 CAAUdleproofs

The protocol in Curdleproofs [3] introduces a lot of multiscalar multiplications. As such, CAAUrdleproofs also introduces these multiplications. This allows 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 introduced a slight difference on this topic in regard to the IPAs, SamePerm and SameMSM. In each recursive round, both the folded vectors and the commitments are being multiplied by verification scalars, γ_j. To keep track of which elements of the

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

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 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 though the recursive protocol. Though, this should not have a big impact on performance as it is run over vectors of small integers, and never actually has to do any multiplications. It is simply used as a measuring tool.

The protocol used in the implementation can be seen in Listing 4. A list, `ActivePos`, on line 32, keeps track of the original index placement and its position after each fold. 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 line 21 and 23, is used for the multiscalar multiplication optimization. For a thorough explanation of these, we refer to Curdleproofs [3].

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. This includes 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 is also using 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 same code as Curdleproofs to run the rest of the protocol.

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 [15], 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 authors of the shuffle's terminology, 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, also randomly. Meanwhile, an average of the water between the active indices of the ℓ-shuffle is found. All active indices are given this amount of water.

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 $\mathbf{s}$*: 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

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 [15]. 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 denotes how many rounds it took 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 can reduce the shuffle size used in Whisk and still prove it secure, then we expect to see some 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, though. 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 [2]. 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).
- 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 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.

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

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 tweak in the different experiments that we have.

The experiments are run on a virtual machine hosted on Strato CLAAUDIA through Aalborg University. The machine is using an Intel Xeon Cascadelake processor, CPU family 6, model 85. It has 16 virtual CPUs and 64 GB of RAM available. 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 own 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 $\ell = \{8, 16, 32, 64, 128, 256\}$.

5.2 Shuffle security

In this experiment we run the shuffle protocol with varying shuffle sizes and varying number of adversarial tracked ciphertexts. The purpose of this experiment is to find the lowest possible shuffle size that is still secure. We therefore run the experiment with shuffle sizes, ℓ, between 64 and 128. 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, all the experiments were done with a maximum of 8192 shuffles. Also, the experiments shuffle over a set of 16,384 ciphertexts. Both of these numbers come from the Ethereum Whisk proposal [2].

Every experiment is run 1000 times to avoid statistical uncertainty. As done by the shuffle authors [15], we will denote the 20th, 40th, 60th, 80th, and 100th percentile on when the shuffle is deemed secure by the experimental runs.

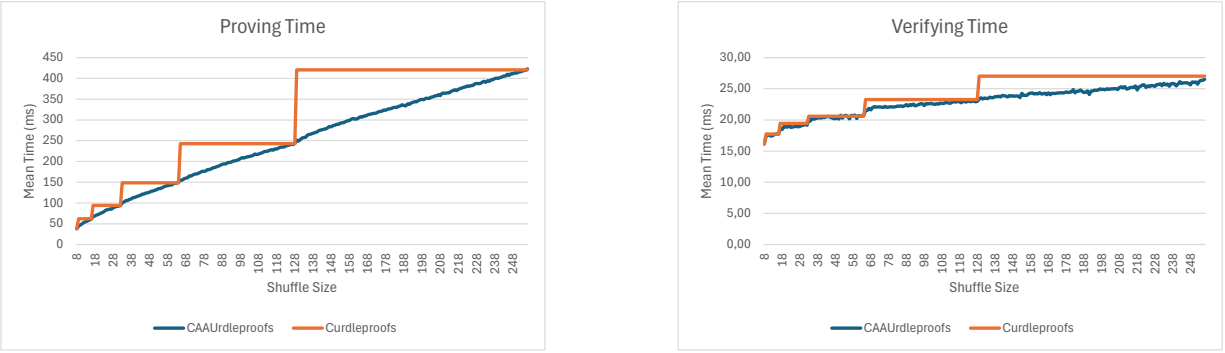
6 Results

6.1 Proving and Verifying Times

After running the experiment where Curdleproofs and CAAUrdleproofs were compared across different shuffle sizes, we got the results shown in Figure 4.

As mentioned in section 5.1, CAAUrdleproofs was run with a shuffle size ℓ of $\{8, 9, \ldots, 256\}$. Curdleproofs was only run with a shuffle size ℓ of $\{8, 16, 32, 64, 128, 256\}$, as it is only able to run in powers of 2. This is why the results for Curdleproofs show the shuffle size, ℓ, instantly going up to the next power of 2, because it theoretically would have to pad the input set until it reached the next power of 2.

From the results, we can see that CAAUrdleproofs and Curdleproofs have similar proving and verifying times when ℓ is a power of 2. However, when ℓ is not a power of



(a) Proving Time

(b) Verifying Time

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

2, CAAUrdleproofs is faster. When ℓ is below a power of 2, we see that the performance advantage of CAAUrdleproofs contra Curdleproofs grows the lower ℓ is.

The results for the verifying time also show that the verifying time jumps up quite significantly the first four times it reaches a power of 2. Though, this seems to not be the case, at least not as aggressively, when increasing ℓ from 128. We find, however, that the bump is smaller the higher ℓ is.

Additional to the proving and verifying times, the time used on shuffling is also lower for any ℓ that is not a power of 2; see Appendix D. Though, 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 2 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 1000 runs of each shuffle size ℓ as well as one standard deviation higher and lower.

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 8192 shuffles, meaning that the maximum number of honest shuffles that can be used is 8192. Therefore, the results of the experiment also find $T_H = T - \beta$. This is how many of the T shuffles, available during the shuffling phase, are needed to be honest T_H shuffles. The rest could then be the number of dishonest shuffles, β. We also see that the bigger the shuffle size, the narrower the standard deviation gets.

From the results of the experiment, with $\alpha = 8192$ we can see that the number of honest shuffles necessary to make the shuffle secure sharply goes down until the size of $\ell = 64$, and then it starts to flatten out. we can see that with a size of $\ell = 75$ we need about 1/3 of the shuffles to be honest to 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 happens, but the later it happens as well. There are two things, however, that are different between the experiments. At an α of 4096 we see that at the start, with $\ell = 32$, the mean number of honest shuffles necessary to make the shuffle secure is ~ 500 lower than the 2 others. 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 see a trend that the lower the α is, the earlier the dip happens.

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.

It is worth noting that there is a spike in the distribution of the necessary honest shuffles at $\ell = 32$ for $\alpha = 4096$. This spike is not present for the other two α values, and is due to the probabilistic nature of the shuffling method.

Another thing that is notable is that in the setting of Ethereum, the maximum number of shuffles available is 8192. This means that in the cases where more than the 8192 shuffles were necessary to make the shuffle secure, the shuffle would not have been secure within the Ethereum setting. Therefore, it is also possible to see the experiment as running 1000 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 $\alpha = 8192$ and $\ell = 40$ for $\alpha = 5462$ and $\alpha = 4096$.

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.

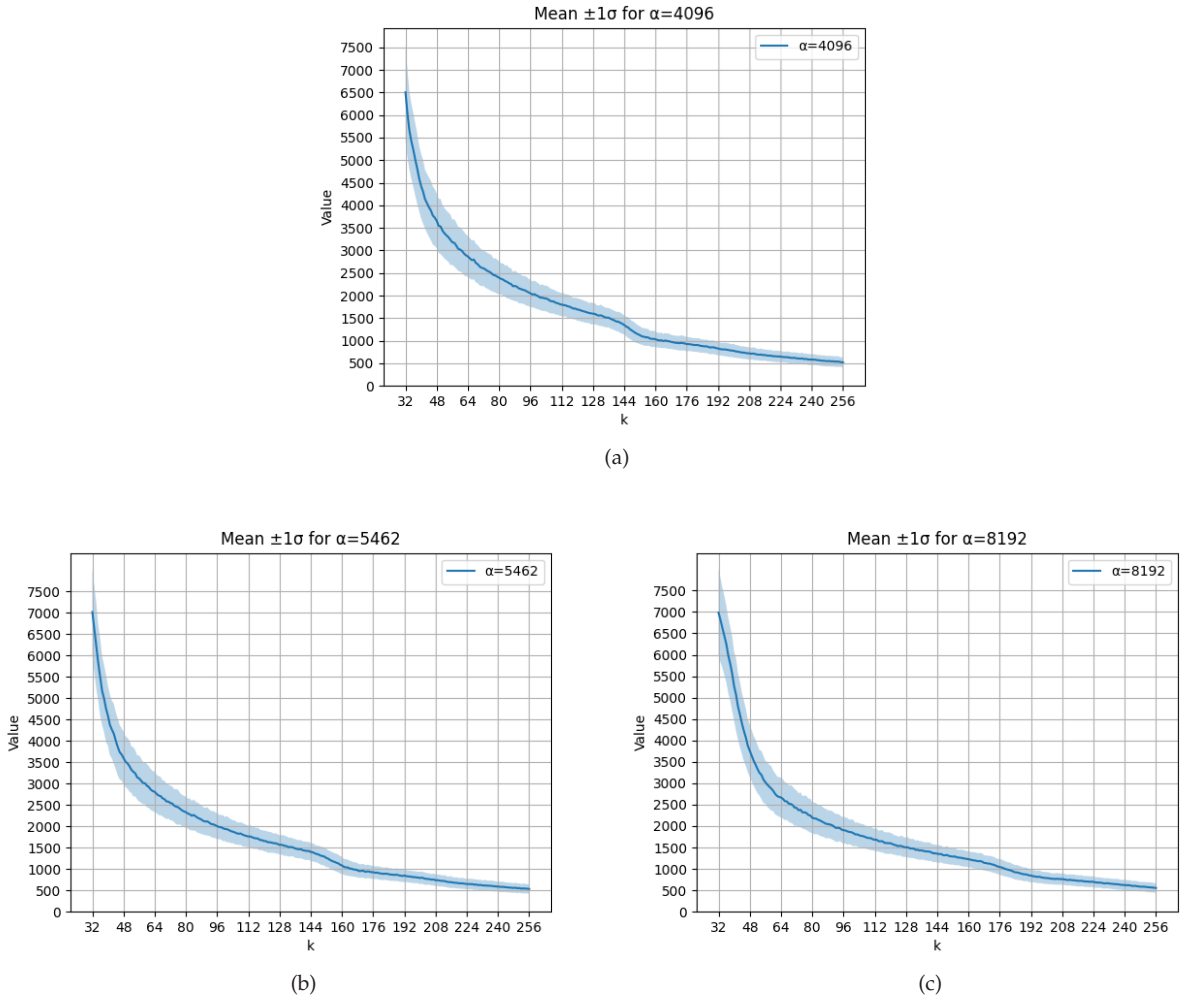


Fig. 5: The results of the shuffle security experiment showing the mean amount of honest shuffles necessary with one standard deviation

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. We expect that this is because the added computation is negligible compared to other computations that are present in the original Curdleproofs protocol.

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

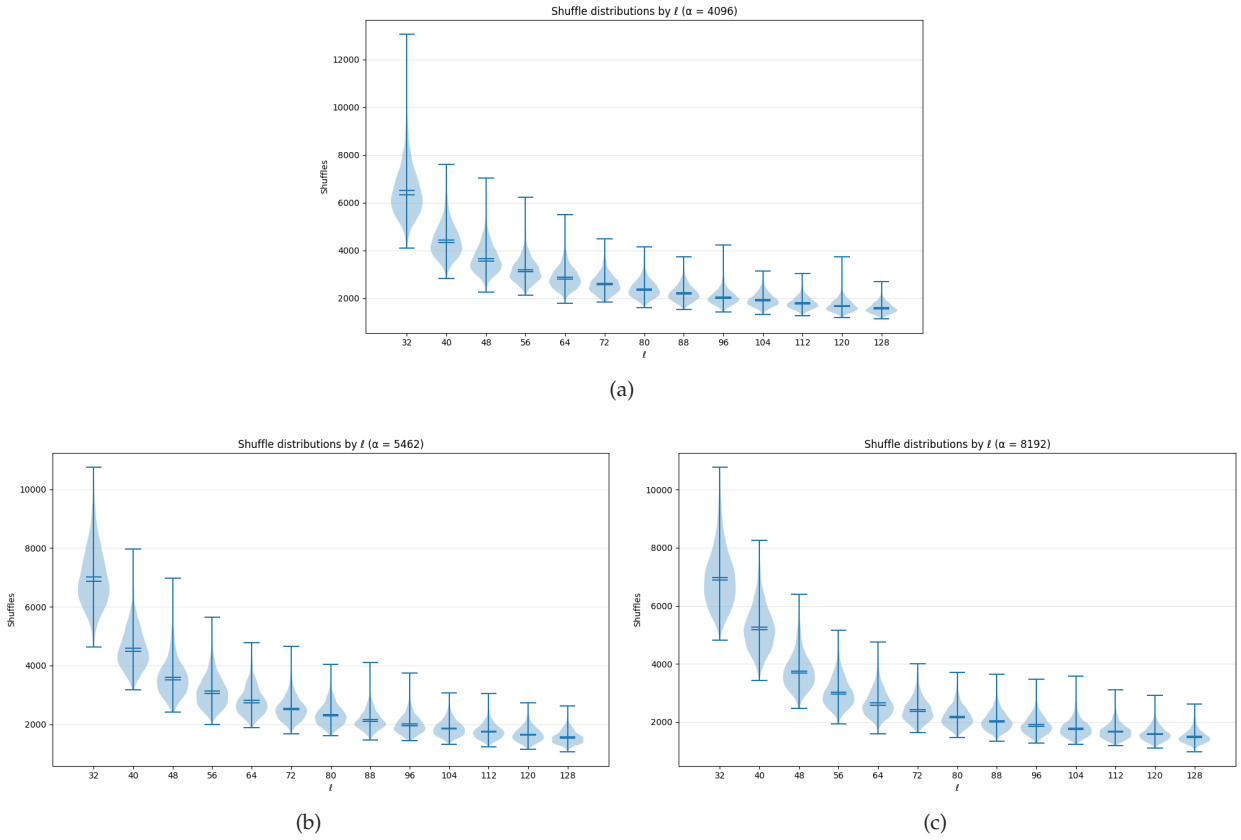
Also, we mentioned in section 4.4 that, every round after the first, runs the same code as Curdleproofs. Thus, only the first round should be able to introduce some computational overhead. But, as mentioned before, the overhead should be negligible.

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(b), we see that the only difference in computation between CAAUrdleproofs and Curdleproofs stems from the calculation of $\mathbf{s}$. 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, need some more integer variables for splitting as well as an array for keeping track of the vector elements' active positions 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 see some more aggressively increasing verifying times. We expect this to be a result of m being set to $\lceil \log n \rceil$ in line 8 of Listing 4. E.g., when ℓ is 65, it will have to handle computations from an additional recursive round, in comparison to when ℓ is 64. Additionally, this also explains why the increase in running time flattens when ℓ is increases.

Though the pattern shows that this bump has a decreasing impact on the running time, the higher ℓ is, as mentioned in section 6.1. In theory, the addition of an extra proof element should introduce a constant amount of work for the verifier.

Therefore, we believe this to be an artifact of memory



(a)

(b)

(c)

Fig. 6: The results of the shuffle security experiment showing the spread of nessecary shuffle need for the shuffle to be secure

optimizations done either by hardware or Rust. This could, for instance, include pre-fetching, in which the memory system can optimize access if it suspects some values in memory are going to be used some time later on[4]. 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.

We would however not recommend using an ℓ lower than 80, as the worst case scenario uses a little under half the available shuffles, and you would only need one third to get within the standard deviation. This would still lead to the size of the block overhead and the speed of the protocol being significantly reduced.

8 Conclusion

This is the conclusion

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

9 Future work

This is the future work.

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

REFERENCES

[1] G. D. Greenwade, "The Comprehensive Tex Archive Network (CTAN)," *TUGBoat*, vol. 14, no. 3, pp. 342–351, 1993.

[2] G. Kadianakis, "Whisk: A practical shuffle-based ssle protocol for ethereum," 2024, Accessed: 22-10-2024.

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

[4] 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.

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

[6] 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].

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

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

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

APPENDIX A
DEFINITIONS OF ZERO-KNOWLEDGE ARGUMENT OF KNOWLEDGE

Definition 4 (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*

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

Definition 5 (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 $\mathtt{RO}$,*

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

Definition 6 (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 is the Curdleproofs DL IPA on which the Springproofs protocol has been applied. So to help show that it is HVZK, we refer to Theorem 5 of Springproofs [20].

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 $\mathtt{poly}(\lambda)$ such that the number of rounds $m < \mathtt{poly}(\lambda)$, then $SIPA_{IPA}(f)$ is HVZK when $n \geq 2$.*

Given this Theorem, we interpret IPA_k as the Curdleproofs DL IPA.

In Theorem 5.3.1 of the Curdleproofs paper, they prove their IPA to be zero-knowledge [3]. They do this by help of a simulator and show that the prover's and simulator's response are distributed identically. This matches the definition of HVZK from Definition 2, hence 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 [20]. Springproofs show 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 zero-knowledge in the random oracle model provided $|\mathbf{G}| \geq 8$ [3]. 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 [20].

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

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

Though, 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 [17]. So, before the addition of blinding values and challenges, the relation they want to prove is:

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

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

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

This exactly 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.

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

We have made code for the IPA in the Curdleproofs repository which actually works with Bulletproofs+' Weighted Inner Product Argument.

Appendix D
Shuffling Results

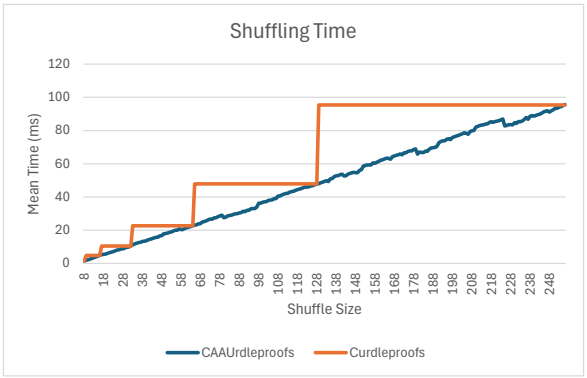


Fig. 7: The shuffling times at each benchmark