Zero-Knowledge Shuffle Improvement in Ethereum Single Secret Leader Election

Anders Malta Jakobsen*, Oliver Holmgaard†

Abstract—This is the abstract Zero-Knowledge Proof (ZKP) [1].

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

1 Introduction

This is the introduction

2 Related Work

2.1 Single Secret Leader Election

A Single Secret Leader Election (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 [2]. One of the use cases of SSLE is to make Proof of Stake (PoS) cryptocurrencies more secure due to the added privacy that the proposer has.

One PoS cryptocurrency that uses an SSLE is Polkadot which uses Safrole as their SSLE protocol [3].

2.2 Shuffling algorithm

The Håstad square shuffle [4] is one of the proposed ways of integrating an SSLE. The Håstad square shuffle is a shuffling algorithm that shuffles a n long vector with a shuffle size of $\sqrt{n}$. The algorithm works by splitting the vector into $\sqrt{n}$ times $\sqrt{n}$ square matrix and for each step in the algorithm it switches between shuffling a row and a column. The Håstad shuffle is more rigid than the shuffling algorithm used in curdleproofs [5] because of the fixed size of the shuffle being $\sqrt{n}$.

The Feistel shuffle [6] is the previous shuffle method used in the Whisk protocol [7]. The Feistel shuffle is a shuffling algorithm that works by taking n number of trackers and arranging 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 $F(x,y) = (y, x+y^3 modk)$. The Feistel shuffle was then later replaced by the shuffle proposed be Larsen et al. [5] because of the Feistel shuffle being too slow to shuffle the list of proposers.

2.3 Bulletproofs

A big inspiration for the curdleproofs protocol is bulletproofs [8]. 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 [8] 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+ [9] which uses a weighted inner product argument instead of the standard inner product argument to achieve a better performance. Bulletproofs+ is also different because it is zero-knowledge proof by itself unlike the original bulletproofs. A third version of the bulletproofs is Bulletproofs++ [10] which uses a new type of argument called the norm argument to achieve a better performance. Unlike the two other proofs Bulletproofs++ is a binary range proof, which means that even if it is the fastest proof it is not suitable for the curdleproofs protocol due to the binary nature of the bulletproofs++.

3 Background

In this section, we provide the necessary background information on the Curdleproofs protocol [11], the Whisk protocol [7] and an overview of the notation used in the paper.

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

Since this work is based on the existing Curdleproofs protocol [11], 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 [12].

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 $c = ab$.*

3.1 Whisk

Ethereum: 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

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

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)
$\{$*Public Input*, *Witness*$\}$: Relation	Relation using the specified public input and witness

TABLE 1: Notation used throughout the paper.

a large and active developer community. Ethereum uses a proof-of-stake consensus mechanism, which allows users to validate transactions and create new blocks by staking their Ether (ETH) tokens. The Proof-of stake protocol works 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 consensus on the state of the blockchain.

Proposer DoS attack: The proposer 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 oen rewards [13]. As a response to the proposer DoS attack, Ethereum has proposed a new protocol called Whisk [7] as an attempt to mitigate the attack. An attack on the Ethereum network that was discovered by Heimbach et al. [14] is the deanonymization attack on validators. In our preliminary work [15], we have shown that the attack still possible to perform on the Ethereum network, and using the attack, a proposer Denial-of-Service (DoS) can be performed.

The Whisk protocol: Whisk is a zero-knowledge Single Secret Leader Election (SSLE) system that uses a zero-knowledge argument called Curdleproofs [11] to verify the correctness of a shuffle without revealing the input or output [2]. Whisk works by selecting a list of proposers 16384 and shuffling them over 8192 slots (1 day). Then 8192 proposers are selected from the shuffled list to propose blocks for the next 8192 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 zero-knowledge proof 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: Curdleproofs is a zero-knowledge proof 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 zero-knowledge proofs, with one of them relying on two more zero-knowledge proofs.

The first proof is the Same Permutation (SamePerm) proof. The prover first constructs a commitment to the permutation, $\sigma()$, by saying $M = \sigma(1, 2, \ldots, \ell) \times \mathbf{g}$. 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 now consists of convincing the verifier that the same permutation was used for constructing commitment A and M. To do this, the two commitments are used to construct a polynomial equation. Then Neff's trick [16], which observes that two polynomials are equal iff. their roots are the same up to permutation.

This is proven through a grand product argument. To prove that argument, Curdleproofs compiles it down to an Inner Product Argument (IPA) by expressing each multiplication of the grand product as its own equation.

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

The second proof is a Same Multiscalar (SameMSM) argument. The prover should by now have 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, here called multiscalar, committed to in commitment 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 [7]. Also, all validators are still able to open their commitments if they are chosen as block proposers, even after several randomizations. So, 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.

3.2 Zero-knowledge proofs

Curdleproofs is a zero-knowledge proof system, which means that it 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.

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.*

4 APPROACH

As explained in section 3, 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

In Chapter 6 of Curdleproofs [11], they explain the efficiency of the protocol, including also the size of the proof. They specifically mention that the proof has size $18+10\log(\ell+4)\mathbb{G}, 7\mathbb{F}$. As the proof size is dependent on the size of the shuffle, ℓ, an interest in the possibility of reducing this parameter arises. The current proposal of curdleproofs only works on shuffles, where the size is a power of 2. The reason is that the underlying proofs, such as the IPA, needs to fold recursively down to 1, by halving the size in every round.

The Springproofs protocol [17] can be used very effectively in this scenario. 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 Figure 1. The computation is for finding the set, T.

This can also visually be seen in Figure 2(b), which is figure 1 of the Springproofs paper [17]. 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 Figure 1.

```
input: $n$, where $n > 0$

$\{n\} \leftarrow n$
$N \leftarrow 2^{\lceil \log n \rceil - 1}$
$i_h \leftarrow \lfloor (2N - n)/2 \rfloor + 1$
$i_t = \lfloor n/2 \rfloor$
if $n \neq N$:
    $\{T\} \leftarrow (i_h : i_t) \cup (N + 1 : n)$
else if $n = N$:
    $\{T\} \leftarrow (1 : n)$
$\{S\} \leftarrow \{n\} - \{T\}$
```

Fig. 1: Scheme function f used in CAAUrdleproofs

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.

First of all, we have the prover computation, where the proof is constructed. The construction can be seen in Figure 3.

First, we have step 1, which is the setup phase. It is done exactly the same way as in Curdleproofs. To ensure zero-knowledge, two blinding vectors for each commitment are constructed. 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.

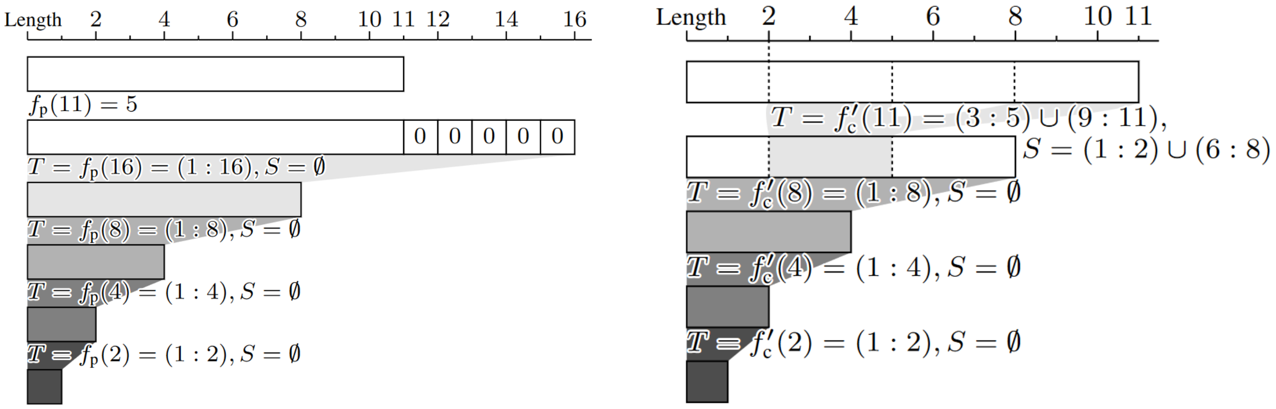
From the public input, hash values α, β are then computed. These are used to ensure the soundness of the protocol.

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 begins. As explained, at the start of the recursive round, the `while`-loop, we find the split of the vectors, with $f(n)$ being the function from Figure 1. Then, we find half the length of the T set, as this is the set, we are doing the recursive round on. Equally we split our witness vectors and the group vectors using T and S.

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

This value is used 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).


Length
2
4
6
8
10 11 12
14
16
$f_p(11) = 5$
0 0 0 0 0
$T = f_p(16) = (1:16), S = \emptyset$
$T = f_p(8) = (1:8), S = \emptyset$
$T = f_p(4) = (1:4), S = \emptyset$
$T = f_p(2) = (1:2), S = \emptyset$
Length
2
4
6
8
10 11
$T = f'_c(11) = (3:5) \cup (9:11),$
$S = (1:2) \cup (6:8)$
$T = f'_c(8) = (1:8), S = \emptyset$
$T = f'_c(4) = (1:4), S = \emptyset$
$T = f'_c(2) = (1:2), S = \emptyset$

(a) Padding Method

(b) Optimal Pre-Compression Method

Fig. 2: Folding visualization as seen in the Springproofs paper

At last, n is updated to the length of the concatenated vectors before starting a new round.

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

The now constructed proof is then supposed to be added to the block in the chain at the given time slot [7]. 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 Figure 4.

The changes to the verifier protocol are equivalent to the ones made to the prover protocol. First, the vectors are divided into the two sets, $|T|, |S|$. The verifier then retrieves the cross-product commitment update values, $L_{C,j}, L_{D,j}, R_{C,j}, R_{D,j}$. These are used for constructing a new commitment according to the fold made at round i. The corresponding left and right side cross-product are multiplied by a 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 updated as in Figure 3 before the protocol updates n to be the length of the newly constructed vectors.

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

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

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

4.3 Shuffle security

The shuffle method proposed by Larsen et al. [5] 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 5 [5].

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 $i_1, \ldots, i_k$ 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 adversarial shuffle can be seen as no shuffling being done. Therefore, the number of honest shuffles that happen during the shuffle process is $T_H = T - \alpha$, where α is the number of adversarial tracked ciphertexts.

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 $T \geq 20n/k \ln(n/\delta) + \beta$ and $k \geq 256 \ln^2(n/\delta)(1 - \alpha/n)^{-2}$. If T and k are chosen such that the above two conditions are met, then the protocol is an (ϵ, δ)-secure (T, n, k)-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 k. 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 k, but this has to be done experimentally.

```
Step 1:
$\mathbf{r}_C, \mathbf{r}_D \xleftarrow{\$} \mathbb{F}^n$
  where $(\mathbf{r}_C \times \mathbf{d} + \mathbf{r}_D \times \mathbf{c}) = 0$ and $\mathbf{r}_C \times \mathbf{r}_D = 0$
$B_C \leftarrow \mathbf{r}_C \times \mathbf{G}$
$B_D \leftarrow \mathbf{r}_D \times \mathbf{G}'$
$\alpha, \beta \leftarrow \mathrm{Hash}(C, D, z, B_C, B_C)$
$\mathbf{c} \leftarrow \mathbf{r}_C + \alpha\mathbf{c}$
$\mathbf{d} \leftarrow \mathbf{r}_D + \alpha\mathbf{d}$
$H \leftarrow \beta H$
Step 2:
$m \leftarrow n$
while $1 \leq j \leq \lceil \log m \rceil$ :
    $T, S \leftarrow \boldsymbol{f}(n)$
    $n \leftarrow \frac{|T|}{2}$
    $\mathbf{c} = \mathbf{c}_T$ ,  $\mathbf{cS} = \mathbf{c}_S$
    $\mathbf{d} = \mathbf{d}_T$ ,  $\mathbf{dS} = \mathbf{d}_S$
    $\mathbf{G} = \mathbf{G}_T$ ,  $\mathbf{GS} = \mathbf{G}_S$
    $\mathbf{G}' = \mathbf{G}'_T$ ,  $\mathbf{GS}' = \mathbf{G}'_T$
    $L_{C,j} \leftarrow \mathbf{c}_{[:n]} \times \mathbf{G}_{[n:]} + (\mathbf{c}_{[:n]} \times \mathbf{d}_{[n:]})H$
    $L_{D,j} \leftarrow \mathbf{d}_{[n:]} \times \mathbf{G}'_{[:n]}$
    $R_{C,j} \leftarrow \mathbf{c}_{[n:]} \times \mathbf{G}_{[:n]} + (\mathbf{c}_{[n:]} \times \mathbf{d}_{[:n]})H$
    $R_{D,j} \leftarrow \mathbf{d}_{[:n]} \times \mathbf{G}'_{[n:]}$
    $\pi_j \leftarrow (L_{C,j}, L_{D,j}, R_{C,j}, R_{D,j})$
    $\gamma_j \leftarrow Hash(\pi_j)$
    $\mathbf{c} \leftarrow \mathbf{cS} \| \mathbf{c}_{[:n]} + \gamma_j^{-1}\mathbf{c}_{[n:]}$
    $\mathbf{d} \leftarrow \mathbf{dS} \| \mathbf{d}_{[:n]} + \gamma_j\mathbf{d}_{[n:]}$
    $\mathbf{G} \leftarrow \mathbf{GS} \| \mathbf{G}_{[:n]} + \gamma_j\mathbf{G}_{[n:]}$
    $\mathbf{G}' \leftarrow \mathbf{GS}' \| \mathbf{G}'_{[:n]} + \gamma_j^{-1}\mathbf{G}'_{[n:]}$
    $n \leftarrow len(c)$
Step 3:
$c \leftarrow c_1$
$d \leftarrow d_1$

return $(B_C, B_D, \pi, c, d)$
```

Fig. 3: Prover computation for CAAU-IPA in CAAUrdleproofs

```
Step 1:
$(\mathbf{G}, \mathbf{G}', H) \leftarrow \mathtt{parse}(crs_{dl_{inner}})$
$(C, D, z) \leftarrow \mathtt{parse}(\phi_{dl_{inner}})$
$(B_C, B_D, \pi, c, d) \leftarrow \mathtt{parse}(\pi_{dl_{inner}})$
$\alpha, \beta \leftarrow \mathrm{Hash}(C, D, z, B_C, B_D)$
$H \leftarrow \beta H$
$C \leftarrow B_C + \alpha C + (\alpha^2 z)H$
$D \leftarrow B_D + \alpha D$

Step 2:
$m \leftarrow \lceil \log n \rceil$
for $1 \leq j \leq m$
    $T, S \leftarrow \boldsymbol{f}(n)$
    $n \leftarrow \frac{|T|}{2}$
    $\mathbf{G} = \mathbf{G}_T$ ,  $\mathbf{GS} = \mathbf{G}_S$
    $\mathbf{G}' = \mathbf{G}'_T$ ,  $\mathbf{GS}' = \mathbf{G}'_T$
    $(L_{C,j}, L_{D,j}, R_{C,j}, R_{D,j}) \leftarrow \mathtt{parse}(\pi_j)$
    $\gamma_j \leftarrow \mathrm{Hash}(\pi_j)$
    $C \leftarrow \gamma_j L_{C,j} + C + \gamma_j^{-1} R_{C,j}$
    $D \leftarrow \gamma_j L_{D,j} + D + \gamma_j^{-1} R_{D,j}$
    $\mathbf{G} \leftarrow \mathbf{GS} \| \mathbf{G}_{[:n]} + \gamma_j\mathbf{G}_{[n:]}$
    $\mathbf{G}' \leftarrow \mathbf{GS}' \| \mathbf{G}'_{[:n]} + \gamma_j^{-1}\mathbf{G}'_{[n:]}$
    $n \leftarrow \mathrm{len}(\mathbf{G})$

Step 3:
Check  $C = c \times G_1 + cdH$
Check  $D = d \times G'_1$
return 1 if both checks pass, else return 0
```

Fig. 4: Verifier computation for CAAU-IPA in CAAUrdleproofs

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

For $t \in [T]$ :
    $S_t$  picks random $\{i_1, \ldots, i_k\} \subset [n]$
    $S_t$  computes $(\tilde{c}_{i_1}, \ldots, \tilde{c}_{i_k}) \leftarrow$ Shuffle$(c_{i_1}, \ldots, c_{i_k})$
    $S_t$  publishes $(\tilde{c}_{i_1}, \ldots, \tilde{c}_{i_k})$
```

Fig. 5: Distributed shuffling protocol.

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 [1]. 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 [11] introduces a lot of multiscalar multiplications. As such, CAAUrdleproofs also introduces these multiplications. This allows for checking calculations of the form:

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

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 `same multiscalar`. In each recursive round, both the folded

[1] https://github.com/AAU-Dat/curdleproofsplus/tree/SIPA

vectors and the commitments are being 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 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 are 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 Figure 6. A list, `ActivePos`, 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, $\mathbf{u}$, is used for optimization in the grand product argument rather than $\mathbf{G}'$, and the `AccumulateCheck` function is used for the multiscalar multiplication optimization. For a thorough explanation of these, we refer to Curdleproofs [11].

In Curdleproofs, both the `SamePerm` and `SameMultiscalar` proof are recursive IPAs. So, the modifications and optimization used on the `SamePerm` argument are also used on the `SameMultiscalar` 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.

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 [5], 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 k ciphertexts and shuffles them, also randomly. Meanwhile, an average of the water between the active indices of the k-shuffle is found. All active indices 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 [5]. 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.

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, k.

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

Since Curdleproofs is unable to run benchmarks, unless the shuffle size is a power of two, those benchmarks will be run on values $k = \{8, 16, 32, 64, 128, 256\}$.

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, k, 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 [7].

Every experiment is run 1000 times to avoid statistical uncertainty. As done by the shuffle authors [5], we will denote the 20th, 40th, 60th, 80th, and 100th percentile on when the shuffle is deemed secure by the experimental runs.

6 Results

These are the results And they are very good

7 Discussion

This is the discussion

8 Conclusion

This is the conclusion

9 Future work

This is the future work.

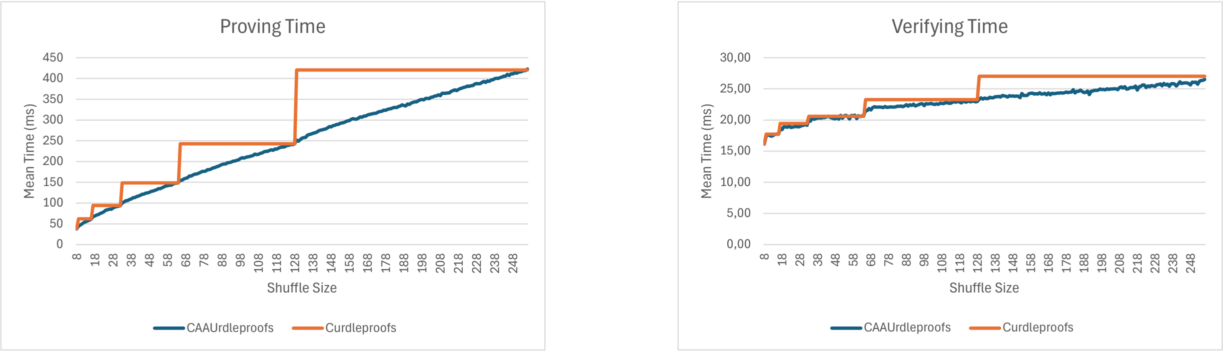
10 Acknowledgements

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.

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

**Step 2:**
$m \leftarrow \lceil \log n \rceil$
for $1 \leq j \leq m$
    $T, S \leftarrow \boldsymbol{f(n)}$
    $n \leftarrow \frac{|T|}{2}$
    $(L_{C,j}, L_{D,j}, R_{C,j}, R_{D,j}) \leftarrow \texttt{parse}(\pi_j)$
    $\gamma_j \leftarrow \text{Hash}(\pi_j)$
    $n \leftarrow n + \text{len}(S)$

**Step 3:**
**CP**: $\gamma \leftarrow (\gamma_m, ..., \gamma_1)$
**CAAUP**: $\gamma \leftarrow (\gamma_1, ..., \gamma_m)$
*Compute* $\mathbf{s}$*: see below for difference*

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

**s-step Curdleproofs:**
for $1 \leq j \leq n$:
    $s_i = \sum_{j=1}^{m} \delta_j^{b_{i,j}}$, $b_{i,j} \in \{0,1\}$ 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:**
$ActivePos \leftarrow [(i,i), i = 1, \ldots, n]$
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 $k \geq h$:
            $b_{i,j} \leftarrow 1$
            $newPos = k - h - fs$
        else:
            $b_{i,j} \leftarrow 0$
            $newPos = k$
        $nextActivePos.push((i, newPos))$
    $ActivePos \leftarrow nextActivePos$
    $n \leftarrow h$
for $1 \leq j \leq n$:
    $s_i = \sum_{j=1}^{m} \delta_j^{b_{i,j}}$
    $s'_i = \sum_{j=1}^{m} \delta_j^{-b_{i,j}}$
```

Fig. 6: Optimized verifier computation for CAAU-IPA in CAAUrdleproofs



(a) Proving Time

(b) Verifying Time

Fig. 7: The timed results compared between CAAUrdleProofs and Curdleproofs

References

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

[2] 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.

[3] P. W. Foundation, “-safrole,” Accessed: 16-05-2025.

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

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

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

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

[8] 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.

[9] 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.

[10] 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.

[11] T. E. F. C. R. Team, “Curdleproofs,” Accessed: 24-04-2025.

[12] 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.

[13] ethereum.org, “Secret leader election,” 2024, Accessed: 22-10-2024.

[14] 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]`.

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

[16] 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.

[17] 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 3 (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*

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

Definition 4 (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 5 (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. CAAUrdleproofs is the Curdleproofs DL IPA on which the Springproofs protocol has been applied. So to help show that it is Honest-Verifier Zero-Knowledge (HVZK), we refer to Theorem 5 of Springproofs [17].

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 $\mathit{SIPA}_{\mathit{IPA}}(f)$ 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 $\mathit{SIPA}_{\mathit{IPA}}(f)$ 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 [11]. 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 [17]. 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 $|\mathbf{G}| \geq 8$ [11]. Therefore, following Theorem 1, CAAUrdleproofs must be HVZK when $n \geq 8$

For soundness and completeness, we refer to Theorem 3 of Springproofs [17].

Theorem 3 (Springproofs Theorem 3)**.** *Given a terminative SIPA(f), if the number of compression steps in SIPA(f) is $O(\log n)$, 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

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

or analogously for an additive cryptographic group:

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

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

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

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

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

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:

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

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 $O(\log n)$ 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 $|k| \geq 8$. □