On the Applicability of STARKs to Counted-as-Collected Verification
in Existing Homomorphically E-Voting Systems

Max Harrison and Thomas Haines*

Australian National University, Canberra, Australia thomas.haines@anu.edu.au

Abstract. Scalable Transparent ARguments of Knowledge (STARKs) are a kind of succinct zero-knowledge proof which do NOT require trusting any party to generate a Common Reference String (CRS). In this work, we examine the applicability of STARKs to improving Counted-as-Collected verification in the homomorphically tallied elections. In particular we are interested in using STARKs to allow very efficient tally verification while providing everlasting privacy to the information made available for public verification. This work provides a useful reference for the computational and verifiability trade-offs of using STARKs.

1 Introduction

A slew of various electronic voting protocols and systems (Helios, STAR-Vote, Belenios, ElectionGuard, etc.) follow a general approach pioneered by Benaloh and Yung [BY86] that utilises homomorphic encryption to preserve privacy while maintaining the verifiability of these systems. In this general approach, voters submit additively homomorphic ciphertexts which are then composed into an output ciphertext representing the vote tally. This output ciphertext can then be decrypted to reveal the vote tally, all while the underlying votes are never exposed and remain secure in their encrypted form. This approach preserves the privacy of voters while also maintaining verifiability of the voting protocol.

The exact verification procedure differs between voting systems but in general the verifier checks zero-knowledge proofs to ensure the (encrypted) ballots are well formed and then homomorphically combines these checking that the accumulated ciphertext decrypts to the claimed result, with the aid of more zero-knowledge proofs; optionally, the verifier may also check that the ballots came from valid voters by checking digital signatures or the like.

The intuition behind the approach we evaluate is that we get the tallier, or a third-party, to produce a STARK proof which says that all the evidence is valid with respect to the normal verification procedure; the verifier then checks the STARK proof rather than original evidence. We enhance this approach by committing (within a Merkle tree) to the original verification data in constant

* Thomas Haines is the recipient of an Australian Research Council Australian Discovery Early Career Award (project number DE220100595).

size which reduces the data required to verify dramatically; the statement proved by the STARK proof is then that the prover can open the commitment to values which satisfy the normal verification procedure.

1.1 Limitations of the existing approaches

On a per ballot basis the standard homomorphic tally approach is fairly efficient, at least for simple elections, but the cost of verification grows linearly in the number of ballots cast which quickly become prohibitive. In our running example of a referendum with 2^{24} voters (roughly the number of eligible voters in the Australian 2023 Voice referendum) the verification time is approximately 77 core-days and the size of the inputs to the verification is roughly 38 GB; these values will be explained in Section 3. While these values are certainly manageable they do pose a hurdle to the idealised world where each voter checks the election result for themselves.

In addition to the computational costs of verification, many Governments have been hesitant to release the data required for verification even when it existed; examples of this include the Estonian IVXV system [oE], the Swiss Post system [Swi21], and iVote system as used in New South Wales and Western Australia. While this hesitation is certainly multifaceted the privacy risk of doing so is certainly a significant part of the equation; for example, the lack of randomness in ballot generation [Gjø16] in the Norwegian system would have been much more disastrous if these ballots were published. Ideally, we like a system where the information released for verification perfectly hid the individual ballots à la everlasting privacy [HMMP23].

1.2 STARKs

There has been an orthogonal thread of research and development in cryptographic proof schemes within the last decade, enabling the construction of *succinct* ZK proofs - these novel proof schemes enable the construction and verification of ZK proofs in radically shorter time frames and data sizes. A prominent example of such a system is one introduced by [BSBHR18]: STARKs (Scalable Transparent ARguments of Knowledge) are a proof-of-computation scheme that have a number of attractive properties relevant to the development of e-voting systems. They are fully transparent, meaning they do NOT rely on any complex trusted setup ceremonies. They rely on very few cryptographic assumptions and are very modular to any specific implementation. STARKs are also highly parallelisable, allowing huge workloads to be distributed efficiently. Crucially, the STARK-constructed succinct ZK proofs are very efficient to both generate and verify, with verification time and proof size sublinear in the size of input data.

1.3 STARKs for Counted-As-Collected

There are many places in an electronic voting protocol where one might consider using STARKS, or Zero-Knowledge Succinct Non-Interactive Argument of

Knowledge (SNARKs) if one is willing to accept the trust assumption; to our knowledge all other previous works considered SNARKs but STARKs could be used instead at an increased computational cost. Huber *et al.* [HKK$^+$22] use them to prove that an election result (the outcome of the social choice function) is correct with regards to commitments to votes. Both Sheikhi *et al.* [SGS23] and Devillez *et al.* [DPP22] use it as part of ballot validity proofs. We, however, are interested in the applicability of STARKs to enhance existing homomorphically tallied election systems by increasing the efficiency and privacy of checking that the election result is correct with regards to the collected ballots (Counted-as-collected), and optionally eligibility verification as well.

While STARKs immediately fix the computational-time issue for the verifier they do not immediately fix the issue with verification input size since we would still need to send all the encrypted votes; we address this issue by committing to all the encrypted votes in a Merkle tree and sending the root of the tree along with the STARK proof, see Section 2 for details. In addition, there is no guarantee that the prover can produce the proofs for the statements we care about within the time-frame of an election. Our analysis shows that primary constraint on the prover, at least on the current state-of-the-art implementations, is not the execution time but rather the availability of Random Access Memory (RAM); we address this limitation by producing proofs which prove that batches of votes are: well formed, come from valid voters, and are correctly homomorphically accumulated. The homomorphically accumulated ciphertexts from each batch are then themselves combined and verifiable decrypted.

One of the advantage of using a Merkle tree is that we can provide logarithmic sized proofs of inclusion to voters that their ballot is among those tallied. In the specific instance we evaluated, which used digital signatures, this is not necessary since the fact that one of ballots has a signature valid with respect to the voter's public key serves to prove inclusion. However, in many cases digital signature are not used and the ability to prove Collected-as-Cast in sublinear time and space is important to the overall efficiency of the system; we note that care must be taken to ensure all internal nodes of the Merkle tree are perfectly hiding if this approach is taken and everlasting privacy is desired.

Our construction reduces the amount of time and data required to verify an election result by l orders of magnitude, at the cost of increasing the required RAM of the talliers by l orders of magnitude. Our construction hides the verification information behind a perfectly hiding commitment: a computationally unbounded adversary cannot extract any more information about individual votes than what is revealed by the vote result. We also present evaluation data for an example implementation of the protocol to practically characterise the asymptotic behaviour of the protocol in both proving and verification.

1.4 Contribution

We believe the approach we evaluate is essentially the obvious way to use STARKs in the context of existing homomorphically tallied voting systems; in that sense, we consider this work's primary contribution to be carefully presenting and evaluating this approach rather than the approach itself. The only downside of the approach we evaluate, that we are aware of, is the marked increase in difficulty in building an independent verifier; we note that this could be mitigated by having some verifiers directly verify the original proofs though we would need to trust those verifiers for everlasting privacy.

- We formalise a construction which reduces the amount of data and time required to verify by l orders of magnitude at the cost increasing the RAM the bulletin board needs by l orders of magnitude.
- The construction hides the verification information behind a perfectly hiding commitment so that even an unconstrained adversary cannot recover the individual votes being each commitment.
- The construction also allows proofs of collected-as-cast which are logarithmic in the number of voters.
- We characterise the asymptotic behaviour of both the prover and the verifier.

In our running example a tallier machine with $\sim$6TB RAM (relatively cheap in the context of modern data centres), reduces the verification time from approximately 77 core-days for a naive execution of the verification procedure to a projected time of 2.57 core-minutes. In addition, it reduces the sizes of the inputs from the naive 37.92GB to a total proof size of 1.54 GB, being the collective size of the STARK proofs for each batch.

Our evaluation code can download from the following link https://github.com/gerlion/STARKs_for_Homomorphic_Tallying.

2 Construction

For completeness we present a somewhat simplified protocol demonstrating the technique; this construction is not intended to be a contribution and is essentially [CGS97] with some small modifications. The e-voting protocol consists of the following participants:

- the election authority EA.
- the set of voters $V_1, \ldots, V_n$.
- the set of talliers $T_1, \ldots, T_m$.

The protocol requires some form of communication between the participants. We model this (following [CPP13]) through two append-only *bulletin boards*: the public bulletin board $\mathcal{PB}$ and the secret bulletin board $\mathcal{SB}$. All participants can read from and publish to the public bulletin board $\mathcal{PB}$. We assume the content of the secret bulletin board $\mathcal{SB}$ is only able to be read by the talliers $T_1, \ldots, T_m$, but anyone can write to the board $\mathcal{SB}$ via a private (but not anonymous) channel.[1]

[1] i.e. an outside observer can store the *metadata* of the channel but not the actual correspondence *data*. This corresponds to the assumptions made in the literature for *practical* everlasting privacy. We have practical constructions to realise such channels long-term, e.g. post-quantum TLS [SSW20].

The election authority EA is responsible for setting up the election (date, set of candidates, voting methods, etc.). The election authority EA provides the following election public information: the candidate list, the list of eligible voters' signature public keys, the list of valid talliers' signature public keys, and the public voting parameters. We assume that the authority EA has a valid signature public key pk_{EA} known to all participants.

The voters interact with the authority EA to register for an election. They publish their ballot information to $\mathcal{SB}$. The talliers are responsible for tallying the results - they read the ballot from the $\mathcal{SB}$, verify voter signatures and the ballot correctness proofs, compute the tally and commitment values, and publish the resulting tally, commitment values, and STARK proofs to the $\mathcal{PB}$.

For the election scheme, PK denotes the public ElGamal encryption key and sk denotes the secret decryption key such that $PK = sk \cdot G$, note we are using multiplicative notation since ElGamal will be defined over an elliptic curve in our evaluation. The secret key sk is jointly generated by the talliers using standard techniques, we denote sk_j as the jth share of the secret key.

2.1 Cryptographic primitives

The proposed e-voting protocol is composed of several cryptographic *primitives* and relies on several *proof systems.*

Definition 1 (ElGamal Encryption Scheme). *The* ***ElGamal encryption scheme*** *is a triple of PPT algorithms defined as follows:*

- $\mathrm{KeyGenE}(\lambda) \to (\mathcal{P}, K, sk)$*: on input of security parameter λ, it derives the encryption parameters $\mathcal{P} = (\mathcal{G}, q, g)$, chooses the secret key $sk \in \mathbb{Z}_q$, and computes the public key $K = sk \cdot G \in \mathcal{G}$, where $sk \cdot G$ denotes the generator element $G \in \mathcal{G}$ composed with itself sk times. The parameters $\mathcal{P}$ contains a cyclic group $\mathcal{G}$ of prime order q generated by G with group composition $\circ$, and $\mathbb{Z}_q$ is the additive subgroup of integers modulo q.*
- $\mathrm{Enc}(\mathcal{P}, m, K) \to (\alpha, \beta)$*: on input of a message $m \in \mathcal{G}$ and public key $K \in \mathcal{G}$, it chooses $r \in \mathbb{Z}_q$ and computes $(\alpha, \beta) = (r \cdot G, m \circ (r \cdot K))$.*
- $\mathrm{Dec}(\mathcal{P}, (\alpha, \beta), sk) \to m$*: on input a ciphertext (α, β) and secret key sk, it computes $m = \beta \circ (-sk \cdot \alpha) = (m \circ (r \cdot K) \circ ((r - sk) \cdot G)$.*

Definition 2 (Digital Signature Scheme). *A* ***digital signature scheme*** *$\mathcal{S}$ is a triple of PPT algorithms defined over a (finite) message space $\mathcal{M}$ and a signature space Σ as follows:*

- $\mathrm{KeyGenS}(\lambda) \to (sk, pk)$*: on input of security parameter λ, it chooses a signature key pair (sk, pk).*
- $\mathrm{Sign}(m, sk) \to \sigma$*: on input a message $m \in \mathcal{M}$ and a signing key sk, it outputs a signature $\sigma \in \Sigma$.*
- $\mathrm{VerifyS}(\sigma, m, pk) \to 0/1$*: on input a signature σ, message m, and verification key pk, it outputs 1 if it accepts the signature and 0 otherwise.*

We require that for a validly generated key pair $(sk, pk) \leftarrow \mathrm{KeyGenS}(\lambda)$ and for any message $m \in \mathcal{M}$, a valid signature $\sigma \leftarrow \mathrm{Sign}(m, sk)$ must be accepted:

$$\Pr[\mathrm{VerifyS}(\sigma, m, pk) = 1] = 1$$

The signature scheme used by the protocol should be secure: participants in the scheme should be convinced that the respective parties actually authored any published information in order for verification to be valid.

Authenticated Data Structures An authenticated data structure (ADS) allows a party to compute a short hash $H(L)$ of some sequence $L = (x_1, \ldots, x_n)$ so that the party can:

1. prove properties (e.g. membership and non-membership) of L with respect to $H(L)$.
2. *commit* to the sequence L, i.e. another party can receive $H(L)$ and know that the committing party cannot change the sequence L without changing $H(L)$.

An ADS scheme can be seen as an extension of the more general *commitment scheme*[2].

Definition 3 (ADS Scheme). *An* ***authenticated data structure scheme*** *$\mathcal{D}$ is a quadruple of PT algorithms defined over some input element space $\mathcal{X}$ as follows:*

- $\mathrm{SetupA}(\lambda) \rightarrow \mathcal{AP}$*: on input of security parameter λ, it derives commitment parameters $\mathcal{AP}$ including a description of some collision-resistant hash function $H : \mathcal{X} \rightarrow \mathcal{Y}$ where the input to H can either be single element in $\mathcal{X}$ or a pair of elements in $\mathcal{Y}$.*
- $\mathrm{Commit}(\mathcal{AP}, L) \rightarrow r$*: on input of commitment parameters $\mathcal{AP}$ and some list of elements $L = (x_1, \ldots, x_n) \in \mathcal{X}^n$, it derives a commitment value $r \in \mathcal{Y}$.*
- $\mathrm{Open}(\mathcal{AP}, i, x, L) \rightarrow \phi$*: on input of commitment parameters $\mathcal{AP}$, an index $1 \leq i \leq n$, an element $x \in \mathcal{X}$, and a list $L \in \mathcal{X}^n$, it outputs a membership proof ϕ that $x = x_i$ for $L = (x_1, \ldots, x_n)$.*
- $\mathrm{VerifyA}(\mathcal{AP}, i, x, r, \phi) \rightarrow 0/1$*: on input of commitment parameters $\mathcal{AP}$, an index $1 \leq i \leq n$, an element $x \in \mathcal{X}$, a commitment value $r \in \mathcal{Y}$, and a membership proof ϕ, it outputs 1 if it accepts the proof and 0 otherwise.*

For any sequence $L \in \mathcal{X}^n$ with a commitment value $r \leftarrow \mathrm{Commit}(\mathcal{AP}, L)$ and any element $x_i \in L$, a valid membership proof $\phi \leftarrow \mathrm{Open}(\mathcal{AP}, i, x, L)$ must be accepted:

$$\Pr[\mathrm{VerifyA}(\mathcal{AP}, i, x, r, \phi) = 1] = 1$$

[2] In a commitment scheme, a party commits to a general message m instead of committing to a sequence L. We can view the ADS scheme as a commitment scheme with the additional functionality of proving some desired properties about the committed message.

It should be hard for a party to commit to some sequence and be able to prove false properties about that sequence.

NIZK Ballot Correctness Proofs. To maintain voter privacy, the protocol aggregates voter ballots in their encrypted form. This is susceptible to voters encrypting non-valid plaintext votes: if a voter encrypts $m' = 100 \cdot G$ and submits this to a simple referendum (where valid votes are either 0 or 1) this could unfairly impact the final vote result. We thus want to able to form proofs of *ballot correctness* that $m \in \{0 \cdot G, 1 \cdot G\}$, but in such a way that we do not reveal the underlying vote value. This is achieved through a NIZK proof system $\Phi = (\mathrm{ProveB}, \mathrm{VerifyB})$ where the two PPT algorithms are defined as follows:

- $\mathrm{ProveB}(v, (\alpha, \beta), PK) \to \pi$: on input of a vote v, an ElGamal ciphertext (α, β), and an encryption public key PK, it outputs a NIZK **ballot correctness** proof π.
- $\mathrm{VerifyB}(\pi, (\alpha, \beta), PK) \to 0/1$: on input of a NIZK ballot correctness proof π, a ciphertext (α, β), and an encryption public key PK, it outputs 1 if it accepts the proof and 0 otherwise.

A *valid* proof $\pi \leftarrow \mathrm{ProveB}(v, (\alpha, \beta), PK)$ is one for which the vote v is a valid value for the respective election, the ciphertext (α, β) hides v, and the public key PK is the one used to encrypt (α, β). The NIZK proof system used by the protocol should be sound: only valid proofs (with overwhelming probability) should be accepted by VerifyB.

2.2 Definition of the Protocol

We first explicitly specify a relation R proved by the STARK framework in the protocol. For ciphertext c, NIZK ballot correctness proof π, signature σ, voter public signature key pk, and public encryption key PK, define the predicate p to accept valid signature and proof tuples, i.e.

$$p(c, \pi, \sigma, pk, PK) = \begin{cases} 1 & \text{if } [\mathrm{VerifyS}((c, \pi), \sigma, pk) = 1] \wedge [\mathrm{VerifyB}(\pi, c, PK) = 1] \\ 0 & \text{otherwise} \end{cases}$$

Then for private witness $\omega = (pk_1, \dots, pk_n, c_1, \dots, c_n, \eta)$ where

- $pk_1, \dots, pk_n$ is the list of voter public keys.
- $c_1, \dots, c_n$ is the list of voter ciphertexts.
- $\eta \in \mathbb{Z}_q$ is the secret hiding value.

and public inputs $\rho = (r_p, r'_c, C)$ where

- r_p is the commitment value to the list of voter public keys.
- r'_c is the product of the commitment value to the list of voter ciphertexts and the secret hiding value η.
- C is the product ciphertext of the voter ciphertexts.

define the relation R over public encryption key PK by the following conjunction:

$$\begin{aligned} R(\rho, \omega) : & [r_p = \text{Commit}(pk_1, \ldots, pk_n)] \\ & \bigwedge [r'_c = \text{Commit}(c_1, \ldots, c_n) \cdot \eta] \\ & \bigwedge \left[\bigwedge_{i=1}^{n} p(c_i, \pi_i, \sigma_i, pk_i, PK) \right] \bigwedge \left[C = \prod_{i=1}^{n} c_i \right] \end{aligned}$$

The protocol (see Fig. 1) is then defined in terms of four functions, i.e. $EP = (\text{SETUP}, \text{VOTE}, \text{TALLY}, \text{EXTRACT})$ and proceeds in five phases: *Setup*, *Voting*, *Tallying*, *Result*, *Verification.* The functions are specified as follows:

– $\text{SETUP}(\lambda, l, l') \rightarrow (\mathcal{PP}, SK, m_{EA}, \sigma_{EA})$ - on input a security parameter λ, list of voter public signature keys $l = \{pk_1, \ldots, pk_n\}$, and list of tallier public signature keys $l' = \{pk'_1, \ldots, pk'_m\}$:
  1. generates the public parameters $\mathcal{PP} = (\mathcal{P}, \mathcal{AP}, \mathcal{SP}, PK)$ where $\mathcal{AP} \leftarrow \text{SetupA}(\lambda)$, $\mathcal{SP} \leftarrow \text{SetupT}(\lambda)$, $(\mathcal{P}, PK, SK) \leftarrow \text{KeyGenE}(\lambda)$.
  2. for the parameter message $m_{EA} = (\mathcal{PP}, l, l')$, produces a signature $\sigma_{EA} \leftarrow \text{Sign}(m_{EA}, sk_{EA})$.

– $\text{VOTE}(v, pk, m_{EA}, \sigma_{EA}) \rightarrow (m_v, \sigma_v)$ - on input a vote value v, a voter private signature key sk, and a parameter message-signature pair (m_{EA}, σ_{EA}):
  1. attempts to verify the message $\text{VerifyS}(m_{EA}, \sigma_{EA}, pk_{EA}) \rightarrow 0/1$. On success it extracts the public parameters $\mathcal{PP}$, and halts on failure.
  2. encrypts the vote into a ciphertext $c \leftarrow \text{Enc}(v, PK)$.
  3. produces a NIZK proof of ballot correctness $\pi \leftarrow \text{ProveB}(v, c, PK)$.
  4. for the ballot message $m_v = (c, \pi)$, produces a signature $\sigma_v \leftarrow \text{Sign}(m_b, sk)$.

– $\text{TALLY}(m_{EA}, \sigma_{EA}, M_v, \Sigma_v, pk', SK_j) \rightarrow (m_t, \sigma_t)$ - on input a parameter message-signature pair (m_{EA}, σ_{EA}), a set of voter ballots $M_b = \{m_{v1}, \ldots, m_{vn}\}$ with signatures $\Sigma_v = \{\sigma_{v1}, \ldots, \sigma_{vn}\}$, a tallier public signature key pk', and a portion of the secret decryption key SK_j:
  1. verify both the parameter message $\text{VerifyS}(m_{EA}, \sigma_{EA}, pk_{EA}) \rightarrow 0/1$ and the voter ballots $\text{VerifyS}(m_{vi}, \sigma_{vi}, pk_i) \rightarrow 0/1$ together with NIZK correctness proofs $\text{VerifyB}(\pi_i, c_i, PK) \rightarrow 0/1$. For any ballots which fail either the signature or proof check reject the ballot.
  2. takes the product of all verified ciphertexts $C_j = \prod_{i=1}^{n} c_i$.
  3. computes commitment values $r_p \leftarrow \text{Commit}(pk_1, \ldots, pk_n)$, $r_c \leftarrow \text{Commit}(c_1, \ldots, c_n)$ for the included ballots.
  4. composes the ciphertext commitment r_c with a random hiding value $\eta \in \mathbb{Z}_q$ to obtain the output commitment value $r'_c = \eta \cdot r_c$.
  5. produces a STARK proof $\psi_j \leftarrow \text{ProveT}(\tau_j, \rho_j, \omega_j)$ for public inputs $\rho_j = (r_p, r'_c, C_j)$, private witness $\omega_j = (pk_1, \ldots, pk_n, c_1, \ldots, c_n, m)$, and computational trace τ_j asserts the relation $R(\rho, \omega)$.
  6. decrypts the vote result C_j into the partial plaintext Γ_j with the portion of the secret key $\Gamma_j \leftarrow \text{Dec}(\Gamma_j, SK_j)$.
  7. for the message $m_t = (\psi, \rho, \Gamma_j)$, produces the signature $\sigma_t \leftarrow \text{Sign}(m_t, sk')$.

– $\text{EXTRACT}(M_t, \Sigma_t) \rightarrow (\Gamma, \sigma_C)$ - on input a set of potential tally messages $M_t = \{m_{t1}, \ldots, m_{tm}\}$ with corresponding signatures $\Sigma_t = \{\sigma_{t1}, \ldots, \sigma_{tn}\}$:

1. attempts to verify the tally messages $\text{VerifyS}((\psi_j, \rho_j, \Gamma_j), \sigma_j, pk'_j) \to 0/1$ and verify the STARK proofs $\text{VerifyT}(\psi_j, \rho_j) \to 0/1$. If fewer than k tally messages succeed both checks, it halts execution.
2. aggregates the partial results $\Gamma_1, \ldots, \Gamma_m$ into the final result $\Gamma = \prod_{j=1}^{m} \Gamma'_j$.
3. produces the signature $\sigma_C \leftarrow \text{Sign}(\Gamma, sk_{EA})$.

The relation R is equivalent to the computational integrity of the function TALLY on input-output pair (ω, ρ): if the relation R holds then c is the valid product of the input ciphertexts, r_p and r'_c are valid commitment values, and each ciphertext is proven to be correct and from a valid public key.

Assuming that the used signature scheme is secure, the ADS scheme is secure, binding, and hiding, and the non-interactive proof systems used by the protocol are both sound and zero-knowledge, we intend the protocol to have:

– privacy against an efficient adversary, with access to the ballot ciphertexts.
– everlasting privacy against a computationally unbounded adversary with access to only published information and communication metadata.
– efficient collected-as-cast verification through the ADS membership proofs.
– efficient counted-as-cast verification through the STARK proofs.

3 Evaluation

This section discusses the details of an example implementation of the protocol, presents the collected performance data and projects it to larger inputs, and compares the resulting data to a naive verification of an election.

3.1 Implementation Details

The implementation is primarily written in Cairo0 [GPR21]. The Cairo runner is given the implementation program and voting input data ω, and outputs a compiled Cairo program and public inputs ρ. The compiled Cairo program, private witness ω, and public inputs ρ are passed to a STARK prover outputting a STARK proof ψ. The proof ψ and public inputs ρ can then be passed to a STARK verifier for verification of results. The implementation instantiates the protocol's cryptographic primitives using:

– **Elliptic Curve:** the STARK elliptic curve [Sta23a].
– **Hash Function:** the Pedersen hash function [HBHW22, Section 5.4.1.7].
– **Digital Signature Scheme:** the Elliptic Curve Digital Signature Algorithm (ECDSA) [JMV01].
– **ADS Scheme:** Functionality is provided either using Merkle trees or hash chains. The presented performance data is collected using Merkle trees. If the hash function used to construct a Merkle tree is collision resistant, the Merkle tree ADS scheme is provably secure [BS23, Theorem 8.8].
– **NIZK Proofs:** NIZK proofs of ballot correctness are implemented according to the ElectionGuard (Version 2.0.0) specification [BN23].

The Protocol

(*Setup Phase*) To setup an election, EA performs the following steps:

1: Aggregates $l = \{pk_1, \ldots, pk_n\}$ and $l' = \{pk'_1, \ldots, pk'_m\}$.

2: Generates $(PK, SK, m_{EA}, \sigma_{EA}) \leftarrow \text{SETUP}(\lambda, l, l')$.

3: Chooses a threshold tuple (k, m) and distributes PK_j to $T_1, \ldots, T_m$.

4: Writes (m_{EA}, σ_{EA}) to $\mathcal{PB}$.

(*Voting Phase*) To cast a vote v_i, the voter V_i with public key pk_i performs the following steps:

1: Reads a potential message-signature pair m_{EA}, σ_{EA} from $\mathcal{PB}$.

2: Generates $(m_{vi}, \sigma_{vi}) \leftarrow \text{VOTE}(v_i, pk_i, m_{EA}, \sigma_{EA})$. If execution is halted due to a failed
signature, restart at step 1.

3: Writes (m_{vi}, σ_{vi}) to $\mathcal{SB}$ via a private channel.

(*Tallying Phase*) To produce a partial vote result $\mathit{\Gamma}_j$, the tallier T_j with public key pk'_j performs the following steps:

1: Reads a potential message-signature pair m_{EA}, σ_{EA} from $\mathcal{PB}$.

2: Reads the set of submitted message-signature pairs $M_v, \mathit{\Sigma}_v$ from $\mathcal{SB}$.

3: Generates $(m_t, \sigma_t) \leftarrow \text{TALLY}(m_{EA}, \sigma_{EA}, M_v, \mathit{\Sigma}_v, pk', SK_j)$. If execution is halted due to a
failed parameter message signature, restart at step 1.

4: Writes (m_t, σ_t) to $\mathcal{PB}$.

(*Result Phase*) To announce an election result $\mathit{\Gamma}$, EA performs the following steps:

1: Reads the set of submitted tallier message-signature pairs $M_t, \mathit{\Sigma}_t$ from $\mathcal{PB}$.

2: Generates $(\mathit{\Gamma}, \sigma_C) \leftarrow \text{EXTRACT}(M_t, \mathit{\Sigma}_t)$. If execution is halted due to
too many failed signatures, restart at step 1.

3: Writes $(\mathit{\Gamma}, \sigma_C)$ to $\mathcal{PB}$.

(*Verification Phase*) To verify a result, each voter V_i can optionally perform the following steps:

1: Query the tallier T_j for the membership proof $\phi_j \leftarrow \text{Open}(i, c_i, \omega)$. The voter V_i can then
verify this proof $\text{VerifyA}(i, c_i, r'_c) \rightarrow 0/1$.

2: Verify the signature of the announced election result $\text{VerifyS}(\mathit{\Gamma}, \sigma_C, pk_{EA}) \rightarrow 0/1$.

3: Verify the set of submitted tallier message-signature pairs $M_t, \mathit{\Sigma}_t$ from $\mathcal{PB}$,
i.e. $\text{VerifyS}((\psi_j, \rho_j, \mathit{\Gamma}_j), \sigma_j, pk'_j) \rightarrow 0/1$.

4: Verify the set of submitted STARK proofs $\psi_1, \ldots, \psi_m$ by $\text{VerifyT}(\psi_j, \rho_j) \rightarrow 0/1$.

Fig. 1. Protocol Summary

– **Commitment Scheme:** Pedersen commitments [Ped91]; standard measures need to be taken to ensure the commitments are binding [HLPT20].

The group points of the STARK elliptic curve over the Cairo field provides the finite cyclic group used for ElGamal. The StarkWare STONE prover and verifier [Sta23b] is used to provide the STARK functionality. The STONE prover supports multithreading, so the implementation is easily able to take advantage of parallelism to reduce the proving time for higher core CPUs. The main limitation of the implementation is the amount of system memory required by the STONE prover.

3.2 Performance Data

Default STONE proof & prover parameters were used to produce STARK proofs at an estimated security level of 128 bits. There is a tradeoff in these parameters between proof security, proof size, proving time, and memory usage. Much more aggressive memory optimisations are possible at the cost of proving time and vice versa. The evaluation data was collected on the following machine:

Evaluation Machine Specifications

– CPU: AMD Ryzen 3 2200G - 4 cores, 4 threads @ 3.5GHz
– RAM: 16GB DDR4 - 2400 MT/s
– Operating System: Fedora Linux 38 (Workstation Edition) x86_64

We are limited in data collection by the system memory requirements: an input of 2^7 votes would have a minimum memory requirement for proving of $(2 \cdot 32 \cdot 2^{23} \cdot 27) \approx 14.5$GB, which cannot be run on the evaluation machine. We are thus limited to lengths of inputs up to 2^6 votes. We present implementation timing data is in Table 1, and the implementation memory data is in Table 2. Actual measured observations are stated in black, and projected values for larger inputs are presented in red. Projections are simplified versions of the proven theoretical asymptotic behaviour for STARKs: verification time and proof size are modelled logarithmically in the number of votes, whereas runner time, proving time, and peak memory usage are modelled linearly. This is likely slightly optimistic but presents a useful contextualisation of the asymptotic behaviour.

Due to the parallelism of the STONE prover, proving time is measured in *core-minutes* (core-min): the total amount of time measured for all assigned cores. This can be divided by the number of CPU cores assigned to a workload to yield the actual proving time - e.g. a proving time of 6.46 core-minutes is equivalent to a proving time of 1.62 minutes on the 4-core evaluation machine's CPU. Verification time is also measured in core-seconds.

We see the two main draw of STARKs in the performance data: verification time remains practically constant for all sizes of inputs, and proof size grows at an exponentially slower rate as compared to the sizes of the inputs. This comes at the cost of a linear increase in the amount of system memory required by the talliers. We can mitigate this memory requirement by some degree by configuring the prover parameters (at the cost of increasing the proving time), but we

Input Votes	Runner (min)	Proving (core-min)	Verification (core-s)
4	0.08	1.77	0.18
8	0.14	3.23	0.16
16	0.26	6.46	0.17
32	0.50	12.91	0.17
64	0.98	26.25	0.22
128	1.93	52.35	0.23
256	3.84	104.73	0.24
512	7.65	209.49	0.25
1024	15.28	419.01	0.27
2048	30.53	838.04	0.28
4096	61.03	1676.10	0.29
8192	122.04	3352.22	0.31
16384	244.06	6704.46	0.33
32768	488.09	13408.94	0.34

Table 1. Implementation timing data. Projections in red.

are fundamentally limited by the minimum lower bound memory requirement. Proving time represents the majority of the computational workload, but is likely fairly trivial for a server-grade machine that is able to readily exploit the prover's parallelism. We can handle larger tallies by running the protocol on batches of the input votes and combining each batch's results. This comes at the cost of a linear increase in verification time and proof size in the number of batches, but still allows protocol verification efficiency with orders of magnitude smaller than the naive verification. This results in even very large elections exhibiting significant reductions in the amount of time and size of the data required to verify when compared to naive verification.

We can exhibit this through a direct comparison between naive recomputation and the proposed protocol, see Table 3. As the EC point exponentiations represent the majority of the computational workload involved in the naive recomputation, we can form a estimated lower bound on the naive verification time for a given size of input votes: we simply take the product of the time taken for EC point exponentiations per vote and the number of input votes.

We present simple measurements taken for EC point exponentiations on the evaluation machine in Table 3. This allows us to construct direct comparisons between naive recomputation for verification and the proposed protocol: we present projected log-log plots comparing the verification times and proof sizes of the proposed protocol and the lower bounds of naive verification in Figure 2, assuming that the protocol is performed by a tallier who has access to 6TB of RAM (which can be acquired relatively cheaply in modern data centres).

We can give a concrete comparison for a real-life example: for a referendum of 2^{24} voters (roughly the number of eligible voters in the Australian 2023 Voice referendum) our implementation on the assumed tallier machine would batch this into about 2^9 batches of the proposed protocol; *note that the batches only include*

Input Votes	Inputs Size (MB)	Proof Size (MB)	Peak RAM (GB)
4	0.009	1.009	0.72
8	0.018	1.062	1.38
16	0.037	1.148	2.82
32	0.073	1.317	5.50
64	0.145	1.620	11.00
128	0.290	1.824	21.96
256	0.579	1.972	43.88
512	1.158	2.120	87.73
1024	2.315	2.267	175.41
2048	4.629	2.415	350.79
4096	9.258	2.563	701.53
8192	18.516	2.711	1403.03
16384	37.031	2.855	2803.02
32768	74.062	3.004	5612.01

Table 2. Implementation memory data. Projections in red.

validity and accumulation, decryption is still only done once on a ciphertext which accumulates all batches. This reduces the verification time from the naive recomputation's lower bound of approximately 77 **core-days** to a projected time of 2.57 **core-minutes** for the proposed protocol, and reduces the sizes of the inputs from the naive 37.92 GB to a total proof size of 1.54 GB.

Measurement	Value
Time for 1024 EC point exponentiations:	36.899 (core-s)
Time per EC point exponentiation:	0.036 (core-s)
Number of exponentiations per input vote:	11
Time of exponentiations per input vote:	0.396 (core-s)

Table 3. Measurements on the evaluation machine for naive estimation.

4 Conclusion

We hope to have convinced the reader that STARKs provide very interesting trade-offs for homomorphically tallied voting systems. The advantage confers large reductions in several metrics for verification in practice, dramatically decreasing the level of necessary computational resources to verify a given result - enabling election verification on consumer-grade hardware (e.g. laptops and smartphones). This empowers voters to be able to verify an election result independently, increasing trust in the overall democratic process; the main downside of the approach appears to be the increased difficult in implementing a fully independent verifier. The approach is ultimately limited by the computational resources that the talliers have access to (both in computing power and system

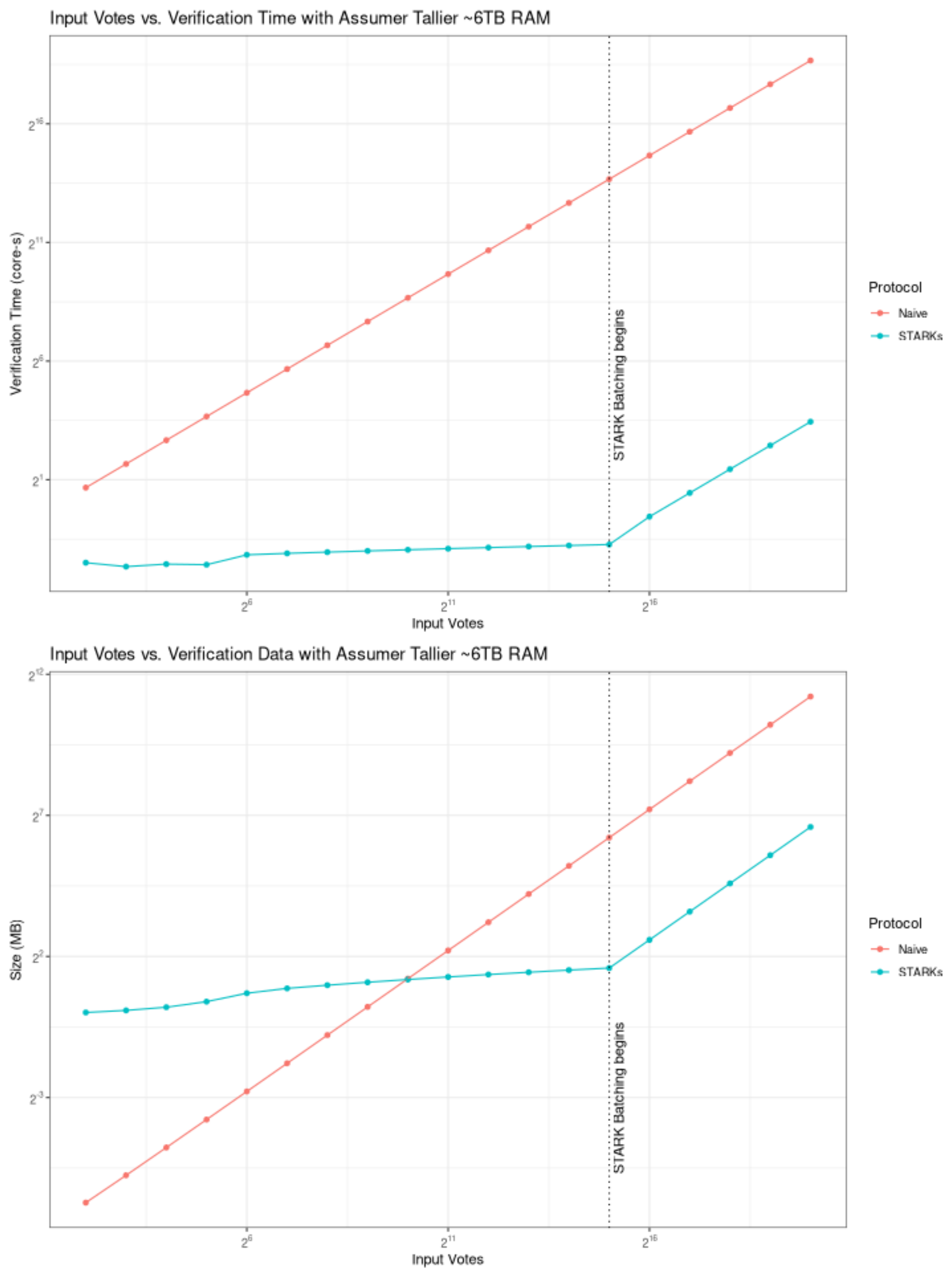

Fig. 2. Double logarithm plots of projected verification time and data with an assumed tallier of ∼6 TB RAM

memory), but this is assuaged by the fact that this role would likely be performed by a set of server-grade machines. Our key takeaways are as follows:

Prover Effort The proving time does not appear to become prohibitive even if we consider large elections, particularly since the prover parallelises very efficiently. However, the RAM required quickly becomes a constraint. In some cases this can efficiently be dealt with by proving the normal verification procedure in batches; however, this technique does not seem to generalise easily beyond homomorphically tallied elections.

Verifier Effort The verifier's computational time is not constant as might have been hoped for due to the requirement of batching the prover. However, the many orders of magnitude which is practicality achievable is enough make the verifiers work attractively small. The amount of data which needs to be sent to prover can be made very small for homomorphically tallied elections.

4.1 Future work: Mix-net Based Systems

Adapting the approach for *mix-net* based systems would provide support for ranked voting elections or systems supporting write-ins. Based on our analysis we would expect the prover time to increase compared to homomorphic tallied elections by the same factor the normal verification procedure time is increased. However, the issue of batching the mixing would result in a significant loss of privacy; finding a batching method for mix-net verification which does not incur a privacy penalty would be of great use.

The attractiveness of STARKs for mix-nets compared to homomorphic tallying is limited by the normal inclusion of all the ballots in plaintext in the tally; the number of ballots is linear in the number of voters which limits the size advantage of STARKs; there doesn't seem to be anyway to compress this in general without also proving the correction execution of the underlying voting methods' social choice function on the plaintext ballots.

References

[BN23] Josh Benaloh and Michael Naehrig. Electionguard design specification (version 2.0.0). Technical report, Microsoft Research, 2023.

[BS23] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. Version 0.6, January 2023.

[BSBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive*, 46, 2018.

[BY86] Josh Cohen Benaloh and Moti Yung. Distributing the power of a government to enhance the privacy of voters (extended abstract). In *PODC*, pages 52–62. ACM, 1986.

[CGS97] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. *European Transactions on Telecommunications*, 8(5):481–490, 1997.

[CPP13] Edouard Cuvelier, Olivier Pereira, and Thomas Peters. Election verifiability or ballot privacy: Do we need to choose? In *ESORICS*, volume 8134 of *Lecture Notes in Computer Science*, pages 481–498. Springer, 2013.

[DPP22] Henri Devillez, Olivier Pereira, and Thomas Peters. How to verifiably encrypt many bits for an election? In *ESORICS (2)*, volume 13555 of *Lecture Notes in Computer Science*, pages 653–671. Springer, 2022.

[Gjø16] Kristian Gjøsteen. *Real-World Electronic Voting: Design, Analysis and Deployment*, chapter E-voting in Norway, page 103. CRC Press, 2016.

[GPR21] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a turing-complete stark-friendly cpu architecture. *Cryptology ePrint Archive*, 1063, 2021.

[HBHW22] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification 2022.3.8 [nu5]. Technical report, Electric Coin Company, 2022.

[HKK$^+$22] Nicolas Huber, Ralf Küsters, Toomas Krips, Julian Liedtke, Johannes Müller, Daniel Rausch, Pascal Reisert, and Andreas Vogt. Kryvos: Publicly tally-hiding verifiable e-voting. In *CCS*, pages 1443–1457. ACM, 2022.

[HLPT20] Thomas Haines, Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. How not to prove your election outcome. In *IEEE Symposium on Security and Privacy*, pages 644–660. IEEE, 2020.

[HMMP23] Thomas Haines, Rafieh Mosaheb, Johannes Müller, and Ivan Pryvalov. Sok: Secure e-voting with everlasting privacy. *Proc. Priv. Enhancing Technol.*, 2023(1):279–293, 2023.

[JMV01] Doc Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1:36–63, 2001.

[oE] State Electoral Office of Estonia. Ivxv online voting system. https://github.com/vvk-ehk/ivxv.

[Ped91] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 1991.

[SGS23] Maryam Sheikhi, Rosario Giustolisi, and Carsten Schürmann. Receipt-free electronic voting from zk-snark. In *SECRYPT*, pages 254–266. SCITEPRESS, 2023.

[SSW20] Peter Schwabe, Douglas Stebila, and Thom Wiggers. Post-quantum tls without handshake signatures. *Cryptology ePrint Archive*, 534, 2020.

[Sta23a] StarkEx. STARK curve. https://docs.starkware.co/starkex/crypto/stark-curve.html, 2023. Accessed: 2023-10-12.

[Sta23b] StarkWare. Stone prover. https://github.com/starkware-libs/stone-prover, 2023. Accessed: 2023-10-12.

[Swi21] Swiss Post. Swiss post voting system. https://evoting-community.post.ch/, 2021.