Verifiability Analysis of CHVote

David Bernhard, Véronique Cortier, Pierrick Gaudry, Mathieu Turuani, Bogdan Warinschi

Verifiability Analysis of CHVote

David Bernhard[1], Véronique Cortier[2], Pierrick Gaudry[2], Mathieu Turuani[2], and Bogdan Warinschi[1]

[1]University of Bristol
[2]Université de Lorraine, Inria, CNRS

October 30, 2018

Executive summary

This document details analyses of verifiability properties of the CH-Vote v1.3 electronic voting protocol, as defined by the preprint publication [12]. Informally, these properties are:

- Individual verifiability: a voter is convinced that a ballot confirmed as coming from the voter contains his intended vote
- Ballot verifiability: all ballots that are confirmed contain correct votes
- Eligibility uniqueness: there are no two distinct entries in the list of confirmed ballots which correspond to the same voter
- Confirmed as intended: if a confirmed ballot is on the bulletin board for some voter, then that ballot records that voter's voting intention
- Universal verifiability: any party can verify that the votes on this board were tallied correctly

The analyses employ the currently well-established approach used within the scientific community. Specifically, they rely on mathematical abstractions for the adversary and for the system under analysis, as well as mathematical formulations of the properties to be established. Mathematical proofs are then used to establish that (under certain assumptions) the security properties hold.

We provide two types of analysis (which differ in the level of abstraction at which they operate). Part I contains a pen-and-paper computational/cryptographic analysis. Part II describes an automated symbolic analysis.

Broadly speaking, both the symbolic and the computational analyses conclude that CH-Vote satisfy the desired security properties under several assumptions. The assumptions include, for example, computational assumptions (which mathematical problems are assumed to be hard), trust assumptions (which parties, if any, are assumed to behave honestly and what are parties assume to know before they interact with the system).

Besides the concrete mathematical statements the analyses led to a number of recommendations which aim to improve the security. Part III concludes with a number of recommendations which reflect assumptions made in the analyses and weaknesses that were identified. The recommendations also sum up the results of a (light) code review of the code available via GitHub[1] – commit `9b0e7c9fcd409`, from April 2017.

[1] `https://github.com/republique-et-canton-de-geneve/chvote-protocol-poc`

Contents

Part I.
Cryptographic Analysis

1. Preliminaries

Here we introduce some of the notation used throughout the document.

Notation	*Meaning*
$[n]$	The set $\{1, \ldots, n\}$, where n is a positive integer.
$x \leftarrow a$	Assignment to a variable.
$x \twoheadleftarrow S$	Uniform random sampling from a set.
$\perp$	An output that denotes an algorithm has failed.
	$\perp$ evaluates to false when used as a condition, e.g. in an IF statement.
$f : X \to Y$	A function (e.g. deterministic algorithm) with domain X and range Y.
$\mathbf{d}_i$	The i'th entry in the vector $\mathbf{d}$
$A : X \twoheadrightarrow Y$	A randomised algorithm with domain X and range Y.
$\mathbf{d}_{\ominus I}$	All positions except the I-th of vector $\mathbf{d}$.
$\langle\,\rangle$	An empty list.
$\langle a, b, \ldots \rangle$	A list with elements $a, b, \ldots$
$\mathcal{A}$	An adversary in a security game.

We view an adversary as a stateful process: if $\mathcal{A}$ is called multiple times, its state is shared across the different invocations. To be able to easily refer to the different invocations of the same adversary, we sometimes call them $\mathcal{A}_1, \mathcal{A}_2, \ldots$ etc.

2. Principles of Game-Based Proofs

Cryptographic games

In the approach used in this part of our analysis, security of cryptographic schemes is defined by considering a "game" which involves an adversary against the scheme. The game defines precisely how the adversary can interact with the scheme and also specifies when the adversary wins. A proof of security is a mathematical argument that bounds the probability of the adversary winning by some (acceptable) quantity.

Many cryptographic proofs proceed as follows: start with any adversary $\mathcal{A}$ who wins a security game with some probability α. Repeatedly modify the game until it admits a reduction to a known hard problem, such as taking discrete logarithms. All the time, keep track of how the adversary's winning probability is affected by the changes to the security game. The result is a theorem in a style generally known as *concrete security* which says that if the adversary breaks the security game with probability α then the reduction breaks the hard problem with probability at least $f(\alpha)$. The function f is the loss function of this security theorem.

We use the following types of game transformations in our security proofs, which are commonly called *game hops*. We also give the loss function associated with each type of game hop.

1. *Code rewriting.* Modify the code of the game in ways that does not affect its input/output behaviour, for example renaming variables, expanding subroutine definitions etc. Hops of this kind serve mainly as preparation for other hops. If the adversary had probability α of winning the game before the hop, then it still has probability α of winning the game afterwards.

2. *Information-theoretically perfect hops.* These are mainly applications of Lemma 6.3. Technically a subset of subsystem switching hops, an i.t. perfect hop does not change the input/output distribution of the game, so an adversary with winning probability α before the hop retains the same probability after the hop.

3. *Fail early.* This is the first of three types of hops in which we introduce extra conditions that make the adversary lose the game. In this case, we introduce a line "if E then abort" where E is some event after which we know the adversary cannot win the game; usually this means checking one of the winning conditions early in the game rather than at the end. The benefit of this hop is that for the rest of the game we can assume $\neg E$ in our analysis. The adversary's winning probability is unaffected by a hop of this type.

4. *Independent abort.* In this type of hop, we introduce a line "if E then abort" where E is some event that is independent of the adversary's winning probability. This reduces the winning probability after the hop from α to $\alpha \cdot (1 - e)$ where e is the probability of event E occurring.

   If W is the event that the adversary wins before the hop and W' the event that it wins after the hop, then $\mathbf{Pr}[W'] = \mathbf{Pr}[W \wedge \neg E] = \mathbf{Pr}[W](1 - \mathbf{Pr}[E])$ using independence of W and E.

   Often, this hop takes the form "$d \twoheadleftarrow D$; if $d = d_0$ then abort" where d_0 is some previously computed value. However d_0 was chosen, since d is uniform in D the probability of aborting is $1/|D|$ so the adversary's winning probability drops from α to $\alpha' = \alpha \cdot (1 - 1/|D|)$.

5. *Shoup's Difference Lemma.* This is an alternative to the previous technique that gives an additive rather than multiplicative loss function. Shoup [16] defines the lemma as follows: if A, B, E are events on some probability space and $A \wedge \neg E \iff B \wedge \neg E$ then $|\mathbf{Pr}[A] - \mathbf{Pr}[B]| \leq \mathbf{Pr}[E]$.

   If we take a game that the adversary wins with probability α and add a line "if E then abort", then define A to be the event that the adversary wins the original game and B the event that the adversary wins the modified game, then if the conditions of the lemma are satisfied we get that the advantage against the modified game is at least $f(\alpha) := \alpha - e$ where e is the probability of event E.

   Typically, the difference lemma is used when the probability of event E is negligible and we wish to argue that if the original advantage α was negligible, then so is the new advantage $\alpha - e$.

6. *Subsystem switching.* In this type of hop, we write our game G as a composition RS of two systems R, S and change the game to $G' = RS'$ where R can be read as *reduction* and S, S' as *subsystems*. An example would be replacing a subsystem that produces a DDH triple with one that produces a triple of random, independently chosen group elements. Where, as in this example, the subsystems represent a problem assumed to be cryptographically hard, the subsystem switching hop reduces to the hard problem.

   If we know that the distinguishing advantage of subsystems S, S' is bounded by δ then any adversary $\mathcal{A}$ who wins against $G = RS$ with probability α will still win[2]against $G' = RS'$ with probability at least $\alpha' = \alpha - \delta$, since $\mathcal{A}R$ can be used as a distinguisher between S and S'.

7. *The forking lemma.* In games where the adversary outputs a Fiat-Shamir-Schnorr style ZK proof of knowledge, we can modify the game to assume that the adversary also outputs a witness, using the forking lemma of Bellare and Neven [1].

   Since such proofs are simulation-sound extractable, we can perform this hop even if we have previously simulated proofs in the random oracle model, as long as we prove that the adversary cannot return one of our simulated proofs.

   The forking lemma is the most costly type of reduction: if the adversary previously succeeded with probability α then the new success probability is $\alpha^2/N_Q - \alpha/|C|$ where N_Q is the number of random oracle queries made by the adversary and C is the challenge space of the PoK.

8. *Removing an abort condition.* If a game contains a line "if E then abort" and some adversary wins the game with probability α, then the adversary will certainly still win the game with probability α if we modify the game by removing this line. The reason that we would do this is that we sometimes use events E that are not efficiently checkable or have a dependency on some secret, which we want to get rid of in order to turn the game into a reduction against some problem in the next step where we no longer have the secret.

[2]To formalise winning, one could define it as the game returning 1. Since the value returned to the adversary is produced by the reduction R and not the subsystem, the event that the adversary wins the switched game is still well-defined.

The Random Oracle Model

CH-Vote employs hash functions in several different ways. In our analysis we model these as random oracles. The idea is to model each hash function as an oracle which can be accessed by all parties in the game (adversary and protocols); the oracle implements a random function (i.e. for each input x a freshly selected uniformly random output is selected if x was not queried before; otherwise the oracle returns the previously selected output associated to x).

In this model, security games are in charge of initializing and maintaining the random oracle and answering the questions of all parties (i.e. honestly run algorithms which need access to the hash function and the adversary). A typical way to obtain games in the random oracle model is to design them ignoring the random oracle and then, when analyzing systems which employ hash functions modeled as random oracles, compute all honest invocations of the hash functions via calls to this oracle and allow the adversary to call the oracle as often as it wishes.

In the next section we will describe security games ignoring the random oracle. However, in the instantiation of games where we analyze CH-Vote we will consider games which, as explained above, will maintain the random oracle, even if we do not always show this explicitly.

The Forking Lemma

The Forking Lemma of Bellare and Neven [1] is a tool for reasoning about the soundness of Schnorr-style Zero-Knowledge proofs. In its original, highly abstracted form it considers the probability that two runs of an algorithm will lead to the same result:

Lemma 2.1 (Forking Lemma of [1]) *Let ν be an integer. Let $\mathcal{A}$ be a randomised algorithm that can sample up to ν elements uniformly from a domain of size d, and which outputs either $\perp$ (failure) or an integer in $[\nu]$, which we call a success. If α is the probability that a single run of $\mathcal{A}$ succeeds then the probability β that two successive runs of $\mathcal{A}$ succeed and both output the same integer value is at least*

$$\beta \geq \frac{\alpha^2}{\nu} - \frac{\alpha}{d} \quad \textit{or equivalently} \quad \alpha \leq \frac{\nu}{d} + \sqrt{\nu \cdot \beta}$$

The lemma is used to reason about Schnorr-type proofs in the random oracle model where d is the size of the challenge space and ν is the number of random oracle queries that the adversary[3] can make. The adversary itself need not be computationally bounded, as long as the number of random oracle queries that it makes is not exponentially large. The event that the adversary outputs an integer i is mapped to the event that the adversary produces a valid proof using its i-th random oracle query.

This way, we can conclude that if an adversary makes a valid proof with probability α — which is often the winning condition for a security game — then we can replace the adversary with a reduction that additionally outputs the witness to the proof with probability at least $\beta \approx \alpha^2$.

[3] To be precise, usually this is the number of random oracle queries made by both the adversary and any reductions we are currently working with.

Simulation-Sound Extraction

A common way to build verifiable protocols is for a party who should compute something (like the tally of an election) to attach a zero-knowledge proof that they have computed correctly. The most common implementation of such proofs are known as Fiat-Shamir proofs, which require the random oracle model for their security analysis.

The properties of these proofs together with the Forking Lemma of Bellare and Neven [1] imply that, if we have an adversary A who outputs a computation result and a valid proof then there also exists an adversary A' who is related to A in all respects except that the probability of successfully making a proof is roughly squared (thus making it smaller) but when A' does make a proof then it additionally outputs a witness to the proof.

Sometimes, a reduction needs to provide the adversary with a "proof" even though it does not have access to the necessary witness, or such a witness does not exist in the first place. In the random oracle model, a reduction can *simulate* a proof in this case by modifying the random oracle.

The question of whether one can still extract a witness from an adversary who has interacted with a proof-simulating reduction is an important one in the theory of Zero-Knowledge proofs. For Fiat-Shamir proofs, the answer is yes; the property that allows this is known as *Simulation-Sound Extraction* (SSE). We will make use of SSE in some of our security games.

The discrete logarithm problem

The security of the CH-Vote protocol relies on the hardness of the discrete logarithm problem in finite group. Given group $\mathbb{G}_{\widehat{q}}$ of order q and a generator g of the group, we define the advantage of an adversary $\mathcal{A}$ against the discrete logarithm problem by:

$$\mathbf{Adv}^{\mathsf{dlog}}_{\mathbb{G}_{\widehat{q}},\mathcal{A}} = \mathbf{Pr}[x \twoheadleftarrow [q]; y \leftarrow \mathcal{A}(g^x) \ : \ x = y]$$

3. Principles of Verifiability Games

In voting schemes, we assume a vote space V for all voters (though one could restrict different classes of voters to different subsets of V w.l.o.g.), a result space R and a result function $\rho : V^* \to R$ which maps lists of votes to results (where $V^* := \cup_{n\in\mathbb{N}} \times_{i=1}^{n} V$). This allows us to define the correct result for a list of votes as the result of applying the result function to these votes. A voting protocol should aim to calculate the result $r = \rho(v_1, \ldots, v_n)$, where v_i is the vote of user i in such a way that each vote stays secret, yet there are clear guarantees that the r that had been calculated is the real result. Verifiability refers to different aspects of these latter guarantees.

Before we come to the specific properties that make up verifiability, we recall "single-pass voting" [3, 4, 2], an existing model for voting schemes used in current literature to analyze voting schemes. In particular, the state of the art of verifiability is a 2016 publication [9] by Cortier et al. which defines different verifiability properties for single-pass voting schemes. While CH-Vote is not a single pass scheme, we use this model to discuss our approach to defining the different verifiability properties and later in this document we explain how we extend the model to encompass schemes like the CH-Vote voting protocol.

3.1. Single-Pass Voting

Roughly, one can describe single-pass schemes as a collection of protocols. Different papers vary in the exact model, but the essential features are the same in most models. In particular, it is always assume that the voting protocols uses a bulletin board to which voters send their ballot. In the single-pass model, a real voting scheme contains at least the following protocols:

1. Setup is run jointly by the election administrators, producing as output some election public data pk, private data for each administrator $\mathbf{sk}$ and secret data for each voter $\mathbf{vsk}$. Depending on the trust model, the adversary may be involved in the setup procedure: we write $(pk, \mathbf{sk}, \mathbf{vsk}) \leftarrow \mathsf{Setup}_{\mathcal{A}}(\mathsf{params})$ for the process of running the setup algorithm, with the adversary's involvement, but leave unspecified how precisely is the adversary allowed to interact with the schem. When defining the various security properties that we consider, we spell out the trust model and the abilities of the adversary in tampering with the setup phase.
2. Vote is run by each voter, taking as input the election public data and her vote and producing a ballot as output.
3. Check is run by the bulletin board on input a new ballot and the election public data. It outputs a Boolean value to indicate if the ballot is valid for this election.
4. Check2 is run by the bulletin board on input a new ballot and its current state. Again it outputs a Boolean value (the purpose of this algorithm is to model a board rejecting anotherwise valid ballot from a voter who has already voted).
5. Tally is jointly run by the administrators, taking as input their private inputs and a bulletin board. It outputs either an error symbol $\perp$ (if something is wrong with the board) or a tally, usually comprising an election result and auxiliary data (which contains e.g. proofs of correct tallying).

6. Verify takes as input election public data, a bulletin board and a tally. It outputs a Boolean value to indicate if the election was conducted correctly or not. This algorithm is deterministic.

The protocols for the different parties are as follows:

- A voter reads election public data from the board, runs Vote on this public data and her vote to get a ballot, then sends the ballot to the board.
- An election authority starts by participating in Setup, receiving some private data. The authority then waits until the election has finished and participates in Tally, run on her private data and the bulletin board.
- The bulletin board begins by receiving public data from the administrators. Then, for each voter who sends a ballot, it runs first Check on the ballot alone and then Check2 on the ballot and its current state. If both these algorithms return true, it appends the ballot to its current state, otherwise it rejects the ballot and leaves its state unchanged. The bulletin board sends its entire state to any party that requests it, at any time.
- A verifier runs Check on each ballot on the board individually, then Verify on the whole board. If these algorithms return true, the verifier accepts the election, otherwise it rejects the election.

3.2. Formatting assumptions

To formally specify the verifiability games we use, we make several assumptions on the protocol. In particular, these assumptions are satisfied by CH-Vote.

Explicit ballots We assume that following an election, any bulletin board BB determines a unique sequence of voter-ballot pairs, that is BB yields a unique sequence

$$((v_1, b_1), (v_2, b_2), \ldots, (v_t, b_t))$$

where v_i is a voter identifier and b_i is a corresponding ballot.

Explicit confirmation For each pair (v, b) on BB it is possible to publicly and efficiently determine if the ballot has been *confirmed* (and therefore that it should be included in the tally).

Committing ballots and Extractors Bernhard et al. [2] define the concept of an extractor which allows one to establish a mathematical definition of "a ballot b contains a vote v". Their extractor is an algorithm which takes a ballot and the election secret data and outputs either a vote (indicating a valid ballot) or a special symbol $\perp \notin V$ to indicate an invalid ballot. Extracting from an honestly created ballot must return the vote used to create that ballot, e.g. we have a property such as $\mathsf{Extract}(\mathsf{Vote}(pk, v), sk) = v$ for any valid election keypair (pk, sk) and any $v \in V$. If ballots contain encryptions of votes, the extractor corresponds to decrypting a ballot individually, which would never happen in a real election for privacy reasons but this property is useful in security modelling.

In this document we consider a similar extractor, but one which takes only a ballot and public election data as input (and makes use of no secret data). Instead we let the

extractor be a computationally unbounded algorithm[4]. This gives us an even cleaner mathematical definition of a ballot containing a vote that is impervious to e.g. malicious administrators preparing two distinct secret keys matching the same public key, which is possible in some encryption schemes[5]. We write $\mathsf{Extract}(b, pk)$ for the result of running the extraction algorithm on ballot b and public key pk. The result is either a vote in V or $\perp$.

A properly specified extractor yields some useful correctness definitions:

- The Vote algorithm is correct if extracting from a ballot created by this algorithm returns the original vote.
- A ballot is correct if the extractor, running on this ballot, does not return $\perp$.
- An election tally is correct if the claimed result matches what one would get by extracting the vote from each ballot[6] and then running the result function ρ on the extracted votes.

Since different voters in CH-Vote may have different voting rights, we will actually split our extractor into two parts: an algorithm E which (inefficiently) obtains the encrypted vote and the algorithm Extract itself which takes the output of E as well as the voter-specific eligibility information.

[4]This is fine as we will only ever use it in security games — no-one ever needs to run the extractor in reality.

[5]For example, one could modify ElGamal to have an extra bit on the secret key that has the effect of adding 1 to any decryption. Together with a compatible ZK proof scheme one could create a voting scheme that allows the decryptors to selectively add votes to the result, which verifiable voting schemes should obviously not tolerate.

[6]This formulation would declare a tally incorrect if any of the ballots on the board is incorrect. Alternatively, one could formulate it as "the result matches the result function run on all the *correct* ballots on the board".

4. Verifiability Notions

Early papers on verifiability take one of two approaches. In the first approach that is sometimes called *chain of custody*, it is required that a vote is cast as intended, recorded as cast and tallied as recorded. If all these properties hold, one may call a scheme *end-to-end verifiable*.

The second approach traditionally identifies two properties: Individual Verifiability (IV) deals with checks that can be done only be a voter themselves (such as that their ballot appears on the bulletin board) and Universal Verifiability (UV) which deals with checks that can be performed by an independent judge or any member of the public.

Some papers alternatively use the term UV to refer to the union of all verifiability properties that one wants a scheme to have. Other more recent work splits part of what was earlier considered part of UV into a separate property called eligibility verifiability (EV). When encountering the term UV it is worth noting therefore that this term means different things in different papers.

Both of these approaches have not, until recently, been formulated at a level of precision that is expected of cryptographic security proofs nowadays. It turns out that both approaches fail to adequately capture the ways in which a dishonest voter might be able to contribute a ballot containing something other than a valid vote, for example multiple votes.

We base our security model on the latest work by Cortier et al. [9] in their review paper of verifiability notions and we identify four distinct verifiability properties:

- Individual Verifiability (IV): a voter can check that their vote has been recorded correctly. This is the only property that can only be checked by the voter, not the general public and it roughly matches the cast-as-intended and recorded-as-cast properties in the chain of custody approach.
- Ballot Verifiability (BV): it is possible to determine if a ballot contains a correct vote or not. This property protects against dishonest voters.
- Eligibility Verifiability (EV): all ballots (that contribute to the tally) have been cast by eligible voters, and each eligible voter has at most one contributing ballot which corresponds to their intention.
- Universal Verifiability (UV): in our formulation, this is the property that anyone can check if the claimed election result corresponds to the ballots on the board (roughly matching tallied-as-recorded).

The property that we want a voting scheme to have, *end-to-end verifiability*, is that the scheme has all four of our properties.

Below we take each security property in turn and explain how we formalize it as a cryptographic game. Later in this document we specialize these games to the case of CH-Vote.

4.1. Individual verifiability

Individual verifiability is the property that a voter is convinced that his vote has been confirmed as intended. It prevents for example a dishonest voting machine to cast a ballot which does not correspond to the voter's intention. The CH-Vote requirement is as follows.

$\mathbf{Exp}^{\mathsf{iv}}_{\mathcal{A},\Pi}(\mathsf{params})$
$(pk, \mathbf{sk}, \mathbf{vsk}, \mathsf{BB}) \leftarrow \Pi.\mathsf{Setup}_{\mathcal{A}}(\mathsf{params})$
$\mathbf{s} \leftarrow \mathcal{A}(pk, \mathbf{sk}_{\ominus 1})$
$\mathcal{A}^{V(\mathbf{vsk}_1,\mathbf{s}),\mathsf{BB},E(\mathbf{sk}_1)}(pk, \mathbf{vsk}_{\ominus 1})$
If $\mathsf{bad}_{\mathsf{iv}}$ return 1

Figure 1: Game for defining individual verifiability

> "After submitting an encrypted vote, the voter receives conclusive evidence that the vote has been cast and recorderd as intended. This evidence enables the voter to exclude with high probability the possibility that the vote has been manipulated by a compromised voting client. [...] The probability of detecting a compromised vote must be 99.9% or higher."

We define *individual verifiability* using the game in Figure 1. The game starts with the initialization of the public voting parameters, as well as individual voters. These include generating the public keys for the election autorities **pk** and the corresponding secret keys **sk**. We write $\Pi.\mathsf{Setup}_{\mathcal{A}}$ to indicate that the adversary may take part in the overall setup of the execution. The adversary takes over all but one voter and all but one election authorities. The game therefore models the interaction of the adversary with the following parties:

- The honest voter 1 which runs the voting protocol using voter secret information $\mathbf{vsk}_1$ and voter selection **s** chosen by the adversary. The selection is made by the adversary after he sees the public election information.
- An honest election authority E which has as input some authority secret information **sk**. Without loss of generality we assume that this is the election authority 1 and its secret key is $\mathbf{sk}_1$.
- The bulletin board BB.

The adversary engages in arbitrary executions with these parties. When its execution finishes, the game returns 1 (to indicate that the adversary has achieved his goal) if event $\mathsf{bad}_{\mathsf{iv}}$ (defined below) occurs.

To define this event, we require the assumptions set forth in Section 3.2, namely that the bulletin board is well structured, and that ballots are committing (and therefore that one can extract, inefficiently, the vote that they encode). Furthermore, we assume that each voter outputs a boolean variable happy which indicates if it believes that the voting process has completed successfully.

Event $\mathsf{bad}_{\mathsf{iv}}$ occurs if the output of voter 1 is $\mathsf{happy} = \mathit{true}$ yet the bulletin board does not contain a confirmed ballot $(1, b)$ such that $\mathsf{Extract}(b, pk) = \mathbf{s}$. This corresponds to voter 1 having successfully completed the voting process, yet his vote not appearing on the board as intended.

We define the advantage of an adversary against individual verifiability of scheme Π by:

$$\mathbf{Adv}^{\mathsf{iv}}_{A,\Pi} = \mathbf{Pr}[\mathbf{Exp}^{\mathsf{iv}}_{A,\Pi} = 1]$$

We make two remarks about the security definition. First, the definition of the game essentially assumes that the adversary controls all other voters and all remaining trusted authorities. This is a minimal assumption for CH-Vote. Second, we note that our choice to let the adversary interact with voter 1 as oppose to an arbitrary voter of his chosing is with a small loss of generality. We could have considered a game where the adversary determines the honest voter on the fly (after seeing the public information). Standard techniques would help relate security in this adaptive setting with security in the setting which we consider with a loss linear in the number of users.

4.2. Ballot Verifiability

Call a ballot valid if it passes the checks mandated by an election scheme (and assume that invalid ballots should not be included in the tally). Call a ballot correct if it contains a legal vote. Ballot verifiability is the property that valid ballots are also correct. It prevents a dishonest voter from casting a ballot that has any other effect than one allowed in the election, for example casting multiple votes or erasing another ballot.

Ballot Verifiability was not stated as a separate property in most of the earlier voting literature. The CH-Vote requirements for example subsume parts of Ballot Verifiability into Universal Verifiability (emphasis ours):

> "Universal Verifiability: The correctness of the election result can be tested by independent verifiers. The verification includes checks that only votes cast by eligible voters have been tallied, *that every eligible voter has voted at most once*, and that every vote cast by an eligible voter has been tallied as recorded."

While the case of a voter submitting two ballots is handled under EV, the BV property addresses the case where a voter tries to submit one ballot that contains a "double vote".

However, voting at most once is not sufficient as we would also like to prevent e.g. a voter from casting a ballot that has the effect of erasing another voter's ballot. A sufficient property is that no voter can cast a ballot that has any effect other than what is explicitly allowed by the election specifications.

We can write this as an implication in our model for single-pass voting:

$$\forall b: \ \mathsf{Check}(pk, b) = 1 \longrightarrow \mathsf{Extract}(b, pk) \neq \perp.$$

We formalise this implication as a security game in which the adversary wins if it can falsify the assumption. In the syntax of single-pass voting, the game is defined in Figure 2.

$\mathbf{Exp}^{\mathsf{bv}}_{\mathcal{A}}(\mathsf{params})$
$(pk, b) \leftarrow \mathcal{A}(\mathsf{params})$
If $\mathsf{Check}(pk, b) = false$ then return 0
If $\mathsf{Extract}(b, pk) = \perp$ then return 1 else return 0

Figure 2: Single-pass ballot verifiability (BV) game.

The above game simply assumes that the adversary is in complete control of the Setup phase. This form, is achievable for schemes such as Helios where ballots must carry a ZK-PoK of correctness, ballot verifiability holds even if everyone (except the bulletin board) is dishonest and so the setup procedure can be run entirely by the adversary, i.e. the adversary can essentially choose the public key by itself.

As we explained in Section 3.2, an extractor should also satisfy the property that any ballot generated by the scheme's voting algorithm, on input a correct vote $v \in V$, extracts to the vote v. In particular such ballots are correct.

4.3. Eligibility verifiability

Informally, eligibility means that following the voting process an independent verifier is convinced that every eligible voter has voted at most once and each vote recorded is as cast by an eligible voter. CH-Vote follows the existing literature in subsuming this into Universal Verifiability, although we prefer to separate Eligibility Verifiability out as a separate property (emphasis ours):

> "Universal Verifiability: The correctness of the election result can be tested by independent verifiers. The verification includes checks that *only votes cast by eligible voters have been tallied, that every eligible voter has voted at most once*, and that every vote cast by an eligible voter has been tallied as recorded."

We formalize this property as two distinct properties. The first one, which we call *eligibility uniqueness*, demands that each eligible voter casts at most one vote. For schemes where the bulletin board can be structured as a list of pairs (v, b) with v a voter's identity and b its corresponding ballot the formalization considers an adversary who controls all voters and all but one election authorities. The adversary interacts with the bulletin board and the election authority and aims that at the end of the interaction the bulletin board contains two confirmed votes for the same identity.

We define eligibility uniqueness via the experiment in the left-side of Figure 3. An adversary first participates in the setup algorithm of the scheme (in a way prescribed by the trust assumptions of the system). Next, the adversary takes over all of the voters and interacts with the bulletin board and one honest authority (wlog we let that authority be the first authority). At the end of the interaction the experiment returns 1 if event $\mathsf{bad}_{\mathsf{el\text{-}u}}$ occurs: the game sets this event true if at the end of the execution the list of confirmed ballots is $((v_1, b_1), (v_2, b_2), \ldots, (v_n, b_n))$ and for some $i, j \in [n]$ we have that $v_i = v_j$. Notice that our formulation uses the assumption that the bulletin board is well structured (Section 3.2).

The second property we want to capture is the idea that all votes that are on the board correspond to an eligible voter. We prove a stronger property namely that if a confirmed ballot for some honest voter is on the board then the vote it encodes corresponds to the intent of the voter.

The formal model for this property, which we call *confirmed as intended* is given in the right side of Figure 3. As before, the adversary participates in the setup of the scheme. Then it interacts with an honest user (for which it decides on some vote intention), one honest election authority and the bulletin board; the adversary takes over all other voters and election authorities.

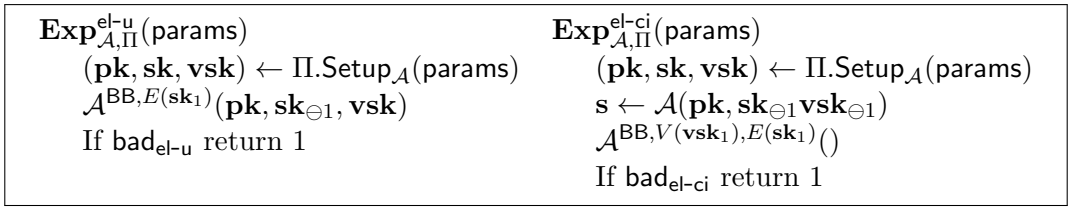

Figure 3: Games for defining eligibility uniqueness (left) and recorded as cast (right)

The game sets event $\mathbf{bad}_{\mathsf{el\text{-}ci}}$ (which indicates that the adversary has won) if the board contains a confirmed ballot b for voter 1, but the ballot does not encode the voter's intention, that is $\mathsf{Extract}(b, pk) \neq \mathbf{s}$.

We define the advantage of the adversary in breaking eligibility uniqueness by

$$\mathbf{Adv}^{\mathsf{el\text{-}u}}_{A,\Pi} = \mathbf{Pr}[\mathbf{Exp}^{\mathsf{el\text{-}u}}_{A,\Pi} = 1]$$

and confirmed as intended by

$$\mathbf{Adv}^{\mathsf{el\text{-}ci}}_{A,\Pi} = \mathbf{Pr}[\mathbf{Exp}^{\mathsf{el\text{-}ci}}_{A,\Pi} = 1].$$

4.4. Universal Verifiability

Universal Verifiability (UV) is the property that one can check whether an election was tallied correctly. It protects the voting process from dishonest authorities who wish to claim a different election result than the one that was voted for.

The term *Universal Verifiability* has been used differently at different times in the literature. Some early work uses the term UV to refer to what we would call end-to-end verifiability, that is roughly IV, EV, BV and UV together. Our definition follows the more recent literature in using the term UV more narrowly. Thus our UV covers the highlighted requirements from the CH-Vote specification:

> "Universal Verifiability: *The correctness of the election result can be tested by independent verifiers.* The verification includes checks that only votes cast by eligible voters have been tallied, that every eligible voter has voted at most once, and that *every vote cast* by an eligible voter *has been tallied as recorded.*"

We assume that the output of an election in a public-board scheme can be divided into three components: pre-election data BB_1 (such as public keys, lists of candidates and eligible voters), election data BB_2 (the cast ballots) and post-election data BB_3 (a claimed result and auxiliary data). We further assume that from the post-election data one can compute a claimed result r.

Our UV property says that you can check if BB_3 is correct under the assumption that BB_1 and BB_2 are correct. In other words, our formulation of UV does not guarantee that only eligible voters have voted (already captured by EV), that votes were recorded correctly (already captured by IV) or even that the ballots in BB_2 are correct (that is BV). What UV does say is that once you have seen (BB_1, BB_2) then the tallying authorities cannot convince you of

any BB'_3 value that would tally to a different result r' than the one mathematically implied by (BB_1, BB_2).

We call an election output (e.g. a complete board containing all three parts) *valid* if it passes the public checks mandated by the scheme and *correct* if it matches some possibly inefficient, but mathematically precise definition of "tallied correctly". Universal verifiability can then be defined as: valid election outputs are also correct.

To define the correct result for some election data and a list of ballots, we again make use of the extractor. Recall that we assume a result function $\rho : V^* \to R$ that takes a list of votes (not ballots) and returns the correct result $r \in R$. We make a slight change to express that the result of any election containing incorrect votes is also incorrect[7], by defining $\rho' : (V \cup \{\perp\})^* \to R$:

$$\rho'(e_1, \ldots, e_n) := \begin{cases} \rho(e_1, \ldots, e_n) & \text{if } (e_1, \ldots, e_n) \in V^* \\ \perp & \text{otherwise, e.g. } \exists i : e_i = \perp \end{cases}$$

This lets us formalise the correct tally of a list $L = (b_1, \ldots, b_n)$, of ballots:

$$\mathsf{CorrectTally}(b_1, \ldots, b_n) := \rho'(\mathsf{Extract}(b_1, pk), \ldots, \mathsf{Extract}(b_n, pk))$$

In words: the correct tally of a list L is the result of extracting a vote individually from each ballot, then applying ρ' to the result. If any ballot was incorrect (the extractor returned $\perp$) then we return $\perp$ as the correct tally since something has clearly gone wrong in this election.

We formalise the implication as follows.

$$\forall (pk, L, r, aux) : \ \mathsf{Verify}(pk, L, r, aux) = 1 \longrightarrow r = \mathsf{CorrectTally}(pk, L)$$

We can transform this into a simple game which the adversary wins if and only if they can falsify the implication, given in Figure 4. In its simplest form, the game allows the adversary to create all the election data.

$\underline{\mathbf{Exp}^{\mathsf{uv}}_{A,\Pi}}$
$(pk, L, r, aux) \leftarrow \mathcal{A}()$
If $\mathsf{Verify}(pk, L, r, aux) = 0$ then return 0
If $\mathsf{CorrectTally}(pk, L) = r$ then return 0 else return 1

Figure 4: Single-pass universal verifiability (UV) game.

CH-Vote comes with several complications, neither of which have been considered in much depth in the literature to our knowledge. The first is that not all voters may have the same voting rights, so the extractor will have to take the voter's identity into account. Secondly, ballot verifiability in CH-Vote is based on the assumption that at least one trustee is honest. Finally, due to the way the mixnet and counting circles are implemented, the election result is only defined up to a permutation. We will address all these points when we present the UV notion for CH-Vote.

[7] This does not mean incorrect in the sense of a voter spoiling their ballot. We will use this property to deal with ballots that violate ballot verifiability.

5. Verifiability of CH-Vote

In this section we define the verifiability properties that we will prove for CH-Vote. We start with a high-level view of the protocol and its execution. Then explain how we derive the security game which we use for our analysis, and for each security property we state the theorem which captures the guarantees offered by CH-Vote. As much as possible, we use the notation used in the specification document [12].

5.1. Parties and parameters

The parties in an election are

1. One administrator, who has the responsibility of publishing the election specification (number of candidates, eligible voters etc.) and the final election result. The administrator does not need to hold any secrets beyond perhaps a signing key to authenticate its messages.
2. A number s of election authorities (index: j), each holding a share of the election secret key. The authorities contribute parts of the information that goes on the voter's secret voting cards. Authorities are also available during the election to interact with the voters who need to cast their ballots and they shuffle and jointly decrypt the election results.
3. A printing authority, who prints and delivers the voting cards to the voters. The printing authority is assumed to be honest
4. A number N_E of voters (index: i). Each voter holds a vote, has access to a voting client and to a voting card.
5. A bulletin board. The board allows voters and authorities to communicate and collects a transcript of the election.

We divide the public parameters of a CH-Vote election into the following classes:

- *Public, non cryptographic parameters.* These are the inputs $(\mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E})$ to the setup phase containing the voter descriptions, counting circle descriptions, candidate descriptions, numbers of candidates and allowed votes per election and the voter eligibility matrix. We assume that all these parameters are correctly and honestly generated.
- *Public group and security parameters.* These are the values

  $$(p, q, g, h, k, \widehat{p}, \widehat{q}, \widehat{g}, \widehat{k}, p', \sigma, \tau, \varepsilon, \mathbf{p}).$$

  We assume that the security level $(s, \sigma, \tau, \varepsilon)$ and parameters of the appropriate level are chosen appropriately and correctly. We include the list of primes $\mathbf{p}$ here as it can be computed from the other parameters.
- We write params for the set of public parameters.
- *Election keys.* These are $(pk, \mathbf{pk}, \widehat{\mathbf{D}}, \widehat{\mathbf{x}}, \widehat{\mathbf{y}})$. Since these parameters are generated in a distributed fashion by the election authorities to assure secrecy of the related secret parameters, we do not assume they are honestly generated *a priori*. Rather, our later security analysis will assume that at least one authority is honest.

5.2. Phases of an election

We start with a high-level overview of the execution of the protocol (full details are in the specification document [12]).

Pre-election

In this phase, the election authorities together with the print server execute a distributed key-generation protocol: for user i, authority j generates a polynomial $\mathbf{p}_{ij}$ and two points x_{ij} and y_{ij}. It also selects n random points on the polynomial $\mathbf{p}_{ij}$. The secret voting key of user i is then set to $X_i = \sum_{j=1}^{s} x_{ij}$ and his secret confirmation key is set to $Y_i = \sum_{j=1}^{s} y_{ij}$. The corresponding public keys are g^{X_i} and $g^{Y_i + \sum_{j=1}^{s} \mathbf{p}_{ij}(0)}$, where g is a generator of $\mathbb{G}_{\hat{q}}$.

The verification codes of user i combine (hashes of) the points across the different polynomials held by the election authority (for each individual selection); the finalization code combines (hashes of) all polynomials corresponding to the voter.

During the election

An eligible voter starts out with a serial number $i \in 1, \ldots, N_E$, a vote $\mathbf{s}$ (= selection) and a voting card containing two secret strings X_i and Y_i as well as a list of confirmation codes, one for each possible selection.

The voter starts their voting client and enters their serial number i. The client displays the voting page and the voter makes their selection $\mathbf{s}$. The voting client then asks the voter to enter their first secret X_i.

The client posts some data on the bulletin board that functions both as an OT request and as an encryption of the voter's selection. The OT request aims to transfer points on the different polynomials that are held by the election authority to the user. This data is authenticated by a Schnorr-type signature using X_i which doubles as a proof of a correctly formed request.

Each authority, when they see an OT request that is correctly signed and proven with a voter key, replies by posting an OT response to the board. Each authority will do this at most once per eligible voter.

The client monitors the board and collects the OT responses. When it has all responses, it can compute the verification codes for the voter's selection, which it displays to the voter. The client then asks the voter for their second secret Y_i.

The voter compares the verification codes with the ones on her card and types Y_i into the client if the codes match.

The client uses Y_i (together with additional information about the polynomials held by the election authorities) to post a confirmation message to the board. All authorities, on seeing such a message, reply with a finalisation message. The client combines the finalisation messages to produce the finalisation code, which it displays to the voter. The voter compares this to the code on her card — if it matches, her vote has been cast successfully.

In a diagram, the voting process can be abstracted as follows:

Voter		Authority
	$\xrightarrow{\alpha}$	CheckBallot
CheckReturnCodes	$\xleftarrow{\beta}$	
	$\xrightarrow{\gamma}$	CheckConfirmation
CheckFinalisationCode	$\xleftarrow{\delta}$	

Post-election

After the voting period ends, the election authorities each perform a shuffling and a partial decryption operation. In the shuffling phase, each authority in turn takes the ballots on the board and applies a mixnet. The result is that if at least one authority is honest then the output of this phase is a list of ballots whose votes are a permutation of the votes in the original ballots, but no-one can link an output ballot to an input ballot.

In the decryption phase, the authorities each produce a partial decryption for each ballot in the output of the final shuffle. Anyone can now take such a ballot and all s of its partial decryptions and combine these values to recover the encrypted vote.

The verifiability of the post-election phase is covered by our UV property. Both the shuffling and decryption phases require the authorities to produce zero-knowledge proofs that they have operated correctly. UV ensures that, even if all authorities are dishonest, they cannot produce valid proofs unless they have shuffled and decrypted correctly.

5.3. Sequence diagrams

We present sequence diagrams for the pre-election, election and tallying phases of a CH-Vote election in Figures 5, 6, and 7, respectively. For simplicity we model an election with two authorities A1, A2 and one voter comprised of a voting device D1 and a human V1. We present our diagrams in a token-based execution model, although in reality the parties may execute concurrently. Since parties mostly communicate by reading and writing to the bulletin board, we give the board the additional role of "scheduler" in our diagrams to simplify the presentation. We write $\perp$ for flows in our diagram that do not model communication, but serve only to transfer the fictional "execution token".

In all diagrams, whenever we indicate a party receiving a message from the board/scheduler, we mean that said party takes a "turn" in the election process and may begin by reading the entire state of the board so far. However, we write out only the information on the board that is actually relevant for the next step.

In the pre-election phase shown in Figure 5, each authority takes two "turns": one to generate voter key data and one to generate its own key shares.

In the election phase in Figure 6, the voter begins (after fictionally receiving the token) by giving their identity i, a selection s and the first voter key x to the voting device, which creates the first ballot component α and sends it to the board. Each authority in turn responds with a value β_j. We call this interaction the vote casting phase.

In the vote confirmation phase of the election phase, the voting device gets the responses β

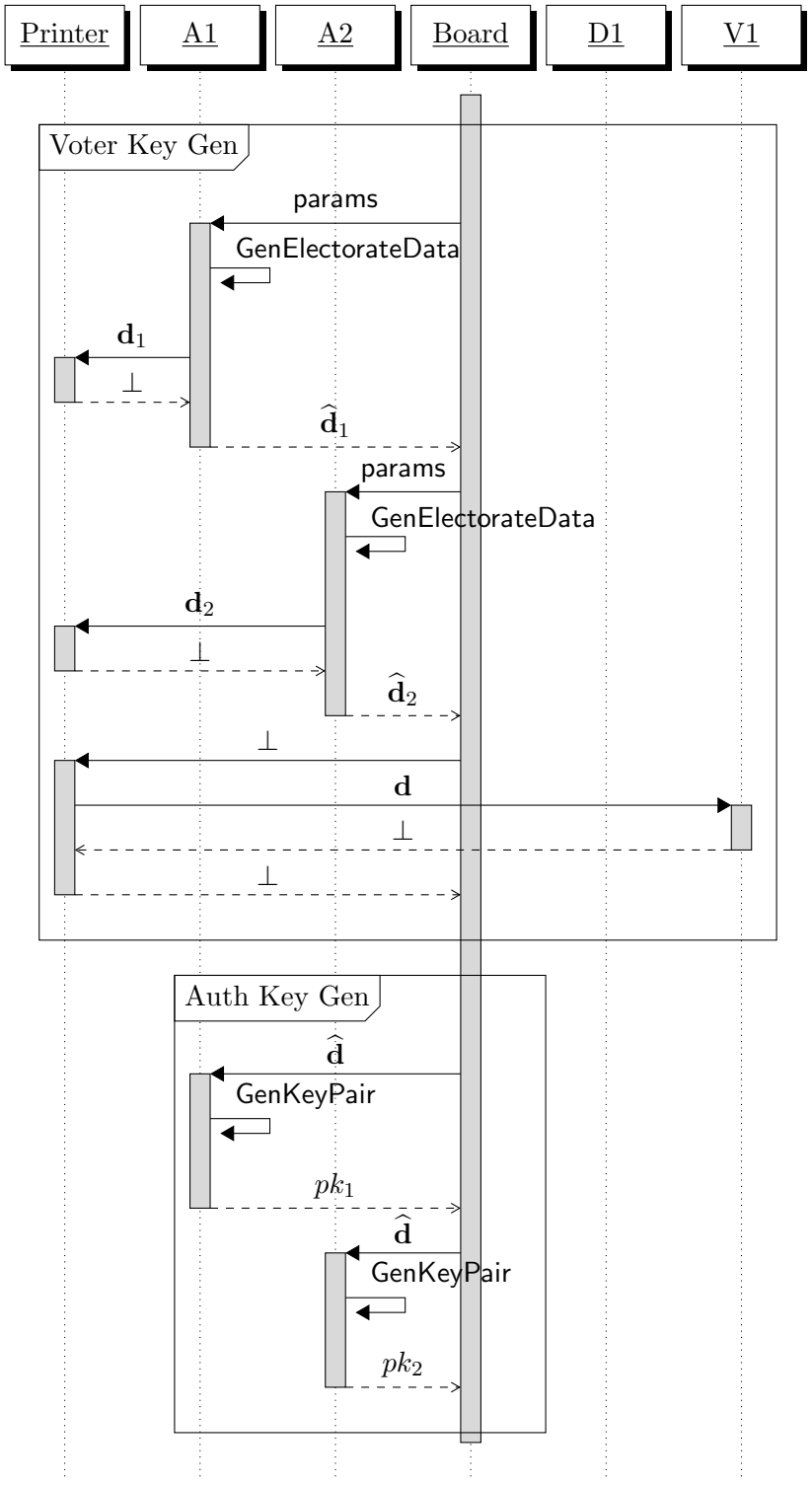

Figure 5: Sequence diagram for the pre-election phase.

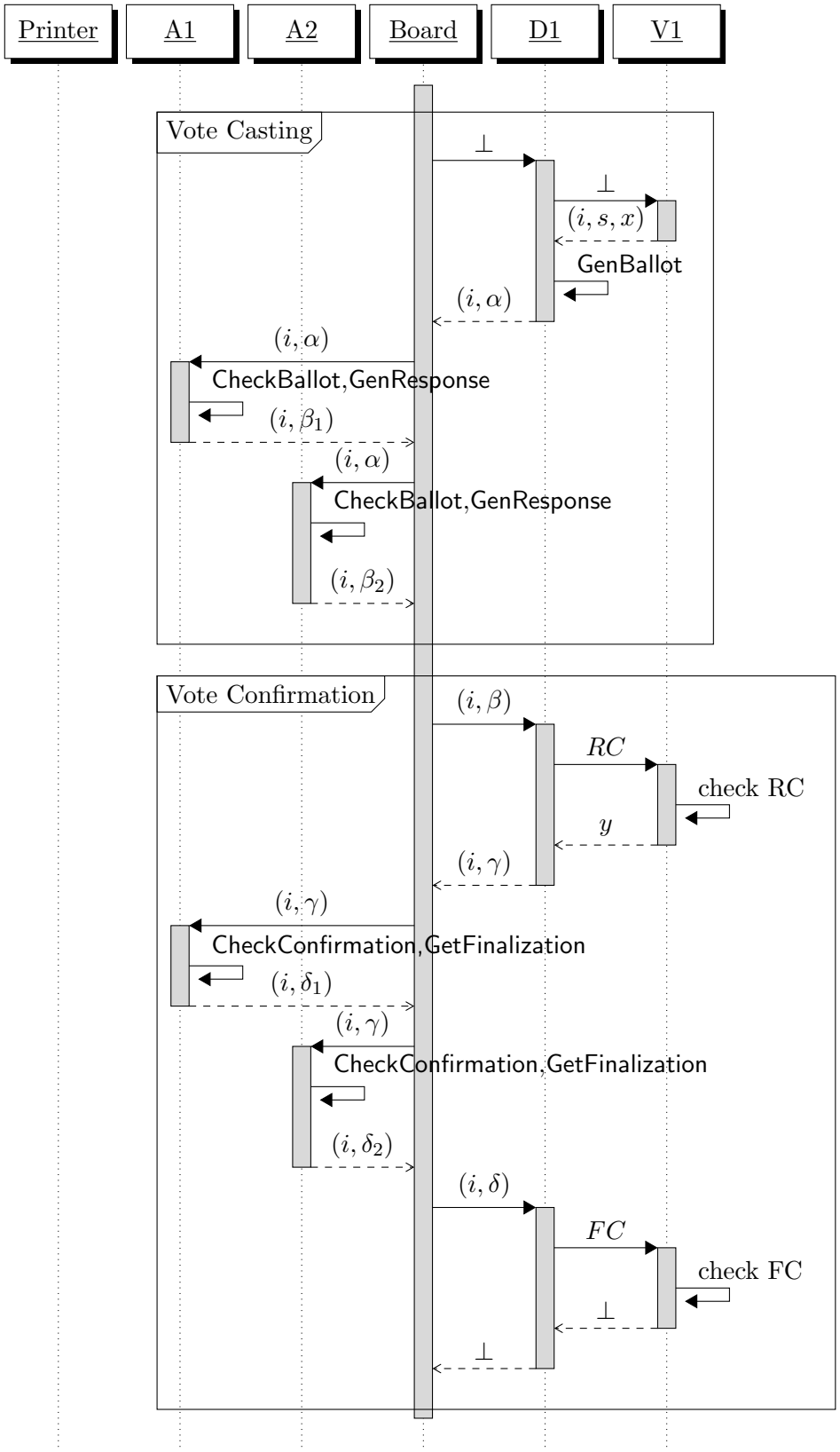

Figure 6: Sequence diagram for the election phase.

from which it computes the return codes RC and sends them to the voter for inspection. If she is happy, the voter inputs her second key y and the interaction with the board and authorities repeats to create values γ and δ on the board, from which the device computes the finalisation code FC and sends this to the voter.

In the tallying phase in Figure 7, there are two rounds: mixing and decryption. In each round, each authority takes a turn. (While mixing has to be sequential, the production of decryption shares could be done concurrently by the authorities.)

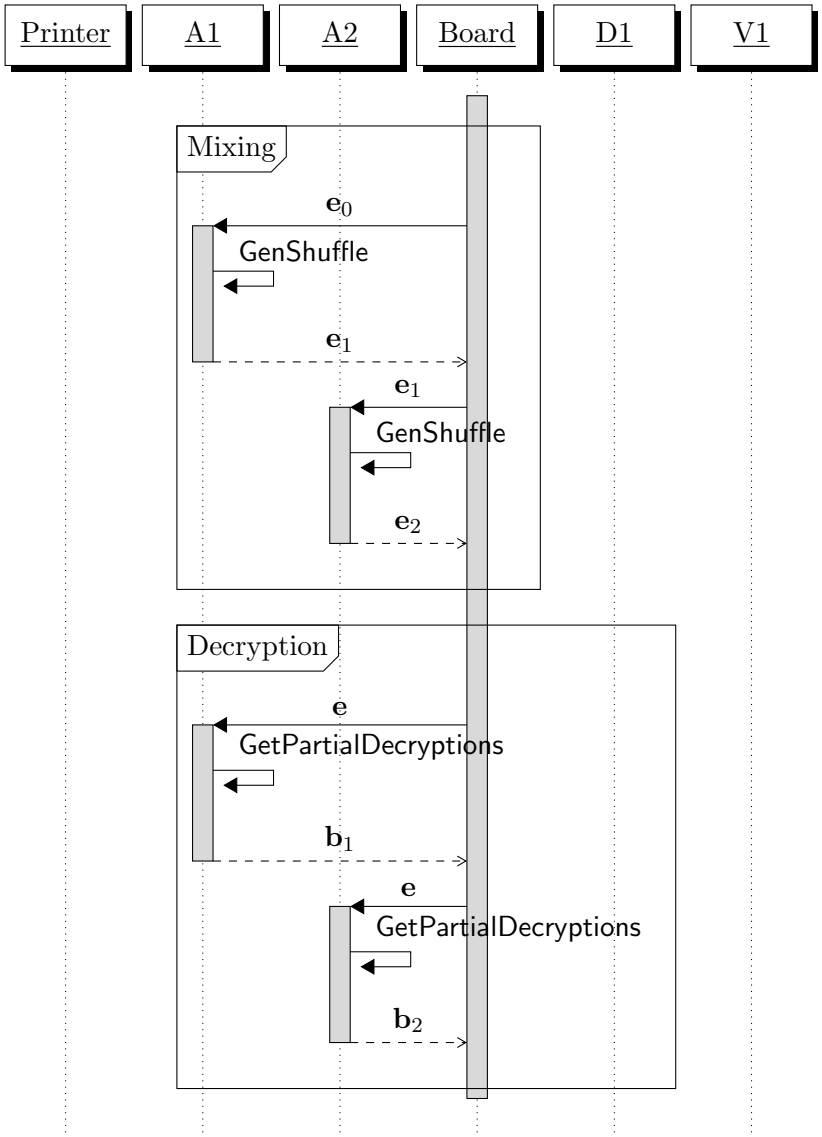

Figure 7: Sequence diagram for the tallying phase.

5.4. Informal security implications

To understand our modeling, it helps to have an informal understanding of the security implications of the protocol. We provide a brief discussion below.

In these descriptions, the term *valid* refers to an element that passes the associated checking function, e.g. a valid α is one that makes CheckBallot succeed. The term *correct* refers to an element that satisfies the intended properties, e.g. a correct ballot for a voter is one that encodes her intended vote. Roughly put, the purpose of verifiability is to ensure that "valid implies correct".

Guarantees for the voter

A voter's ballot on the board consists of two parts: an OT request and encryption, signed by her first secret X_v and a confirmation, signed by her second secret Y_v.

If the confirmation codes returned by the client match those on the voter's card, for the candidates that she voted for [e.g. CheckReturnCodes succeeds], then the voter can be assured that the first part of her ballot is a correct encryption of her vote and that all authorities have seen this first part. The voting client cannot have suppressed or modified her vote. The vote is cast as intended, but not recorded yet (it will not be tallied yet).

If the finalisation code matches that on a voter's card [CheckFinalisationCode succeeds], she can be assured that the second part of her ballot has been posted to the board and that all authorities have seen it there. Her client cannot have cast a malicious second part (or not sent the second part at all) in order to suppress her vote. Her vote has been recorded as cast.

If the voter does not release her Y_v without checking the confirmation codes, her client cannot impersonate her or cast a different vote to the one that she wanted.

Guarantees for the authorities

If the authorities see a valid first part α [CheckBallot succeeds] then the authorities know that α encodes a valid[8] vote (Ballot Verifiability) and that the voter must have been involved in the casting process. The authorities do not know that the ballot matches the voter's intention yet, however.

If the authorities see a valid second part γ [CheckConfirmation succeeds] then they know that the voter saw the correct return codes, hence her ballot is correct.

During tallying, the authorities should count only those ballots (α, γ) with valid components (in the sense mentioned above).

5.5. Formatting assumptions

As explained earlier, our games are meaningful for protocols which satisfy certain formatting assumptions (as stated in Section 3.2). Here we argue that CH-Vote satisfies those assumptions.

[8] A *valid vote* is simply an element of the vote space. A ballot either encodes a valid vote or not — the former case is called a *correct ballot*.

Explicit parsing CH-Vote ballots contain a voter identity i and components $(\alpha, \beta, \gamma, \delta)$. Each component contains the voter identity to which it corresponds. We assume that there is a way to tell (i) if a voter has voted or not and (ii) in case of more than one ballot for a voter, which of the ballot(s) to count[9]. With this assumption, the board can be mapped to a sequence of pairs containing an identity and a ballot, as required in our models.

Explicit confirmation In the interaction with the authorities the voter (or rather his voting device) signs twice, once with the voting key and once with the confirmation key, and both signatures can be publicly verified. A ballot is confirmed (i.e. intended to be tallied) if both of these signatures are valid and can be publicly verified.

Committing ballots and extractors The first component α contains the ElGamal-encrypted votes and, once ballot validity has been confirmed, the election can be tallied based on the α components alone. For a fixed public key, ballots are therefore committing and the selection can be (inefficiently) recovered. Our extractor is defined (below) as decrypting the ciphertexts in α and then reversing the encoding scheme (based on prime numbers) to obtain the vote.

5.6. Correctness and Extractors

Recall that *correctness* is the property one would like to have and *validity* is the property that one can easily check; *verifiability* means that validity implies correctness.

We define correctness of a vote relative to two vectors $\mathbf{k}', \mathbf{n}$. The intention here is to check a particular voter's vote relative to their "personal" vector $\mathbf{k}'$ which is defined for voter i as the vector with $\mathbf{k}'_j = e_{i,j} \cdot \mathbf{k}_j$. Recall that t is defined as the length of the vector $\mathbf{n}$ and $n = \sum_{i=1}^{t} n_i$.

Definition 5.1 (correct vote) *A correct vote for a voter with respect to vectors $\mathbf{k}', \mathbf{n}$ of length t is a vector $\mathbf{S} = (S_1, \ldots, S_t)$ of selections such that $S_j \in [n]$ for all $j = 1, \ldots, t$ and, if we partition the set $[n]$ into t subsets of size n_i each preserving the ordering, e.g.*

$$P_1 = \{1, \ldots, n_1\}, P_2 = \{n_1 + 1, \ldots, n_2\}, \ldots, P_t = \{n_{t-1} + 1, \ldots, n_t\}$$

then $\mathbf{S}$ contains at most k'_i elements of each P_i.

We define the following extraction algorithm. An extraction algorithm is a deterministic but inefficient algorithm that can be used in security games and proofs. Our algorithm takes an ElGamal ciphertext and a public key as input, recovers the secret key by taking a discrete logarithm and then decrypts the ciphertext.

$$\mathsf{E} : (\mathbb{G}_q)^2 \times \mathbb{G}_q \to \mathbb{Z}_q;\ ((a, b), h) \mapsto a/b^{\mathsf{DLOG}_g(h)}$$

We also allow ourselves to use the following variations on the extraction algorithm E:

[9] In CH-Vote, a voter has voted if there is an α and a γ component for this voter on the board and the proofs in both of these components verify. In case of multiple ballots for a voter, we take the first α component that contains a valid proof. We will discuss this matter in more detail when we give the detailed model for eligibility verifiability.

- $\mathsf{E}(\mathbf{a}, h)$ where $\mathbf{a}$ is a vector of ciphertexts runs E on each individual ciphertext and returns the vector of decryptions.
- $\mathsf{E}(\alpha, h)$ where α is the first part of a voter's ballot, runs E on the vector of contained ciphertexts and returns their decryptions.

Next, we define the following algorithm $\mathsf{Extract}$. It takes the following inputs: a vector $\mathbf{v}$ coming from decrypting or extracting from a ballot, vectors $\mathbf{k}', \mathbf{n}$ determining a voter's eligibility and a vector $\mathbf{q}$ of encodings, e.g. $q_i = \Gamma(i)$. $\mathsf{Extract}(\mathbf{v}, \mathbf{k}', \mathbf{n}, \mathbf{q})$ operates as follows: it sets $\mathbf{S}$ as the preimages of $\mathbf{v}$ under the encoding Γ provided in $\mathbf{q}$, e.g. $\Gamma(S_i) = v_i$ for all i. Then, if $\mathbf{S}$ is a correct vote w.r.t. $\mathbf{k}', \mathbf{n}$ in the sense of Definition 5.1, it returns $\mathbf{S}$, otherwise it returns $\perp$.

It follows that if $\mathsf{Extract}(\mathsf{E}(\alpha, pk), \mathbf{k}', \mathbf{n}, \mathbf{q}) \neq \perp$ then the ballot component α encodes a correct vote w.r.t. $\mathbf{k}', \mathbf{n}$. This corresponds to the notion of an extractor in the literature.

5.7. Individual verifiability

In this section we study the individual verifiability of the CH-Vote protocol. We specialize the individual verifiability game described in Section 4.1 for the different algorithms that define CH-Vote. First, we spell out the algorithms run by the parties involved in the experiment defining individual verifiability. Specifically, we define the setup algorithm, that of the honest voter (without loss of generality we assume this is the voter with identity 1), and that of the honest authority (without loss of generality we assume this is tally authority 1). The entire experiment takes as input parameters:

1. s the number of trusted parties
2. t the number of distinct elections
3. $\mathbf{n} = (n_1, n_2, \ldots, n_t)$ the vector that records the total number of candidates for each election
4. $\mathbf{k} = (k_1, k_2, \ldots, k_t)$ the number of selections for each election
5. $\mathbf{E} = (e_{ij})_{N_E \times t}$ the boolean matrix which indicates for each voter in which elections he is allowed to take part in.

The setup part of the experiment models the generation of voting cards. We model that all, but one election authority may behave maliciously; we make no assumption on how the data for voter 1 is generated by the remaining authorities. The details of the setup procedure are in Figure 9. It starts with the execution of the generation of voter data (for voter 1) by election authority 1; the adversary provides data for voter 1. The remaining of the setup corresponds to he printing authority which calculates data for voter 1 and to the distributed generation of the public key for the election, by the election authorities: our modeling assumes that the adversary generates his shares independently of that generated by the honest election authority.

> ***Note.*** *The* CH-Vote *1.3 protocol, in our reading, does not prevent the public key of a dishonest authority from depending on those of the other authorities. We have made a recommendation to mitigate this, and assume the recommendation has been implemented for the following proof.*

Deriving the game

Individual verifiability for CH-Vote considers the case where a voter, the printing authority, the bulletin board and at least one authority is honest. The voter's voting device and all other authorities may be dishonest.

For CH-Vote the voter is satisfied with the outcome if both of his checks (the check of the return codes and the check of the confirmation code) succeed. So, individual verifiability says that if both of these codes succeed for voter 1, then there must exist a ballot on the board attributed to voter 1 which encodes the intended selection of this voter.

Starting with the sequence diagrams in Section 5.3, we collapse all honest parties into one entity "Game" and all dishonest parties into one entity "Adversary". We assume that authority 1 is the honest one. Since we are not interested in the tallying phase for this property, we stop the interaction at the end of the voting phase. Further, we omit any communication via the honest board that goes from one dishonest party to another. For example, the dishonest voting device sends (i, α) to the board, which then sends it on to the dishonest authorities and gets back their (i, β_j) values. We omit these steps as the honest parties never need to use the dishonest β_j values and we assume that the adversary, if it wishes to simulate the dishonest voter and authorities separately, will handle communication between them. This gives us the sequence diagram in Figure 8.

We make the following modifications to the sequence diagram to derive the individual verifiability game in Section 5.7.

- We move the setup for the honest authorities into a separate algorithm Setup. In this algorithm, we interleave the voter and authority key generation and we force the dishonest authorities to send all their public key material and the honest voter's secret keys to the game before the game sends its own key material out. This models the requirement that the dishonest authorities' keys cannot depend on the keys of the honest authorities.
- We remove the voter identity i from the flows as we are w.l.o.g. attacking voter $i = 1$.
- We let the adversary specify the voting selection for voter 1.
- We make the voter and authority checks more explicit by introducing event variables acheck_1, acheck_2, ucheck_1 and ucheck_1. We model explicitly that if acheck_1 is set to false then the authority will not engage further with the protocol. Similarly, we model explicitly that if ucheck_1 is set to false (that is the return codes are not valid) the voter will not continue executing the protocol.
- We give the adversary the extra power to choose the honest voter's selection s.
- The adversary wins if both checks performed by the user (checking the return codes and the finalization code) succeed, yet the ballot (α, γ) "on the board" — defined as the values (α, γ) sent from the adversary to the game — is either not confirmed or, it is not a correct ballot for the selection $\mathbf{s}$ that the adversary sent to the game earlier.

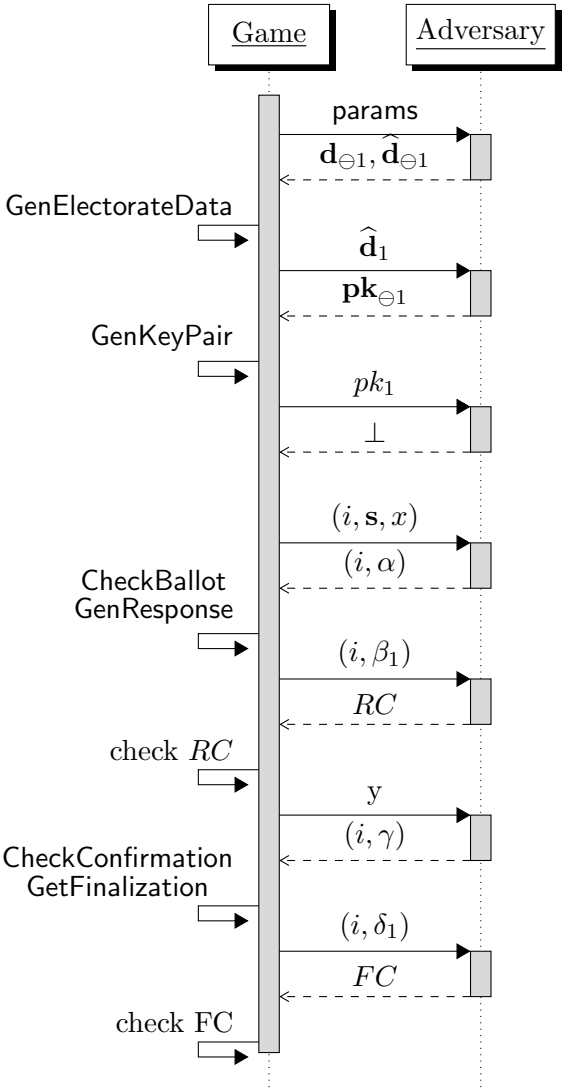

Figure 8: Sequence diagram for pre-election and election phases for one honest authority and a honest voter with a dishonest voting device. $\mathbf{d}_{\ominus 1}$ means all components of vector $\mathbf{d}$ except $\mathbf{d}_1$. The setup is arranged so that the honest authority goes last, modelling that the dishonest authorities' keys are not allowed to depend on the honest ones.

The security games for IV

The process which models voter 1 is in Figure 10. It takes as input the information on the voting card together with a selection $\mathbf{s}$ supplied by the adversary. First, the process outputs the voting code X_1. It then expects to receive a set of verification codes which correspond to the voter selection; the boolean variable ucheck_1 records if the verification codes match those on the voting card. If the check does not succeed, the process aborts. Otherwise, the voter provides the confirmation code Y_1. In return, it expects to receive a finalization code which should match the finalization code on the voting card. Boolean variable ucheck_2 records the result of this match, and therefore indicates that if the voter is satisfied with the result of the process.

In Figure 11 we give the process run by the election authority when interacting with voter 1. The interaction starts when the authority receives a first message $(1, \alpha)$ on behalf of user 1; it checks that $(1, \alpha)$ is valid for user 1 (using $\mathsf{CheckBallot}$): the result of the check is recorded in boolean variable acheck_1. If the check succeeds, the authority calculates and returns an answer β_1 (using the algorithm $\mathsf{GenReponse}$). It then expects to receive the confirmation message $(1, \gamma)$. It verifies that it is a valid confirmation message – the result of the check is stored in boolean variable acheck_2. If this is the case it replies with the finalization message δ_1.

To avoid reasoning about an execution which involves several oracles we inline the execution in a way that reflects the trust assumption in the bulletin board (namely that it forwards the messages between parties and that it does not erase messages). The resulting experiment, $\mathbf{Exp}^{\mathsf{iv}}_{\mathcal{A}}$, which we provide in Figure 12 inlines the execution of the oracles for honest voter 1 and election authority 1.

The inlining reflects the trust assumption on the bulletin box, namely that it is honest and forwards the messages between the voter's machine and the election authority. We note that in the description, the adversary $\mathcal{A}$ is invoked at different points in the execution: we make the assumption that the adversary passes its state from one invocation to the next, but we do not show this dependency in the figure.

The experiment maintains boolean variables $\mathsf{acheck}_1, \mathsf{acheck}_2, \mathsf{ucheck}_1, \mathsf{ucheck}_2$ with the same semantics as explained earlier. It first generates the voter data and public key for the election via the $\mathsf{Setup}_{\mathcal{A}}$ algorithm as explained earlier.

The adversary then provides a selection $\mathbf{s}$ for voter 1. The voter provides its signing key X_1 to the adversary who produces a ballot $\alpha = (\mathbf{a}, \pi)$ on behalf of user 1. The game extracts the underlying vote s^* (which will be used later to determine if the adversary has won or not). Next, we model the check that the election authority makes to see that the ballot provided by 1 is valid; if this is the case the experiment calculates and returns to the adversary β_1, the response of the authority. The adversary provides verification codes $(rc^*_1, \ldots, rc^*_{|\mathbf{s}|})$. The experiment checks that these match the codes corresponding to the vote which the user intended to cast. If this is not the case, the adversary looses the game (the experiment returns 0). Otherwise, the experiment provides the confirmation key Y_1 to the adversary who produces confirmation message γ on behalf of the voter.

The game checks that the confirmation message is valid. If both this check and the earlier check succeed the game returns to the adversary the finalization code produced by the election authority. Finally, the adversary provides a finalization code FC^* which the game checks against the one of the voter.

$\mathsf{Setup}_{\mathcal{A}}(s, t, \mathbf{n}, \mathbf{k}, \mathbf{E})$

// Election authority 1 generates credentials for voters

$n \leftarrow \sum_{j=1}^{t} n_j$

for $i = 1, \ldots, N_E$

$k'_i \leftarrow \sum_{j=1}^{t} e_{ij} k_j$

$(\mathbf{p}_{i1} = ((x_{i11}, y_{i11}), \ldots, (x_{i1n}, y_{i1n}), y'_{i1})), y'_i) \leftarrow \mathsf{GenPoints}(n, k'_i)$

$d_{i1} = (x_{i1}, y_{i1}, F_{i1}, \mathbf{r}_{i1}) \leftarrow \mathsf{GenSecretVoterData}(\mathbf{p}_{i1})$

$\hat{d}_{i1} = (\hat{x}_{i1}, \hat{y}_{i1}) \leftarrow \mathsf{GetPublicVoterData}(x_{i1}, y_{i1}, y'_{i1})$

$\mathbf{P}_1 \leftarrow (x_{i1j}, y_{i1j})_{1 \le i \le N_E, 1 \le j \le n}$

// The adversary impersonating all other authorities.

for $i = 1, \ldots, N_E$

for $j \leftarrow 2, \ldots, s$ **do**

$d_{ij} = (x_{ij}, y_{ij}, F_{ij}, \mathbf{r}_{ij}) \leftarrow \mathcal{A}()$

$(\hat{x}_{ij}, \hat{y}_{ij}) \leftarrow A()$

$\hat{x}_i \leftarrow \prod_{j=1}^{s} \hat{x}_{ij}$

$\hat{y}_i \leftarrow \prod_{j=1}^{s} \hat{y}_{ij}$

$\hat{\mathbf{x}} = (\hat{x}_1, \ldots, \hat{x}_{N_E})$

$\hat{\mathbf{y}} = (\hat{y}_1, \ldots, \hat{y}_{N_E})$

// Generation of voters cards by printing authority

for $i = 1, \ldots, N_E$ do

$X_i \leftarrow \sum_{j=1}^{s} x_{ij}$

$Y_i \leftarrow \sum_{j=1}^{s} y_{ij}$

$FC_i \leftarrow \oplus_{j=1}^{s} F_{ij}$

for $k \leftarrow 1, \ldots, n$ **do**

$RC_{ik} \leftarrow \oplus_{j=1}^{s} \mathbf{r}_{ijk}$

$\mathbf{rc}_i \leftarrow (RC_{i1}, RC_{i2}, \ldots, RC_{in})$

$VC_i = (X_i, Y_i, \mathbf{rc}_i, FC_i)$

// Generation of the election public key

$(sk_1, pk_1) \leftarrow \mathsf{GenKeyPair}()$

$(pk_2, \ldots, pk_s) \leftarrow \mathcal{A}(\hat{\mathbf{d}}_1, \mathbf{VC}_{\ominus 1})$

$pk \leftarrow \Pi_{i=1}^{s} pk_i$

Figure 9: The adversarial setup algorithm for the experiments defining individual verifiability and confirmed as intended property of the CH-Vote election scheme. For the entries in $\mathbf{rc}_i$, we have used RC_{ij} instead of $\mathbf{rc}_{ij}$ to keep notation close to the one in specification. We also wrote VC_i for the voter card of voter i; $\mathbf{VC}$ is the vector formed of all voter cards

$\mathsf{Voter}(X_1, Y_1, FC_1, \mathbf{rc}_1, \mathbf{s})$
out $\mathbf{s}, X_1$
in $\mathbf{rc}^*$
$\mathsf{ucheck}_1 \leftarrow \mathsf{CheckReturnCodes}(\mathbf{rc}_1, \mathbf{rc}^*, \mathbf{s})$
If $\mathsf{ucheck}_1 = false$ then abort
else out Y_1
in FC^*
$\mathsf{ucheck}_2 \leftarrow \mathsf{CheckFinalizationCode}(FC_1, FC^*)$
If $\mathsf{ucheck}_2 = false$ then abort

Figure 10: The protocol run by the voter 1

$\mathsf{EA}_1(\mathbf{P}_1, \hat{\mathbf{x}})$
input $(1, \alpha = (\hat{x}, \mathbf{a}, \pi))$
$\mathsf{acheck}_1 \leftarrow \mathsf{CheckBallot}(1, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}}, B_1)$
if $\mathsf{acheck}_1 = false$ then **abort**
else $(\beta_1, z) \leftarrow \mathsf{GenResponse}(1, \mathbf{a}, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P}_1)$; out $(1, \beta_1)$
input $(1, \gamma)$
$\mathsf{acheck}_2 \leftarrow \mathsf{CheckConfirmation}(v, \gamma, \hat{\mathbf{y}})$
if $\mathsf{acheck}_2 = false$ then **abort**
else $\delta_1 \leftarrow \mathsf{GetFinalization}(1, \mathbf{P}_1, B_1)$; out $(1, \delta_1)$

Figure 11: The protocol run by election authority EA_1

One important aspect of our model is that the individual lists B_1, C_1 maintained by election authority 1 are initialized with the empty list at the beginning of the exection. In particular, this means that their content does not change except through the calls considered in the game which is a departure from real executions where B_1 and C_1 change by interaction with other users as well. We note that, nonetheless, all of these interactions do not interfere with the interaction between the election authority and user 1: all of the additional entries in these lists are of the form (i, α), respectively (i, γ) for $i \neq 1$ – any such entry does not affect the interaction with user 1.

Before we explain how our game encodes the winning condition for the generic individual verifiability game, we make the following remark:

Remark 5.1 *Notice that the order in which the different checks are performed in the protocol implies that* $\mathsf{ucheck}_2 \Longrightarrow \mathsf{ucheck}_1$ *and* $\mathsf{acheck}_2 \Longrightarrow \mathsf{acheck}_1$ *(that is, if the latter checks succeed, it must be the case that the earlier checks have also succeeded). Indeed, neither the user nor the authority would engage in the second check if the first one fails. Furthermore, by the definition of confirmed ballots, the ballot* $(1, \alpha, \gamma)$ *submitted on behalf of user* 1 *by the adversary is confirmed if and only if* $\mathsf{ucheck}_1 = \mathsf{ucheck}_2 = true$.

Recall the the adversary wins the individual verifiability game if the process of the voter finishes successfully (for our model of CH-Vote this means that $\mathsf{ucheck}_2 = true$) and yet, the bulletin

$\mathbf{Exp}^{\mathsf{iv}}_{\mathcal{A}}(s,t,\mathbf{n},\mathbf{k},\mathbf{E})$
$\mathsf{acheck}_1 \leftarrow false, \mathsf{acheck}_2 \leftarrow false$
$\mathsf{ucheck}_1 \leftarrow false, \mathsf{ucheck}_2 \leftarrow false$
$B_1 \leftarrow []; C_1 \leftarrow []$
// Setup data for voter data
$\mathsf{Setup}_{\mathcal{A}}(s,t,\mathbf{n},\mathbf{k},\mathbf{E})$
$\mathbf{s} = (s_1, s_2, \ldots, s_{|\mathbf{s}|}) \leftarrow \mathcal{A}(pk, \hat{\mathbf{d}}_1, \mathbf{d}_{\ominus 1})$
// The adversary provides a ballot on behalf of voter 1
$\alpha = (\mathbf{a}, \pi) \leftarrow \mathcal{A}(X_1)$
$\mathbf{s}^* \leftarrow \mathsf{Extract}(\alpha, pk)$
// The election authority responds
$\mathsf{acheck}_1 \leftarrow \mathsf{CheckBallot}(1, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}}, B_1)$
if $(\mathsf{acheck}_1 = true)$ then $(\beta_1, z) \leftarrow \mathsf{GenResponse}(1, \mathbf{a}, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P}_1)$
// The adversary provides verification codes
$(rc^*_1, rc^*_2, \ldots, rc^*_{|\mathbf{s}|}) \leftarrow \mathcal{A}(\beta_1)$
// If codes match the adversary gets the confirmation key
// and provides a confirmation message
if $(\forall i \in [|\mathbf{s}|])\ rc^*_i = \mathbf{rc}_{1\mathbf{s}_i}$ then $\mathsf{ucheck}_1 \leftarrow true$; $\gamma \leftarrow \mathcal{A}(Y_1)$
else return 0
// If the confirmation message succeeds, the authority sends the finalization code
$\mathsf{acheck}_2 \leftarrow \mathsf{CheckConfirmation}(1, \gamma, \hat{\mathbf{y}}, B_1, C_1)$
if $(\mathsf{acheck}_1 = true)$ and $(\mathsf{acheck}_2 = true)$ then $\delta_1 \leftarrow \mathsf{GetFinalization}(1, \mathbf{P}_1, B_1)$
// The adversary provides a finalization code
$FC^* \leftarrow \mathcal{A}(\delta_1)$
// The user checks that the finalization code matches
if $\mathsf{ucheck}_1 = true$ then $\mathsf{ucheck}_2 \leftarrow (FC_1 = FC^*)$
if $(\mathsf{ucheck}_2 = true)$ and $((\mathsf{acheck}_2 = false)$ or $(\mathbf{s}^* \neq \mathbf{s}))$ return 1
else return 0

Figure 12: Experiment for defining individual verifiability of CH-Vote. Adversary $\mathcal{A}$ shares its state between the different invocations.

board does not contain a confirmed ballot which encodes the vote the voter intended to cast. For our model of CH-Vote this means that either $\mathsf{acheck}_1 = false$ or $\mathsf{acheck}_2 = false$ (the ballot of the voter is not confirmed) or that $\mathsf{Extract}(\alpha, pk) \neq \mathbf{s}$. By the previous remark, if $\mathsf{acheck}_1 = false$ then $\mathsf{acheck}_2 = false$, so the adversary wins (and the experiment returns 1) if $\mathsf{ucheck}_2 = true$ (the voter is happy) and either $\mathsf{acheck}_2 = false$ (his ballot is not confirmed) or $\mathsf{Extract}(\alpha, pk) \neq \mathbf{s}$ (the vote that is confirmed does not correspond to the intention of the voter).

For any adversary, we define its advantage against individual verifiability of CH-Vote by

$$\mathbf{Adv}^{\mathsf{iv}}_{\mathcal{A},\mathsf{CH\text{-}Vote}}(s,t,\mathbf{n},\mathbf{k},\mathbf{E}) = \mathbf{Pr}[\mathbf{Exp}^{\mathsf{iv}}_{\mathcal{A},\mathsf{CH\text{-}Vote}}(s,t,\mathbf{n},\mathbf{k},\mathbf{E}) = 1]$$

The following theorem which provides a bound on the advantage of any (even unbounded) adversary formally captures the individual verifiability property of CH-Vote. Its proof is in Section 7.1.

Theorem 5.2 *For any adversary $\mathcal{A}$ it holds that*

$$\mathbf{Adv}^{\mathsf{iv}}_{\mathcal{A}}(s,t,\mathbf{n},\mathbf{k},\mathbf{E}) \leq \binom{n}{|\mathbf{s}|} \cdot \max\left(\frac{1}{2^{8L_R}}, \frac{1}{p'}\right)^{|\mathbf{s}|} + |\mathbf{s}| \cdot \max\left(\frac{1}{2^{8L_R}}, \frac{1}{p'}\right) + \max\left(\left(\frac{1}{p'-|\mathbf{s}|}\right)^{n-|\mathbf{s}|}, \frac{1}{2^{8L_F}}\right)$$

where $|\mathbf{s}| = k'_1 = \sum_{i=1}^{t} e_{1,i} \cdot k_i$, L_R is the byte length of the return codes and L_F is the byte length of the finalisation codes.

The following corollary provides specific guarantees for the bounds on the parameters specified for CH-Vote.

Corollary 5.3 *If $||p'|| \geq 2\tau$, $8L_R \geq \log \frac{n-1}{1-\epsilon}$, $8L_F \geq \log \frac{1}{1-\epsilon}$, $|\mathbf{s}| \leq n-1$, $8L_R \leq 2\tau$, and $8L_F \leq 2\tau$ then the advantage of any adversary in breaking individual verifiability of* CH-Vote *is upperbounded by $4 \cdot (1-\epsilon)$.*

If additionally $k \geq 2$ and $\epsilon \geq 2$ then the advantage is upperbounded by $3 \cdot (1-\epsilon)$.

The proofs for the Theorem and Corollary are in Section 7.1.

5.8. Ballot verifiability

Ballot verifiability is the property that, on a board that verifies, all ballots contain correct votes and nothing else — the contribution of each individual ballot to the tally cannot be anything else than adding at most one vote. For example, ballot verifiability prohibits ballots that have the effect of cancelling out another ballot, or that contribute multiple votes.

Considerations for CH-Vote

In CH-Vote, three factors are important for ballot verifiability. First, the definition of a correct vote depends on the voter identity as the eligibility matrix $\mathbf{E}$ may assign different voters different voting rights. Secondly, unlike e.g. Helios or other homomorphic voting schemes, the validity of a ballot is not evident from the voter's contribution alone: the OT process with the voting authorities also plays a role. Finally, since CH-Vote uses a mixnet, even in the case that a voter were to be able to cast a ballot containing multiple votes (which we prove to be infeasible if at least one authority is honest) then this could be detected by inspecting the decrypted ballots at the end of the election.

Correct and Valid Ballots

A correct ballot is one that contains a correct vote. We additionally require the parameters $\widehat{x}, \widehat{y}$ in order to verify the ZK proofs in the ballot and a hash function H to compute the ZK proof challenges. Since the vote is contained solely in the α component, we define correctness for this component.

Definition 5.4 (correct ballot) *A ballot component α is correct w.r.t. a public key pk, eligibility vectors $\mathbf{k}'$ and $\mathbf{n}$ and a vector $\mathbf{q}$ of primes if $\mathsf{Extract}(\mathsf{E}(\alpha, pk), \mathbf{k}', \mathbf{n}, \mathbf{q}) \neq \perp$.*

A valid ballot is simply one that passes verification:

Definition 5.5 (valid ballot) *A ballot b is valid w.r.t. parameters pk, $\mathbf{k}'$, $\mathbf{n}$ and public keys $(\widehat{x}_i, \widehat{y}_i)$ if it parses as in point (1.) of the validity definition and the following two checks succeed:*

$$\mathsf{CheckBallotProof}(\pi, \widehat{x}, \mathbf{a}, pk) = 1 \text{ and } \mathsf{CheckConfirmationProof}(\pi', \widehat{y}) = 1.$$

Defining the security game

A general principle of security game design is that the game manages the parties that are required to be honest and the adversary can manage all other parties. In CH-Vote, for ballot verifiability we require the non-cryptographic election parameters (e.g. numbers of candidates, voters etc.) to be honestly generated and we require the bulletin board and at least one trustee to be honest.

Starting from the sequence diagrams in Figures 5 and 6 for the interaction up to the end of the voting phase, imagine merging the dishonest parties (printer, authority A1, voter V1 and device D1) into a single entity called the adversary and the board and the honest authority A2 into a single entity called the game. For the authorities, we let A1 stand for the authorities $A_{\ominus s} := A_1, \ldots, A_{s-1}$ and A2 for the honest auhority A_s. Further, although the dishonest authorities would post their β and δ values on the board where the honest authority could see them, we omit these flows as they are not required for this security property. This is legitimate as the values $\beta_{\ominus s}, \delta_{\ominus s}$ are sent from the dishonest authorities and meant for the dishonest voter, so the adversary can be assumed to handle these values by itself. Although we normally assume that the printing authority is honest, in the case of BV we are considering a single dishonest voter so we can allow the printing authority to be dishonest too — which only strengthens the security property. This leaves the interactions in Figure 13.

We modify the game in Figure 13 as follows to get the ballot verifiability game described in Section 5.8.

- The setup parameters are provided to the adversary at the start of the game, so we do not need a flow for these.
- We "inline" the adversary by passing explicit state back and forth and splitting the adversary into algorithms $\mathcal{A}_1, \mathcal{A}_2, \ldots$
- To model one honest authority, we let the adversary play all other authorities. In its first message, the adversary announces the index of the one authority that it wishes to remain honest.
- In CH-Vote, the notion of "correct ballot" depends on the voter identity as CH-Vote comes with an eligibility matrix $\mathbf{E}$ that may give different voters rights to vote in different elections. We model this by making the adversary declare which voter they are casting a ballot for.
- Certain algorithms that are run locally by the honest authority and hence omitted from the sequence diagram such as GetPublicCredentials and GetPublicKey are written out explicitly in the game.
- We abort the game if the CheckBallot or CheckConfirmation checks fail. In this case, the adversary loses.
- We stop the game the moment we get γ from the adversary since at this point, we have all the information we need to decide if the adversary has won or not. The game invokes the

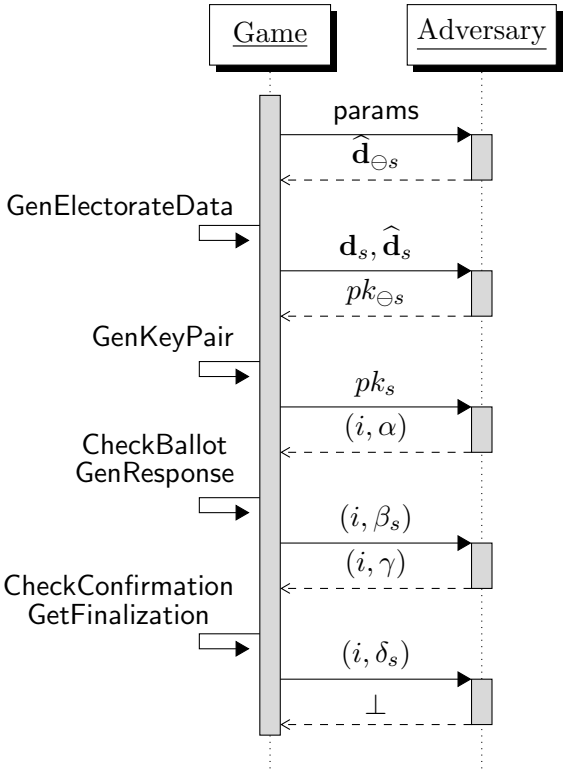

Figure 13: Sequence diagram for pre-election and election phases with one honest authority and a dishonest voter. Compared to the diagram for an honest voter, in this diagram the adversary does not send the game the voter secrets $\mathbf{d}_{\ominus s}$ but the game must now send the voter secrets $\mathbf{d}_s$ to the adversary.

extractor (with the eligibility information for the particular voter that we are attacking) and uses the result to decide the winning condition, namely that the adversary has *not* created a correct ballot for this voter but the earlier checks still passed.

From these principles, we derive the following ballot verifiability game.

The security game for BV

We model ballot verifiability as the game in Figure 14. We want to show that if at least one of the authorities is honest then a voter cannot create a valid but incorrect ballot.

The adversary operates in several stages. In stage $\mathcal{A}_1$ it chooses an index i_0 for a particular voter and the voting card data from all other authorities; we write $\widehat{\mathbf{D}}_{\ominus 1}$ to mean all entries except the first of the vector $\widehat{\mathbf{D}}$ (since the first component will be provided by an honest authority). This models the phase of the election setup where all authorities generate their share of the voting card data. The ordering (honest authority goes last) models that the adversarial authority data cannot depend on the honest data. Since the voter is assumed to be dishonest in this experiment, we let the adversary see all the voting card data.

$\mathbf{Exp}^{\mathsf{bv}}_{A}(s, t, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{q})$
// setup
$(i_0, \widehat{\mathbf{D}}_{\ominus 1}) \leftarrow \mathcal{A}_1()$
$(\mathbf{D}_1, \widehat{\mathbf{D}}_1, \mathbf{P}, \mathbf{K}) \leftarrow \mathsf{GenElectorateData}(\mathbf{n}, \mathbf{k}, \mathbf{E})$
$(\widehat{\mathbf{x}}^*, \widehat{\mathbf{y}}^*) \leftarrow \mathsf{GetPublicCredentials}(\widehat{\mathbf{D}})$
$(\mathbf{pk}_{\ominus 1}) \leftarrow \mathcal{A}_2(\mathbf{D}_1, \widehat{\mathbf{D}}_1)$
$(\mathbf{sk}_1, \mathbf{pk}_1) \leftarrow \mathsf{GenKeyPair}()$
$pk \leftarrow \mathsf{GetPublicKey}(\mathbf{pk})$
// Adversary chooses the first part of the ballot
$(i, \alpha) \leftarrow \mathcal{A}_3(\mathbf{pk}_1)$
if $i \neq i_0$ then return 0
Parse α as $(\widehat{x}, \mathbf{a}, \pi)$
$\mathsf{acheck}_1 \leftarrow \mathsf{CheckBallot}(i, \alpha, pk, \mathbf{K}, \mathbf{E}, \widehat{\mathbf{x}}^*, \langle\,\rangle)$
If $\mathsf{acheck}_1 = 0$ then return 0
$(\beta, z) \leftarrow \mathsf{GenResponse}(i, \mathbf{a}, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P})$
// Adversary gets the response and produces the second part
$(i', \gamma) \leftarrow \mathcal{A}_4(\beta)$
Parse γ as $(\widehat{y}, \pi')$
$\mathsf{acheck}_2 \leftarrow \mathsf{CheckConfirmation}(i', \gamma, \widehat{\mathbf{y}}^*, \langle (i, \alpha, z) \rangle, \langle\,\rangle)$
If $\mathsf{acheck}_2 = 0$ then return 0
// Finalisation
$\mathbf{v} \leftarrow \mathsf{E}(\alpha, pk)$
$vote \leftarrow \mathsf{Extract}(\mathbf{v}, \mathbf{K}_{i_0}, \mathbf{n}, \mathbf{q})$
If $vote = \perp$ then return 1 else return 0

Figure 14: The ballot verifiability (BV) game for CH-Vote.

Note. *The* CH-Vote *1.3 protocol, in our reading, does not prevent the public key of a dishonest authority from depending on those of the other authorities. We have made a recommendation to mitigate this, and assume the recommendation has been implemented for the following proof.*

In stage $\mathcal{A}_2$ we let the adversary choose public keys for all authorities but one; again the honest authority chooses last to prevent a dishonest authority from choosing their key depending on the honest authority's one. This models the phase of the setup where the authorities create their own keys and publish their public keys, from which the election public key pk can be derived.

In stage $\mathcal{A}_3$ the adversary presents us with a ballot for the voter i_0 that it chose earlier. We check the ballot and return the next component β if the checks pass, in response to which $\mathcal{A}_4$ must give us the next component γ. In this game we do not need to return δ as by the time we have γ, the adversary either has made us a correct ballot or it has not. This stage models the election phase in which a dishonest voter interacts with a honest authority. In a real election this interaction would take place via the bulletin board; in our security game the game and adversary just exchange the necessary messages directly.

Finally, we extract the vote from α, using the voter's specific choice vector $\mathbf{K}_{i_0}$ to check if the ballot is correct for this particular voter. The adversary wins if the ballot components (α, γ) are valid but not correct.

The following theorem, which we prove in Section 7.2, establishes a bound on the advantage of an arbitrary adversary $\mathcal{A}$ against ballot verifiability of CH-Vote.

Theorem 5.6 *For any adversary $\mathcal{A}$ which makes ν random oracle queries, there exists an adversary $\mathcal{E}$ against the DLOG assumption in $\mathbb{G}_{\hat{q}}$ such that:*

$$\mathbf{Adv}^{\mathsf{bv}}_{\mathcal{A}} \leq \sqrt{\nu \cdot \mathbf{Adv}^{\mathit{dlog}}_{\mathcal{E},\mathbf{G}_{\hat{q}}}} + \frac{\nu}{2^{\tau}} + \nu\frac{(n+1)}{\hat{q}} + \frac{n}{\hat{q}-n+1}$$

where τ is the length of the challenge in the zero knowledge proof in the confirmation message.

5.9. Eligibility verifiability

In this section we study eligibility verifiability of the CH-Vote protocol.

5.9.1. Eligibility: uniqueness

Eligibility uniqueness is the property that there are no two distinct entries in the list of confirmed ballots which correspond to the same voter. In CH-Vote this property is enforced without using cryptography: each election authority $j \in [s]$ records locally the ballots it received on behalf of each voter. Specifically, list B_j (held locally by some honest election authority j) maintains pairs (v, α) and list C_j maintains pairs (v, γ) such that for any v there exist unique entries $(v, \alpha) \in B_j$ (per the description of CheckBallot and GenResponse) and $(v, \gamma) \in C_j$ (per the description of CheckConfirmation and GetFinalization).

Under the assumption that at least one of the election authorities is honest and that the list of confirmed ballots on the bulletin board is consistent with these records, it follows that for any voter identity v, the list of confirmed ballots contain a unique entry (v, α, γ).

> ***Note.*** *NB: this consistency check and the way the bulletin board deals with multiple ballots for the same voter identity are not in the current specification of the protocol.*

5.9.2. Confirmed as intended

This is the property that no-one can cast ballots on behalf of a voter, except the voter herself: if a confirmed ballot is on the bulletin board for some voter, then that ballot records that voter's voting intention.

The setup algorithm is the same as the individual verifiability game: the adversary impersonates all but one of the election authorities whereas one election authority (w.l.o.g. election authority 1) runs the algorithms as prescribed by the protocol. The details are in Figure 9. Similarly, the processes that define the behaviour of voter 1 and of election authority 1 are as prescribed by the protocol (Figures 10,11).

As for individual verifiability, we inline the execution of the different oracles to get a manageable specification of the execution. Since we work under the same trust assumptions as in individual verifiability, the execution is essentially the same with a single difference: since the adversary against eligibility does not have to ensure that the process run by the voter finishes successfully, the game does not abort if the user checks fails. Instead, if the first user check (if the return codes were correct) fails then the adversary does not receive the user confirmation key and has to produce the confirmation message γ on its own.

To define the winning condition we make use of Remark 5.1. Since $\mathsf{acheck}_2 \implies \mathsf{acheck}_1$ the ballot cast on behalf of user 1 is confirmed if and only if the second check of the authority holds, i.e. $\mathsf{acheck}_2 = true$.

In this case, the winning condition for the adversary is that this second check is true, yet the vote recorded by the ballot cast on behalf of the voter is different from the one he intended to cast. That is, the adversary wins if $\mathsf{acheck}_2 = true$ and $\mathsf{E}(\alpha, pk) \neq \mathbf{s}$ (where $\mathbf{s}$ is the intended vote).

The resulting game is in Figure 15.

$\mathbf{Exp}^{\mathsf{el\text{-}ci}}_{\mathcal{A}}(s, t, \mathbf{n}, \mathbf{k}, \mathbf{E})$
- $\mathsf{acheck}_1 \leftarrow false, \mathsf{acheck}_2 \leftarrow false$
- $\mathsf{ucheck}_1 \leftarrow false, \mathsf{ucheck}_2 \leftarrow false$
- $\mathsf{Setup}_{\mathcal{A}}(s, t, \mathbf{n}, \mathbf{k}, \mathbf{E})$
- $\mathbf{s} = (s_1, s_2, \ldots, s_{|\mathbf{s}|}) \leftarrow \mathcal{A}(\hat{\mathbf{d}}_1)$
- If $\neg\mathsf{CorrectVote}(1, \mathbf{s}, \mathbf{k}, \mathbf{n}, \mathbf{E})$ then abort
- $\alpha = (\mathbf{a}, \pi) \leftarrow \mathcal{A}(X_1)$
- $\mathbf{s}^* \leftarrow \mathsf{Extract}(\alpha, pk)$
- $\mathsf{acheck}_1 \leftarrow \mathsf{CheckBallot}(1, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}}, B_1)$
- if $(\mathsf{acheck}_1 = true)$ then $(\beta_1, z) \leftarrow \mathsf{GenResponse}(1, \mathbf{a}, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P}_1)$
- $(rc^*_1, rc^*_2, \ldots, rc^*_{|\mathbf{s}|}) \leftarrow \mathcal{A}(\beta_1)$
- if $(\forall i \in [|\mathbf{s}|])\ rc^*_i = \mathbf{rc}_{1\mathbf{s}_i}$ then $\mathsf{ucheck}_1 \leftarrow true$; $\gamma \leftarrow \mathcal{A}(Y_1)$
  - else $\gamma \leftarrow \mathcal{A}()$
- $\mathsf{acheck}_2 \leftarrow \mathsf{CheckConfirmation}(1, \gamma, \hat{\mathbf{y}}, B_1, C_1)$
- if $(\mathsf{acheck}_1 = true)$ and $(\mathsf{acheck}_2 = true)$ then $\delta_1 \leftarrow \mathsf{GetFinalization}(1, \mathbf{P}_1, B_1)$
- $FC^* \leftarrow \mathcal{A}(\delta_1)$
- if $\mathsf{ucheck}_1 = true$ then $\mathsf{ucheck}_2 \leftarrow (FC_1 = FC^*)$
- if $(\mathsf{acheck}_2 = true)$ and $(\mathbf{s}^* \neq \mathbf{s})$ return 1
  - else return 0

Figure 15: Experiment for defining confirmed as intended property of the CH-Vote; the highlighted line is the only difference from the game for individual verifiability

We define the advantage of adversary $\mathcal{A}$ as

$$\mathbf{Adv}^{\mathsf{el\text{-}ci}}_{\mathcal{A}}(s, t, \mathbf{n}, \mathbf{k}, \mathbf{E}) = \mathbf{Pr}[\mathbf{Exp}^{\mathsf{el\text{-}ci}}_{\mathcal{A}}(s, t, \mathbf{n}, \mathbf{k}, \mathbf{E}) = 1]$$

The confirmed as intended property of CH-Vote is formally captured by the following theorem. Its proof is in Section 7.3.

Theorem 5.7 *For any adversary $\mathcal{A}$ against confirmed as intended of* CH-Vote *there exists an adversary $\mathcal{B}$ against the discrete logarithm problem in $\mathbf{G}_{\hat{q}}$ such that*

$$\mathbf{Adv}_{\mathcal{A}}^{\mathsf{el\text{-}ci}} \leq |\mathbf{s}| \cdot \max\left(\frac{1}{2^{8L_R}}, \frac{1}{p'}\right) + \sqrt{\nu \cdot \mathbf{Adv}_{\mathcal{E},\mathbf{G}_{\hat{q}}}^{dlog}} + \frac{\nu}{\hat{q}}$$

5.10. Universal verifiability

In our formulation, Universal Verifiability (UV) is the property that given a bulletin board, one can verify that the votes on this board were tallied correctly. For suitable definitions of correctly tallied and valid board (where validity is efficiently checkable) UV should guarantee the implication

$$\textit{valid board} \Longrightarrow \textit{correctly tallied}$$

We follow the literature in considering a scenario in which all parties in the election may be dishonest. Our UV game simply consists of the adversary giving us a bulletin board containing data (BB_1, BB_2, BB_3) from the pre-election, election and post-election phases respectively. We enforce that the post-election data BB_3, in particular the claimed election result, is consistent with the (pre-)election data (BB_1, BB_2), e.g. the public key and the ballots.

Other properties were counted under the term UV in some previous papers. For example, our UV does not check whether the ballots came from eligible voters (that is EV) or contain correct votes (that is BV). What our UV does ensure is that for example, if the election data in (BB_1, BB_2) contains a majority of votes for candidate A then we would not accept an election result in BB_3 that declares another candidate B to be the winner.

We consider this terminology appropriate as we will show that CH-Vote, under suitable assumptions, has all four of our properties {IV, BV, EV, UV}. All these properties together we may call *end-to-end verifiability*.

The bulletin board format

We assume that the election parameters $(s, \mathbf{w}, \mathbf{n}, \mathbf{k}, \mathbf{E})$ as well as the group sizes and security levels are honestly generated and known to all participants.

For the purposes of UV, we assume that a bulletin board BB at the end of an election can be parsed into three components (BB_1, BB_2, BB_3) containing the data from the pre-election, election and post-election phases respectively.

- The pre-election data BB_1 contains the public key share matrix $\widehat{\mathbf{D}}$ of dimension $s \times N_E$ with a row from each authority and a column for each voter and the vector of authority public key shares $\mathbf{pk}$ of length s.
- The election data BB_2 contains a list of items each containing a voter identity i, a data component and an optional signature component.
- The post-election data BB_3 contains the following lists, each of length s: a list $\mathbf{e}'$ of shuffles, a list π of shuffle proofs, a list $\mathbf{b}'$ of partial decryptions and a list π' of decryption proofs. In addition, the post-election data contains[10] a claimed election result $(\mathbf{V}, \mathbf{W})$.

We define $(\widehat{x}_{ij}, \widehat{y}_{ij})$ to be the components of the element $\widehat{\mathbf{D}}_{ij}$ and we set, for $i = 1, \ldots, N_E$, the values $\widehat{x}_i := \prod_{j=1}^{s} \widehat{x}_{ij}$ and $\widehat{y}_i := \prod_{j=1}^{s} \widehat{y}_{ij}$ to represent the two public keys for voter i. Further we let $\widehat{\mathbf{x}}$ and $\widehat{\mathbf{y}}$ be the lists of voter public keys, that is $\widehat{x}_i$ is the first and $\widehat{y}_i$ the second public key for voter i.

We assume that the component BB_2 representing the election phase data can be parsed as a list of items which are either a pair (i, d) or a triple (i, d, s) where i is a voter identity and d is the item's data (e.g. the α, β, γ or δ value) and s is a signature, which is provided for the authority-supplied items (β, δ). We assume that each ballot component can be uniquely classified as one of four types $(\alpha, \beta, \gamma, \delta)$.

We further assume that for each component of type β or type δ, the authority that created this component can be identified. Although the authority identity is not written directly into the ballot, unlike the voter identity, the authority could be identified by looking at the signature s in their component resp. which authority public key verifies this signature. Let $\mathsf{AUTHORITY}(e)$ be an algorithm that takes an item of type β or type δ and returns either an index $j \in [s]$ identifying the authority, or $\perp$ if this item is invalid. CH-Vote 1.3 leaves open exactly how this is to be implemented, therefore so do we.

The result format

The theory of cryptographic elections considers a result function $\rho : V^* \to R$ that takes a list of votes $v_i \in V$ and outputs a result $r \in R$. The result function must be deterministic and efficiently computable.

In CH-Vote, the election returns a pair of matrices $(\mathbf{V}, \mathbf{W}) \in \{0,1\}^{N \times n} \times \{0,1\}^{N \times w}$ representing the candidates' vote counts and the counting circles. However, due to the way the mixnet is used, tallying the same election twice will produce different matrices as the random permutation of the votes will be different. To define the result function formally and to check whether two tallies match, we define $\mathsf{NORMALISE}(\mathbf{V}, \mathbf{W})$ to be the algorithm that combines $(\mathbf{V}, \mathbf{W})$ into a single $\{0,1\}^{N \times (n+w)}$ matrix $\mathbf{Z}$ and then sorts the rows of $\mathbf{Z}$ lexicographically. This lets us define the result set R as the set of equivalence classes under normalisation. Further, we define $\mathsf{MATCH}((\mathbf{V}, \mathbf{W}), (\mathbf{V}', \mathbf{W}'))$ to return the result of the comparison $\mathsf{NORMALISE}(\mathbf{V}, \mathbf{W}) == \mathsf{NORMALISE}(\mathbf{V}', \mathbf{W}')$. Universal verifiability will ensure that the result on the board matches the correct result for the ballots on the board in the sense of the MATCH algorithm.

> ***Note.*** *Since the permutation of the ballots in the mixnet is independent of the votes (as long as at least one authority is honest), normalisation as described above does not lose any information, e.g. seeing* $(\mathbf{V}, \mathbf{W})$ *does not give you any more information than seeing the normalised result* $\mathbf{Z}$*.*
>
> *We originally hoped to be able to define the normalised result as the sum of votes for each candidate, grouped by counting circle. However, this process does lose information: from the matrices* $(\mathbf{V}, \mathbf{W})$ *one could additionally derive statistics such as that voters who chose candidate A over B in election one were more likely to choose C over D in election two.*
>
> *It is not within the scope of this document to comment on whether this behaviour is de-*

[10] The claimed result can be computed deterministically from the other post-election data using GetVotes, so it does not need to be included on the board itself.

sired or not — it is a privacy issue rather than a verifiability one — but we have made a recommendation to check this issue and a related one where even more information can be revealed if voters within a counting circle have different eligibility rights.

Which ballots should be tallied?

To define that a board was tallied correctly, we first need to say which ballots should be counted. A ballot for a voter consists of a first entry (i, α) of type α, a reply (i, β_j, s_j) of type β from each authority, a second entry (i, γ) of type γ and a reply (i, δ_j, s'_j) of type δ from each authority. According to our reading of CH-Vote, a ballot should be tallied if and only if the voter-supplied components are valid and it is the first such ballot for this voter. In more detail, we define validity for the voter-supplied data.

Definition 5.8 (valid component for tallying) *Let an election public key pk and a matrix $\widehat{\mathbf{D}}$, which defines two lists $(\widehat{x}_i, \widehat{y}_i)_{i=1}^{N_E}$, be given as election parameters.*

- *A component $(i', \alpha = (\widehat{x}', \mathbf{a}, \pi))$ of type α is valid for tallying w.r.t. the given parameters if $\widehat{x}' = \widehat{x}_{i'}$ and $\mathsf{CheckBallotProof}(\pi, \widehat{x}', \mathbf{a}, pk) = 1$.*
- *A component $(i', \gamma = (\widehat{y}', \pi'))$ of type γ is valid for tallying w.r.t. the given parameters if $\widehat{y}' = \widehat{y}_{i'}$ and $\mathsf{CheckConfirmationProof}(\pi', \widehat{y}') = 1$.*
- *We say that a voter i has cast a valid ballot if there is an entry (i, α) and an entry (i, γ) on the board, both of which are valid for tallying. In this case the ballot for this voter is the first entry (i, α) on the board that is valid for tallying.*

Note. *This definition assumes that the algorithm CheckBallotProof has been patched to take all of $\mathbf{a}$ and not just the product e as input, to mitigate malleability issues that can arise otherwise.*

Valid for tallying is a weaker condition than being a correct ballot, since if all authorities are dishonest then it does not ensure that the ballot contains a correct vote.

We define the following algorithm PublicGetBallots to get the ballots that should be tallied.

PublicGetBallots(BB)
$A \leftarrow \{\}$; $S \leftarrow \{\}$; $C \leftarrow \{\}$; $L \leftarrow \langle\rangle$
Parse BB as (BB_1, BB_2, BB_3)
For each entry e of BB_2
  If $e \sim (i, \alpha = (\widehat{x}', \mathbf{a}, \pi))$ then
    If $\mathsf{CheckBallot}(i, \alpha, pk, \mathbf{k}, \mathbf{E}, \widehat{\mathbf{x}}, BB_{\text{prev}}) = true$ then
      If $i \notin A$ then $(A, S) \leftarrow (A \cup \{i\}, S \cup \{(i, \mathbf{a})\})$
  Else if $e \sim (i, \gamma = (\widehat{y}', \pi))$ then
    If $\mathsf{CheckConfirmationProof}(\pi, \widehat{y}') = true$ then
      If $i \in A \wedge i \notin C$ then $(C, L) \leftarrow (C \cup i, \langle L, (S(i), i)\rangle)$
Return L

We use the notation $\boxed{\text{if } e \sim (i, \alpha = (\widehat{x}, \mathbf{a}, \pi))}$ (read $\sim$ as "matches") to mean that the condition is true if e is of type α, in which case we bind the variables $i, \widehat{x}, \mathbf{a}, \pi$ to the relevant components of e for the block of code associated with this condition.

We construct a mapping S of voter identities i to ciphertexts $\mathbf{a}$ by constructing a set of pairs $(i, \mathbf{a})$ which can be read when the corresponding γ component is found. By $S(i)$ we mean the component $\mathbf{a}$ associated with the identity i in set S; we only use this notation where it is well-defined. Each entry in the list L of ballots to be tallied is a pair $(\mathbf{a}, i)$ where $\mathbf{a}$ are the ciphertexts for voter i and i is a voter identity.

The set A contains all voters for whom we have found a component of type α that is valid for tallying and the set C contains all voters for whom we have also found a component of type γ that is valid for tallying.

In the loop over all entries of BB, we use BB_{prev} to mean all previously seen entries, e.g. for the j-th entry of BB we have $BB_{\text{prev}} = \langle BB_1, \ldots, BB_{j-1} \rangle$. This lets us reuse[11] CheckBallot to perform the ballot checks as well as to check whether this is the first α component for this voter that is valid for tallying.

If PublicGetBallots identifies N valid ballots on a board, then it will return a vector L of length N, each component of which is a pair containing a vector of ciphertexts and a voter identity.

Checking the post-election data

We define the algorithms in Figure 16 to check the post-election data. They are based on the algorithms in CH-Vote run by the authorities themselves except that the algorithms here check all authorities' proofs.

PublicCheckShuffleProofs checks the mixnet proofs. It is similar to CheckShuffleProofs except that it does not skip any of the proofs. The parameter domains are $\pi \in \left(\mathbb{G}_q^{5+N} \times \mathbb{Z}_q^{4+2N} \times \mathbb{G}_q^{2N}\right)^s$, $e_0 \in (\mathbb{G}_q^2)^{N_E}$, $e \in (\mathbb{G}_q^2)^{N_E \times s}$ and $pk \in \mathbb{G}_q$.

PublicCheckResult checks a claimed result. The parameter domains are $\mathbf{e} \in (\mathbb{G}_q^2)^{N_E}$ for the final list of shuffled ciphertexts; $\mathbf{b} \in (\mathbb{G}_q)^{N_E \times s}$ for the partial decryptions; $\mathbf{V} \in \{0,1\}^{N_E \times N}$ for the election result and $\mathbf{W} \in \{0,1\}^{N_E \times w}$ for the counting circle results.

PublicCheck is the algorithm that checks a bulletin board. We call a board valid w.r.t. some parameters if PublicCheck accepts the board under these parameters; the UV property will enforce that valid boards are correctly tallied.

Definition 5.9 (valid board) *We call a board* BB *valid w.r.t. parameters* $(s, \mathbf{w}, \mathbf{n}, \mathbf{k}, \mathbf{E})$ *if* *PublicCheck*$(\mathsf{BB}, s, \mathbf{w}, \mathbf{n}, \mathbf{k}, \mathbf{E})$ *returns* 1.

Correct Tally

We define the correct tally for a board via the extractor.

[11] Although CheckBallot formally takes triples as input where the third component is a value z_i created by the authority, the code of CheckBallot does not make use of the z_i so we assume that it will still run on our pairs (i, α). One could also convert each pair (i, α) to a triple $(i, \alpha, \perp)$.

$\underline{\mathsf{PublicCheck}(\mathsf{BB}, s, \mathbf{w}, \mathbf{n})}$
Parse BB as $BB_1 = (\widehat{\mathbf{D}}, \mathbf{pk}), BB_2,$
$BB_3 = (\mathbf{e}', \pi, \mathbf{b}', \pi', \mathbf{V}, \mathbf{W})$
$pk \leftarrow \mathsf{GetPublicKey}(\mathbf{pk})$
// Get the ballots and do the public steps
$L \leftarrow \mathsf{PublicGetBallots}(\mathsf{BB})$
for $i = 1, \ldots, |L|$
  $(\mathbf{e}'_0)_i \leftarrow \mathsf{AddCircles}(L_i, \mathbf{n}, \mathbf{w})$
$\mathbf{e}_0 \leftarrow \mathsf{Sort}(\mathbf{e}'_0)$ // $\mathbf{e} : N \times 2$
// Check the proofs and the result
$\mathsf{c}_S \leftarrow \mathsf{PublicCheckShuffleProofs}(\pi, \mathbf{e}_0, \mathbf{e}', pk)$
$\mathsf{c}_D \leftarrow \mathsf{CheckDecryptionProofs}(\pi', \mathbf{pk}, \mathbf{e}'_s, \mathbf{b}')$
$\mathsf{c}_R \leftarrow \mathsf{PublicCheckResult}(\mathbf{e}'_s, \mathbf{b}', \mathbf{V}, \mathbf{W})$
return $(\mathsf{c}_S \wedge \mathsf{c}_D \wedge \mathsf{c}_R)$

$\underline{\mathsf{PublicCheckShuffleProofs}(\pi, e_0, \mathbf{e}, pk)}$
$c \leftarrow \mathsf{CheckShuffleProof}(\pi_1, e_0, \mathbf{e}_1, pk)$
If $c = 0$ then return 0
for $j = 2, \ldots, s$
  $c \leftarrow \mathsf{CheckShuffleProof}(\pi_j, \mathbf{e}_{j-1}, \mathbf{e}_j, pk)$
  If $c = 0$ then return 0
return 1

$\underline{\mathsf{PublicCheckResult}(\mathbf{e}, \mathbf{b}, \mathbf{V}, \mathbf{W})}$
for $i = 1, \ldots, |\mathbf{e}|$
  $b^*_i \leftarrow \prod_{j=1}^{s} \mathbf{b}_{i,j}$
  $m_i \leftarrow (\mathbf{e}_i)_1 / b^*_i$
$(\mathbf{V}', \mathbf{W}') \leftarrow \mathsf{GetVotes}(\mathbf{m}, \mathbf{n}, \mathbf{w})$
return $(\mathbf{V} = \mathbf{V}' \wedge \mathbf{W} = \mathbf{W}')$

Figure 16: Algorithms for public checking of post-election data.

Definition 5.10 (correct tally) *The correct tally for a board BB w.r.t. public parameters $(\mathbf{n}, \mathbf{w})$ is given by the algorithm CorrectTally.*

$\underline{\mathsf{CorrectTally}(\mathsf{BB}, \mathbf{n}, \mathbf{w})}$
parse $pk, \widehat{\mathbf{D}}$ from BB
$L \leftarrow \mathsf{PublicGetBallots}(\mathsf{BB})$; $M \leftarrow \langle\rangle$
for $e = (\mathbf{a}, i)$ in L do
  $\mathbf{m} \leftarrow \mathsf{E}(\mathbf{a}, pk)$
  $m \leftarrow p_{n+w_i} \cdot \prod_{j=1}^{|\mathbf{m}|} \mathbf{m}_j$
  $M \leftarrow \langle M, m \rangle$
return $\mathsf{GetVotes}(M, \mathbf{n}, \mathbf{w})$

First, the CorrectTally algorithm extracts the ballots that should be tallied using PublicGetBallots. This step is efficiently computable without any secret data. Next, the algorithm extracts from each ballot individually, applies the counting circles and adds the ballot to a list. The list is finally processed with the GetVotes algorithm.

The algorithm GetVotes is part of the CH-Vote specification and computes a matrix of votes per counting circle. To enable this, the ballot of a voter i is marked as belonging to the counting circle w_i by multiplying its vote by prime number p_{n+w_i} before the ballots are mixed, which loses any further identifying information.

We deliberately only apply the "inner" extractor E here and not the full Extract algorithm. Checking that the ballots on the board are correct is covered under the BV property; what UV is asserting is that the authorities have correctly tallied the ballots on the board, whether they are correct or not. This would of course mean that a scheme satisfying UV but not BV is vulnerable to dishonest voters, but we will show that CH-Vote has both the UV and BV properties which together guarantee that only correct votes are tallied, since invalid ones are stripped by PublicGetBallots.

The security game for UV

Universal verifiability says at a high level that valid tallies are also correct, that is a tally that passes the public and efficiently computable validity checks would also pass the inefficient correctness check.

Since we assume that all parties may be dishonest, the sequence diagram for this property is almost trivial: the adversary hands us the entire election transcript and wins if the election passes public verification but the claimed result is not the one that one would obtain by using the extractor. Since tallying is a randomised process due to the mixnet, correctness of the result is only defined up to permutation of the final matrices, so we apply the MATCH algorithm to account for this complication.

The experiment for defining universal verifiability is in Figure 17; the advantage of some adversary $\mathcal{A}$ against universal verifiability of CH-Vote is defined by:

$$\mathbf{Adv}^{\mathsf{uv}}_{\mathcal{A}}(s, \mathbf{w}, \mathbf{n}) = \mathbf{Pr}[\mathbf{Exp}^{\mathsf{uv}}_{\mathcal{A}}(s, \mathbf{w}, \mathbf{n}) = 1]$$

$\underline{\mathbf{Exp}^{\mathsf{uv}}_{\mathcal{A}}(s, \mathbf{w}, \mathbf{n})}$
$\mathsf{BB} \leftarrow \mathcal{A}()$
Parse BB as $(BB_1 = (\widehat{\mathbf{D}}, \mathbf{pk}), BB_2, BB_3 = (\mathbf{e}', \pi, \mathbf{b}', \pi', \mathbf{V}, \mathbf{W}))$
$pk \leftarrow \mathsf{GetPublicKey}(\mathbf{pk})$
If $\mathsf{PublicCheck}(\mathsf{BB}, s, \mathbf{w}, \mathbf{n}) = 0$ then return 0
$(\mathbf{V}', \mathbf{W}') \leftarrow \mathsf{CorrectTally}(BB_2, pk, \widehat{\mathbf{D}}, \mathbf{n}, \mathbf{w})$
Return $\mathsf{MATCH}((\mathbf{V}, \mathbf{W}), (\mathbf{V}', \mathbf{W}'))$

Figure 17: Game for defining universal verifiability of CH-Vote

The following theorem establishes universal verifiability of CH-Vote. Its proof is in Section 7.4.

Theorem 5.11 *For any adversary $\mathcal{A}$ we have*

$$\mathbf{Adv}^{\mathsf{uv}}_{\mathcal{A}} \leq \frac{2s \cdot \nu}{2^{\tau}}$$

where s is the number of authorities, ν is the total number of random oracle queries made by $\mathcal{A}$ and τ is the entropy (bit-length) of the challenge space for the ZK proofs of correct mixing.

5.11. Summary of the results

We summarise the results of our cryptographic proofs.

For all security properties, we make the following general assumptions.

- The public, non-cryptographic parameters

  $$(\mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E})$$

  are correctly generated and publicly available to all parties.

- The public group and security parameters, excluding election-specific keys, are correctly generated at an appropriate security level and publicly available to all parties. These parameters are
$$(p, q, g, h, k, \widehat{p}, \widehat{q}, \widehat{g}, \widehat{k}, p', s, \sigma, \tau, \varepsilon)$$

Individual Verifiability

Under the assumptions that

- At least one of the authorities is honest.
- The dishonest authorities cannot choose their keys and voter key shares as a function of the keys and key shares of the honest authorities

it is computationally infeasible for any dishonest party or coalition of parties to convince an honest voter that he's intended vote has been properly recorded without this being the case.

More specifically, no adversary can win our IV game with a greater advantage than

$$\binom{n}{|\mathbf{s}|} \cdot \max\left(\frac{1}{2^{8L_R}}, \frac{1}{p'}\right)^{|\mathbf{s}|} + |\mathbf{s}| \cdot \max\left(\frac{1}{2^{8L_R}}, \frac{1}{p'}\right) + \max\left(\left(\frac{1}{p' - |\mathbf{s}|}\right)^{n-|\mathbf{s}|}, \frac{1}{2^{8L_F}}\right)$$

where n is the number of candidates, $|\mathbf{s}|$ is the number of selections the voter in question can make and L_R, L_F and p' are system parameters.

Ballot Verifiability

Under the assumptions that

- At least one of the authorities is honest.
- The dishonest authorities cannot choose their keys and voter key shares as a function of the keys and key shares of the honest authorities.
- It is infeasible to take discrete logarithms in the group $\mathbb{G}_{\widehat{q}}$.

it is computationally infeasible for any dishonest party or coalition of parties to generate a ballot that is valid, e.g. passes verification and so would be tallied, but does not contain a correct vote for the voter that it is claimed to be from.

More specifically, no adversary can win our BV game with a higher advantage than

$$\sqrt{\nu \cdot \mathbf{Adv}^{\mathsf{dlog}}_{\mathcal{E}, \mathbf{G}_{\hat{q}}}} + \frac{\nu}{2^{\tau}} + \nu\frac{(n+1)}{\hat{q}} + \frac{n}{\hat{q} - n + 1}$$

where ν is the number of random oracle (a.k.a. "hash") queries made by the adversary and n is the total number of candidates in the election; τ and $\widehat{q}$ are system parameters. This quantity is negligible if taking discrete logarithms is hard and the adversary is efficient (thus bounding the number hash queries it can make).

Universal Verifiability

It is computationally infeasible, even if all parties are dishonest, to generate an election transcript that passes our proposed PublicCheck verifier but on which the claimed election result does not match the ballots on the board.

Put another way, given a bulletin board at the end of the voting phase, it is computationally infeasible even if all authorities are dishonest and colluding for them to claim any result but the one defined by the ballots on the board, in such a way that PublicCheck would accept when run on the given board and the authorities' claimed results and proofs.

In the case where all authorities are dishonest, UV does not guarantee that each ballot on the board contains a correct vote — that property is covered under BV, which in CH-Vote holds only if at least one authority is honest. Thus, assuming at least one honest authority, we can take BV and UV together to conclude that each ballot to be tallied contains exactly one correct vote and that the election result is exactly the tally of these ballots.

More specifically, no adversary can win our UV game with a better advantage than

$$\frac{2s \cdot \nu}{2^\tau}$$

where ν is the number of hash queries made by the adversary, s is the number of election authorities and τ is a system parameter. For an efficient adversary, this quantity is negligible.

Eligibility Verifiability

Under the assumption that

- At least one of the authorities is honest
- The bulletin board is consistent with the honest authority's view

it is impossible for any coalition of parties to cast two valid ballots for the same eligible voter.

Under the assumption that

- At least one of the authorities is honest
- The discrete logarithm problem is hard in $\mathbb{G}_{\widehat{q}}$

it is computationally infeasibible for any coalition of parties to produce a valid ballot of some honest eligible user, which does not record that user's voting intention.

More precisely, no adversary can win our EL-CI game with a better advantage than

$$|\mathbf{s}| \cdot \max\left(\frac{1}{2^{8L_R}}, \frac{1}{p'}\right) + \sqrt{\nu \cdot \mathbf{Adv}^{\mathsf{dlog}}_{\mathcal{E},\mathbf{G}_{\hat{q}}}} + \frac{\nu}{\hat{q}}$$

where $|\mathbf{s}|$ is the number of selections the voter can make, ν is the number of hash queries the adversary can make and $p', \widehat{q}$ and L_R are system parameters. For an efficient adversary and assuming that the discrete logarithm problem is hard, this quantity is negligible.

6. Sender Security of Robust Batch Chu-Tzeng Oblivious Transfer

A core component of CH-Vote is a variant of the Chu-Tzeng oblivious transfer protocol. In this section we study the security of this protocol. We split the analysis in three parts.

First, we study the security of the protocol without considering the distribution of the messages $(m_1, m_2, \ldots, m_n)$ held by the sender. We prove information theoretic security for all messages which are not queried by the adversary. Furthermore, we show that each query unambiguously encodes a set of requests $i_1, i_2, \ldots, i_k$ and that an adversary can obtain m_i if and only if i is one of these indexes.

Then, we consider the setting where these messages are distributed as in the CH-Vote protocol (that is they are all points on a random polynomial of degree $k-1$). Here, we study the security of the sender's messages under two different scenarios. The first considers an adversary who is not allowed to request k valid indexes. Here we show that all of the remaining points on the polynomial, including the value of the polynomial at 0 are hidden. The second scenario considers an adversary who can request k valid indexes, and therefore recover the polynomial. Here we show that all remaining points are information theoretically hidden.

6.1. Robust Chu-Tzeng OT

CH-Vote uses the OT protocol on selections of a particular form relating to two vectors $\mathbf{k}, \mathbf{n}$ which we define first.

Definition 6.1 (correct selection σ) *Let $\mathbf{k}, \mathbf{n}$ be vectors of the same length, which we denote by κ, such that for all $i \in [\kappa]$ we have $k_i \leq n_i$. We define $k := \sum_{i=1}^{\kappa} k_i$ and $n := \sum_{i=1}^{\kappa} n_i$.*

A correct selection w.r.t. such vectors $\mathbf{k}, \mathbf{n}$ is a vector σ of length k with elements in $[n]$ such that the elements of σ are pairwise distinct and for each $i \in [k]$, we have that at most k_i elements of σ lie in the interval from $1 + \sum_{j=1}^{i-1} n_j$ to n_i, boundaries included.

Let q be a prime, let $\mathbb{G}_q$ be a group of order q and let an injective mapping $\Gamma : \{1, \ldots, n\} \to \mathbb{G}_q$ and two elements $g, h \in \mathbb{G}_q$ be given. Let $\mathbf{k}, \mathbf{n}$ be vectors as described above. The robust batch Chu-Tzeng $OT^{\mathbf{k}}_{\mathbf{n}}$ protocol is the following protocol.

$\underline{\mathsf{Query}(\sigma)}$
for $i = 1, \ldots, k$
  $t_i \twoheadleftarrow \mathbb{Z}_q$
  $q_{i,1} \leftarrow \Gamma(\sigma_i) \cdot h^{t_i}$
  $q_{i,2} \leftarrow g^{t_i}$
return $(\mathbf{q}, \mathbf{t})$

$\underline{\mathsf{Open}(\sigma, \mathbf{q}, \mathbf{t}, r)}$
parse r as $(w, \mathbf{C}, \mathbf{D})$
for $i = 1, \ldots, k$
  $K_{\sigma_i,i} \leftarrow D_i / w^{t_i}$
  $M'_{\sigma_i} \leftarrow C_{\sigma_i,i} \oplus H(K_{\sigma_i,i})$
return $\mathbf{M}'$

$\underline{\mathsf{Respond}(\mathbf{q}, \mathbf{M})}$
$z, x \twoheadleftarrow \mathbb{Z}_q$
$w \leftarrow h^z g^x$
for $i = 1, \ldots, k$
  $b_i \twoheadleftarrow \mathbb{G}_q$
  $D_i \leftarrow q_{i,1}^z q_{i,2}^z b_i$
$\mu_0 \leftarrow 1$; $\nu_0 \leftarrow 1$
for $\beta = 1, \ldots, \ell$
  for $\mu = \mu_0, \ldots, \mu_0 + n_\beta$
    for $\nu = \nu_0, \ldots, \nu_0 + k_\beta$
      $K_{\mu,\nu} \leftarrow \Gamma(\mu)^z b_\nu$
      $C_{\mu,\nu} \leftarrow M_\mu \oplus H(K_{\mu,\nu})$
  $\mu_0 \leftarrow \mu_0 + n_\beta$; $\nu_0 \leftarrow \nu_0 + k_\beta$
return $(w, \mathbf{C}, \mathbf{D})$

The Respond algorithm creates matrices K and C of dimension $n \times k$ which consist of blocks (batches) on the diagonal such that the i-th block has size $n_i \times k_i$. In the nested loop of the Respond algorithm, the index β is over the batches and μ, ν are the current row and column of the matrices K and C that are being created. μ_0, ν_0 are the indices of the first row and column belonging to the current batch.

The robust non-batch protocol OT_n^k is the variation in which k, n are integers with $k \leq n$ and we set $k_1 = k, n_1 = n$. In this case, the loop over β is redundant.

In an execution of the protocol, one party called the sender has a vector $\mathbf{M}$ of n messages that lie in some domain $\mathcal{M} = \{0,1\}^\lambda$, which we also assume to be the range of the hash function H. The other party called the receiver has a vector σ that is a correct selection w.r.t. $\mathbf{k}, \mathbf{n}$. The receiver begins the protocol by computing $(\mathbf{q}, \mathbf{t}) \leftarrow \mathsf{Query}(\sigma)$ and sends $\mathbf{q}$ to the sender. The sender computes $(y, \mathbf{C}, \mathbf{D}) \leftarrow \mathsf{Respond}(\mathbf{q}, \mathbf{M})$ and sends this triple back to the receiver. The receiver computes $\mathbf{M}' \leftarrow \mathsf{Open}(\sigma, \mathbf{q}, \mathbf{t}, (y, \mathbf{C}, \mathbf{D}))$. The result is that the receiver has $\mathbf{M}_i$ for all $i \in \sigma$ and the sender gains no information on which messages the receiver has obtained.

Theorem 6.2 *The robust batch OT protocol is information-theoretically secure for the sender in the random oracle model. Specifically,*

1. *No receiver can recover more than k of the keys $K_{\mu,\nu}$, or more than k of the messages M_μ with a better probability than a random guess. All the remaining messages and keys are information-theoretically hidden from the receiver.*
2. *If the receiver recovers k messages then the receiver's query must have been of the form $(q_{i,1}, q_{i,2}) = (\gamma_i h^{t_i}, g^{t_i})$ for k distinct values $\gamma_1, \ldots, \gamma_k$ in the image of Γ such that their preimages $\sigma_1, \ldots, \sigma_k$ under Γ formed a correct selection w.r.t. $\mathbf{k}, \mathbf{n}$.*

To prove information-theoretic security we use the following lemma.

Lemma 6.3 *Let $\mathbb{F}$ be a finite field. Let integers k, m, n be given and let $A \in \mathbb{F}^{m \times k}, B \in \mathbb{F}^{n \times k}$ be matrices. We write $\mathbf{span}(A)$ for the span of the rows of a matrix A.*

If $\mathbf{span}(A) \cap \mathbf{span}(B) = \{\mathbf{0}\}$ then we have

$$(\forall \alpha \in \mathbb{F}^m)(\forall \beta \in \mathbb{F}^n) \ \underset{r \leftarrow \mathbb{F}^k}{\mathbf{Pr}} [Br = \beta \mid Ar = \alpha] = \underset{r \leftarrow \mathbb{F}^k}{\mathbf{Pr}} [Br = \beta]$$

In words, learning Ar reveals no information about Br.

We defer the proof to Section 6.2.

Proof of Theorem 6.2. In the random oracle model, since we assume H to be a random oracle then the only way to get $H(K_{i,j})$ with better probability than a random guess is to query the oracle on $K_{i,j}$. Therefore, if we can show for any i that all $K_{i,j}$ are information-theoretically hidden, then so is M_i.

We operate in the field $\mathbb{F} = \mathbb{Z}_q$, embedded in a multiplicative group over $\mathbb{Z}_p$. Write $[[x]]$ for $g^x \pmod p$ and set h' to be the discrete logarithm of h to basis g, e.g. $h = [[h']]$. We consider the following (computationally inefficient) variation on the sender, parameterised by an indexing function f and two matrices $\mathcal{B}, \mathcal{K}$:

$\underline{\mathsf{Respond}(\mathbf{q}, \mathbf{M})}$
$z, x \twoheadleftarrow \mathbb{Z}_q$
$w \leftarrow h^z g^x$
for $i = 1, \ldots, k$
  $b_i \twoheadleftarrow \mathbb{G}_q$
  $D_i \leftarrow q_{i,1}^z q_{i,2}^x b_i$
$\mu_0 \leftarrow 1;\ \nu_0 \leftarrow 1$
for $\beta = 1, \ldots, \ell$
  for $\mu = \mu_0, \ldots, \mu_0 + n_\beta$
    for $\nu = \nu_0, \ldots, \nu_0 + k_\beta$
      $K_{\mu,\nu} \leftarrow \Gamma(\mu)^z b_\nu$
      $C_{\mu,\nu} \leftarrow M_\mu \oplus H(K_{\mu,\nu})$
  $\mu_0 \leftarrow \mu_0 + n_\beta;\ \nu_0 \leftarrow \nu_0 + k_\beta$
return $(w, \mathbf{C}, \mathbf{D})$

$\underline{\mathsf{Respond'}(\mathbf{q}, \mathbf{M})}$
$\mathbf{r} \twoheadleftarrow \mathbb{Z}_q^{k+2}$
$(w', D'_1, \ldots, D'_k) \leftarrow \mathcal{B} \cdot \mathbf{r}$
$(w, D_1, \ldots, D_k) \leftarrow ([[w']], [[D'_1]], \ldots, [[D'_k]])$
$K' \leftarrow \mathcal{K} \cdot \mathbf{r}$

$\mu_0 \leftarrow 1;\ \nu_0 \leftarrow 1$
for $\beta = 1, \ldots, \ell$
  for $\mu = \mu_0, \ldots, \mu_0 + n_\beta$
    for $\nu = \nu_0, \ldots, \nu_0 + k_\beta$
      $i \leftarrow f(\mu, \nu, \mu_0, \nu_0)$
      $C_{\mu,\nu} \leftarrow M_\mu \oplus H([[K'_i]])$
  $\mu_0 \leftarrow \mu_0 + n_\beta;\ \nu_0 \leftarrow \nu_0 + k_\beta$
return $(w, \mathbf{C}, \mathbf{D})$

The inefficient sender creates the following matrices $\mathcal{B}$ and $\mathcal{K}$. For $\mathcal{B}$, let h' be the discrete logarithm of h and let q'_{ij} be the discrete logarithm of q_{ij}, all to basis g.

$$\mathcal{B} = \left(\begin{array}{cc|ccccc} h' & 1 & 0 & 0 & \cdots & 0 \\ \hline q'_{11} & q'_{12} & 1 & 0 & \ldots & 0 \\ q'_{21} & q'_{22} & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ q'_{k1} & q'_{k2} & 0 & 0 & \cdots & 1 \end{array}\right)$$

The distribution of the values $(w, \mathbf{D})$ is identical to that of the real sender: write $(z, x, b'_1, \ldots, b'_k)$ for the components of $\mathbf{r}$, then we have $w = [[h' \cdot z + x]] = h^z g^x$ and $D_i = [[q'_{i,1} z + q'_{i,2} x + b'_i]] = q_{i,1}^z q_{i,2}^x b_i$ for $b_i = [[b'_i]]$. Since the distribution of the b'_i is uniform in $\mathbb{Z}_q$, that of the b_i is uniform in $\mathbb{G}_q$.

For the matrix $\mathcal{K}$ we first consider the case $\ell = 1$, e.g. a single election where the voter can pick any k of n choices. Setting $\Gamma'(i)$ to be the discrete logarithm of $\Gamma(i)$ we have:

$$\mathcal{K} = \left(\begin{array}{cc|ccc} \Gamma'(1) & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \Gamma'(1) & 0 & 0 & \cdots & 1 \\ \hline \vdots & \vdots & \vdots & \ddots & \vdots \\ \hline \Gamma'(n) & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \Gamma'(n) & 0 & 0 & \cdots & 1 \end{array}\right)$$

For the indexing function $f(\mu, \nu, \mu_0, \nu_0) := \nu + (\mu - 1) \cdot k$ we compute

$$[[K'_{f(\mu,\nu)}]] = [[K'_{\nu+(\mu-1)\cdot k}]] = [[\Gamma'(\mu) z + b'_\nu]] = \Gamma(\mu)^z b_\nu$$

since the effect of $(\mu - 1) \cdot k$ is to skip to the μ-th block of k rows the matrix which contains

the $\Gamma'(\mu)$ entries; adding ν then selects the row in the block that chooses the b'_ν value. For this indexing function and matrix $\mathcal{K}$, this shows that that the real and alternative Respond algorithms (for $\ell = 1$) are equivalent.

Using Lemma 6.3, we claim that $\mathcal{B} \cdot \mathbf{r}$ contains no mutual information on $\mathcal{K}' \cdot \mathbf{r}$ where $\mathcal{K}'$ is the subset of the rows of $\mathcal{K}$ whose span does not intersect the span of the rows of $\mathcal{B}$ except in $\{0\}$. It remains to identify these rows.

Lemma 6.4 *For any w and for choice of the submatrix $Q = (q'_{ij})$ for $\mathcal{B}$,*

1. *At most k rows of $\mathcal{K}$ lie in the span of $\mathcal{B}$.*
2. *For every row of $\mathcal{K}$ in the span of $\mathcal{B}$, there is a distinct index j such that we have $q'_{j1} - h' \cdot q'_{j2} = \gamma$ where γ is the element in the first column of the relevant $\mathcal{K}$-row.*

Consider any row of $\mathcal{K}$. If this row is in the span of $\mathcal{B}$ then because of the identity matrix covering the last k components of $\mathcal{B}$, the row must be a linear combination $u \cdot (h', 1, 0, \ldots, 0) + v \cdot (q'_{j1}, q'_{j2}, 0, \ldots, 1, 0, \ldots, 0)$ for some $u, v \in \mathbb{F}$. In particular the rows with the 1 in a different column cannot contribute to this linear combination. This gives us the following equations.

$$\left| \begin{array}{rcl} u \cdot h' + v \cdot q'_{j,1} & = & \Gamma'(i) \\ u + v \cdot q'_{j,2} & = & 0 \\ v & = & 1 \end{array} \right|$$

where $\Gamma'(i)$ is the appropriate entry for the row, which we can solve for $q'_{1j} - h' \cdot q'_{2j} = \Gamma'(i)$. Therefore, each row of $\mathcal{K}$ in the span of $\mathcal{B}$ must be due to a matching pair $(q_{j,1}, q_{j,2})$ which is an ElGamal ciphertext for message $\Gamma(i)$. Since Γ is injective, one pair $(q_{j,1}, q_{j,2})$ cannot match more than one row of $\mathcal{K}$; it cannot match two rows with the same $\Gamma(i)$ either as the values in the later columns are distinct. There are only k such rows overall in $\mathcal{B}$ and $\mathcal{K}$ has size $(n \times k) \times (k + 2)$ so at least $(n - 1)k$ rows of $\mathcal{K}$ lie outside the span of $\mathcal{B}$, completing the proof of Lemma 6.4.

It follows that all but at most k of the keys K_{ij} are information-theoretically hidden from the receiver and every non-hidden row has a matching ElGamal ciphertext in the receiver's query. This proves sender security and robustness of the robust OT scheme for a single election.

For multiple elections where $\mathbf{k}$ is a vector of length greater than one, the matrix $\mathcal{K}$ has dimensions $(\sum_{i=1}^{\ell} n_i \cdot k_i) \times (k + 2)$. We give an example for $\mathbf{n} = (3, 4, 4)$ and $\mathbf{k} = (2, 3, 2)$, where empty blocks are all zeroes:

$$\mathcal{K} = \left(\begin{array}{cc||cc|ccc|cc}
\Gamma'_1 & 0 & 1 & 0 & & & & & \\
\Gamma'_1 & 0 & 0 & 1 & & & & & \\
\hline
\vdots & \vdots & \vdots & & & & & & \\
\hline
\Gamma'_3 & 0 & 1 & 0 & & & & & \\
\Gamma'_3 & 0 & 0 & 1 & & & & & \\
\hline\hline
\Gamma'_4 & 0 & & & 1 & 0 & 0 & & \\
\Gamma'_4 & 0 & & & 0 & 1 & 0 & & \\
\Gamma'_4 & 0 & & & 0 & 0 & 1 & & \\
\hline
\vdots & \vdots & & & \vdots & & & & \\
\hline
\Gamma'_7 & 0 & & & 1 & 0 & 0 & & \\
\Gamma'_7 & 0 & & & 0 & 1 & 0 & & \\
\Gamma'_7 & 0 & & & 0 & 0 & 1 & & \\
\hline\hline
\Gamma'_8 & 0 & & & & & & 1 & 0 \\
\Gamma'_8 & 0 & & & & & & 0 & 1 \\
\hline
\vdots & \vdots & & & & & & \vdots & \\
\hline
\Gamma'_{11} & 0 & & & & & & 1 & 0 \\
\Gamma'_{11} & 0 & & & & & & 0 & 1
\end{array}\right)$$

The matrix $\mathcal{K}$ is created as follows:

1. There are ℓ main blocks of rows. The i-th main block consists of n_i sub-blocks of k_i rows each.
2. Within the j-th sub-block of the i-th main block, the first column is filled with Γ'_S for $S = \sum_{s=1}^{i-1} n_s + (j-1)$. (This simply means that the index S increases by 1 as you move from one sub-block to the next.)
3. The second column is all zeroes.
4. For the remaining columns, each sub-block in the i-th main block carries an identity matrix in the k_i columns for this main block, that is the columns from T to $T + (k_i - 1)$ where $T = \sum_{t=1}^{i-1} k_t$.

Lemma 6.5 *We have the following facts about the matrices $\mathcal{B}$ and $\mathcal{K}$:*

1. *At most k rows of $\mathcal{K}$ lie in the span of $\mathcal{B}$.*
2. *For every row of $\mathcal{K}$ in the span of $\mathcal{B}$ there is a unique index j and pair (q'_{j1}, q'_{j2}) such that $q'_{j1} - h' \cdot q'_{j2} = \gamma$ where γ is the value in the first column of the relevant row of $\mathcal{K}$.*

Proof. Pick a row of $\mathcal{K}$ that is in the span of $\mathcal{B}$. This row consists of a value Γ'_m for some index m in the first column and exactly one 1 in one of the last k columns. The last k columns of $\mathcal{B}$ are still an identity matrix so the same argument as in Lemma 6.4 applies and we find a matching index j and pair (q'_{j1}, q'_{j2}). Since no two rows of $\mathcal{K}$ with the 1 in the same column have the same value in their first colums, the indices j for all rows of $\mathcal{K}$ in the span of $\mathcal{B}$ must be distinct. This proves the lemma.

It remains to define the indexing function $f(\mu, \nu, \mu_0, \nu_0)$. Define $\beta_* : [n] \to [\ell]$ to return the index of the batch in which a particular row of the matrices K and C lie and $\beta^* : [k] \to [\ell]$ to

be the same for colums, that is

$$\beta_*(i) := 1 + \max\{j \mid j = 0 \vee \textstyle\sum_{s=1}^{j} n_s < i\}$$
$$\beta^*(i) := 1 + \max\{j \mid j = 0 \vee \textstyle\sum_{s=1}^{j} k_s < i\}$$

We claim that in the modified Respond algorithm, we only access values $f(\mu, \nu, \mu_0, \nu_0)$ for which $\beta_*(\mu) = \beta_*(\mu_0) = \beta^*(\nu) = \beta^*(\nu_0)$. Specifically, while the outer loop has $\beta = i$ we only access values μ, μ_0 with $\beta_*(\mu) = i$ and values ν, ν_0 with $\beta^*(\nu) = i$. This is because we start μ_0, ν_0 both at 1 and after every iteration through the β loop we increment μ_0 by n_β and ν_0 by k_β. We define

$$f(\mu, \nu, \mu_0, \nu_0) := \begin{cases} \left(\sum_{s=1}^{\beta-1} n_s \cdot k_s\right) + (\mu - \mu_0) \cdot k_\beta + (\nu - \nu_0) & \text{if } \beta_*(\mu) = \beta^*(\nu) = \\ & \beta_*(\mu_0) = \beta^*(\nu_0) = \beta \\ \bot & \text{otherwise.} \end{cases}$$

The effect of the sum is to "skip" the first $\beta - 1$ main blocks. $(\mu - \mu_0)$ is the "local" row index, that is the index of the sub-block we are looking for within the current main block. Similarly $(\nu - \nu_0)$ is the index of the row within the sub-block. If $f(\mu, \nu, \mu_0, \nu_0) \neq \bot$ then the resulting row will be in the μ-th sub-block of the β-th main block. The value in the first column is therefore Γ'_S for $S = \sum_{s=1}^{\beta-1} k_s + (\mu - \mu_0) = \mu_0 + (\mu - \mu_0) = \mu$, i.e. this indeed picks a row containing Γ'_μ. The term $(\nu - \nu_0)$ selects the position of the 1 in the later rows by selecting a row within the sub-block; when $\nu = \nu_0$ we are in the first row of the Γ'_μ sub-block for example.

This means that for all parameters on which we call f in the modified Respond algorithm, we have

$$[[K_{f(\mu,\nu,\mu_0,\nu_0)}]] = [[\Gamma'_\mu \cdot z + b'_\nu]] = \Gamma(\mu)^z b_\nu$$

which shows that the modified algorithm acts identically to the original one. To complete the proof of Theorem 6.2 we apply Lemma 6.3 to the matrices $\mathcal{B}$ and $\mathcal{K}$ in the modified algorithm.

q.e.d.

6.2. Proof of Lemma 6.3

We thank Marx Stampfli from the University of Applied Science, Bern for comments on this proof. He produced an independent proof of the Lemma which we reproduce here with his permission.

Proof. Consider B as a linear map $\mathbb{F}^k \to \mathbb{F}^n$, sending r to Br. The kernel of this map is a subspace of $\mathbb{F}^k$ of the form $\mathbb{F}^b$ with $b \leq k$ and therefore has cardinality $|\mathbb{F}|^b$. The preimage sets $B_\beta := \{s \in \mathbb{F}^k \mid Bs = \beta\}$ are cosets of the kernel of B, so we can write $B_\beta = \{s + t \mid Bs = \beta \wedge t \in \mathbf{Ker}(B)\}$ (Lagrange). This shows that all these cosets have the same cardinality, namely $|\mathbf{Ker}(B)|$.

Similarly, the cosets $A_\alpha := \{s \in \mathbb{F}^k \mid As = \alpha\}$ all have cardinality $|\mathbf{Ker}(A)|$. Let a be such that $|\mathbf{Ker}(A)| = |\mathbb{F}|^a$, e.g. a is the dimension of the kernel of A as a $\mathbb{F}$-vector space. It suffices to prove for all $r \in \mathbb{F}^k$ that

$$\Pr[Br = 0 \mid Ar = 0] = \Pr[Br = 0]$$

Or equivalently, for uniform $r \in \mathbb{F}^k$,

$$\Pr[r \in \mathbf{Ker}(A) \cap \mathbf{Ker}(B)] = \Pr[r \in \mathbf{Ker}(A)] \times \Pr[r \in \mathbf{Ker}(A)]$$

Setting $C := \mathbf{Ker}(A) \cap \mathbf{Ker}(B)$ and letting c be the dimension of C as a $\mathbb{F}$-vector space, we see that it suffices to prove $a + b - c = k$. Writing **row** for the span of the rows of a matrix, since the row-span and the kernel of a matrix are orthogonal we have

$$\mathbf{Ker}(A) \times (\mathbf{row}(A) \cap \mathbf{row}(B)) = \mathbf{Ker}(B) \times (\mathbf{row}(A) \cap \mathbf{row}(B)) = \{0\}$$

and so

$$\mathbf{span}(\mathbf{Ker}(A) \cup \mathbf{Ker}(B)) = \mathbf{Ker}(\mathbf{row}(A) \cap \mathbf{row}(B))$$

but $\mathbf{row}(A) \cap \mathbf{row}(B) = \{0\}$ so the right-hand side of the equation is $\mathbf{Ker}(\{0\}) = \mathbb{F}^k$. Therefore the span of the union of kernels on the left-hand side is all $\mathbb{F}^k$ and so $a+b \geq k$. The part counted twice on the left-hand side is the intersection of the kernels, which is of dimension c as we defined earlier. Therefore $a + b = c + k$, proving the Lemma. q.e.d.

6.3. OT and Polynomials

Since the $OT^{\mathbf{k}}_{\mathbf{n}}$ protocol will be used to transfer points on a polynomial, we recall the following fact about polynomials.

Lemma 6.6 *Let p be a randomly chosen polynomial of degree-bound $k-1$ over a finite field $\mathbb{F}$. Given up to $k-1$ points $(x_i, y_i = p(x_i))$ on the polynomial, all other points on the polynomial are information-theoretically hidden.*

Proof. A polynomial of degree-bound $k-1$ is defined by coefficients $\mathbf{r} = (r_0, r_1, \ldots, r_{k-1})$. Revealing up to $k-1$ points (x_i, y_i) means revealing $A\mathbf{r}$ for the following matrix A:

$$\begin{pmatrix} 1 & x_1 & (x_1)^2 & \cdots & (x_1)^{k-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{k-1} & (x_{k-1})^2 & \cdots & (x_{k-1})^{k-1} \end{pmatrix}$$

For any set of points $(x^*_j)_{j \in J}$ indexed by some set J, we can express $p(x^*_j)$ by a similar matrix B, known as a Vandermonde matrix. Since a square Vandermonde matrix for k distinct points has maximal rank, any individual row $(1, x^*_j, \ldots, (x^*_j)^{k-1})$ with $x^*_j \notin \{x_1, \ldots, x_{k-1}\}$ must be linearly independent of the rows of A. Therefore $\mathbf{Span}(A)$ and $\mathbf{Span}(B)$ can only intersect in $\{0\}$ as long as the points represented by B are distinct from those represented by A. Using Lemma 6.3, revealing up to $k-1$ points on the polynomial does not reveal any information about the remaining points. q.e.d.

The CH-Vote protocol combines OT, ballot correctness and voter authentication by the use of the second voter key $\widehat{y} = \widehat{g}^{y+y'}$ where y is known only to the voter and y' can only be obtained through the OT protocol.

We define two games POINTS-0 and POINTS-1 which we use in the security proof (the games are in Figures 18,19)). The games are stateful components that begin with a call Initialise(k, n, ℓ)

for fixed parameters. The outputs of all algorithms are sent to the adversary. The adversary can then call Query up to ℓ times and Oracle any number of times. Variables retain their state between calls to the different algorithms. The algorithm Oracle implements a random oracle with range R. The adversary wins an execution against a game by calling Finalise with an input that makes it return 1; calling this algorithm ends the game whether the adversary has won or not.

Game POINTS-0 models the key and polynomial generation for a honest voter by a honest authority. Setting $\ell = k$, would we let the voter learn at most k points by querying the component (this models the OT queries). Once the voter has k points, they can reconstruct the whole polynomial themselves. In the ballot verifiability proof we will consider these games for $\ell = k - 1$, e.g. the voter cannot learn the whole polynomial.

Game POINTS-0

$\underline{\mathsf{Initialise}(k, n, \ell)}$
$H \leftarrow \langle\rangle;\ Q \leftarrow \ell$
$a_0, a_1, \ldots, a_{k-1} \twoheadleftarrow \mathbb{Z}_{p'}$
$X \leftarrow \{\}$
for $i = 1, \ldots, n$
  $x_i \twoheadleftarrow \mathbb{Z}_{p'} \setminus X$

  $X \leftarrow X \cup \{x_i\}$
  $y_i \leftarrow \mathsf{GetYValue}(x_i, \mathbf{a})$
  $p_i \leftarrow (x_i, y_i)$
  $d_i \leftarrow \mathsf{Oracle}(p_i)$
$d_F \leftarrow \mathsf{Oracle}(\mathbf{p})$
$y' \leftarrow \mathsf{GetYValue}(0, \mathbf{a})$
$x, y \twoheadleftarrow \mathbb{Z}_{\lfloor \widehat{q}/s \rfloor}$
$\mathbf{d} \leftarrow (x, y, d_F, (d_1, \ldots, d_n))$
$\widehat{\mathbf{d}} \leftarrow (\widehat{g}^x, \widehat{g}^{y+y'})$
return $(\mathbf{d}, \widehat{\mathbf{d}})$

$\underline{\mathsf{Query}(i)}$
$Q \leftarrow Q - 1$
if $Q \leq 0$ then return $\perp$
if $1 \leq i \leq n$ then
  return (x_i, y_i)
return $\perp$

$\underline{\mathsf{Oracle}(u)}$
if $\exists\ (u, r) \in H$ then return r
$r \twoheadleftarrow R; H \leftarrow \langle H, (u, r)\rangle$; return r

$\underline{\mathsf{Finalise}(z)}$
return $(z = y')$

Figure 18: Game POINTS-0.

Game POINTS-1 (Figure 19) is obtained by modifying POINTS-0 as follows: the value c is chosen randomly instead of by computing $p(0)$; the values d_i and d_F are generated independently of the points p_i and we keep track of all queried points in a list L and abort the game if the adversary makes a random oracle query on a point that it is not supposed to know yet.

Lemma 6.7 *For any positive integers $k \leq n$ and $\ell \leq k - 1$, the two games POINTS-0 and POINTS-1 are indistinguishable in the random oracle model. For any adversary winning against the game POINTS-0 on such parameters (k, n, ℓ) with advantage α and making at most ν oracle*

Game POINTS-1

Initialise(k, n, ℓ)
$H \leftarrow \langle\rangle$; $Q \leftarrow \ell$; $L \leftarrow \langle\rangle$
$a_0, a_1, \ldots, a_{k-1} \twoheadleftarrow \mathbb{Z}_{p'}$
$X \leftarrow \{\}$
for $i = 1, \ldots, n$
  $x_i \twoheadleftarrow \mathbb{Z}_{p'} \setminus X$
  if $x_i = 0$ then abort
  $X \leftarrow X \cup \{x_i\}$
  $y_i \leftarrow$ GetYValue$(x_i, \mathbf{a})$
  $p_i \leftarrow (x_i, y_i)$
  $d_i \twoheadleftarrow R$; $H \leftarrow \langle H, (p_i, d_i)\rangle$
$d_F \twoheadleftarrow R$; $H \leftarrow \langle H, (\mathbf{p}, d_F)\rangle$
$c \twoheadleftarrow \mathbb{Z}_{p'}$; $C \leftarrow \widehat{g}^c$
$x, y \twoheadleftarrow \lfloor \mathbb{Z}_{\widehat{q}}/s \rfloor$
$\mathbf{d} \leftarrow (x, y, d_F, (d_1, \ldots, d_n))$
$\widehat{d} \leftarrow (\widehat{g}^x, \widehat{g}^y C)$
return $(\mathbf{d}, \widehat{\mathbf{d}})$

Query(i)
$Q \leftarrow Q - 1$
if $Q \leq 0$ then return $\perp$
if $1 \leq i \leq n$ then
  $L \leftarrow \langle L, x_i\rangle$; return (x_i, y_i)
return $\perp$

Oracle(u)
if $u = \mathbf{p} \vee (u = (x, y) \in \mathbf{p} \wedge x \notin L)$
then abort
if $\exists\ (u, r) \in H$ then return r
$r \twoheadleftarrow R$; $H \leftarrow \langle H, (u, r)\rangle$; return r

Finalise(z)
return $(z = c)$

Figure 19: Game POINTS-1.

queries, the adversary must also win against POINTS-1 with probability at least

$$\alpha \left(1 - \nu \frac{(n+1)}{p'}\right) \left(1 - \frac{n}{p' - n + 1}\right).$$

Proof. The proof is by "game hopping". Begin with game POINTS-0 and replace the generation of the values d_F and $d_1, \ldots, d_k$ by the method in which they are generated in game POINTS-1. This does not change the distribution of these values, which is uniform in R and independent of $\mathbf{p}$ in both games.

Next, add the condition that we abort if an x_i is 0. The probability of this event is the highest when we are picking the last point x_n as the sampling domain has shrunk to a size of $p' - (n-1)$, albeit for $n \ll p$. We then use the union bound for the probability that any of the n values hits zero to get $n/(p' - n + 1)$.

Next, add the list L and the abort condition in the random oracle. For an adversary making up to ν random oracle queries, the probability that a query hits a "forbidden" point which causes an abort is the highest when no points have been revealed yet, that is no Query queries have been made yet. Since the list L is only ever added to, making Query queries can only reduce the probability that a particular random oracle query Oracle(u) causes an abort and Query queries do not reveal anything about the values x_i that are still forbidden to call. Before the first such query, since there are $n + 1$ forbidden points and since the adversary's view is independent of these points, the probability of aborting is the number of points times the inverse of the size of the space of the points. The points x_i for $1 \leq i \leq n$ are in $\mathbb{Z}_{p'}$ and the point $\mathbf{p}$ is in $\mathbb{Z}_{p'}^{2\times n}$ whose size is at least that of $\mathbb{Z}_{p'}$. Therefore the probability of aborting with ν queries is at most

$(n+1)\nu/p'$.

Remark 6.1 *It would be tempting to get a bound with a p'^2 in the denominator by only aborting if the adversary has hit an exact point (x_i, y_i). The problem with this approach is that if the adversary knows exactly $k-1$ points and has somehow learnt $y' = p(0)$, for example by taking a discrete logarithm, then the adversary can query $(x, p(x))$ for any value of x that it knows; if it hits one of the remaining $n-k+1$ of our chosen points then we are stuck. What protects us here is that the remaining x_i values are information-theoretically hidden so the adversary has at most a $(n+1)/p'$ chance of hitting one per query; guessing $\mathbf{p}$ means guessing all x-values correctly which is certainly no easier than guessing one of them.*

The value $\mathbf{a}$ is a random polynomial of degree-bound $k-1$. Lemma 6.6 says that an adversary making at most $k-1$ Query queries and thus learning at most $k-1$ points on the polynomial gains no information on the point $p(0) = y'$ from the points (x_i, y_i). We have already ensured that the game aborts if the adversary makes a random oracle query that could reveal information on further points. This allows us to switch $y' = p(0)$ for c since both look like uniformly random points in $\mathbb{Z}_{p'}$ to the adversary. Specifically, if we define the view of the adversary to be all inputs and outputs they have seen from the game so far e.g. $(\mathbf{d}, \widehat{\mathbf{d}}, x, y)$ and the lists $(i, (x_i, y_i))$ and (u_j, v_j) from the Query and Oracle queries respectively, then conditionally on the game not having aborted, the adversary's view is indepdendent of $p(0)$. Multiplying the adversary's original advantage α with the probability of *not* aborting gives the desired bound. q.e.d.

Lemma 6.8 *For any adversary winning against game POINTS-1 with probability α, there is a lossless reduction to winning the following discrete logarithm game where the discrete logarithms are sampled from the subset $\mathbb{Z}_{p'}$ of $\mathbb{Z}_{\widehat{q}}$:*

$$c \twoheadleftarrow \mathbb{Z}_{p'};\ c' \leftarrow \mathcal{A}(\widehat{g}^c);\ \textit{return}\ (c = c')$$

Proof. Game POINTS-1 never uses the value c except to create $C = \widehat{g}^c$. We can therefore simply obtain C from the discrete logarithm challenger; the winning condition for the second game is exactly to find the discrete logarithm of C. q.e.d.

6.4. The OT experiments for ballot verifiability

We can now define the normal and simulated OT experiments that we will use to prove the verifiability properties. As usual we use n, k for the component-wise sums of $\mathbf{n}, \mathbf{k}$.

The normal experiment OT-BV represents an honest election authority that is willing to interact once with a voter (played by the adversary). The election authority takes the role of sender in the OT protocol to transfer the points on the voter's polynomial corresponding to the voter's ballot. The simulated experiment OT-SIM is constructed with the following difference: instead of using $p(0)$ to create the voter's second key pair, the experiment picks a random y' in $\mathbb{Z}_{p'}$. The two experiments can easily be distinguished by any adversary who makes a correct OT query, as the keys presented in $\widehat{\mathbf{d}}$ and the points recovered from the OT will not match up. However, the point of the experiment OT-SIM is that we will show it is indistinguishable from OT-BV in any execution where the adversary does not make a valid OT query. We use a third experiment, the reduction OT-RED, to prove this.

We define the following encoding function (which is how CH-Vote encodes the differnt pairs of points held by election authorities)

$$\mathsf{Encode}(x, y) := \mathsf{ToByteArray}(x, L_M/2) \| \mathsf{ToByteArray}(y, L_M/2)$$

First, we define two algorithms in Figure 20 that we will use in the experiments.

$\underline{\mathsf{Prepare}(\mathbf{k}, \mathbf{n})}$
$a_0, a_1, \ldots, a_{k-1} \twoheadleftarrow \mathbb{Z}_{p'}$
$X \leftarrow \{\}$
for $i = 1, \ldots, n$
  $x_i \twoheadleftarrow \mathbb{Z}_{p'} \setminus X$
  $X \leftarrow X \cup \{x_i\}$
  $y_i \leftarrow \mathsf{GetYValue}(x_i, \mathbf{a})$
  $p_i \leftarrow (x_i, y_i)$
  $d_i \leftarrow H(p_i)$
  $M_i \leftarrow \mathsf{Encode}(x_i, y_i)$
$d_F \leftarrow H(\mathbf{p})$
$y' \leftarrow \mathsf{GetYValue}(0, \mathbf{a})$
$x, y \twoheadleftarrow \lfloor \mathbb{Z}_{\widehat{q}}/s \rfloor$
$\mathbf{d} \leftarrow (x, y, d_F, (d_1, \ldots, d_n))$
$\widehat{\mathbf{d}} \leftarrow (\widehat{g}^x, \widehat{g}^{y+y'})$
return $(x, y, y', \mathbf{d}, \widehat{\mathbf{d}})$

$\underline{\mathsf{Respond}(\mathbf{q}, h)}$
$z, x \twoheadleftarrow \mathbb{Z}_q$
$w \leftarrow h^z g^x$
for $i = 1, \ldots, k$
  $b_i \twoheadleftarrow \mathbb{G}_q$
  $D_i \leftarrow q_{i,1}^z q_{i,2}^z b_i$
$\mu_0 \leftarrow 1;\ \nu_0 \leftarrow 1$
for $\beta = 1, \ldots, \ell$
  for $\mu = \mu_0, \ldots, \mu_0 + n_\beta$
    for $\nu = \nu_0, \ldots, \nu_0 + k_\beta$
      $K_{\mu,\nu} \leftarrow \Gamma(\mu)^z b_\nu$
      $C_{\mu,\nu} \leftarrow M_\mu \oplus H(K_{\mu,\nu})$
  $\mu_0 \leftarrow \mu_0 + n_\beta;\ \nu_0 \leftarrow \nu_0 + k_\beta$
return $(w, \mathbf{C}, \mathbf{D})$

Figure 20: The algorithms for the OT experiments.

Definition 6.9 (OT experiments) *The OT experiments are as follows. They are in the random oracle model where H is the random oracle.*

- *The OT experiment OT-BV is the following experiment:*

$\underline{\mathsf{OT\text{-}BV}_{\mathcal{A}}(\mathbf{k}, \mathbf{n})}$
$(x, y, y', \mathbf{d}, \widehat{\mathbf{d}}) \leftarrow \mathsf{Prepare}(\mathbf{k}, \mathbf{n})$
$(\mathbf{q}, h) \leftarrow \mathcal{A}_1^{H(\cdot)}(\mathbf{d}, \widehat{\mathbf{d}})$
$(w, \mathbf{C}, \mathbf{D}) \leftarrow \mathsf{Respond}(\mathbf{q}, h)$
$z \leftarrow \mathcal{A}_2^{H(\cdot)}(w, \mathbf{C}\ \mathbf{D})$
if $z = y'$ then return 1 else return 0

- *The simulated OT experiment OT-SIM is the experiment OT-BV modified as follows: the first highlighted line (in Prepare) is replaced by* $\boxed{y' \twoheadleftarrow \mathbb{Z}_{p'}}$.
- *The OT reduction OT-RED is the algorithm OT-BV modified as follows:*
  - *The reduction takes an extra parameter $C \in \mathbb{G}_{\widehat{q}}$ as input.*
  - *The first highlighted line that creates y' (in Prepare) is removed.*
  - *The (highlighted) second component of $\widehat{\mathbf{d}}$ is replaced by* $\boxed{\widehat{g}^y C}$.

– *The* Prepare *algorithm returns* $\perp$ *in place of the removed* y'.
– *Instead of defining its own winning condition, the reduction simply returns the value* z *output by the adversary.*

When we work with the OT games in later security games, we will use the following syntax. The security game will play the role of the adversary towards the OT game. In a security game, the line $(\mathbf{d}, \widehat{\mathbf{d}}) \leftarrow \mathsf{OT\text{-}BV.Prepare}(\mathbf{k}, \mathbf{n})$ initialises an instance of the OT game and runs it until the point where it returns $(\mathbf{d}, \widehat{\mathbf{d}})$ to the adversary. Later in the security game, the line $(w, \mathbf{C}, \mathbf{D}) \leftarrow \mathsf{OT\text{-}BV.Respond}(\mathbf{q}, h)$ passes $(\mathbf{q}, h)$ to the OT game which continues executing until the point where it returns $(w, \mathbf{C}, \mathbf{D})$ to the adversary. From this point on we will argue abstractly about the probability of winning the OT game. We emphasise that the calls to OT-BV.Prepare and OT-BV.Respond correspond to stages in the execution of the OT game, not direct calls to the algorithms that make up this game; this is because one of the functions of the game is to hide some of the algorithms' output (such as y') from the adversary.

We prove the following theorem about our OT experiments for ballot verifiability.

Theorem 6.10 *Call an OT query* $(\mathbf{q}, h)$ *correct if it is a vector of* k *ElGamal ciphertexts with public key* h *such that each ciphertext encodes a distinct point in the range* $\{\Gamma(1), \ldots, \Gamma(n)\}$ *such that their preimages under* Γ *are a correct selection w.r.t.* $\mathbf{k}, \mathbf{n}$.

1. *No adversary who makes an incorrect query can distinguish games* OT-BV *and* OT-SIM. *Specifically, if such an adversary wins against* OT-BV *with advantage* α *then it must win against* OT-SIM *with advantage at least*

$$\alpha \left(1 - \nu \frac{(n+1)}{p'}\right) \left(1 - \frac{n}{p' - n + 1}\right).$$

2. *Winning* OT-SIM *without producing a corect OT query reduces without any loss to the breaking the following discrete logarithm property*

$$c \twoheadleftarrow \mathbb{Z}_{p'};\ c' \leftarrow \mathcal{A}(\widehat{g}^c);\ \textit{return}\ (c = c').$$

The second point follows immediately from the first as the combination of the reduction OT-RED and the discrete logarithm challenger is equivalent to OT-SIM.

To argue the first point, we proceed as follows. Take the OT-BV game and consider the "hop" to the following inefficient game OT-HOP-1 in Figure 21, where POINTS-0 handles random oracle queries. We let $\mathcal{M}$ be the space of messages M_i.

We are arguing that the normal and simulated OT experiments, both of which are efficient, are hard to distinguish. For this we are allowed to introduce inefficient games as intermediary "hops". This game OT-HOP-1 takes a discrete logarithm to find the secret key for the ElGamal encryptions and then answers the OT queries in such a way that the messages which the adversary is allowed to see are correctly formed.

We claim that the normal OT experiment OT-BV is indistinguishable from this hop OT-HOP-1 for adversaries that do not make a correct OT query. Actually, the indistinguishability would also hold for correct queries if we used k instead of $k-1$ as the last parameter to the initialiser of the sub-game POINTS-0.

Game OT-HOP-1

Initialise$(\mathbf{k}, \mathbf{n})$
return POINTS-0.Initialise$(\mathbf{k}, \mathbf{n}, k-1)$

Finalise(z):
return POINTS-0.Finalise(z)

Respond$(\mathbf{q}, h)$
$s \leftarrow \mathsf{DLOG}_g(h)$
for $i = 1, \ldots, k$
  $a_i \leftarrow q_{i,1}/(q_{i,2})^s$
  $M_i \twoheadleftarrow \mathcal{M}$
  for $j = 1, \ldots, n$
    if $a_i = \Gamma(j)$ then
      $(x_i, y_i) \leftarrow$ POINTS-0.Query(i)
      $M_i \leftarrow \mathsf{Encode}(x_i, y_i)$

$z, x \twoheadleftarrow \mathbb{Z}_q$
$w \leftarrow h^z g^x$
for $i = 1, \ldots, k$
  $b_i \twoheadleftarrow \mathbb{G}_q$
  $D_i \leftarrow q_{i,1}^z q_{i,2}^z b_i$
$\mu_0 \leftarrow 1;\ \nu_0 \leftarrow 1$
for $\beta = 1, \ldots, \ell$
  for $\mu = \mu_0, \ldots, \mu_0 + n_\beta$
    for $\nu = \nu_0, \ldots, \nu_0 + k_\beta$
      $K_{\mu,\nu} \leftarrow \Gamma(\mu)^z b_\nu$
      $C_{\mu,\nu} \leftarrow M_\mu \oplus H(K_{\mu,\nu})$
  $\mu_0 \leftarrow \mu_0 + n_\beta;\ \nu_0 \leftarrow \nu_0 + k_\beta$
return $(w, \mathbf{C}, \mathbf{D})$

Figure 21: The Game OT-HOP-1.

We know from Theorem 6.2 that all but at most k messages M_i are information-theoretically hidden from the adversary and that the non-hidden ones correspond to correct points in the query. Therefore it does not matter if we pick the remaining messages randomly.

Assuming an incorrect OT query, the second highlighted line gets called at most $k-1$ times. The game POINTS-0 therefore does not abort here. The messages thus obtained are distributed and encoded identically to before. Therefore this game is perfectly indistinguishable from the normal OT experiment.

For our next hop we define OT-HOP-2 to be OT-HOP-1 with POINTS-0 replaced by POINTS-1. Game OT-HOP-1 can be seen as a reduction to the POINTS games. Assuming that adversary $\mathcal{A}$ wins OT-BV with probability α, then OT-HOP-1$^{\mathcal{A}}$ must win POINTS-0 with probability α too; therefore by Lemma 6.7 the reduction OT-HOP-1$^{\mathcal{A}}$ must win POINTS-1 with probability at least $\alpha(1 - \nu(n+1)/p')(1 - n/(p' - n + 1))$. This reduction is not efficient, but efficiency is not a precondition in Lemma 6.7.

For OT-HOP-3, we remove the following abort conditions in OT-HOP-2: we no longer abort if $x_i = 0$ or if the adversary makes a "forbidden" oracle query. Since these abort conditions cannot occur in any execution of OT-HOP-2 in which the adversary wins the game, the probability of winning cannot be reduced by removing these conditions.

We claim that OT-HOP-3 is indistinguishable from OT-SIM and so the winning probabilities in either game must be the same. This is essentially the same argument as comparing OT-BV and OT-HOP-1: instead of discovering the messages that the adversary will learn and randomising the rest, we just let the OT protocol take care of that. The values of all the remaining messages are information-theoretically hidden by Theorem 6.2.

$\underline{\mathsf{OT\text{-}IV}_{\mathcal{A}}(\mathbf{k},\mathbf{n})}$
$(x, y, y', \mathbf{d}, \widehat{\mathbf{d}}) \leftarrow \mathsf{Prepare}(\mathbf{k}, \mathbf{n})$
$(\mathbf{q}, h) \leftarrow \mathcal{A}_1^{H(\cdot)}(x, y, y', \widehat{\mathbf{d}})$
$(w, \mathbf{C}, \mathbf{D}) \leftarrow \mathsf{Respond}(\mathbf{q}, h)$
$(i^*, r^*) \leftarrow \mathcal{A}_2^{H(\cdot)}(w, \mathbf{C}\ \mathbf{D})$
$\mathbf{s} \leftarrow \mathsf{Extract}(\mathbf{q}, h)$
if $i^* \notin \mathbf{s} \wedge r^* = \mathsf{Truncate}(H(M_{i^*}, L_R))$ then return 1 else return 0

Figure 22: Game for defining OT-IV security of the OT protocol that underlies CH-Vote

In conclusion, if an adversary $\mathcal{A}$ wins OT-BV on an incorrect OT query with probability α then $\mathcal{A}$ must still win OT-SIM on an incorrect OT query with probability at least $\alpha(1 - \nu(n + 1)/p')(1 - n/(p' - n + 1))$ as demanded in the theorem. q.e.d.

6.5. The OT game for individual verifiability

Here we consider a variation on the OT-BV game, where the adversary gets to see the voter secrets and aims to recover the (truncated) hash of one of the messages held by the sender which she did not query. The game uses the inefficient algorithm $\mathsf{E}(\mathbf{q}, h)$ which recovers the discrete logarithm of h and then decrypts the ciphertexts in $\mathbf{q}$ using the recovered value as the secret key. We also let $\mathsf{Truncate}(\cdot, L_R)$ be the algorithm which on an input m outputs the first L_R bytes of m.

We define the game $\mathsf{OT\text{-}IV}_{\mathcal{A}}(\mathbf{k}, \mathbf{n})$ in Figure 22. As before, it considers a setting where the sender OT messages are prepared as in the CH-Vote protocol. The hidden messages, held by the sender, are n random points on a random polynomial $\mathbf{p} \in \mathbb{Z}_{p'}[X]$, encoded as $M_i = \mathsf{Encode}(x_i, y_i)$ with $x_i \twoheadleftarrow \mathbb{Z}'_p$ and $y_i = \mathbf{p}(x_i)$. The adversary gets as input x, y (which coorespond to) the votin code and finalization code and also gets $y' = \mathbf{p}(0)$ and $\hat{\mathbf{d}} = (\hat{g}^x, \hat{g}^{y+y'})$ [12] The adversary then makes up an OT request $(\mathbf{q}, h)$ and receives the OT reponse $(w, \mathbf{C}, \mathbf{D})$. His goal is to predict the value of $\mathsf{Truncate}(H(M_{i^*}, L_R))$ for some point i^* which is not among the points queried in the OT request.

We define the advantage of an adversary against the scheme in the above game by:

$$\mathbf{Adv}_{\mathcal{A}}^{\mathsf{OT\text{-}IV}} = \mathbf{Pr}[\mathsf{OT\text{-}IV}_{\mathcal{A}}(\mathbf{k}, \mathbf{n}) = 1]$$

The following Theorem establishes the security of the modified Chu-Zheng scheme with respect to the notion above.

Theorem 6.11 *For any adversary $\mathcal{A}$ it holds that:*

$$\mathbf{Adv}_{\mathcal{A}}^{\mathsf{OT\text{-}IV}} \leq \max\left(\frac{1}{2^{8L_R}}, \frac{1}{p'}\right)$$

[12] Strictly speaking $\hat{\mathbf{d}}$ is redundant but we include it since it eases the later simulation

Proof. By Theorem 6.2 all messages $M_i = \mathsf{Encode}(x_i, y_i)$ that the adversary does not query are information theoretically hidden, subject to the restriction that $\mathbf{p}(x_i) = y_i$, where $\mathbf{p}$ is the $k-1$ degree polynomial that passes through the n points. Since the adversary can learn $\mathbf{p}$, the only source of uncertainty is in the selection of the x_i's. If L is the length of the output of H then the probability that the adversary guesses the value $\mathsf{Truncate}(H_L(M_{i^*}), L_R)$ is upperbounded by $\max(\frac{1}{2^{8L_R}}, \frac{1}{p'})$ corresponding to either guessing directly the value $\mathsf{Truncate}(H_L(M_{i^*}, L_R)$ or guessing x_i directly.

7. Cryptographic Proofs

7.1. Proof of IV – Theorem 5.2

To prove the theorem we provide a sequence of games which we use to bound the advantage of any adversary in $\mathbf{Exp}^{\mathsf{iv}}_{\mathcal{A}}$. We consider three experiments, denoted $\mathbf{Exp}^{\mathsf{iv}\text{-}x}_{\mathcal{A}}$ for $x = 1, 2, 3$, obtained by incrementally changing $\mathbf{Exp}^{\mathsf{iv}}$. For each experiment we write $\mathbf{Adv}^{\mathsf{iv}\text{-}x}_{\mathcal{A}}$ for $\mathbf{Pr}[\mathbf{Exp}^{\mathsf{iv}\text{-}x}_{\mathcal{A}} = 1]$.

In the first non-trivial change to the experiment we argue that if the first check of the user yields *true* then the first check of the authority is also true, or otherwise the adversary has guessed all of the verification codes. Intuitively, this is the case since if the first check of the authority is false, then the authority does not answer the OT request hence the adversary needs to guess them (or the contribution of the authority to these codes).

Technically, we modify the game $\mathbf{Exp}^{\mathsf{iv}}_{\mathcal{A}}$ by introducing the event $\mathbf{bad}_1$ which is set to *true* if ucheck_1 is true but acheck_1 is false. In this case the experiment will abort. We call the resulting experiment $\mathbf{Exp}^{\mathsf{iv}\text{-}1}_{\mathcal{A}}$. We detail it in Figure 23 – the shadowed line indicates the change over $\mathbf{Exp}^{\mathsf{iv}}_{\mathcal{A}}$.

$\mathbf{Exp}^{\mathsf{iv}\text{-}1}_{\mathcal{A},\Pi}(s, t, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{q})$
- $\mathsf{acheck}_1 \leftarrow false, \mathsf{acheck}_2 \leftarrow false$
- $\mathsf{ucheck}_1 \leftarrow false, \mathsf{ucheck}_2 \leftarrow false$
- $\mathsf{Setup}_{\mathcal{A}}(s, t, \mathbf{n}, \mathbf{k}, \mathbf{E})$
- $\mathbf{s} = (s_1, s_2, \ldots, s_{|\mathbf{s}|}) \leftarrow \mathcal{A}(\hat{\mathbf{d}}_1)$
- $\alpha = (\mathbf{a}, \pi) \leftarrow \mathcal{A}(X_1)$
- $\mathbf{s}^* \leftarrow \mathsf{Extract}(\alpha, pk)$
- $\mathsf{acheck}_1 \leftarrow \mathsf{CheckBallot}(1, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}}, B_1)$
- if $(\mathsf{acheck}_1 = true)$ then $(\beta_1, z) \leftarrow \mathsf{GenResponse}(1, \mathbf{a}, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P}_1)$
- $(rc^*_1, rc^*_2, \ldots, rc^*_{|\mathbf{s}|}) \leftarrow \mathcal{A}(\beta_1)$
- if $(\forall i \in [|\mathbf{s}|])\ rc^*_i = \mathbf{rc}_{1 s_i}$ then $\mathsf{ucheck}_1 \leftarrow true$; $\gamma \leftarrow \mathcal{A}(Y_1)$
  - else return 0
- if $\mathsf{ucheck}_1 = true$ and $\mathsf{acheck}_1 = false$ then $\mathbf{bad}_1 \leftarrow true$ abort
- $\mathsf{acheck}_2 \leftarrow \mathsf{CheckConfirmation}(1, \gamma, \hat{\mathbf{y}}, B_1, C_1)$
- if $(\mathsf{acheck}_1 = true)$ and $(\mathsf{acheck}_2 = true)$ then $\delta_1 \leftarrow \mathsf{GetFinalization}(1, \mathbf{P}_1, B_1)$
- $FC^* \leftarrow \mathcal{A}(\delta_1)$
- if $\mathsf{ucheck}_1 = true$ then $\mathsf{ucheck}_2 \leftarrow (FC_1 = FC^*)$
- if $(\mathsf{ucheck}_2 = true)$ and $((\mathsf{acheck}_2 = false)$ or $(\mathbf{s}^* \neq \mathbf{s}))$ return 1
  - else return 0

Figure 23: The first game hop: the first check of the election authority fails, yet the return codes are valid

Lemma 7.1 *For any adversary $\mathcal{A}$ it holds that*

$$\left|\mathbf{Adv}^{\mathsf{iv}\text{-}1}_{\mathcal{A}} - \mathbf{Adv}^{\mathsf{iv}}_{\mathcal{A}}\right| \leq \binom{n}{|\mathbf{s}|} \cdot |\mathbf{s}|! \cdot \max\left(\frac{1}{2^{8L_R}}, \frac{1}{p'}\right)^{|\mathbf{s}|}$$

Return codes
$(\mathbf{p}_{11} = (x_{111}, y_{111}), \ldots, (x_{11n}, y_{11n}), y'_{11}) \leftarrow \mathsf{GenPoints}(n, k'_1)$
$d_{11} = (x_{11}, y_{11}, F_{11}, \mathbf{r}_{11}) \leftarrow \mathsf{GenSecretVoterData}(\mathbf{p}_{11})$
$\hat{d}_{11} = (\hat{x}_{11}, \hat{y}_{11}) \leftarrow \mathsf{GetPublicVoterData}(x_{11}, y_{11}, y'_{11})$
for $j \leftarrow 2, \ldots, s$ **do**
  $d_{1j} = (x_{1j}, y_{1j}, F_{1j}, \mathbf{r}_{1j}) \leftarrow \mathcal{A}()$
$X_1 \leftarrow \sum_{j=1}^{s} x_{1j}$
$Y_1 \leftarrow \sum_{j=1}^{s} y_{1j}$
for $k \leftarrow 1, \ldots, n$ **do**
  $\mathbf{rc}_{1k} \leftarrow \oplus_{j=1}^{s} \mathbf{r}_{1jk}$

Figure 24: Computation of the return codes for voter 1; we ignore conversion to strings and watermarking of the codes, since these do not affect in a relevant way the probability distribution of the codes

where $\mathbf{s}$ *is the selection of voter* 1*.*

Proof Since $\mathbf{Exp}^{\mathsf{iv}\text{-}1}_{\mathcal{A}}$ and $\mathbf{Exp}^{\mathsf{iv}}_{\mathcal{A}}$ are identical except if even $\mathbf{bad}_1$ occurs we get that:

$$\left|\mathbf{Adv}^{\mathsf{iv}\text{-}1}_{\mathcal{A}} - \mathbf{Adv}^{\mathsf{iv}}_{\mathcal{A}}\right| \leq \mathbf{Pr}[\mathbf{bad}_1]$$

and we obtain the desired inequality by bounding $\mathbf{Pr}[\mathbf{bad}_1]$. Informally, this event is raised when the first message sent by the adversary on behalf of voter 1 (i.e. the signed OT request) does not pass the ballot check performed by the entity authentication yet the adversary returns valid verification codes for all of the voter selections. Since the election authority does not answer invalid requests, its contribution to each of the verification codes stays information theoretically hidden: the only way for the adversary to return valid codes is to guess them.

In other words, if $\mathbf{bad}_1$ occurs (so $\mathsf{acheck}_1 = false$) then the view of the adversary up to when it returns codes $rc^*_1, \ldots, rc^*_s$ contains no additional information about $\mathbf{rc}_{11}, \mathbf{rc}_{12}, \ldots, \mathbf{rc}_{1n}$, except the inputs $\mathbf{r}_{121}, \mathbf{r}_{131}, \ldots, \mathbf{r}_{1s1}$ which it provided as input into the computation of $\mathbf{rc}_1$. To bound the likelyhood that the adversary returns valid codes we recall how the verification codes for voter 1 are computed (Figure 24).

Unpacking the definitions of $\mathsf{GenPoints}$ and $\mathsf{GenSecretVoterData}$ we have that the k'th return code of voter 1 is calculated as:

$$\mathbf{rc}_{1k} = \mathsf{Truncate}(\mathsf{H}_L(x_{11k}, y_{11k}), L_R)) \oplus \bigoplus_{j=2}^{s} \mathbf{r}_{1jk}$$

for points $(x_{111}, y_{111}), \ldots, (x_{11n}, y_{11n})$ sampled according to $\mathsf{GenPoints}$ and $\mathbf{r}_{12}, \ldots, \mathbf{r}_{1s}$ provided by the adversary. Points (x_{11k}, y_{11k}) are obtained by first sampling a polynomial $\mathbf{p}_{11} \in \mathbb{Z}_{p'}[X]$ of degree $k'_1 - 1$ (where $k'_1 = \mathbf{E}[1] \cdot \mathbf{k}$, i.e. the dot-product of the first row in $\mathbf{E}$ with the vector $\mathbf{k}$ which define how many valid choices there are for each of the t elections.) Then, each x_{11k} is sampled uniformly at random from $\mathbb{Z}_{p'}$ and y_{11k} is computed as $\mathbf{p}_{11}(x_{11k})$.

Since $\mathsf{acheck}_1 = false$, the authority does not respond to the OT request on behalf of user 1

so the adversary has no information about x_{11k}, y_{11k}. It follows that the probability that the adversary outputs even one valid return code $rc_j^* \in \mathbf{rc}_1$ (we abuse notation to signify that rc_j^* occurs in $\mathbf{rc}_1$) is upperbounded (using a union bound) by:

$$\begin{aligned}\mathbf{Pr}[rc_j^* \in \mathbf{rc}_1] &\leq \mathbf{Pr}[rc_j^* = \mathbf{rc}_{11} \vee rc_j^* = \mathbf{rc}_{12} \vee \ldots \vee rc_j^* = \mathbf{rc}_{1n}] \\ &\leq \sum_{i=1}^{n} \mathbf{Pr}[rc_j^* = \mathbf{rc}_{1i}] \\ &\leq n \cdot \max\left(\frac{1}{2^{8L_R}}, \frac{1}{p'}\right)\end{aligned}$$

the two different terms under the maximization correspond, respectively, to guessing the value $\mathsf{Truncate}(\mathsf{H}_L(x_{11k}, y_{11k}), L_R)$ or even one of the values $x_{1k}, y_{1k} \in \mathbb{Z}_{p'}$.

A tighter bound for $\mathbf{Pr}[\mathbf{bad}_1]$ can be obtained by bounding the probability that all of the $|\mathbf{s}|$ verification codes supplied by the adversary are valid (let alone match those corresponding to the user's selection). We get that

$$\begin{aligned}\mathbf{Pr}[(rc_1^*, rc_2^*, \ldots, rc_{|\mathbf{s}|}^*) \in \mathbf{rc}_1] &\leq \sum_{(i_1, i_2, \ldots, i_{|\mathbf{s}|}) \subseteq [n]} \mathbf{Pr}[rc_1^* = \mathbf{rc}_{1i_1}, rc_2^* = \mathbf{rc}_{1i_2}, \ldots, rc_{|\mathbf{s}|}^* = \mathbf{rc}_{1i_{|\mathbf{s}|}}] \\ &\leq \binom{n}{|\mathbf{s}|} \cdot |\mathbf{s}|! \cdot \max\left(\frac{1}{2^{8L_R}}, \frac{1}{p'}\right)^{|\mathbf{s}|}\end{aligned}$$

The right-hand side is obtained by an union bound over all distinct ordered subsets of size $|\mathbf{s}|$ of the probability that the codes selected by the adversary match the codes corresponding to the subset, in order.

We conclude that

$$\left|\mathbf{Adv}_{\mathcal{A}}^{\mathsf{iv\text{-}1}} - \mathbf{Adv}_{\mathcal{A}}^{\mathsf{iv}}\right| \leq \binom{n}{|\mathbf{s}|} \cdot |\mathbf{s}|! \cdot \max\left(\frac{1}{2^{8L_R}}, \frac{1}{p'}\right)^{|\mathbf{s}|}$$

Next, we argue that if ucheck_1 is true (that is the user has received the correct return codes) then the vote recorded in α is precisely $\mathbf{s}$. If this is not the case, then the adversary must have guessed (at least) one of the return codes. Technically, we modify the game by introducing event $\mathbf{bad}_2$ which is set to true if ucheck_1 is true yet $\mathbf{s} \neq \mathbf{s}^*$; the game returns 0 in this case. The resulting game, which we call $\mathbf{Exp}_{\mathcal{A}}^{\mathsf{iv\text{-}2}}$ is depicted in Figure 25.

The difference between the two games is that the game $\mathbf{Exp}^{\mathsf{iv\text{-}2}}$ aborts if the users' first check succeeds (i.e. the adversary returns the correct return codes) yet the vote that is cast does not correspond to the users' selection. There must therefore be an entry i^* in the voter's selection $\mathbf{s}$ which is not recorded in α and for which the adversary can somehow predict the corresponding code – so the adversary knows/learns one of the messages of the OT sender which he should not learn.

This idea helps upperbound the difference between the adversary's advantages in $\mathbf{Exp}^{\mathsf{iv\text{-}1}}$ and

$\mathbf{Exp}^{\mathsf{iv\text{-}2}}_{\mathcal{A},\Pi}(s,t,\mathbf{n},\mathbf{k},\mathbf{E},\mathbf{q})$
$\mathsf{acheck}_1 \leftarrow false, \mathsf{acheck}_2 \leftarrow false$
$\mathsf{ucheck}_1 \leftarrow false, \mathsf{ucheck}_2 \leftarrow false$
$\mathsf{Setup}_{\mathcal{A}}(s,t,\mathbf{n},\mathbf{k},\mathbf{E})$
$\mathbf{s} = (s_1, s_2, \ldots, s_{|\mathbf{s}|}) \leftarrow \mathcal{A}(\hat{\mathbf{d}}_1)$
$\alpha = (\mathbf{a}, \pi) \leftarrow \mathcal{A}(X_1)$
$\mathbf{s}^* \leftarrow \mathsf{Extract}(\alpha, pk)$
$\mathsf{acheck}_1 \leftarrow \mathsf{CheckBallot}(1, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}}, B_1)$
if $(\mathsf{acheck}_1 = true)$ then $(\beta_1, z) \leftarrow \mathsf{GenResponse}(1, \mathbf{a}, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P}_1)$
$(rc_1^*, rc_2^*, \ldots, rc_{|\mathbf{s}|}^*) \leftarrow \mathcal{A}(\beta_1)$
if $(\forall i \in [|\mathbf{s}|])\ rc_i^* = \mathbf{rc}_{1\mathbf{s}_i}$ then $\mathsf{ucheck}_1 \leftarrow true$; $\gamma \leftarrow \mathcal{A}(Y_1)$
else return 0
if $\mathsf{ucheck}_1 = true$ and $\mathbf{s}^* \neq \mathbf{s}$ then $\mathbf{bad}_2 \leftarrow true$; abort
if $\mathsf{ucheck}_1 = true$ and $\mathsf{acheck}_1 = false$ then $\mathbf{bad}_1 \leftarrow true$ abort
$\mathsf{acheck}_2 \leftarrow \mathsf{CheckConfirmation}(1, \gamma, \hat{\mathbf{y}}, B_1, C_1)$
if $(\mathsf{acheck}_1 = true)$ and $(\mathsf{acheck}_2 = true)$ then $\delta_1 \leftarrow \mathsf{GetFinalization}(1, \mathbf{P}_1, B_1)$
$FC^* \leftarrow \mathcal{A}(\delta_1)$
if $\mathsf{ucheck}_1 = true$ then $\mathsf{ucheck}_2 \leftarrow (FC_1 = FC^*)$
if $(\mathsf{ucheck}_2 = true)$ and $((\mathsf{acheck}_2 = false)$ or $(\mathbf{s}^* \neq \mathbf{s}))$ return 1
else return 0

Figure 25: Second game hop: the return codes are valid, yet the cast ballot does not contain the intended vote

$\mathbf{Exp}^{\mathsf{iv\text{-}2}}$ and is formalized in the following Lemma.

Lemma 7.2 *For any adversary $\mathcal{A}$ it holds that*

$$\left|\mathbf{Adv}^{\mathsf{iv\text{-}2}}_{\mathcal{A}} - \mathbf{Adv}^{\mathsf{iv\text{-}1}}_{\mathcal{A}}\right| \leq |\mathbf{s}| \cdot \max\left(\frac{1}{2^{8L_R}}, \frac{1}{p'}\right)$$

Proof. Since the output of the two experiments only differs if $\mathbf{bad}_2$ is set we have that

$$\left|\mathbf{Adv}^{\mathsf{iv\text{-}2}}_{\mathcal{A}} - \mathbf{Adv}^{\mathsf{iv\text{-}1}}_{\mathcal{A}}\right| \leq \mathbf{Pr}[\mathbf{bad}_2].$$

We construct an adversary $\mathcal{B}$ against the security of the underlying OT protocol in the sense of the game defined in Figure 22 such that

$$\mathbf{Pr}[\mathbf{bad}_2] \leq |\mathbf{s}| \cdot \mathbf{Adv}^{\mathsf{OT\text{-}IV}}_{\mathcal{B}}$$

and we conclude using the bound provided by Theorem 6.11.

Adversary $\mathcal{B}$ is defined in Figure 26. It emulates the environment of adversary $\mathcal{A}$ as used in $\mathbf{Exp}^{\mathsf{iv\text{-}2}}$. Specifically, adversary $\mathcal{B}$ runs (most of) the setup of the experiment for the experiment of $\mathcal{A}$. That is, it generates all of the data for election authority 1, except the data for voter 1. Instead, part of this data namely $(x_{11}, y_{11}, \hat{d_1})$ is provided by the experiment of $\mathcal{B}$. The rest of the data, namely the n points on a random polynomial in $\mathbb{Z}_{p'}[X]$ are held by the sender in the experiment of $\mathcal{B}$.

Adversary $\mathcal{B}$ receives a vote selection $\mathbf{s}$ for voter 1. Next, $\mathcal{B}$ provides $\mathcal{A}$ with a voting code X_1 (constructed essentially as in the experiment of $\mathcal{A}$) to which the adversary responds with a ballot $(\mathbf{a}, \pi)$. This is an OT request for the OT scheme which underlies CH-Vote.

Adversary $\mathcal{B}$ obtains in return some message (β, z) calculated using the OT response (which should, by the security of the OT protocol only reveal information about messages $\mathsf{Encode}(x_i, y_i)$ for the positions i encoded in the OT request (α, pk)). Adversary $\mathcal{A}$ returns a set of return codes intended for user 1. Adversary $\mathcal{B}$ selects one of these codes rc_{i^*} uniformly at random, and attempts a response to the OT security game. The details are in Figure 26.

$\mathcal{B}_1((x_{111}, y_{111}, y'_{11}, \hat{\mathbf{d}}_{11}))$
  $n \leftarrow \sum_{j=1}^{t} n_j$
  for $i = 2, \ldots, N_E$
    $k'_i \leftarrow \sum_{j=1}^{t} e_{ij} k_j$
    $(\mathbf{p}_{i1} = ((x_{i11}, y_{i11}), \ldots, (x_{i1n}, y_{i1n}), y'_{i1})), y'_i) \leftarrow \mathsf{GenPoints}(n, k'_i)$
    $d_{i1} = (x_{i1}, y_{i1}, F_{i1}, \mathbf{r}_{i1}) \leftarrow \mathsf{GenSecretVoterData}(\mathbf{p}_{i1})$
    $\hat{d}_{i1} = (\hat{x}_{i1}, \hat{y}_{i1}) \leftarrow \mathsf{GetPublicVoterData}(x_{i1}, y_{i1}, y'_{i1})$
  // The adversary impersonating all other authorities.
  for $i = 1, \ldots, N_E$
    for $j \leftarrow 2, \ldots, s$ **do**
      $d_{ij} = (x_{ij}, y_{ij}, F_{ij}, \mathbf{r}_{ij}) \leftarrow \mathcal{A}()$
      $(\hat{x}_{ij}, \hat{y}_{ij}) \leftarrow A()$
    for $k \leftarrow 1, \ldots, n$ **do**
      $RC_{ik} \leftarrow \oplus_{j=1}^{s} \mathbf{r}_{ijk}$
    $\mathbf{rc}_i \leftarrow (RC_{i1}, RC_{i2}, \ldots, RC_{in})$
    $VC_i = (X_i, Y_i, \mathbf{rc}_i, FC_i)$
  // Generation of the election public key
  $(sk_1, pk_1) \leftarrow \mathsf{GenKeyPair}()$
  $(pk_2, \ldots, pk_s) \leftarrow \mathcal{A}(\hat{\mathbf{d}}_1, \mathbf{VC}_{\ominus 1})$
  $pk \leftarrow \Pi_{i=1}^{s} pk_i$
  $\mathbf{s} = (s_1, s_2, \ldots, s_{|\mathbf{s}|}) \leftarrow \mathcal{A}(\hat{\mathbf{d}}_1)$
  $\alpha = (\mathbf{a}, \pi) \leftarrow \mathcal{A}(X_1)$
  $\mathsf{acheck}_1 \leftarrow \mathsf{CheckBallot}(1, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}}, B_1)$
  if $(\mathsf{acheck}_1 = true)$ then output $(\mathbf{a}, pk)$

$\mathcal{B}_2(\beta, z)$
  $(rc^*_1, rc^*_2, \ldots, rc^*_{|\mathbf{s}|}) \leftarrow \mathcal{A}(\beta)$
  $i^* \twoheadleftarrow [|\mathbf{s}|]$
  return $(i^*, rc^*_{i^*} \oplus \bigoplus_{j=2}^{s} \mathbf{r}_{1ji^*})$

Figure 26: Reduction for bounding $\mathbf{bad}_2$: adversary $(\mathcal{B}_1, \mathcal{B}_2)$ is the against the underlying OT. Its wining probability is proportional to the probability of event $\mathbf{bad}_2$

The simulation provided to the adversary $\mathcal{A}$ is indistinguishable from his own execution in $\mathbf{Exp}^{\mathsf{iv\text{-}2}}$ so event $\mathbf{bad}_2$ occurs with the same probability. We note that if event $\mathbf{bad}_2$ is triggered by $\mathcal{A}$ in the execution of $\mathbf{Exp}^{\mathsf{iv\text{-}2}}_{\mathcal{A}}$, then for the execution of $\mathcal{B}$ in $\mathbf{Exp}^{\mathsf{OT\text{-}IV}}_{\mathcal{B}}$ it holds that there exists some i_0 such that $i_0 \in \mathbf{s}$ and $i_0 \notin \mathsf{Extract}(\alpha, pk)$ and yet $rc^*_{i_0} = \oplus_{j=1}^{s} \mathbf{r}_{1ji_0}$, where $\mathbf{r}_{11i_0} =$

$\mathsf{Truncate}(\mathsf{Encode}(x_{i_0}, y_{i_0}), L_R)$.

It follows that, provided that $i^* = i_0$ (which happens with probability $\frac{1}{|\mathbf{s}|}$) the message returned by the adversary $M_{i^*} = rc^*_{i_0} \oplus^s_{j=2} \mathbf{r}_{1ji_0}$ is equal to $\mathbf{r}_{11i_0}$ and adversary $\mathcal{B}$ wins $\mathbf{Exp}^{\mathsf{OT\text{-}IV}}_{\mathcal{B}}$, that is:

$$\mathbf{Adv}^{\mathsf{OT\text{-}IV}}_{\mathcal{B}} \geq \frac{1}{|\mathbf{s}|} \cdot \Pr[\mathbf{bad}_2]$$

The desired bound follows.

Finally, in the last hop we show that if the second check by the user succeeds, then the second check by the authority also succeeds, or the adversary must have guessed the contribution of the election authority to the finalization code of user 1. Technically, we set the flag $\mathbf{bad}_3$ to *true* if the second user check succeeds but the second authority check fails. We call the resulting experiment $\mathbf{Exp}^{\mathsf{iv\text{-}3}}_{\mathcal{A},\Pi}$ and specify it in Figure 27.

$\mathbf{Exp}^{\mathsf{iv\text{-}3}}_{\mathcal{A},\Pi}(s, t, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{q})$
- $\mathsf{acheck}_1 \leftarrow false, \mathsf{acheck}_2 \leftarrow false$
- $\mathsf{ucheck}_1 \leftarrow false, \mathsf{ucheck}_2 \leftarrow false$
- $\mathsf{Setup}_{\mathcal{A}}(s, t, \mathbf{n}, \mathbf{k}, \mathbf{E})$
- $\mathbf{s} = (s_1, s_2, \ldots, s_{|\mathbf{s}|}) \leftarrow \mathcal{A}(\hat{\mathbf{d}}_1)$
- $\alpha = (\mathbf{a}, \pi) \leftarrow \mathcal{A}(X_1)$
- $\mathbf{s}^* \leftarrow \mathsf{Extract}(\alpha, pk)$
- $\mathsf{acheck}_1 \leftarrow \mathsf{CheckBallot}(1, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}}, B_1)$
- if $(\mathsf{acheck}_1 = true)$ then $(\beta_1, z) \leftarrow \mathsf{GenResponse}(1, \mathbf{a}, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P}_1)$
- $(rc^*_1, rc^*_2, \ldots, rc^*_{|\mathbf{s}|}) \leftarrow \mathcal{A}(\beta_1)$
- if $(\forall i \in [|\mathbf{s}|])\ rc^*_i = \mathbf{rc}_{1\mathbf{s}_i}$ then $\mathsf{ucheck}_1 \leftarrow true$; $\gamma \leftarrow \mathcal{A}(Y_1)$
  - else return 0
- if $\mathsf{ucheck}_1 = true$ and $\mathbf{s}^* \neq \mathbf{s}$ then $\mathbf{bad}_2 \leftarrow true$; abort
- if $\mathsf{ucheck}_1 = true$ and $\mathsf{acheck}_1 = false$ then $\mathbf{bad}_1 \leftarrow true$ abort
- $\mathsf{acheck}_2 \leftarrow \mathsf{CheckConfirmation}(1, \gamma, \hat{\mathbf{y}}, B_1, C_1)$
- if $(\mathsf{acheck}_1 = true)$ and $(\mathsf{acheck}_2 = true)$ then $\delta_1 \leftarrow \mathsf{GetFinalization}(1, \mathbf{P}_1, B_1)$
- $FC^* \leftarrow \mathcal{A}(\delta_1)$
- $\mathsf{ucheck}_2 \leftarrow (FC_1 = FC^*)$
- If $\mathsf{ucheck2} = true$ and $\mathsf{acheck}_2 = false$ then $\mathbf{bad}_3 \leftarrow true$; abort
- if $(\mathsf{ucheck}_2 = true)$ and $((\mathsf{acheck}_2 = false)$ or $(\mathbf{s}^* \neq \mathbf{s}))$ return 1
  - else return 0

Figure 27: Third game hop: the second check by the election authority fails, yet the adversary provides a valid finalization code

The next lemma bounds the difference between the adversary's advantage in these two games by the probability of guessing the finalization code.

Lemma 7.3 *For any adversary $\mathcal{A}$*

$$\left|\mathbf{Adv}^{\mathsf{iv\text{-}3}}_{\mathcal{A}} - \mathbf{Adv}^{\mathsf{iv\text{-}2}}_{\mathcal{A}}\right| \leq \max\left(\left(\frac{1}{p' - |\mathbf{s}|}\right)^{n-|\mathbf{s}|}, \frac{1}{2^{8L_F}}\right)$$

Proof. Since the two games differ only if $\mathbf{bad}_3$ is set to true we have that

$$\left|\mathbf{Adv}^{\mathsf{iv}\text{-}3}_{\mathcal{A}} - \mathbf{Adv}^{\mathsf{iv}\text{-}2}_{\mathcal{A}}\right| \leq \mathbf{Pr}[\mathbf{bad}_3]$$

To bound the probability on the right hand side we note that $\mathbf{bad}_3$ is set if $\mathsf{acheck} = \mathit{false}$ in which case the adversary does not obtain $\delta_1 = H(\mathbf{p_{11}})$ from the election authority. Since up to this point in the execution the adversary learns at most k out of the n points from $\mathbf{p}_{11}$ the remaining points are information theoretically hidden. We can then upperbound the probability of event $\mathbf{bad}_3$ by the probability that the adversary can guess the value of $FC_1 = \mathsf{Truncate}(H_L(\mathbf{p}_{11}), L_F) \oplus_{j=2}^{s} FC_{1j}$ which is the same as predicting the value of $\mathsf{Truncate}(\delta_1, L_R) = \mathsf{Truncate}(H_L(\mathbf{p}_{11}, L_F))$. Since the adversary can lean at most :

$$\mathbf{Pr}[\mathbf{bad}_3] \leq \max\left(\left(\frac{1}{p'-k}\right)^{n-k}, \frac{1}{2^{8L_F}}\right)$$

Finally, we note that $\mathbf{Adv}^{\mathsf{iv}\text{-}3} = 0$. Indeed, if $\mathsf{ucheck}_2 = \mathit{true}$ and $\mathsf{acheck}_2 = \mathit{false}$ then event $\mathbf{bad}_3$ is raised and the game aborts. If $\mathsf{ucheck}_2 = \mathit{true}$ and $\mathbf{s}^* \neq \mathbf{s}$ then (by Remark 5.1) $\mathsf{ucheck}_1 = \mathit{true}$ and event $\mathbf{bad}_2$ is raised and the game also aborts. We conclude, that $\mathbf{Adv}^{\mathsf{iv}\text{-}3} = 0$.

The desired bound on $\mathbf{Adv}^{\mathsf{iv}}_{\mathcal{A}}$ follows by triangle inequality.

7.1.1. Proof of Corollary 5.3

Theorem 5.2 bounds the advantage of any adversary $\mathcal{A}$ by

$$\mathbf{Adv}^{\mathsf{iv}}_{\mathcal{A},\mathsf{CH\text{-}Vote}}(s,t,\mathbf{n},\mathbf{k},\mathbf{E}) \leq \binom{n}{|\mathbf{s}|}\cdot\max\left(\frac{1}{2^{8L_R}}, \frac{1}{p'}\right)^{|\mathbf{s}|} + |\mathbf{s}|\cdot\max\left(\frac{1}{2^{8L_R}}, \frac{1}{p'}\right) + \max\left(\left(\frac{1}{p'-|\mathbf{s}|}\right)^{n-|\mathbf{s}|}, \frac{1}{2^{8L_F}}\right)$$

Using the bounds on L_R, L_F and $|\mathbf{s}|$ we get that $\frac{1}{2^{8L_R}}$ and $\frac{1}{2^{8L_F}}$ are the dominating factors under the maxima so we get that:

$$\mathbf{Adv}^{\mathsf{iv}}_{\mathcal{A}} \leq \binom{n}{|\mathbf{s}|} \cdot |\mathbf{s}|! \cdot \left(\frac{1-\epsilon}{n-1}\right)^{|\mathbf{s}|} + (n-1)\cdot\left(\frac{1-\epsilon}{n-1}\right) + (1-\epsilon)$$

We get the desired inequalities by bounding the first term in the above sum (we let $k = |\mathbf{s}|$):

$$\begin{aligned}
\binom{n}{k} \cdot k! \cdot \left(\frac{1-\epsilon}{n-1}\right)^{k} &= \frac{n!}{k!\cdot(n-k)!} \cdot k! \cdot \left(\frac{1-\epsilon}{n-1}\right)^{k} \\
&= (n-k+1)\cdot(n-k+2)\cdot\ldots\cdot n \cdot \left(\frac{1-\epsilon}{n-1}\right)^{k} \\
&= \frac{(n-k+1)}{n-1} \cdot \frac{(n-k+2)}{n-1} \cdot \ldots \frac{n-1}{n-1} \cdot \frac{n}{n-1} \cdot (1-\epsilon)^{k} \\
&\leq \frac{n}{n-1} \cdot (1-\epsilon)^{k}
\end{aligned}$$

Since $\frac{n}{n-1} \leq 2$ and $(1-\epsilon)^k \leq (1-\epsilon)$ the first bound follows.

The second bound follows since for $\epsilon \geq 2$ and $k \geq 2$ we have that $\frac{n}{n-1} \cdot (1-\epsilon)^k \leq (1-\epsilon)^{k-1} \leq (1-\epsilon)$.

7.2. Proof of BV — Theorem 5.6

The proof is based on Theorem 6.10 and proceeds by game hopping.

1. Since the adversary cannot win the game with a correct vote, we abort immediately after receiving α if the vote was correct.
2. The ballot verifiability game can be rewritten as a reduction to the OT-BV game in the appendices that handles the polynomial generation and OT response for voter i_0. This is just code rewriting so it does not change the adversary's success probability.
3. By Theorem 6.10 we switch y' for a random group element, incurring the aforementioned loss in the adversary's advantage.
4. We rewrite the game to cancel out the contributions of the other (dishonest) authorities.
5. We rewrite the game to split it into a game and a reduction.
6. We apply Theorem 12.6 to the ZK proof returned by the reduction to extract the witness. This lets us reduce to the discrete logarithm problem.

Steps 1 and 2

We give the reduction to OT-BV. After generating the electorate data, we overwrite that of voter i_0 with that from the game we are reducing to. We also abort the game early if the ballot is correct (Step 1) and use the game to generate the OT response. Since this no longer gives us the value z from the OT, we simply set it to $z = \perp$ for checking purposes. The algorithms CheckConfirmation and HasBallot do not require at this value.

$\mathbf{Exp}_A^{\text{bv-2}}(s, t, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{q})$
// setup
$(i_0, \widehat{\mathbf{D}}_{\ominus 1}) \leftarrow \mathcal{A}_1()$
$(\mathbf{D}_1, \widehat{\mathbf{D}}_1, \mathbf{P}, \mathbf{K}) \leftarrow \mathsf{GenElectorateData}(\mathbf{n}, \mathbf{k}, \mathbf{E})$
$(\mathbf{D}_{1,i_0}, \widehat{\mathbf{D}}_{1,i_0}) \leftarrow \text{OT-BV.}\mathsf{Prepare}(\mathbf{K}_{i_0}, \mathbf{n})$
$(\widehat{\mathbf{x}}^*, \widehat{\mathbf{y}}^*) \leftarrow \mathsf{GetPublicCredentials}(\widehat{\mathbf{D}})$
$(\mathbf{pk}_{\ominus 1}) \leftarrow \mathcal{A}_2(\mathbf{D}_1, \widehat{\mathbf{D}}_1)$
$(\mathbf{sk}_1, \mathbf{pk}_1) \leftarrow \mathsf{GenKeyPair}()$
$pk \leftarrow \mathsf{GetPublicKey}(\mathbf{pk})$
// Adversary chooses the first part of the ballot
$(i, \alpha) \leftarrow \mathcal{A}_3(\mathbf{pk}_1)$
if $\mathsf{Extract}(\alpha, pk) \neq \perp$ then return 0
if $i \neq i_0$ then return 0
Parse α as $(\widehat{x}, \mathbf{a}, \pi)$
$\mathsf{acheck}_1 \leftarrow \mathsf{CheckBallot}(i, \alpha, pk, \mathbf{K}, \mathbf{E}, \widehat{\mathbf{x}}^*, \langle\,\rangle)$

If $\mathsf{acheck}_1 = 0$ then return 0
$\beta \leftarrow \mathsf{OT\text{-}BV.Respond}(\mathbf{a}, pk)$
// Adversary gets the response and produces the second part
$(i', \gamma) \leftarrow \mathcal{A}_3(\beta)$
Parse γ as $(\widehat{y}, \pi')$
$\mathsf{acheck}_2 \leftarrow \mathsf{CheckConfirmation}(i', \gamma, \widehat{\mathbf{y}}^*, \langle (i, \alpha, \perp) \rangle, \langle\,\rangle)$
If $\mathsf{acheck}_2 = 0$ then return 0
// Finalisation
$\mathbf{v} \leftarrow \mathsf{E}(\alpha, pk)$
$vote \leftarrow \mathsf{Extract}(\mathbf{v}, \mathbf{K}_{i_0}, \mathbf{n}, \mathbf{q})$
If $vote = \perp$ then return 1 else return 0

Lemma 7.4 *For all adversaries $\mathcal{A}$ we have $\mathbf{Adv}^{\mathsf{bv}}_{\mathcal{A}} = \mathbf{Adv}^{\mathsf{bv\text{-}2}}_{\mathcal{A}}$.*

Proof. By inspecting the definitions of the algorithms involved, one can see that the $\mathbf{Exp}^{\mathsf{bv\text{-}2}}$ game behaves identically to the $\mathbf{Exp}^{\mathsf{bv}}$ one as long as it does not return early (in the modified line). But in an execution which returns early, the adversary would not win the game if we omitted this line. q.e.d.

Step 3

We switch OT-BV for OT-SIM using Theorem 6.10. Call the new game $\mathbf{Exp}^{\mathsf{bv\text{-}3}}$. The cited theorem immediately gives us the following result.

Lemma 7.5 *For all adversaries $\mathcal{A}$ that make up to ν random oracle queries we have*

$$\mathbf{Adv}^{\mathsf{bv\text{-}3}}_{\mathcal{A}} \geq \mathbf{Adv}^{\mathsf{bv\text{-}2}}_{\mathcal{A}} \left(1 - \nu \frac{(n+1)}{\widehat{q}}\right) \left(1 - \frac{n}{\widehat{q} - n + 1}\right)$$

The following equation is an immediate consequence of this lemma:

$$\mathbf{Adv}^{\mathsf{bv\text{-}2}}_{\mathcal{A}} \leq \mathbf{Adv}^{\mathsf{bv\text{-}3}} + \nu \frac{(n+1)}{\hat{q}} + \frac{n}{\hat{q} - n + 1}$$

> ***Note.*** *We have proposed to set $p' = \widehat{q}$. In this case, the $\widehat{y}$-elements of $\widehat{\mathbf{D}}$ were all uniform in $\mathbb{G}_{\widehat{q}}$ previously and the following step is redundant.*
>
> *Unfortunately, the following step does not lend itself nicely to a concrete security analysis (which is why we have recommended setting $p' = \widehat{q}$. In our concrete security analysis, we therefore let $\mathbf{Adv}^{\mathsf{bv}-3^*}_{\mathcal{A}}$ be the advantage of the adversary after this step is performed.*

We pick the value y' uniformly at random from $\mathbb{Z}_{\widehat{q}}$ instead of from $\mathbb{Z}_{p'}$. The result is that the value $\widehat{y}_{I,i_0}$ is now a uniformly random element of $\mathbb{G}_{\widehat{q}}$ which we need for our next step.

Formally, switch OT-SIM for OT-RED together with a DLOG challenger that creates a challenge as $\boxed{y' \twoheadleftarrow \mathbb{Z}_{p'};\ C \leftarrow \widehat{g}^{y'}}$. This does not change the adversary's advantage as the two systems are equivalent. Then, we switch the challenger for one that creates the challenge as $\boxed{y' \twoheadleftarrow \mathbb{Z}_{\widehat{q}};\ C \leftarrow \widehat{g}^{y'}}$. This is an example of subsystem switching.

The justification for this hop is based on a theorem of Boneh and Venkatesan [7] who show that retrieving just the most significant bits of a discrete logarithm reduces to finding the entire logarithm. However, their theorem uses a lattice basis reduction argument which makes it hard to compute the exact loss function for the reducton.

Step 4

Immediately before the line $(\widehat{\mathbf{x}}^*, \widehat{\mathbf{y}}^*) \leftarrow \mathsf{GetPublicCredentials}(\widehat{\mathbf{D}})$ we add the following code to cancel out the contributions of the other authorities to the voter's key pair. Recall that each entry e of the matrix $\widehat{\mathbf{D}}$ is a pair $(\widehat{x}, \widehat{y})$, the components of which we refer to as $e_{(1)}$ and $e_{(2)}$.

$$\widehat{\mathbf{D}}_{I,i_0} \leftarrow \left((\widehat{\mathbf{D}}_{I,i_0})_{(1)},\ (\widehat{\mathbf{D}}_{I,i_0})_{(2)} / \prod_{i \neq I} (\widehat{\mathbf{D}}_{i,i_0})_{(2)} \right)$$

This changes the I-th row of our local copy of the matrix $\widehat{\mathbf{D}}$, which we return to the adversary in stage $\mathcal{A}_2$. The result is that the discrete logarithm of $\widehat{\mathbf{y}}^*$ is $y + y'$ for the values y, y' that our authority I chose.

Lemma 7.6 *Call the result of the changes in this step* $\mathbf{Exp}^{\mathsf{bv\text{-}4}}$. *For all adversaries* $\mathcal{A}$ *we have* $\mathbf{Adv}_{\mathcal{A}}^{\mathsf{bv\text{-}3}^*} = \mathbf{Adv}_{\mathcal{A}}^{\mathsf{bv\text{-}4}}$.

Proof. Our entry of $\widehat{\mathbf{D}}$ for voter I is still uniformly random in $\mathbb{G}_{\widehat{q}}$ as y' is uniform in $\mathbb{Z}_{\widehat{q}}$. Therefore, this change is undetectable and does not change the adversary's winning probability. q.e.d.

> ***Note.*** *We have proposed to strengthen the* CH-Vote *protocol by adding ZK-PoKs to the matrix* $\widehat{\mathbf{D}}$. *In this case, we simulate the ZK-PoKs associated with authority I here which the game can do as it is in control of the random oracle. Replacing a real ZK-PoK with a simulated one is perfectly indistinguishable to the adversary by the ZK property.*

Step 5

We rewrite our game to delegate most of the work to a reduction $\mathcal{R}$ as described in Figure 28. This introduces a new abstraction boundary, where the discrete logarithm y' of the value C is hidden from the reduction. In order to get the reduction to DLOG in the next step to work, we also pass the voter secret key y_{I,i_0} as well as the ballot component α and the public key pk back to the game across this abstraction boundary.

Since we have only moved code around but not changed the semantics, the adversary's advantage remains unchanged. The only point that may need some explanation is that we have moved the

$\underline{\mathbf{Exp}_A^{\mathsf{bv\text{-}5}}(s,t,\mathbf{n},\mathbf{k},\mathbf{E},\mathbf{q})}$
$y' \twoheadleftarrow \mathbb{Z}_{\widehat{q}};\ C \leftarrow \widehat{g}^{y'}$
$(\widehat{y},\pi',y,\alpha,pk) \leftarrow \mathcal{R}(\mathcal{A},C)$ // if this returns 0, so do we
if $\mathsf{Extract}(\alpha,pk) \neq \perp$ then return 0
Return 1

$\underline{\mathcal{R}(\mathcal{A},C)}$
// setup
$(1,i_0,\widehat{\mathbf{D}}_{\ominus 1}) \leftarrow \mathcal{A}_1()$
$(\mathbf{D}_1,\widehat{\mathbf{D}}_1,\mathbf{P},\mathbf{K}) \leftarrow \mathsf{GenElectorateData}(\mathbf{n},\mathbf{k},\mathbf{E})$
$(\mathbf{D}_{1,i_0},\widehat{\mathbf{D}}_{1,i_0}) \leftarrow \mathsf{OT\text{-}RED.Prepare}(\mathbf{K}_{i_0},\mathbf{n},C)$
$\widehat{\mathbf{D}}_{1,i_0} \leftarrow \left((\widehat{\mathbf{D}}_{1,i_0})_{(1)},\ (\widehat{\mathbf{D}}_{1,i_0})_{(2)}/\prod_{i\neq 1}(\widehat{\mathbf{D}}_{i,i_0})_{(2)}\right)$
$(\widehat{\mathbf{x}}^*,\widehat{\mathbf{y}}^*) \leftarrow \mathsf{GetPublicCredentials}(\widehat{\mathbf{D}})$
$(\mathbf{pk}_{\ominus 1}) \leftarrow \mathcal{A}_2(\mathbf{D}_1,\widehat{\mathbf{D}}_1)$
$(\mathbf{sk}_1,\mathbf{pk}_1) \leftarrow \mathsf{GenKeyPair}()$
$pk \leftarrow \mathsf{GetPublicKey}(\mathbf{pk})$

// Adversary chooses the first part of the ballot
$(i,\alpha) \leftarrow \mathcal{A}_3(\mathbf{pk}_1)$
if $i \neq i_0$ then return 0
Parse α as $(\widehat{x},\mathbf{a},\pi)$
$\mathsf{acheck}_1 \leftarrow \mathsf{CheckBallot}(i,\alpha,pk,\mathbf{K},\mathbf{E},\widehat{\mathbf{x}}^*,\langle\,\rangle)$
If $\mathsf{acheck}_1 = 0$ then return 0
$\beta \leftarrow \mathsf{OT\text{-}RED.Respond}(\mathbf{a},pk)$

// Adversary gets the response and produces the second part
$(i',\gamma) \leftarrow \mathcal{A}_4(\beta)$
Parse γ as $(\widehat{y},\pi')$
$\mathsf{acheck}_2 \leftarrow \mathsf{CheckConfirmation}(i',\gamma,\widehat{\mathbf{y}}^*,\langle(i,\alpha,\perp)\rangle,\langle\,\rangle)$
If $\mathsf{acheck}_2 = 0$ then return 0
Parse $\mathbf{D}_{I,i_0}$ as $(x_{I,i_0},y_{I,i_0},R_{I,i_0},F_{I,i_0})$
return $(\widehat{y},\pi',y_{I,i_0},\alpha,pk)$

Figure 28: The game and reduction for ballot verifiability, step 5.

extractor check to the game, so it executes after the rest of the reduction. However, since the extractor is deterministic, the moment α and pk are determined so is the result of the extraction, so it does not matter if we do this check at a later time. The reason for this step is that the reduction $\mathcal{R}$ is now an efficient algorithm again. We conclude that $\mathbf{Adv}_{\mathcal{A}}^{\mathsf{bv\text{-}5}} = \mathbf{Adv}_{\mathcal{A}}^{\mathsf{bv\text{-}4}}$.

Step 6

The adversary produces a Schnorr ZK-PoK π' in γ. We use Theorem 12.6 to extract the witness $\tilde{y}$ such that $\widehat{g}^{\tilde{y}} = \widehat{y}$. By applying the theorem to the reduction $\mathcal{R}$ we get the following result.

Lemma 7.7 *Let $E_{\mathcal{A},\mathcal{R}}$ be the event that extraction from reduction $\mathcal{R}$ succeeds for a given adversary $\mathcal{A}$. If $\mathcal{A}$ makes up to ν random oracle queries then*

$$\mathbf{Adv}_{\mathcal{A}}^{\mathsf{bv\text{-}5}} \leq \sqrt{\nu \cdot \mathbf{Pr}[E_{\mathcal{A},\mathcal{R}}]} + \frac{\nu}{2^{\tau}}$$

where τ is the length (resp. entropy) of the challenge space in the PoK in γ.

If the adversary's advantage was non-negligible and the adversary makes at most a polynomially bounded number of random oracle queries then the probability of the extraction succeeding will still be non-negligible.

> ***Note.*** *The Simulation-Sound Extraction property (Theorem 12.6) implies that the Forking Lemma still works after the adversary has seen simulated proofs (which implies "programming" access to the random oracle), as long as the proof returned at the end is not one of the simulated proofs.*
>
> *We have made a recommendation to add PoKs to the authority public keys and to the values in $\widehat{\mathbf{D}}$. In this case, our experiment additionally simulates these proofs. Since the returned proof γ is on the statement $\widehat{y}$ and the simulated proofs are on statements generated with high entropy by the experiment, the probability of a collision here (that would let the adversary submit a simulated proof and still pass the confirmation check) is negligible, specifically $1/\widehat{q}$.*

We can now reduce to the discrete logarithm problem. Obtain a challenge C from the DLOG challenger and use it in place of the one that the previous experiment generated by itself. This does not change the distribution of C so the probability of successful forking remains unchanged. This gives us the following reduction $\mathcal{E}$ to the discrete logarithm problem:

Reduction $\mathcal{E}(s, t, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{q})$
obtain C from DLOG challenger
$(\widehat{y}, \pi', y, \alpha, pk) \leftarrow \mathcal{R}(\mathcal{A}, C)$
fork $\mathcal{R}$ to get witness $\tilde{y}$
return $\tilde{y} - y \pmod{\widehat{q}}$ to DLOG challenger

If we succeed in obtaining $\tilde{y} = y + y'$ through the Forking Lemma, then as we already know y, we can return $y' := \tilde{y} - y \pmod{\widehat{q}}$ as the required discrete logarithm. The reduction $\mathcal{E}$, unlike Game 5, does not invoke the (inefficient) extractor or abort if extraction succeeds. This does

not matter — the probability of the event that the adversary produces a valid proof *and* that α is not a correct ballot component cannot have decreased by omitting a check that can only *decrease* the adversary's advantage. This gives the following result.

Lemma 7.8 *For every adversary $\mathcal{A}$ we have $\mathbf{Adv}^{\mathsf{dlog}}_{\mathcal{E},\mathbb{G}_{\widehat{q}}} = \mathbf{Pr}[E_{\mathcal{A},\mathcal{R}}]$ where the left-hand side is the advantage of reduction $\mathcal{E}$ against the discrete logarithm problem.*

Proof of Theorem 5.6.

> ***Note.*** *We are assuming $p' = \widehat{q}$ here.*

By combining steps 1–6 we can upperbound the advantange of adversary $\mathcal{A}$ against ballot verifiability by:

$$\mathbf{Adv}^{\mathsf{bv}}_{\mathcal{A}} \leq \sqrt{\nu \cdot \mathbf{Adv}^{\mathsf{dlog}}_{\mathcal{E},\mathbf{G}_{\hat{q}}}} + \frac{\nu}{2^{\tau}} + \nu\frac{(n+1)}{\hat{q}} + \frac{n}{\hat{q}-n+1}$$

7.3. Proof of confirmed as intended – Theorem 5.7

We obtain the desired bound through two game hops. The first game hop is essentially the same as the second game hop for individual verifiability: we make the game abort if the first user check succeeds (i.e. the user receives the correct return codes) yet the vote that is recorded does not correspond to the voter's intention. For completeness, we detail the resulting game $\mathbf{Exp}^{\mathsf{el\text{-}ci\text{-}1}}_{\mathcal{A}}$ in Figure 29.

To bound $\mathbf{bad}_2$ in $\mathbf{Exp}^{\mathsf{el\text{-}ci\text{-}1}}_{\mathcal{A}}$ (and therefore bound the gap introduced by the first hop) we note that event $\mathbf{bad}_2$ has the same definition in $\mathbf{Exp}^{\mathsf{el\text{-}ci\text{-}1}}_{\mathcal{A}}$ as in $\mathbf{Exp}^{\mathsf{iv\text{-}2}}_{\mathcal{A}}$. Since in the latter game $\mathbf{bad}_2$ can be set before $\mathbf{bad}_1$ and otherwise the experiments are identical, it follows that the same reduction used to bound $\mathbf{Exp}^{\mathsf{el\text{-}ci\text{-}1}}_{\mathcal{A}}$ in $\mathbf{Exp}^{\mathsf{iv\text{-}2}}_{\mathcal{A}}$ can be used to bound it in $\mathbf{Exp}^{\mathsf{iv\text{-}2}}_{\mathcal{A}}$.

Using reduction $\mathcal{B}$ in Figure 26 we get the following Lemma.

$$\left|\mathbf{Adv}^{\mathsf{el\text{-}ci\text{-}1}}_{\mathcal{A}} - \mathbf{Adv}^{\mathsf{el\text{-}ci}}_{\mathcal{A}}\right| \leq |\mathbf{s}| \cdot \max\left(\frac{1}{2^{8L_R}}, \frac{1}{p'}\right)$$

In the next game hop, the game aborts if the first user check fails (i.e. the return codes are not valid) yet the second authority check succeeds (i.e. the authority receives a valid confirmation message). Intuitively, since the first check fails the voter will not pass the confirmation key to the adversary, so the adversary essentially forges a signature under the key encoded in the finalization code. The resulting game is in Figure 30.

The informal discussion above shows that we can bound the probability that $\mathbf{bad}_2$ occurs (and therefore the distance between the two experiments) via the hardness of taking discrete logarithms (in the group from which Y comes from).

Lemma 7.9 *For any adversary $\mathcal{A}$, if ν is the total number of queries to the random oracle in $\mathbf{Exp}^{\mathsf{el\text{-}ci\text{-}2}}_{\mathcal{A}}$ there exists an adversary $\mathcal{E}$ against the discrete logarithm problem in $\mathbf{G}_{\hat{q}}$ such that*

$\mathbf{Exp}_{\mathcal{A}}^{\mathsf{el\text{-}ci\text{-}1}}(s, t, \mathbf{n}, \mathbf{k}, \mathbf{E})$
$\mathsf{acheck}_1 \leftarrow false, \mathsf{acheck}_2 \leftarrow false$
$\mathsf{ucheck}_1 \leftarrow false, \mathsf{ucheck}_2 \leftarrow false$
$\mathsf{Setup}_{\mathcal{A}}(s, t, \mathbf{n}, \mathbf{k}, \mathbf{E})$
$\mathbf{s} = (s_1, s_2, \ldots, s_{|\mathbf{s}|}) \leftarrow \mathcal{A}(\hat{\mathbf{d}}_1)$
$\alpha = (\mathbf{a}, \pi) \leftarrow \mathcal{A}(X_1)$
$\mathbf{s}^* \leftarrow \mathsf{Extract}(\alpha, pk)$
$\mathsf{acheck}_1 \leftarrow \mathsf{CheckBallot}(1, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}}, B_1)$
if $(\mathsf{acheck}_1 = true)$ then $(\beta_1, z) \leftarrow \mathsf{GenResponse}(1, \mathbf{a}, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P}_1)$
$(rc_1^*, rc_2^*, \ldots, rc_{|\mathbf{s}|}^*) \leftarrow \mathcal{A}(\beta_1)$
if $(\forall i \in [|\mathbf{s}|])\ rc_i^* = \mathbf{rc}_{1\mathbf{s}_i}$ then $\mathsf{ucheck}_1 \leftarrow true$; $\gamma \leftarrow \mathcal{A}(Y_1)$
else $\gamma \leftarrow \mathcal{A}()$
$\mathsf{acheck}_2 \leftarrow \mathsf{CheckConfirmation}(1, \gamma, \hat{\mathbf{y}}, B_1, C_1)$
if $(\mathsf{acheck}_1 = true)$ and $(\mathsf{acheck}_2 = true)$ then $\delta_1 \leftarrow \mathsf{GetFinalization}(1, \mathbf{P}_1, B_1)$
$FC^* \leftarrow \mathcal{A}(\delta_1)$
if $\mathsf{ucheck}_1 = true$ then $\mathsf{ucheck}_2 \leftarrow (FC_1 = FC^*)$
if $\mathsf{ucheck}_1 = true$ and $\mathbf{s}^* \neq \mathbf{s}$ then $\mathbf{bad}_1 \leftarrow true$; abort
if $(\mathsf{acheck}_2 = true)$ and $(\mathbf{s}^* \neq \mathbf{s})$ return 1
else return 0

Figure 29: First game hop: the return codes are valid, yet the ballot does not contain the intended vote

$$\left|\mathbf{Adv}_{\mathcal{A}}^{\mathsf{el\text{-}ci\text{-}2}} - \mathbf{Adv}_{\mathcal{A}}^{\mathsf{el\text{-}ci\text{-}1}}\right| \leq \sqrt{\nu \cdot \mathbf{Adv}_{\mathcal{E},\mathbf{G}_{\hat{q}}}^{dlog}} + \frac{\nu}{2^{\tau}}$$

Proof. We bound event $\mathbf{bad}_2$ via an adversary for the Fiat-Shamir-Schnorr proof system for the exponentiation function $\phi : \mathbb{Z}_{\hat{q}} \rightarrow \mathbf{G}_{\hat{q}}$. If the event $\mathbf{bad}_2$ occurs, then the adversary is successful and forges a proof for a discrete logarithm challenge of our choosing. We then use Theorem 12.6 to bound the success of the adversary.

The adversary is in Figure 31. It receives a discrete logarithm challenge $Y^* \in \mathbf{G}_{\hat{q}}$. It simulates the honest election authority for producing the points on the polynomial which corresponds to user 1 and produces the authority's contribution to the voting key and confirmation key. It then expects to receive the (public) contribution of the malicious authorities. The public contribution to the confirmation key of user 1 by authority 1 is then changed in such a way that the public confirmation key of user 1 is Y^*. In the remaining of the execution, adversary $\mathcal{C}$ simulates for $\mathcal{A}$ the execution of the protocol up to the point where $\mathcal{A}$ returns the confirmation codes to the voter. If all of the confirmation codes are valid, then adversary $\mathcal{C}$ aborts. Otherwise (i.e. at least one code is valid), adversary $\mathcal{C}$ expects to receive from $\mathcal{A}$ a valid confirmation message.

Adversary $\mathcal{C}$ simulates for $\mathcal{A}$, perfectly, the experiment $\mathbf{Exp}_{\mathcal{A}}^{\mathsf{el\text{-}ci\text{-}2}}$. Event $\mathbf{bad}_2$ corresponds to the case when $\mathsf{ucheck}_1 = false$, i.e. the check $(\forall i \in [|\mathbf{s}|])\ rc_i^* = \mathbf{rc}_{1\mathbf{s}_i}$ fails, and the adversary needs to produce γ without receiving the confirmation credential, and yet the confirmation message γ is valid.

Since in the experiment y_{11} (and therefore $\hat{y}_{11}$) are generated outside the view of the adversary the substitution of $\hat{y}_{11}$ with $Y^* \left(\mathsf{Prod}_{j=2}^{s} \hat{y}_{1j}\right)^{-1}$ for randomly selected Y^* will result in identical

$\mathbf{Exp}^{\mathsf{el\text{-}ci\text{-}2}}_{\mathcal{A}}(s,t,\mathbf{n},\mathbf{k},\mathbf{E})$
$\mathsf{acheck}_1 \leftarrow false, \mathsf{acheck}_2 \leftarrow false$
$\mathsf{ucheck}_1 \leftarrow false, \mathsf{ucheck}_2 \leftarrow false$
$\mathsf{Setup}_{\mathcal{A}}(s,t,\mathbf{n},\mathbf{k},\mathbf{E})$
$\mathbf{s} = (s_1, s_2, \ldots, s_{|\mathbf{s}|}) \leftarrow \mathcal{A}(\hat{\mathbf{d}}_1)$
$\alpha = (\mathbf{a}, \pi) \leftarrow \mathcal{A}(X_1)$
$\mathbf{s}^* \leftarrow \mathsf{Extract}(\alpha, pk)$
$\mathsf{acheck}_1 \leftarrow \mathsf{CheckBallot}(1, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}}, B_1)$
if $(\mathsf{acheck}_1 = true)$ then $(\beta_1, z) \leftarrow \mathsf{GenResponse}(1, \mathbf{a}, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P}_1)$
$(rc_1^*, rc_2^*, \ldots, rc_{|\mathbf{s}|}^*) \leftarrow \mathcal{A}(\beta_1)$
if $(\forall i \in [|\mathbf{s}|])\ rc_i^* = \mathbf{rc}_{1\mathbf{s}_i}$ then $\mathsf{ucheck}_1 \leftarrow true$; $\gamma \leftarrow \mathcal{A}(Y_1)$
else $\gamma \leftarrow \mathcal{A}()$
$\mathsf{acheck}_2 \leftarrow \mathsf{CheckConfirmation}(1, \gamma, \hat{\mathbf{y}}, B_1, C_1)$
if $(\mathsf{acheck}_1 = true)$ and $(\mathsf{acheck}_2 = true)$ then $\delta_1 \leftarrow \mathsf{GetFinalization}(1, \mathbf{P}_1, B_1)$
If $\mathsf{acheck}_2 = true$ and $\mathsf{ucheck}_1 = false$ then $\mathbf{bad}_2 \leftarrow true$; abort
$FC^* \leftarrow \mathcal{A}(\delta_1)$
if $\mathsf{ucheck}_1 = true$ then $\mathsf{ucheck}_2 \leftarrow (FC_1 = FC^*)$
If $\mathsf{ucheck}_1 = true$ and $\mathbf{s}^* \neq \mathbf{s}$ then $\mathbf{bad}_2 \leftarrow true$; abort
If $(\mathsf{acheck}_2 = true)$ and $(\mathbf{s}^* \neq \mathbf{s})$ return 1
else return 0

Figure 30: The second game hop: the return codes are invalid, yet the confirmation code message is correct

views (as long as y_{11} is not revealed. This is precisely the case when $\mathbf{bad}_2$ is raised, since in this case the adversary does not obtain the confirmation key.

We conclude that if $\mathbf{bad}_2$ is *true* then the adversary will output a valid γ, i.e.$\gamma = (\hat{y}, \pi)$ is such that $\hat{y} = \mathsf{Prod}_{j=1}^{s} \hat{y}_{1j} = Y^*$ and π is a Fiat-Shamir-Schnorr proof of knowledge of discrete logarithm of Y^*. We can now apply Theorem 12.6 to conclude that for any adversary $\mathcal{A}$ in $\mathbf{Exp}^{\mathsf{el\text{-}ci\text{-}2}}_{\mathcal{A}}$ there exists an adversary $\mathcal{E}$ against discrete logarithm in $\mathbf{G}_{\hat{q}}$ such that

$$\mathbf{Adv}^{\mathsf{dlog}}_{\mathcal{E},\mathbf{G}_{\hat{q}}} \geq \frac{\mathbf{Pr}[\mathbf{bad}_2]^2}{\nu} - \frac{\mathbf{Pr}[\mathbf{bad}_2]}{\hat{q}}$$

Alternatively,

$$\mathbf{Pr}[\mathbf{bad}_2] \leq \frac{\nu}{\hat{q}} + \sqrt{\nu \cdot \mathbf{Adv}^{\mathsf{dlog}}_{\mathcal{E},\mathbf{G}_{\hat{q}}}}$$

Finally, we argue that $\mathbf{Adv}^{\mathsf{el\text{-}ci\text{-}2}}_{\mathcal{A}} = 0$. Indeed, if $\mathsf{acheck}_2 = true$ and $(\mathbf{s}^* \neq \mathbf{s})$, then either $\mathsf{ucheck}_1 = false$ and the game aborts immediately after acheck_2 is set, or $\mathsf{ucheck}_1 = true$ and the game aborts in the next to last line.

Reduction $\mathcal{C}(Y^*)$

$n \leftarrow \sum_{j=1}^{t} n_j$

for $i = 1, \ldots, N_E$

$k'_i \leftarrow \sum_{j=1}^{t} e_{ij} k_j$

$(\mathbf{p}_{i1} = ((x_{i11}, y_{i11}), \ldots, (x_{i1n}, y_{i1n}), y'_{i1})), y'_i) \leftarrow \mathsf{GenPoints}(n, k'_i)$

$d_{i1} = (x_{i1}, y_{i1}, F_{i1}, \mathbf{r}_{i1}) \leftarrow \mathsf{GenSecretVoterData}(\mathbf{p}_{i1})$

$\hat{d}_{i1} = (\hat{x}_{i1}, \hat{y}_{i1}) \leftarrow \mathsf{GetPublicVoterData}(x_{i1}, y_{i1}, y'_{i1})$

// The adversary impersonating all other authorities.

for $i = 1, \ldots, N_E$

for $j \leftarrow 2, \ldots, s$ **do**

$d_{ij} = (x_{ij}, y_{ij}, F_{ij}, \mathbf{r}_{ij}) \leftarrow \mathcal{A}()$

$(\hat{x}_{ij}, \hat{y}_{ij}) \leftarrow A()$

// We substitute the public confirmation key for user 1 with the dlog challenge

$\hat{d}_{11} \leftarrow \left(\hat{x}_{11}, Y^* \left(\mathsf{Prod}_{j=2}^{s} \hat{y}_{1j}\right)^{-1}\right)$

// Generation of voter 1 card by printing authority

$X_1 \leftarrow \sum_{j=1}^{s} x_{1j}$

// Generation of the election public key

$(sk_1, pk_1) \leftarrow \mathsf{GenKeyPair}(\hat{\mathbf{d}}_1)$

$(pk_2, \ldots, pk_s) \leftarrow \mathcal{A}()$

$pk \leftarrow \Pi_{i=1}^{s} pk_i$

$\mathbf{s} = (s_1, s_2, \ldots, s_{|\mathbf{s}|}) \leftarrow \mathcal{A}(\hat{\mathbf{d}}_1)$

$\alpha = (\mathbf{a}, \pi) \leftarrow \mathcal{A}(X_1)$

$\mathsf{acheck}_1 \leftarrow \mathsf{CheckBallot}(1, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}}, B_1)$

if $(\mathsf{acheck}_1 = true)$ then $(\beta, z) \leftarrow \mathsf{GenResponse}(1, \mathbf{a}, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P}_1)$

$(rc^*_1, rc^*_2, \ldots, rc^*_{|\mathbf{s}|}) \leftarrow \mathcal{A}(\beta_1)$

if $(\forall i \in [|\mathbf{s}|])\ rc^*_i = \mathbf{rc}_{1\mathbf{s}_i}$ then abort

else $\gamma \leftarrow \mathcal{A}()$

Figure 31: Reduction for bounding $\mathbf{bad}_2$ in $\mathbf{Exp}_{\mathcal{A}}^{\mathsf{el\text{-}ci\text{-}2}}$

7.4. Proof of UV — Theorem 5.11

We begin by analysing the statements of the shuffle and decryption proofs. Let $\mathbf{e}'_0$ be the vector of valid ballots in BB_2 (this is computed by the tallying authorities, but not posted back to the board). Let $N := |\mathbf{e}_0|$ be the number of ballots on the board that should be tallied.

- The j-th shuffle proof (for $j \in [s]$) states that there exists a permutation ϕ_j on $[|\mathbf{e}'_{j-1}|]$ such that for all $i \leq |\mathbf{e}'_{j-1}|$ we have $\mathsf{Extract}((\mathbf{e}'_j)_i, pk) = \mathsf{Extract}((\mathbf{e}'_{j-1})_{\phi_j(i)}, pk)$.
- The j-th decryption proof states that for all $i \leq N$, we have $\mathbf{b}'_{i,j} = (((\mathbf{e}'_s)_i)_2)^{sk_j}$ where sk_j is the value satisfying $g^{sk_j} = pk_j$.

If these statements are correct, then we can infer that $\mathbf{e}'_s$ is a shuffle of $\mathbf{e}'_0$ in the sense that there exists a permutation ϕ such that for all $i < N$ we have $\mathsf{Extract}((\mathbf{e}'_s)_i, pk) = \mathsf{Extract}((\mathbf{e}'_0)_{\phi(i)}, pk)$. Further, since $pk = \prod_{j=1}^{s} pk_j$ we conclude that for $b^*_i = \prod_{j=1}^{s} \mathbf{b}'_{i,j}$ we have $\mathsf{Extract}((\mathbf{e}'_s)_i, pk) = (((\mathbf{e}'_s)_i)_1)/b^*_j$. It follows that $\mathsf{GetDecryptions}(\mathbf{e}'_s, \mathbf{b}')$ returns the same result, up to a permutation, as decrypting $\mathbf{e}'_0$ with sk directly since the decryption of $(a, b) = (pk^r \cdot m, g^r)$ is $m = a/b^{sk}$.

If these statements are correct, then the decryption of $\mathbf{e}'_s$ with the partial decryption factors in $\mathbf{b}'$ will yield a permutation of the same votes $\mathbf{m}$ as would decrypting $\mathbf{e}'_0$ directly. Calling $\mathsf{PublicCheckResult}$ will only succeed if the claimed result $(\mathbf{V}, \mathbf{W})$ is the one that you would get by following this process correctly. In other words, if the statements in the ZK proofs are correct, then $\mathsf{PublicCheck}$ suceeding means that the result is correct too.

Both $\mathsf{PublicCheck}$ and $\mathsf{CorrectTally}$ compute the vector $\mathbf{e}'_0$ (resp. $\mathbf{e}'$) identically as a function of the board BB. $\mathsf{CorrectTally}$ then proceeds to extract from $\mathbf{e}'$ directly, skipping the shuffles and partial decryptions on the board. It follows that, if $\mathsf{PublicCheck}$ succeeded but $\mathsf{CorrectTally}$ produced a different result (or returned $\perp$) then at least one of the statements in one of the ZK proofs of shuffling or decryption must be incorrect.

There are $2s$ of these Fiat-Shamir-Schnorr style proofs, two per authority — one for the shuffle and one for the partial decryption. For each of these proofs, by Corollary 12.5, an adversary making up to ν random oracle queries has a probability of at most $\nu/2^\tau$ of making a proof on an incorrect statement, where τ is the length (more precisely, entropy) in bits of the challenge space.

Consider the following reduction: guess one of the $2s$ proofs at random that you think the adversary will attack, e.g. attempt a proof on an invalid statement. If the adversary had probability α of winning the UV game then the adversary has at least $\alpha/(2s)$ probability of creating a proof on an invalid statement for the particular proof that we have chosen. We know this probability is bounded by $\nu/2^\tau$ so we must have $\alpha \leq 2s \cdot \nu/2^\tau$. q.e.d.

Part II.
Symbolic Analysis

M, N, $M_1, \ldots, M_k$ ::=	terms
$x \mid n \mid f(M_1, \ldots, M_k)$	where $x \in \mathcal{V}$, $n \in \mathcal{N}$, and $f \in \mathcal{C}$
D::=	expressions
$M \mid h(D_1, \ldots, D_k)$	where $h \in \mathcal{C} \cup \mathcal{D}$
ϕ::=	formula
$M = N \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi$	
P, Q::=	processes
0	nil
$\mathsf{out}(N, M); P$	output
$\mathsf{in}(N, x : T); P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\ a : T; P$	restriction
$\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	assignment
$\mathsf{if}\ \phi\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional
$\mathsf{event}(M); P$	event
$\mathsf{get}\ tbl(x_1 : T_1, \ldots, x_n : T_n)\ \mathsf{suchthat}\ D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	table lookup
$\mathsf{insert}\ tbl(M_1, \ldots, M_n); P$	table insertion

Figure 32: Syntax of the core language of ProVerif.

8. ProVerif framework

The symbolic analysis of the CH-Vote protocol has been performed using symbolic cryptography and can be verified automatically using a well-known state-of-the-art automated verification tool called ProVerif [5].

A detailed presentation of the syntax and semantics of ProVerif can be found in [6]. We give here an overview of the ProVerif model, focusing on the parts that are more relevant to our model. Namely, we provide the syntax and semantics of processes in ProVerif, as well as the definitions of correspondence and equivalence properties. Notations and definitions are mainly borrowed from [6].

Readers already familiar with ProVerif or who only wish an intuitive understanding of the symbolic model may skip this section.

8.1. Syntax

We assume a set $\mathcal{V}$ of variables, a set $\mathcal{N}$ of names, a set $\mathcal{T}$ of types. By default in ProVerif, types include *channel* for channel's names, and *bitstring* for bitstrings (also written *any*). The syntax for *terms*, *expressions*, and *processes* is displayed in Figure 32.

Terms and expressions. Symbols for functions are split into two sets of constructors $\mathcal{C}$ and destructors $\mathcal{D}$ respectively. Terms are built over names, variables and constructors and represent actual messages sent over the network, while expressions may also contain destructors representing cryptographic computations that extract or rebuild data. Function symbols are given with their types: $g(T_1, \ldots, T_n) : T$ means that the function g takes n arguments as input of types respectively $T_1, \ldots, T_n$ and returns a result of type T. A substitution is a mapping from variables to terms, denoted $\{U_1/x_1, \ldots, U_n/x_n\}$. The application of a substitution σ to a term U, denoted $U\sigma$, is obtained by replacing variables by the corresponding terms and is defined as usual. We only consider well-typed substitutions.

The evaluation of an expression is defined through rewrite rules. Specifically, each destructor d is associated with a rewrite rule of the form $d(U_1, \ldots, U_n) \to U$, over terms. Then the evaluation of an expression is recursively defined as follows:

- $g(D_1, \ldots, D_n)$ *evaluates* to U, which is denoted by $g(D_1, \ldots, D_n) \Downarrow U$, if $\forall i$, $D_i \Downarrow U_i$, and g is a constructor ($g \in \mathcal{C}$) and $U = g(U_1, \ldots, U_n)$; or g is a destructor ($g \in \mathcal{D}$) and there exists a substitution σ such that $U_i = U_i'\sigma$, $U = U'\sigma$, where $g(U_1', \ldots, U_n') \to U'$ is the rewrite rule associated to g.
- $g(D_1, \ldots, D_n)$ *evaluates* to fail, which is denoted by $g(D_1, \ldots, D_n) \Downarrow \mathsf{fail}$, otherwise.

The evaluation $[\![\phi]\!]$ of a formula ϕ is defined by $[\![M = N]\!] = \top$ if $M = N$ syntactically, or $[\![M = N]\!] = \bot$ otherwise, and is then extended to $\wedge, \vee, \neg$ as expected.

Processes. Figure 32 provides a convenient abstract language for describing protocols (formally modeled as processes). The output of a message M on channel N is represented by $\mathsf{out}(N, M); P$ while $\mathsf{in}(N, x : T); P$ represents an input on channel N, stored in variable x. Process $P \mid Q$ models the parallel composition of P and Q, while $!P$ represents P replicated an arbitrary number of times. $\mathsf{new}\ a : T; P$ generates a fresh name of type T and behaves like P. $\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$ evaluates D and behaves like P unless the evaluation fails, in which case it behaves like Q. The if case is similar. $\mathsf{event}(M); P$ is used to specify security property: the process emits an *event* (not observable by an attacker) to reflect that fact that it reaches some specific state, with some values, stored in M. Finally, ProVerif supports user defined tables declared by their name and the types of their elements, e.g. $tbl(T_1, \ldots, T_n)$. The process $\mathsf{insert}\ tbl(M_1, \ldots, M_n); P$ corresponds to the insertion in the table tbl of the entry $(M_1, \ldots, M_n)$. The process $\mathsf{get}\ tbl(x_1 : T_1, \ldots, x_n : T_n)\ \mathsf{suchthat}\ D\ \mathsf{in}\ P\ \mathsf{else}\ Q$ looks for an entry $(M_1, \ldots, M_n)$ in the table tbl such that $D\sigma$ evaluates to *true* with $\sigma = \{M_i/x_i\}_{i=1}^n$. If such an entry exists then $P\sigma$ is executed otherwise Q is executed.

The set of free names of a process P is denoted $fn(P)$, and the set of it's free variables by $fv(P)$. A *closed* process is a process with no free variables. Following ProVerif's notations, we may write $\mathsf{in}(c, =x).P$ instead of $\mathsf{in}(c, y : T).\mathsf{if}\ x = y\ \mathsf{then}\ P$, where T is the type of x.q Similarly, we may write $\mathsf{in}(c, (x : T, y : T')).P$ instead of $\mathsf{in}(c, z : any).\mathsf{let}\ x : T = proj1(z)\ \mathsf{in}\ \mathsf{let}\ y : T' = proj2(z)\ \mathsf{in}\ P$.

8.2. Semantics

A *configuration* $E, \mathcal{S}, \mathcal{P}$ is given by a multiset $\mathcal{P}$ of processes representing the current state of the processes, a set $E = (\mathcal{N}_{pub}, \mathcal{N}_{priv})$ representing respectively the public and private names used so far, and a set $\mathcal{S}$ of elements of the form $(tbl, M_1, \ldots, M_n)$ representing the entries of user

$$E,\mathcal{S},\mathcal{P} \cup \{0\} \to E,\mathcal{S},\mathcal{P}$$
$$E,\mathcal{S},\mathcal{P} \cup \{P \| Q\} \to E,\mathcal{S},\mathcal{P} \cup \{P, Q\}$$
$$E,\mathcal{S},\mathcal{P} \cup \{!P\} \to E,\mathcal{S},\mathcal{P} \cup \{P, !P\}$$
$$(\mathcal{N}_{pub}, \mathcal{N}_{priv}),\mathcal{S},\mathcal{P} \cup \{\mathsf{new}\ a : T; P\} \to (\mathcal{N}_{pub}, \mathcal{N}_{priv} \cup \{a'\}),\mathcal{S},\mathcal{P} \cup \{P[a'/a]\} \quad \text{where } a' \notin \mathcal{N}_{pub} \cup \mathcal{N}_{priv}$$
$$E,\mathcal{S},\mathcal{P} \cup \{\mathsf{out}(N, M); Q, \mathsf{in}(N, x); P\} \to E,\mathcal{S},\mathcal{P} \cup \{Q, P[M/x]\}$$
$$E,\mathcal{S},\mathcal{P} \cup \{\mathsf{let}\ x = D\ \mathsf{in}\ P\} \to E,\mathcal{S},\mathcal{P} \cup \{P[M/x]\} \quad \text{if } D \Downarrow M \text{ and } M \neq \mathsf{fail}$$
$$E,\mathcal{S},\mathcal{P} \cup \{\mathsf{if}\ \phi\ \mathsf{then}\ P\} \to E,\mathcal{S},\mathcal{P} \cup \{P\} \quad \text{if } [\![\phi]\!] = \top$$
$$E,\mathcal{S},\mathcal{P} \cup \{\mathsf{event}(M); P\} \to E,\mathcal{S},\mathcal{P} \cup \{P\}$$
$$E,\mathcal{S},\mathcal{P} \cup \{\mathsf{get}\ tbl(x_1 : T_1, \ldots, x_n : T_n)\ \mathsf{suchthat}\ D\ \mathsf{in}\ P\ \mathsf{else}\ Q\} \to E,\mathcal{S},\mathcal{P} \cup \{P\{M_i/x_i\}_{i=1}^n\} \quad \text{if } \exists (tbl, M_1, \ldots, M_n) \in \mathcal{S} \text{ such that } D\{M_i/x_i\}_{i=1}^n \Downarrow true$$
$$E,\mathcal{S},\mathcal{P} \cup \{\mathsf{get}\ tbl(x_1 : T_1, \ldots, x_n : T_n)\ \mathsf{suchthat}\ D\ \mathsf{in}\ P\ \mathsf{else}\ Q\} \to E,\mathcal{S},\mathcal{P} \cup \{Q\} \quad \text{if } \forall (tbl, M_1, \ldots, M_n) \in \mathcal{S},\ D\{M_i/x_i\}_{i=1}^n \not\Downarrow true$$
$$E,\mathcal{S},\mathcal{P} \cup \{\mathsf{insert}\ tbl(M_1, \ldots, M_n); P\} \to E,\mathcal{S} \cup \{(tbl, M_1, \ldots, M_n)\},\mathcal{P} \cup \{P\}$$

Figure 33: Transitions between configurations, without types for clarity

.

declared tables. The semantics of processes is defined through a reduction relation $\to$ between configuration, defined in Figure 33. A *trace* is a sequence of reductions between configurations $E_0, \mathcal{S}_0, \mathcal{P}_0 \to \cdots \to E_n, \mathcal{S}_n, \mathcal{P}_n$. We say that a trace $E_0, \mathcal{S}_0, \mathcal{P}_0 \to^* E', \mathcal{S}', \mathcal{P}'$ *executes* an event M if it contains a reduction $E, \mathcal{S}, \mathcal{P} \cup \{\mathsf{event}(M); P\} \to E, \mathcal{S}, \mathcal{P} \cup \{P\}$ for some $E, \mathcal{S}, \mathcal{P}, P$.

8.3. Properties

As usual, we assume that protocols are executed in an untrusted network, meaning that communications over a public network are fully controlled by an attacker who may eavesdrop, intercept, or send messages. This is easily modeled by executing a protocol $\mathcal{P}_0$ in parallel with an arbitrary process Q. Formally, we assume given a set of *public constructors*, subset of the constructors. An *adversarial process* w.r.t. to a set of names $\mathcal{N}_{pub}$ is a process Q such that $fn(Q) \subset \mathcal{N}_{pub}$ and Q uses only public constructors (and destructors). In what follows, all constructors are public, unless otherwise specified.

8.3.1. Correspondence

Many security properties can be stated as "if Alice reaches some state (e.g. finishes her session) then Bob must have engaged a conversation with her". This is for example the case of many variants of agreement properties [13]. In the context of voting, such correspondence properties can be used to express verifiability. For example, we may wish to state that whenever Alice thinks she has voted for v then there is indeed a ballot registered on her name that corresponds to v.

ProVerif allows to specify *correspondence* properties between events.

Definition 8.1 *A closed process P_0 satisfies the correspondence*

$$\mathsf{event}(M) \rightsquigarrow \bigwedge_{i=1}^{m} \bigvee_{j=1}^{l_i} \mathsf{event}(M_{ij})$$

where the $M_{i,j}$ do not contain names, if for any (adversarial) closed process Q such that $fn(Q) \subset fn(P_0)$, for any trace tr of $P_0 \mid Q$, for any substitution σ, if tr executes the event $M\sigma$, then for any i, there exists j such that tr executes event $M_{ij}\sigma'$.

8.3.2. Equivalence

Equivalence properties are crucial to express vote privacy. Observational equivalence of two processes P and Q models the fact that an adversary cannot distinguish between the two processes. Slightly more precisely, whenever P may emit on some channel c (interacting with an adversarial process R), then Q can emit on c as well. For readability, we summarize here the definition of equivalence from [6]. We write $\mathcal{C} \downarrow_N$ when a configuration $\mathcal{C} = E, \mathcal{P}$ with $E = (\mathcal{N}_{pub}, \mathcal{N}_{priv})$ can output on some channel N, i.e. if there exists $\mathsf{out}(N, M); P \in \mathcal{P}$ such that $fn(N) \in \mathcal{N}_{pub}$. Also, an adversarial context $C[\_]$ is a process of the form $\mathsf{new}\ n : any; \_ \mid Q$ where $fv(Q) = \emptyset$ and all functional symbols in Q are public, with $\_$ being a 'hole' expected to be filled by a configuration $\mathcal{C} = (\mathcal{N}_{pub}, \mathcal{N}_{priv}), \mathcal{P}$. Therefore, and assuming that $\mathcal{N}_{priv} \cap fn(Q) = \emptyset$, the application of one to the other is defined by :

$$\begin{aligned} C[\mathcal{C}] &= (\mathcal{N}'_{pub}, \mathcal{N}'_{priv}), \mathcal{P} \cup \{Q\} \\ \text{with } \mathcal{N}'_{pub} &= (\mathcal{N}_{pub} \cup fn(Q)) \backslash \{n\} \\ \text{and } \mathcal{N}'_{priv} &= \mathcal{N}_{priv} \cup \{n\} \end{aligned}$$

From this, the definition of observational equivalence follows :

Definition 8.2 *The* observational equivalence *between configurations, denoted by $\approx$, is the largest symmetric relation such that $\mathcal{C} \approx \mathcal{C}'$ implies :*

- *if $\mathcal{C} \downarrow_N$ then $\exists \mathcal{C}'_1$ s.t. $\mathcal{C}' \rightarrow^* \mathcal{C}'_1$ and $\mathcal{C}'_1 \downarrow_N$;*
- *if $\mathcal{C} \rightarrow \mathcal{C}_1$, then $\exists \mathcal{C}'_1$ s.t. $\mathcal{C}' \rightarrow^* \mathcal{C}'_1$ and $\mathcal{C}_1 \approx \mathcal{C}'_1$;*
- *$C[\mathcal{C}] \approx C[\mathcal{C}']$, for any adversarial context $C[\_]$.*

9. Symbolic analysis of the CH-Vote protocol

To symbolically analyse the CH-Vote protocol, we need a tool that can handle:

- complex ad-hoc equational theories, needed e.g. here to model the oblivious transfer used in CH-Vote;
- states: revote is forbidden and election authorities will answer to at most one request per voter;
- correspondence properties to model verifiability and possibly equivalence properties to model privacy in possible future extension of this study.

The (only) two reasonable candidates satisfying these requirements are ProVerif [5] and Tamarin [15]. It is difficult to predict in advance which one of the tools will be the best choice for a given protocol. We used ProVerif (as initially planned in our contract) since our group has more experience with this tool and since ProVerif has already been used to analyse a large voting protocol [10].

We present here the symbolic model of the CH-Vote protocol. This section is a companion report to the provided ProVerif models.

We detail the modeling of the primitives (Section 9.1), of the protocol itself (Section 9.2), and of the verifiability properties (Section 9.3).

Limitations. As it is well-known, symbolic proofs of cryptographic protocols deal with abstractions thereof, omitting numerous cryptographic and mathematical properties of the underlying primitives. Symbolic proofs are widely accepted as a good indication that the design of a cryptographic protocol is not flawed, and it is considered to be a good sanitization method for complex cryptographic protocols, such as e-voting protocols. *However, symbolic proofs do not cover actual implementations of the security protocols, and might overlook special attacks that make use of specialized properties of the cryptographic primitives.* Our analysis is not an exception to this rule.

Parameters. We consider an arbitrary number of voters and an arbitrary number of elections authorities. We also consider an arbitrary number of candidates but a fixed number k of selections made by the honest voter. We instantiate k to several (small) values. Note however that dishonest voters may vote for an arbitrary number of selections.

Files. The ProVerif model is split in two main files:

- `CHVote_Honest_Voters.pv` models the verifiability properties that correspond to the point of view of the (honest) voter.
- `CHVote_All_Voters.pv` models the verifiability properties that should hold for all voters, honest or not.

These files are parameterized by k and can be compiled into a ProVerif file by instantiating the parameter k (with the script `build`).

9.1. Primitives

The list of symbolic functions, together with their type is displayed in Figure 34.

Standard primitives. The CH-Vote protocol makes use of standard encryption (to secure communications between the election authorities and the printing authority) as well as standard signatures (used throughout the protocol to authenticate data) and hash. These primitives are modeled as usual. $\mathsf{GenSignature}(sk, m)$ denotes the signature of m with signing key sk while $\mathsf{GenCiphertext}(pk, m)$ denoted the (asymmetric) encryption of m with the public key pk. The corresponding reduction rules are the two standard ones

$$\begin{aligned} \mathsf{VerifySignature}(\mathsf{pkey}(sk), \mathsf{GenSignature}(sk, m), m) &\rightarrow ok \\ \mathsf{GetPlaintext}(sk, \mathsf{GenCiphertext}(\mathsf{pkey}(sk), m)) &\rightarrow m \end{aligned}$$

The term $\mathsf{h}(m)$ represents the hash of m. It has no corresponding reduction rule since a hash may not be inverted.

ElGamal encryption. The voting selections made by the voter are encrypted using ElGamal encryption. This is modeled similarly to standard encryption, together with an additional rule that reflects that, in case the attacker knows the randomness used to generate a ciphertext, he may recover the plaintext (by brute-force). This is due to the fact that candidates are encoded through relatively small integers that the adversary may enumerate. Formally, $\mathsf{enc}(pk, m, r)$ represents the ElGamal encryption of m with randomness r and public key pk. The corresponding decryption rule is:

$$\mathsf{dec}(\mathsf{enc}(\mathsf{pkey}(sk), m, r), sk) \rightarrow m.$$

Due to the form of an ElGamal encryption, an attacker may retrieve the plaintext m when he knows the randomness as m is small.

$$\mathsf{retrieve}(\mathsf{enc}(pk, m, r), pk, r) \rightarrow m.$$

Zero-Knowledge proof. The CH-Vote protocol embeds two zero-knowledge proofs. With GenBallotProof, the voter proves that he knows the voting code x corresponding to (left part of) her public voter credential $\hat{x}$. The GenBallotProof algorithm also includes a proof that the voter (her voting device) knows the underlying randomness and links the proof with the ballot. These parts of the proof are however unnecessary for the security properties considered in this analysis and are therefor enot modeled here. With GenConfirmationProof, the voter proves that he knows the secret key corresponds to (right part of) her public voter credential ($\hat{y}$). This is

Standard primitives

$\mathsf{GenSignature}(bitstring, bitstring) : bitstring$	signature
$\mathsf{GenCiphertext}(tPubKey, tElectorateData) : bitstring$	encryption
$\mathsf{h}(bitstring) : bitstring$	hash
$\mathsf{pkey}(bitstring) : tPubKey$	public key
$\mathsf{Sum}(bitstring, bitstring) : bitstring$	sum over two elements

ElGamal encryption

$\mathsf{enc}(tPubKey, bitstring, bitstring) : bitstring$	ElGamal encryption

Zero-Knowledge proof

$\mathsf{GenBallotProof}(bitstring, tPubKey) : bitstring$	ZKP of some voting credential
$\mathsf{GenConfirmationProof}(bitstring, tPubKey) : bitstring$	ZKP of some confirmation credential

Polynomials

$\mathcal{A}(bitstring, bitstring) : bitstring$	Polynomial w.r.t. E.A. + Voter
$\mathsf{Pos}(bitstring, bitstring, bitstring) : bitstring$	n-th evaluation point w.r.t. E.A. + Voter
$\mathsf{eval}(bitstring, bitstring) : bitstring$	Value of a Polynom at some evaluation point

Oblivious Transfer

$G(bitstring) : bitstring$	Selections to Prime numbers
$\mathsf{GenResponse}(bitstring, bitstring, tPubKey, bitstring) : bitstring$	Response to an OT request

Credentials

$x(bitstring, bitstring) : bitstring$	Voting code w.r.t. E.A. + Voter
$y(bitstring, bitstring) : bitstring$	Confirmation code w.r.t. E.A. + Voter
$\mathsf{PublicShare}(bitstring) : bitstring$	Public Share of some E.A.

Verification Codes

$\mathsf{GenRC}(bitstring, bitstring) : bitstring$	List of Return Codes w.r.t. E.A. + Voter
$\mathsf{GetFinalization}(bitstring, bitstring) : bitstring$	Finalization Code w.r.t. E.A. + Voter

Figure 34: List of functions used in the ProVerif model.

reflected by the two following (standard) rewrite rules.

$$\begin{aligned} \mathsf{CheckBallotProof}(\mathsf{GenBallotProof}(x, \mathsf{pkey}(x)), \mathsf{pkey}(x)) &\rightarrow \mathsf{ok}. \\ \mathsf{CheckConfirmationProof}(\mathsf{GenConfirmationProof}(x, \mathsf{pkey}(x)), \mathsf{pkey}(x)) &\rightarrow \mathsf{ok}. \end{aligned}$$

The two equations are identical but use different function names as they correspond to two distinct proofs.

Polynomials. One originality of the CH-Vote protocol is to associate to each voter i and each election authority (with electoral data ED), a polynomial $A(ED, i)$ of degree $k-1$ that the voter will be able to reconstruct only when he will have enough (that is k) distinct points. The evaluation of a polynomial A on a position x is denoted $\mathsf{eval}(A, x)$. More precisely, $\mathsf{eval}(A, x)$ represents the point $(x, A(x))$. The reconstruction of the polynomial A from any k points of the form $(c, A(c))$ is modeled by the rule:

$$\mathsf{RebuildPoly}(\mathsf{eval}(A, x_1), \ldots, \mathsf{eval}(A, x_k)) \rightarrow A. \qquad \text{if } x_i \neq x_j \ \forall i \neq j$$

This rule can be written in ProVerif thanks to the `otherwise` construction.

The parameter k represents the number of selections made by the voter. In our ProVerif model, we instantiate k (and the corresponding equations) to various (small) integers.

This equation is used in the model `CHVote_All_Voters.pv` only (and its variant). Indeed, in `CHVote_Honest_Voters.pv`, we directly provide the polynomial to the adversary. This over-approximation speeds up the analysis and is still sufficient for the properties considered in `CHVote_Honest_Voters.pv`.

Oblivious transfer. When an election authority (with electoral data ED) receives the encrypted selections $s_1, \ldots, s_k$ from a voter i, he obliviously transfers (through his response GenResponse) the points of the polynomial $A(ED, i)$ corresponding to the selections $s_1, \ldots, s_k$. The position associated to voter i for candidate s and electoral data ED is denoted $\mathsf{Pos}(ED, i, s)$.

The voter obtains all the points through the function GetPoints used on a response to his OT request plus the randomness he used for building the encryption of his selection. In our model the OT response is split cipher per cipher, with GenResponse providing each piece one individually, so that the response can easily adjust to any number of selections. Consequently, we define a GetPoint reduction to get each point from it's associated GenResponse individually. The protocol behaviour is captured by iterating GetPoint on a set of GenResponse.

$$\mathsf{GetPoint}(\mathsf{GenResponse}(i, \mathsf{enc}(pk, G(s), r), pk, ED), s, r) = \mathsf{eval}(A(ED, i), \mathsf{Pos}(ED, i, s))$$

To avoid a dependency in the parameter k (costly when k increases), we over-approximate the election authority's behavior. Here, the election authority generates an oblivious answer for each individual ciphertext and the voting device computes the return codes one by one, while in reality, this corresponds to a single algorithm, applied to a list of ciphers.

Setup. We also introduce some functions, together with some rewrite properties, to model the setup. These functions do not necessarily represent actual algorithms.

The election public key is obtained by combining the public keys of the election authorities, which is modeled by the following rule.

$$\mathsf{Prod}(\mathsf{pkey}(a), \mathsf{pkey}(b)) \quad = \quad \mathsf{pkey}(\mathsf{Sum}(a, b)).$$

Note that Prod and Sum are two abstract symbolic functions with no other rules (in particular no commutative nor associative properties). This is due to the fact that ProVerif do not support AC symbols.

Each election authority generates an important set of data, called the electoral data ED. The corresponding published part is denoted $\mathsf{PublicShare}(ED)$. For the electoral data ED of a given election authority, one can compute the (secret) voting code $x(ED, i)$ of a voter i *for* ED and her (secret) confirmation code $y(ED, i)$ *for* ED. Their corresponding public part can be read from the published part of ED. This is modeled by the following equation.

$$\mathsf{GetPublicVoterData}(\mathsf{PublicShare}(ED), i) = \\ (\mathsf{pkey}(x(ED, i)), \mathsf{pkey}(\mathsf{Sum}(y(ED, i), \mathsf{eval}(A(ED, i), c_0)))).$$

Note that the secret key corresponding to the right part of the public voter credential is not directly $y(ED, i)$ but $y(ED, i)$ combined with the voter's polynomial evaluated on some fixed position c_0. Note also that the true voting code (as written on the voting sheet of the voter) is actually obtained by combining the voting code generated by each election authority and similarly for the confirmation code.

A voter i is also provided with a finalization code $\mathsf{GetFinalization}(ED, i)$ and return codes, one for each possible candidate. In order to model an arbitrary number of possible candidates, we denote by $\mathsf{GenRC}(ED, i)$ the list of all possible return codes. To retrieve the return code associated to a particular candidate $c(t)$, one may simply look up in the list by applying the function GetRC, with the following associated rule:

$$\mathsf{GetRC}(\mathsf{GenRC}(ED, i), c(t)) \quad = \quad \mathsf{h}(\mathsf{eval}(A(ED, i), \mathsf{Pos}(ED, i, c(t)))).$$

The finalization code $\mathsf{GetFinalization}(ED, i)$ is in fact the exclusive or of the (hash) of all the points corresponding to $\mathsf{Pos}(ED, i, c(t))$ generated by ED for i. Since the points appear only in the oblivious transfer request, we make here the assumption that the only way for the attacker to compute $\mathsf{GetFinalization}(ED, i)$ by himself is to learn the hash of all the points corresponding to the $\mathsf{Pos}(ED, i, c(t))$. As the total number n of points *is strictly greater* than k, the number of selections made by the voter, this situation should not occur. Therefore, we abstract $\mathsf{GetFinalization}(ED, i)$ by an independent, secret value. To ensure that the attacker can indeed not get all the necessary points, we ask ProVerif to show that if the attacker gets $h(\mathsf{eval}(A(ED, i), \mathsf{Pos}(ED, i, s))$ then ED must have recorded a cipher corresponding to s. This guarantees that the attacker gets at most the (hash) of the points corresponding to the selections made by the voter. This property, called **Valid AbstractFC** is formalized in Section 9.3. Note that in case $k = n$: the voters may select possibly all the candidates then there is an attack against the cast as intended property, as explained in Section 10.

Voting card. Finally, we explain how we model the *voting card* that each voter receives and that

contains all the secret information provided to a voter. The voting card contains the following information:

- the voting code X obtained as the sum of voting codes $x(ED, i)$ computed by each election authority (with electoral data ED),
- the return codes, one for each candidate. A return code is obtained as the exclusive or of (the hash of) the polynom evaluation $\mathsf{GetRC}(\mathsf{GenRC}(ED, i), c(t))$ obtained from the electoral data ED of each election authority.
- the secret confirmation credential Y, obtained as the sum of the secret credentials generated by each election authority.
- the finalization code obtained as the exclusive or of the finalization codes computed by each election authority.

Since we assume all the election authorities but one (EA_1) to be dishonest, we can only trust the contribution computed by EA_1. Therefore, we model the voting card as a card containing only the data coming from EA_1 and the other contributions will be provided by the adversary. An alternative, more standard, model would be to input the adversarial contributions from the network and perform exclusive or with the honest ones. However, this would require to model both the exclusive or, which is an associative and commutative operator. Its modeling in ProVerif would have been (too) abstract, possibly losing attacks. This is why we designed a model without an explicit representation of the exclusive or.

Formally, the information written on the voting card is modelled by $\mathsf{GetVotingCard}(i, ED)$ where $\mathsf{GetVotingCard}$ is defined as follows.

$$\mathsf{GetVotingCard}(i, ED) = (i, x(ED, i), y(ED, i), \mathsf{GetFinalization}(ED, i), \mathsf{GenRC}(ED, i))$$

9.2. Protocol

We only need to model a (honest) voter and a (honest) election authority since the other roles are considered to be compromised in our security analysis, and are therefore under the control of the attacker.

Voter. The role of the voter is simple:

- she reads her voting card;
- she enters her voting code X;
- she checks whether the received return codes correspond to her intent;
- she enters her secret confirmation credential;
- she expects to receive her finalization code.

The corresponding process is displayed in Figure 35. For simplicity, we asume here that the voter has already read her voting card. In order to model an arbitrary number of candidates and to consider an arbitrary selection of candidates, the voter simply inputs her choices from the network. In other words, the attacker choses how the voter votes. Instead of following strictly the specification, we made two modifications.

```
let Voter(i, x1, y1, F1, RCseed1) =
!(
  in(pc, t_u); let s_u = c(t_u) in
  Intended(i, s_u);
  out(pc, i);
  out(pc, x1); out(pc, s_u);
  in(pc, = GetRC(RCseed1, s_u));
  Checked(i, s_u);
  out(pc, y1); in(pc, = F1);
  VoterHappy(i, s_u)
).
```

Figure 35: The Voter Process.

First recall that in our model, the voting card contains the contribution of the honest election authority instead of the full code. Since the voting device is under the control of the attacker, it is sufficient to provide the contribution of the honest election authority and then the attacker may recover the whole code by himself.

Second, to limit the dependency in the parameters, we consider a voter that outputs her (encrypted) voting selection one by one (instead of a single ballot containing k ciphers). Similarly, she checks the return codes one by one and outputs her secret confirmation credential as soon as one is correct. We therefore over-approximate the voter's behavior, making sure that we do capture all honest ones. This over-approximation will be partially tightened when writing the security properties, as we shall see in the next section.

Election authority. The election authority is in charge of generating the election data as explained in the description of the voting card (Section 9.1). During the voting phase, he proceeds as follows.

- When a voter submits an encrypted ballot, he checks that the ballot proof is valid. If this is the case and if the voter did not vote already, he generates the oblivious transfer response.
- When a voter submits a confirmation code, the election authority checks that the corresponding proof is valid. If this is the case and if the voter did not confirm already, he sends the finalization code.

To make sure that a voter does not vote twice, the election authority stores the ballots (and the corresponding voter) in a database B. Similarly, proofs of confirmation are stored in a database C. They are modeled as tables in ProVerif.

As for the voter and to limit the dependency in the parameters, we consider an election authority that will answer to the encrypted selection one by one (instead of a single ballot containing k ciphers). The ciphers are stored in a database Bc. We therefore over-approximate the election authority's behavior, making sure that we do capture all honest ones. Again, this over-approximation will be partially tightened when writing the security properties Finally, the authority signs his two answers. Since each answer is now plit into (arbitrary) many components, we over-approximate this last step by letting the adversary have access to a signature oracle.

```
let Election_Authority(U, ED) =
!(      (* Protocol 6.5 – Vote Casting *)
   in(pc, (i, (X̂, pi)));
   in(pc, X̂_all);
   let X̂_v = Prod(pkey(x(ED, i)), X̂_all) in
   if X̂ = X̂_v ∧ CheckBallotProof(pi, X̂) = ok then
   new st;
   get B(= i, alpha') in 0 else
   insert B(i, (X̂, pi));
   Recorded(i, (X̂, pi), st);
   !(
      in(pc, a_u);
      RecordedCipher(i, a_u);
      insert Bc(i, a_u);
      out(pc, (i, GenResponse(i, a_u, pk, ED)));
      !(in(pc, beta); out(pc, GenSignature(key_EA, (U, i, beta))))
   )
)|
!(      (* Protocol 6.6 – Vote Confirmation *)
   in(pc, (i, (Ŷ, pi')));
   in(pc, Ŷ_all);
   let Ŷ_v = Prod(pkey(Sum(y(ED, i), eval(A(ED, i), c0))), Ŷ_all) in
   if Ŷ = Ŷ_v ∧ CheckConfirmationProof(pi', Ŷ) = ok then
   get B(= i, alpha) in
   get C(= i, gamma') in 0 else
   insert C(i, (Ŷ, pi'));
   Confirmed(i);
   out(pc, (i, GetFinalization(i, ED)));
   !(in(pc, delta); out(pc, GenSignature(key_EA, (U, i, delta))))
).
```

Figure 36: The Election Authority's vote casting and vote confirmation process.

The corresponding process is displayed in Figure 36 (omitting the setup phase). The key key_EA is the private key signature key of this authority.

In the first step, the voter must prove that he knows x, the private key associated to the public key $\hat{x}$. This public key is made of the aggregation (the sum) of the public share computed by each election authority. This is modeled here as follows:

- the election authority inputs the contributions coming from all the other authorities $\hat{x}_{all}$;
- he computes the public key $\hat{x} = \mathsf{Prod}(x(ED, i), \hat{x}_{all})$, by multiplying with his own contribution.

Similarly for $\hat{y}$, the election authority inputs from the network the contributions from the other authorities.

9.3. Verifiability properties

Intuitively, we prove the following verifiability properties. For our analysis, it is important to notice that some properties are from the point of view a voter and therefore must hold for any honest voter while other properties must hold for any voter, honest or not. We categorize accordingly the properties with the keywords *honest* and *all*.

Individual verifiability At the end of the voting process, the voter is guaranteed that his vote has been cast and recorded as intended. *(honest)*

Universal verifiability We show the following sub-properties:

- *Eligibility* (*legitimate voters* and *ballot verifiability*) Only correct votes cast by eligible voters have been tallied. *(all)*
- *Confirmed as intended*: Every confirmed ballot corresponds to the voter's intent. *(honest)*
- *Uniqueness:* Eligible voters have voted at most once. *(all)*

Events in ProVerif We express these properties in ProVerif by adding several events in the description of the processes, that correspond to different steps of the protocol.

- $\mathsf{Intended}(id, s)$: voter id starts the voting process, intending to vote for some candidate s.
- $\mathsf{Checked}(id, s)$: voter id has checked that the return code corresponds to her intended candidate s.
- $\mathsf{VoterHappy}(id, s)$: voter id has completed her voting process and believes she has voted for her intended candidate s.
- $\mathsf{Recorded}(id, b, u)$: the honest election authority has received a (whole) ballot b from voter id at some time u.
- $\mathsf{RecordedCipher}(id, a)$: the honest election authority has received a (valid) cipher a from voter id.
- $\mathsf{Confirmed}(id)$: the honest election authority has a received (valid) confirmation proof from voter id. This means that the ciphers a_i recorded in the name of id are now ready to be tallied.

9.3.1. Verifiability for all voters

We express *Eligibility* as follows in terms of two sub-properties, namely *legitimate voters* and *ballot verifiability*.

$$\mathsf{Recorded}(i, b, u) \rightsquigarrow (i = id) \vee (i = \mathsf{dishonest}(i')) \qquad \textbf{(Legitimate voters)}$$

If a ballot is recorded in the name of some voter i, then either $i = id$ where id is the identity of our (unique) honest voter, or i is a (legitimate) dishonest voter: only identities of the form $\mathsf{dishonest}(i')$ (or id itself) are provided with voting material.

We further show that whenever an election authority confirms the ballot of a (possibly dishonest)

voter id then he has registered at least k ciphers for id, corresponding to k valid distinct selections, where k is the number of selections allowed for voter id.

$$\mathsf{Confirmed}(id) \rightsquigarrow \mathsf{Recorded}(id, \mathsf{enc}(pk, G(s_1), r_1)) \wedge \cdots \wedge \mathsf{Recorded}(id, \mathsf{enc}(pk, G(s_k), r_k)) \wedge \bigwedge_{1 \leq i \neq j \leq k} s_i \neq s_j \quad \textbf{(Ballot verifiability)}$$

The fact that a voter cannot register strictly more than k ciphers is discharged to the audit process (and the checks done by election authorities): a ballot should contain exactly k ciphers, where k is the number of selections allowed for voter id.

We express *Uniqueness* as follows.

$$\mathsf{Recorded}(id, b, u) \wedge \mathsf{Recorded}(id, b', u'))) \rightsquigarrow b = b' \wedge u = u' \quad \textbf{(Uniqueness)}$$

No voter can register two distinct ballots. Thus, a fortiori, no voter can have two ballots confirmed and ready to be tallied.

9.3.2. Verifiability for honest voters

We express *individual verifiability* as follows.

$$\mathsf{VoterHappy}(id, s) \rightsquigarrow \mathsf{RecordedCipher}(id, \mathsf{enc}(pk, G(s), r)) \wedge \mathsf{Confirmed}(id) \quad \textbf{(Individual verifiability)}$$

As soon as a voter id successfully completes her voting process, thinking she has voted for s, then there is indeed a recorded ballot a registered on her name, where a encrypts $G(s)$, that is, the encoding of her intended choice. Moreover, a confirmation $\mathsf{Confirmed}(id)$ has been received so the ballot a is ready to be tallied. Note that we prove this property on confirmed, ready to be tallied, ballots. The fact that confirmed ballots are correctly tallied is discharged to the cryptographic analysis.

Ideally, we would like to express *Confirmed as recorded* as follows.

$$\phi_{TAR} = \mathsf{Confirmed}(id) \wedge \mathsf{Recorded}(id, a) \rightsquigarrow \mathsf{Intended}(id, s) \wedge a = \mathsf{enc}(pk, G(s), r) \quad \textbf{(Confirmed as intended)}$$

For any confirmed ballot, registered in the name of id, then id indeed intended to vote for the content of the ballot. This guarantees that even if the voter did not reach the very final step (with the finalization code), her vote cannot be manipulated. Note that again, we prove this property on confirmed, ready to be tallied, ballots. The fact that confirmed ballots are correctly tallied is discharged to the cryptographic analysis.

However, due to the over-approximation made in the ProVerif model, we can no longer prove ϕ_{TAR}. One reason lies in the fact that the election authority may *in our ProVerif model* record more than k ciphers and thus potentially record a malicious one. So we aim at proving ϕ_{TAR} only when the election authority records at most k distinct ciphers. A registrar misbehaves as

$$
\begin{array}{l}
\mathsf{let\ ValidationProcess}() = \\
\quad \mathsf{if\ Checked}(id, s_1) \wedge \ldots \wedge \mathsf{Checked}(id, s_k) \\
\qquad \wedge\ s_2 \neq s_1 \\
\qquad \wedge\ s_3 \neq s_1 \wedge s_3 \neq s_2 \\
\qquad \ldots \\
\qquad \wedge\ s_k \neq s_1 \wedge \ldots \wedge s_k \neq s_{k-1}\ \mathsf{then} \\
\quad \mathsf{out}(pc, \mathsf{CautiousVoter}(id)).
\end{array}
$$

Figure 37: The cautious process with k selections.

soon as he records $k+1$ ciphers. This is captured by the property ϕ_{MisReg} defined as follows.

$$
\begin{array}{rcl}
\phi_{\mathsf{MisReg}} & = & \mathsf{RecordedCipher}(id, a_0) \wedge \mathsf{RecordedCipher}(id, a_1) \wedge \cdots \wedge \mathsf{RecordedCipher}(id, a_k) \\
& & \wedge \bigwedge_{0 \leq i \neq j \leq k} a_i \neq a_j
\end{array}
$$

Therefore, instead of ϕ_{TAR}, we consider ϕ'_{TAR} defined as follows.

$$
\begin{array}{l}
\phi'_{TAR} = \mathsf{Confirmed}(id) \wedge \mathsf{Recorded}(id, a) \rightsquigarrow \\
\qquad\qquad \mathsf{Intended}(id, s) \wedge a = \mathsf{enc}(pk, G(s), r) \vee \phi_{\mathsf{MisReg}} \quad \textbf{(Confirmed as intended')}
\end{array}
$$

Intuitively, ϕ'_{TAR} guarantees that either ϕ_{TAR} (Confirmed as intended is satisfied) or the (honest) election authority misbehaved. The later case cannot occur for all real executions (since the election authority is honest), hence ϕ_{TAR}.

This transformation is still not sufficient since, symmetrically, due to the over-approximation made in the ProVerif model, the honest voter may send her confirmation code *before* she has checked all her return codes. So instead of the property ϕ'_{TAR}, we first show two other properties. First, if a (honest) election authority has confirmed a ballot from a voter id then this voter must have checked some of her corresponding return code.

$$
\mathsf{Confirmed}(id) \rightsquigarrow \mathsf{Checked}(id, s') \qquad \textbf{(Confirmed implies Check)}
$$

A real honest voter will not stop the checking procedure: if a voter starts checking a return code, she will check exactly that her k distinct return codes correspond to her vote intent before sending her confirmation code.

We write a small process, displayed in Figure 37, that tests whether k distinct $\mathsf{Checked}(id, s')$ events have occurred. If yes, it activates the event $\mathsf{CautiousVoter}(id)$.

Therefore, instead of ϕ_{TAR} or ϕ'_{TAR}, it is sufficient to prove the following ϕ''_{TAR} property.

$$
\begin{array}{l}
\phi''_{TAR} = \mathsf{CautiousVoter}(id) \wedge \mathsf{Recorded}(id, a) \rightsquigarrow \\
\qquad\qquad \mathsf{Intended}(id, s) \wedge a = \mathsf{enc}(pk, G(s), r) \vee \phi_{\mathsf{MisReg}} \quad \textbf{(Confirmed as intended")}
\end{array}
$$

As explained in Section 9.1, we model the finalization code by a fresh secret value. This

abstraction is correct only if the attacker cannot reconstruct the finalization code by himself, that is, if he cannot obtain all the (hash) of the n points of the polynomial associated to a an honest voter i for a given (honest) authority ED. We ask ProVerif to prove that the only points obtained by the attacker are the points corresponding to the k selections of the voter. Provided that $n > k$, this justifies our abstraction. This property is formalized as follows.

$$\mathsf{attacker}(h(\mathsf{eval}(A', \mathsf{Pos}(ED, id, s)))) \rightsquigarrow \mathsf{RecordedCipher}(id, \mathsf{enc}(\mathbf{pk}, G(s), r)) \quad \textbf{(Valid AbstractFC)}$$

where ED is an honest election authority and $\mathsf{attacker}(m)$ is a special predicate in ProVerif that states that the attacker may learn m.

9.4. Analysis with ProVerif

Threat model and assumptions. We consider one honest voter and one honest election authority while all the rest is corrupted. In other words, we consider:

- A dishonest bulletin board. Note however that we assume that the election authorities as well as the talliers see the *same* bulletin board. In particular, we assume the honest election authority can guarantee that the tallied bulletin board is the one he has seen and approved.
- An arbitrary number of dishonest election authorities (and one honest election authority).
- An arbitrary number of dishonest voters.
- All voting devices are dishonest, including the voting device of the honest voter.
- The private key of the election key is compromised (that is, given to the adversary). Note however that we assume that the result of the election provably corresponds the confirmed ballots.
- We assume an abstract and honest tally process. More precisely, we assume that the result of the election corresponds to the set of ballots that have been confirmed (by the honest election authority). This is actually guaranteed by a mixnet and zero-knowledge proofs that go beyond the scope of the ProVerif tool. Indeed, we cannot model the fact that an arbitrary long list of encryptions corresponds to the re-randomization of another list of encryptions.

Results Our model makes use of states, like the ballot box stored by the election authority, to make sure that a voter does not vote twice. States in ProVerif are difficult to handle due to the internal over-approximation of the tool, yielding in general to false attack, hence the absence of proof. Therefore, we used GSVerif [8], a recently developed front-end for ProVerif, that soundly transforms a ProVerif file with states, into another ProVerif file that will suffer less from this over-approximation. We simply need to indicate the data that are used as states (here the ballot box) as `[value]`. We then run the resulting file with a new branch of ProVerif[13] that better handles the treatment of disjunctive queries.

[13] `https://sites.google.com/site/globalstatesverif/`

We consider several values for the (only) parameter k representing the number of selections. We stoped the analysis after 60h, using a dual Intel(R) Xeon(R) CPU E5-2687W v3 @ 3.10GHz with 378GB of memory. Note however that ProVerif is single threaded. The resulting analysis is displayed below.

Property	$k = 1$ to 4	$k = 5$	$k = 6$	$k = 7$	$k = 8$
Confirmed as intended	✓ 1s	✓ 2s	✓ 49s	✓ 25m09	✓ 08h50
Individual verifiability	✓ 1s –	For any value of $k > 0$			
Eligibility	✓ 1s –	For any value of $k > 0$			
Ballot verifiability	✓ 1s	✓ 13s	✓ 4m02	✓ 01h30	✓ –h–

All the model files are in the `Specs_Divided` forder, one per property. The properties Confirmed as intended and Individual verifiability state properties from the point of view of an honest voter and the corresponding files are generated from `CHVote_Honest_Voters.pv`. They include the Valid AbstractFC property. The other three properties (Eligibility, Unicity, Valid votes) should hold for any voter. The three corresponding files are generated from `CHVote_All_Voters.pv`.

Sanity checks To test our symbolic model, we have performed two kinds of test.

1. Executability: we have checked that a normal execution of the voting process can occur, as expected. This is of course a very minimal test but it typically catches typos and mismatch in the order of the arguments in messages.
2. Attacks for weaker models: we have checked that the protocol becomes insecure if we weaken it. We have considered several weakening:
   - We let the adversary rebuild a polynomial with $k-1$ points instead of k points (since the polynomial is of degree $k-1$). Attacks are found, as expected.
   - We consider honest voters that check $k-1$ return codes instead of k. Attacks are found, as expected.
   - We consider an (honest) election authority that accepts $k+1$ ciphers, instead of k. Again, attacks are found, as expected.

The files corresponding to these experiments can be found in the folder `SanityChecks`.

Part III.
Recommendations

10. Recommendations

These recommendations apply to CH-Vote 1.3 as described in the document dated 31 October 2017. They are listed in decreasing order of severity. Of course, this perception is subjective and depends on the context of the election.

10.1. PoK on public key shares

Severity: high
Difficulty: low (using Pedersen DKG)

The algorithm GenKeyPair generates a key share for a threshold ElGamal scheme. Public key shares are published on a bulletin board. This method of generating keys may be vulnerable to the following attack: a dishonest authority with index $I \leq s$ waits for all other authorities to publisk their shares, picks a pair (sk_I, pk_I) and publishes $pk' = pk_I / \sum_{i \neq I} pk_i$. The result is that the combined public key is $pk = pk_I$. Although this attack means the election will not tally correctly, and thus will always be detectable, it does allow the dishonest authority to decrypt and learn everyone's votes using sk_I. This is clearly not desirable.

The established approach to threshold key generation is called Pedersen's Distributed Key Generation (Pedersen DKG) [14]. In essence, each authority publishes a Schnorr proof of knowledge of the secret key matching their public key. This can be combined with a Schnorr signature on another signature keypair, but note that a signature on its own is not enough: a dishonest authority could strip the signatures from everyone else's public key and still perform the above attack, then sign pk' with their own signature keypair. A proof of knowledge of the secret key is required to avoid this attack.

Gennaro et al. [11] point out that even Pedersen DKG leaves a dishonest authority who goes last with the option of biasing the public key. Suppose this authority wants to have a public key where the last two bytes of the public key are all zeros, something that would occur with a probability 2^{-16} in an honestly generated key. The authority repeatedly generates key pairs until they find one that matches the desired pattern; in our example they would expect to have to generate 2^{16} key pairs before finding a suitable one, which they publish as their own key. The mitigation suggested in the paper [11] is to move to a multi-party key generation protocol.

We recommend that CH-Vote implements at least PoKs on the public key shares in GenKeyPair and, for the same reasons, on the voter key shares in GenSecretVoterData and GenPublicVoterData. This is technically easy to implement and since the extra cost falls solely on the authorities during the protocol set-up, it does not impede the voters. The cost of a few extra exponentiations should be manageable.

With PoKs implemented, our security proofs do not require a change to a multiparty protocol for key generation.

> ***Threat.*** *A dishonest authority may compute her share of the public key such that she can decrypt all the votes. Note that in this case the official tally would fail so the attack would be detected (but too late).*

10.2. PoK on shares of $\widehat{\mathrm{D}}$

Severity: high
Difficulty: low (using Pedersen DKG)

This is the same principle as for the last recommendation but concerns the shares $\widehat{d} = (\widehat{x}, \widehat{y})$ for the voter keys.

Suppose that authority s is dishonest. They wait for the previous $s-1$ shares $\widehat{x}_j$ to appear on the bulletin board, then for a particular voter i they pick x_s randomly and set $\widehat{x}_s = \widehat{g}^{x_s} / \prod_{j<s} \widehat{x}_j$. Because $\widehat{x} = \prod_{j=1}^{s} \widehat{x}_j = \widehat{x}_s$, authority s knows the discrete log of $\widehat{x}$.

Similarly, authority s may wait for the previous $s-1$ shares $\widehat{y}_j$ to appear on the bulletin board, then for a particular voter i they pick y_s randomly and set $\widehat{y}_s = \widehat{g}^{x_s} / \prod_{j<s} \widehat{y}_j$. Because $\widehat{y} = \prod_{j=1}^{s} \widehat{y}_j = \widehat{y}_s$, authority s knows the discrete log of $\widehat{y}$.

This breaks eligibility and individual verifiability of the protocol. The attacks (explained below) can be avoided by having each authority provide a Schnorr proof of knowledge of the discrete logarithm of each of their shares along with the share itself. This causes some overhead ($s \cdot N_E$ extra proofs, if the proofs for $\widehat{x}$ and $\widehat{y}$ are combined) but the cost falls on the authorities, not the voters, and these shares and proofs can be precomputed before the election.

We list corresponding threats in decreasing order of severity.

Threat 1. *A dishonest authority may vote on behalf of an honest voter. Indeed, as explained above, they may select their share such that they know the discrete log of $\widehat{x}$ and $\widehat{y}$. Therefore they can submit a ballot and confirm it. It the dishonest authority selects voters that usually do not vote, this attack is hard to detect.*

Note that the bulletin board may detect this attack if it checks consistency of the $\widehat{x}_j$ posted on the bulletin board and the x_j received privately (for each authority and each voter).

Threat 2. *A dishonest authority s, colluding with the voting device of an honest voter, may vote on behalf of this voter, without having to cheat on $\widehat{x}_s$. Therefore this attack would be undetected by the printing authority. For this attack, authority s select a voter i, computes their x_s and $\widehat{x}_s$ for i as expected but chose $\widehat{y}_s$ that such they know the discrete log of $\widehat{y}$. Then as soon as voter i starts to vote, her voting device learns x and therefore authority s has all the material to cast a vote on behalf of voter i. Of course, i will not receive the return codes as expected and may complain. Therefore this attack could be detected due to the complaint of voters.*

Threat 3. *A dishonest authority s may produce inconsistent x_s and $\widehat{x}_s$ or simply inconsistent y_s and $\widehat{y}_s$ for a selected voter i, to prevent i from voting (e.g. because she is likely to vote for a candidate that does not correspond to the authority's goal). This attack could be detected due to the complaint of voters.*

10.3. Strengthen the ballot proofs

Severity: high
Difficulty: low

The ballot proof, implemented by GenBallotProof and CheckBallotProof, in CH-Vote 1.3 takes the product e of all ciphertexts as input to the ZK-PoK. This is vulnerable to vote malleability attacks.

Imagine a voter wants to cast a component $\alpha = (\widehat{x}, \mathbf{a}, \pi)$ containing ciphertexts

$$\mathbf{a} = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix}$$

An attacker intercepts this ballot component and replaces $a_{1,2}$ with $a'_{1,2} := a_{1,2} \cdot g^r$ and $a_{2,2}$ with $a'_{2,2} := a_{2,2} \cdot g^{-r}$ for a randomly chosen r. The result is that the new $\mathbf{a}'$ contains ciphertexts for random elements that are, with overwhelming probabilities, not a valid OT query. However, the value $e = (a_{1,1} \cdot a_{2,1}, a_{1,2} \cdot a_{2,2})$ is identical to $e' = (a_{1,1} \cdot a_{2,1}, a'_{1,2} \cdot a'_{2,2})$ so the ZK-PoK still verifies. This means that the OT authorities will still respond to this modified ballot, causing the voter to forfeit their vote.

This breaks individual verifiability in the sense that the bulletin board and the authorities accept the modified α' even though it did not come from the voter.

The solution to this problem is to hash the entire component $\mathbf{a}$ in place of the sum e in the ZK-PoK. This is computationally almost for free: it only requires a few more elements to be added to an already existing hash input but does not change the size of the element α. The result is that α is now non-malleable and one can conclude that if the PoK (which also functions as a Schnorr signature) verifies then α must have come from the voter.

> ***Threat 1.*** *An attacker may prevent a voter from casting a vote by interfering with the α component of the ballot in such a way that the authorities respond to the modified ballot. The voter will detect that something has gone wrong, but will be unable to cast a vote.*

> ***Threat 2.*** *An attack may intercept Alice's ballot, chose two candidates c_1 and c_2 and learn whether Alice did select these two candidates (among the rest of her selection).*

10.4. Operational concerns

Severity: medium
Difficulty: unknown

The CHVote protocol makes several assumptions on how the system is used in practice. These assumptions should be spelled out clearly to election authorities. In particular, the security of the protocol relies on the following points.

Multiple blank options The analysis assumes the voters check all the return codes they receive. In particular, if they vote blank to a question where they may select up to 4 choices, they

must check 4 distinct blank options. The voters should be properly instructed to do so. There are attacks otherwise: a dishonest device may use the remaining "blank" choices for casting other choices.

Number of selections strictly smaller than the number of candidates The document states that the number of selections k made by voters is strictly smaller than the number of candidates n. This assumption is crucial for the security of the protocol as there is an attack otherwise. Namely, if $k = n$ then an attacker (in particular a corrupted voting device) may forge the finalization code as soon as he learns the return codes provided to the voters, bypassing the last step of the election authority. In other words, he may prevent the confirmation of the voter to reach the election authority (therefore the vote will not be counted), without the voter noticing (she will receive her finalization code as expected).

Therefore authorities should be clearly told that **the system must not be used in a context where** $k = n$, that is, in a context where a voter is allowed to select all the candidates. In particular, the scenario where $k = n = 1$ (voters may only approve a given candidate) seems plausible.

Alternatively, the design of the finalization codes could be changed. It could be simply a fresh value, which would also ease the security proofs.

Threat. *In case there is a question where the voter can select several candidates, say 4 candidates and the voter choses to vote blank, he actually has to check 4 distinct blank options. Otherwise, a dishonest voting device may use the remaining "blank" choices to cast vote for other candidates of her choice.*

Voters should be properly instructed to check as many blank options than the number of selections for the corresponding question. This may raise usability issues.

10.5. Set $p' = \widehat{q}$.

Severity: hard to assess
Difficulty: moderate

The choice of p' to be slightly larger than $\widehat{q}$ and of a particular form makes arithmetic modulo p' slightly more efficient. However, the PoK on the ballot component γ certifies knowledge of a preimage y^* such that $\widehat{y} = g^{y^*} \pmod{\widehat{p}}$ and in the protocol we have $y^* = y + y'$ where y is known to the voter, whereas y' should only be obtainable by a voter making a valid OT query in their ballot component α.

The proof of ballot verifiability must therefore show that a voter cannot obtain y' unless they make a correct α component and the obvious way to do this is to reduce to the well-known discrete logarithm (DLOG) assumption. However, the reduction must also simulate the remaining elements such as the matrix $\widehat{\mathbf{D}}$ to the adversary in a way that is indistinguishable from the distribution of these elements in a protocol execution.

Neither y' nore the components of $\widehat{\mathbf{D}}$ are uniformly random modulo $\widehat{q}$ so the simulated distribution in the reduction is not identical to the real one in the protocol. We have every reason to believe that the distributions are computationally indistinguishable, using a theorem by

Boneh and Venkatesan [7] as evidence, but this theorem does not come with a usable "concrete security" bound as it relies on lattice basis reduction.

Setting $p' = \widehat{q}$ eliminates this problem and provides a reduction to a well-known problem (DLOG), with a satisfactory concrete security bound, and without a theoretical "leap of faith" to argue indistinguishability of distributions.

Although this change is trivial to implement (and even suggested as an option by the CH-Vote authors), it does cause a small performance impact on arithmetic modulo p'. We would suggest making this change, but this suggestion is from a theoretician's point of view — we have not found any attacks that arise if the modification is not made.

10.6. Clarify the relationship between counting circles and the eligibility matrix

Severity: unknown, this is a privacy issue not a verifiability one
Difficulty: unknown

If it is possible for different voters in the same counting circle to have different eligibility rights in different elections, there may be a privacy issue as follows. The problem does not apply if eligibility cannot vary within counting circles, in which case we also get better universal verifiability properties in the (unlikely, but out of scope) case that all authorities are dishonest.

Consider a small community consisting of a single counting circle that is running two elections: one for a community councillor, for which all voters are eligible, and another election for a position in the local Protestant church council for which only a subset of voters (who are members of the church) are eligible. With the current GetVotes implementation of announcing the result, it is possible to compute statistics on whether a particular candidate for the community council was more popular among church members than among non-church members by comparing the proportion of votes for this candidate between rows of the matrix $\mathbf{V}$ that also have entries in the columns for the church election and rows that do not.

For example, if the community council election is between candidates A and B and the church election is between candidates C and D, one might choose the following encoding: $\mathbf{k} = (1, 1), \mathbf{n} = (3, 3)$, p_1 = vote for A, p_2 = vote for B, p_3 = abstain for the first election, p_4 = vote for C, p_5 = vote for D, p_6 = abstain for the second election.

Consider two voters and an eligibility matrix $\mathbf{E} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. At the end of the election, the following matrix results: $\mathbf{V} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$.

From this we can learn that both A and B got one vote each in the first election, which is information that should definitely be revealed. But we can also learn that the voter who was eligible to vote in both elections was the one who supported candidate B, even though they elected to abstain in the second election. This is strictly more information that is revealed than if we conducted both elections with a separate run of the whole protocol.

10.7. Exclude 0 from the domain of sampled points

Severity: low
Difficulty: low

In GenPoints, we recommend to initialise the set X as $\{0\}$, that is to prevent any of the values x_i from being chosen as 0 which could reveal $y' = p(0)$ even in the presence of an incorrect OT query. The probability of this occurring is negligible but it makes for better constants in the security proof to not have to deal with this case separately and it makes the guarantee that an incorrect OT query produces no information on y' an absolute one.

10.8. Check tallies for evidence of attacks

Severity: low
Difficulty: medium

If at least one authority is honest, the probability of an incorrect but confirmed ballot on the board is negligible (due to BV). While applying GetVotes, there is the possibility to check for some classes of incorrect votes and we would recommend doing this. Specifically, we recommend auditing the final matrix $\mathbf{V}$ to check for the following cases, even though we are confident that we are extremely unlikely to encounter them in practice. But since it is possible to check for them, there is little reason not to.

- A row of $\mathbf{V}$ contains more than k ones.
- In a row of $\mathbf{V}$, the columns for a particular election j have more than k_j ones.
- The number of votes cast in any election (set of adjacent columns) is greater than the number of voters eligible for this election.
- The number of voters (rows of $\mathbf{V}$) who cast a vote in a particular counting circle is greater than the number of voters in this circle.
- A row of $\mathbf{V}$ contains more than one counting circle, or no counting circle at all.

10.9. Secure link between the Printing Authority and Election Authority

Severity: low
Difficulty: low/medium

The secure link between the Printing Authority and some honest Election Authority is underspecified. Currently, the secret voter data is first encrypted alone, then signed along with the election event identifier 'U'. However, if the Printing Authority were to be involved in multiple elections with the same decryption key, the attacker could mount an attack as follows. Consider two elections, the real one and another one where the intruder fully controls the Election Administrator and all the Election Authorities (this second election could be a test election). Then the attacker could intercept the cipher texts created by the E.A. of the first election, resign them in the name of the corrupted E.A. of the second election, and make all the resulting voting cards to be sent to him by adjusting the Voter Data in the second election. He would

then receive the exact copy of all the voting cards of all the voters of the first (honest) election, simply because all the private shares sent to the Printing Authority, even if undisclosed, are the same in both elections.

We recommend either to make sure that all the signature and decryption keys are refreshed for each election event, or to modify the encrypted message sent from Election Authorities to the Printing Authority so that it cannot be accepted for another election (through appropriate labeling).

11. Comments on the source code

11.1. Introductory remarks

The CHVote system specification comes with a companion Java implementation by a developer of the project. It is made publicly and freely available under the GNU AGPLv3+ license and is hosted by the GitHub platform[14]. The status of this code is clearly mentioned in the `README.md` file: this is a prototype developed as a proof of concept, and is not supposed to be used *as-is* in production. Still, the developer are prudent and kindly asked to be made aware about security issues in advance to public announcement (e.g. via a pull-request). This is indeed a good idea, because in the future this code could become a *de facto* reference code for the CHVote protocol upon which production codes are based, not necessarily with the current developers being involved.

The current public version is the commit `9b0e7c9fcd409`, dated from April 2017, and this is the version we have studied. The consequence of this 1-year old version is that the implementation corresponds to a version of the specification that is older than the one studied in the rest of this report. In fact, a first suggestion we make is to mention precisely in the source code the version of the specification that it corresponds to, and to update this information with the evolution of the code. We list below the main differences between the current code and the version of the specification we have been working with.

Our work on the source code has been mostly on the cryptographic back-end, corresponding to the numbered algorithms in the specification, and the auxiliary functions they use. Following the terms of the contract, the time devoted to the study of the code was not long and can not be considered as sufficient for an audit of a code that would go in production, even without taking into account the fact that the code is not up-to-date with the specification.

11.2. General comments

The code is written in the Java language, using standard libraries. It requires version 1.8. This is good choice for a proof of concept implementation: Java is one of the most widely known language, so that potential developers of the CHVote protocol will certainly know it and can adapt it to another language is necessary (it is likely to be necessary for the voting client that might have to run in a web browser and be translated to Javascript, for instance). The use of the most recent version of Java at the time of writing the code is also appropriate: this allows a modern implementation style with lambdas, for instance.

We had no difficulty in compiling and running the tests, after installing the appropriate Java version and the Gradle build system on a Debian Linux distribution. The efficiency aspects were not investigated.

The general impression when reading the code is good. Good coding practices have been followed, with a stable coding-style among the various files, reasonable checks on the input parameters for each function, etc. The organization of the source directory is well thought, and one can easily navigate between files and subdirectories and follow the logic of the protocol.

In most cases the code follows exactly the specification. For each numbered algorithm in the

[14] `https://github.com/republique-et-canton-de-geneve/chvote-protocol-poc`

specification there is a function with exactly the same name (with a lower-case for the first letter, following Java tradition) and the same parameter names, with minor and easy to understand differences. Of course, this is no longer the case for the algorithms of the specification that have changed since the development of the code. We strongly encourage the developers to update the code so that it is in perfect accordance with the newest specification (when it will be stabilized).

11.3. Important differences between the code and the specification

We list here the important differences between the version of the code that we have studied and the version of the specification that has been analyzed in this report. As far as we can tell, these differences are due to the fact that the code refers to the version 1.0 of the specification, but we did not fully check this.

Counting circles are not implemented. Although the notion of counting circle is explained in the old version of the specification, the pseudo-code did not take it into account. As a consequence, a few functions do not follow the current specification. Most notably, the `getEncryption` function skips the multiplication by the prime marking the counting circle and the `getVotes` function that decodes the result is also different.

Use of several polynomials instead of one. In the old version, a trustee would assign a new polynomial for each of the t elections. In the current specification, one single large degree polynomial is used, and this is the version that has been analyzed and proved secured in this document. The multiple-polynomial variant of the code should be updated to follow this. This change affects a certain number of functions, including the parameters they take as input and output.

(partly) Weak Fiat-Shamir. The code will have to be updated to take into account the security issue concerning the Fiat-Shamir transform used in the NIKZP included with the ballots.

Weak DL primes. The version 1.0 of the specification proposed recommended field parameters that were much weaker than the announced security level, due to primes having a lot of structure. Some of these primes are still present in the code. For Level 1, the given 1024-bit prime p is indeed $2^{1023} + 1671615$, and discrete logarithms in $GF(p)$ can be computed with moderate academic computing power (see reference [23] in the specification).

In total, a large proportion of the functions in the code are different from the specification. For many of them it is not too difficult to imagine the future modifications, so that we could still read the code with no major difficulties but this is another reason for which this can certainly not be considered as a complete audit of a code that could go in production.

11.4. Important bugs

- The function `isMember_G_q_hat` is wrong. It has been extrapolated from the `IsMember` algorithm 7.2 in the specification, but this algorithm is valid only for a safe prime. For

testing membership in a non-safe prime, one can not use the Jacobi symbol: one has to resort to checking that the element is indeed of order $\hat{q}$ (assuming that $\hat{q}$ divides only once $\hat{p}-1$).

This function is used in particular by the authorities in `checkBallotProof` to validate the NIZKP of the voter. We did not investigate the possible consequence; the fix is easy.

- The file `RandomGenerator.java` contains a few functions that are used in many places of the code. Among them, the function `randomIntInRange` that takes two parameters `from` and `to` is claimed to return a uniform random integer between these bounds, inclusive. However, since the `secureRandom.nextInt` function excludes the upper bound, this is also the case for `randomIntInRange`. It can never return `to`.

  In the same spirit, the function `randomInZq` will never return $q-1$, and therefore is not uniform. This can be fixed by not subtracting 1 to q: just call `randomBigInteger` with parameter q.

  Those two issues have absolutely no practical consequence if the range is so large that the probability of hitting such corner cases is negligible. Still, when fixing them, care must be taken to functions that use them (see below for `genPolynomial`).

  However, the function `randomIntInRange` is used by the function `genPermutation` to generate the permutation used by the mixers. Here, the bug implies a loss of entropy. In the extreme case where there are only 2 ballots to mix, the permutation becomes deterministic.

  Finally and less importantly, the `randomBigInteger` function has the following issue. It uses a `MAX_ITERATIONS` constant that is set to 256. If the `secureRandom` is correct, there is no need for it: each trial will have half a chance to be successful; therefore failing 256 times in a row will occur with probability 2^{-256}, *i.e.* never. If we reach the `MAX_ITERATIONS` limit, then this is most probably a problem with the random number generator, and it is better to abort the computation. Therefore we recommend to remove this `MAX_ITERATIONS` and let the program run forever in an endless loop (or raise an exception, if endless loops are not acceptable in the context).

11.5. Other positive or negative remarks, typos

- The function `modExp` in `BigIntegerArithmetic.java` calls the `Gmp.modPowSecure` function which is the constant-time variant of modular exponentiation. This would indeed be very nice to have everything constant-time to make the code resistant against side-channel attacks. It is however really difficult to ensure that the whole code is safe. I suggest to add a comment around this call to acknowledge that no effort is made anywhere else to be constant-time, and that side-channel leakage is still likely despite the use of `Gmp.modPowSecure`.
- In the `ByteArrayUtils.java` file, the amount of copies involved is really amazing. This is probably not a problem of efficiency, but can be considered inelegant. Also, using `Math.pow` to compute a flag is unusual (and inefficient): in a bit-fiddling function, a shift would be more natural.
- In the comment of `genPolynomial`, the sum should start from 0, not 1. Note also that

this function uses `randomInZq` in a correct but maybe unexpected way, in order to impose a non-zero element. Care should be taken when fixing the above issues with randomness.

This is linked to the recommendation in Section 10.7: the implementation already takes it into account, but does not check for hitting twice the same value. The probability of this event occurring is negligible, and again, if this occurs it is probably better to raise an exception about the random generator having a problem.

- There is a copy-paste bug in the main comment of `getYValue`.
- We slightly regret that all the types that end-up being implemented with a multi-precision integer are just of type `BigInteger`. Having specific types for $\mathbb{G}_q$, $\mathbb{G}_{\widehat{q}}$, $\mathbb{Z}_q$, $\mathbb{Z}_{\widehat{q}}$, $\mathbb{Z}_{p'}$ would provide more guarantee that the modulo operation is always done and always with the good modulus. We did not do a thorough check, but for instance, on line 185 of `DecryptionAuthorityAlgorithms.java` there is a missing `.mod`. Here, this has no consequence on correctness.

  A related issue is in the `getPublicVoterData`, where the implicit cast from $\mathbb{Z}_{p'}$ to $\mathbb{Z}_{\widehat{q}}$ would not be visible in the code (due to version mismatch with the specification, this is anyway not present right now). In the future `genConfirmation` the same problem will arise: it is not clear from the specification but the cast from $\mathbb{Z}_{p'}$ to $\mathbb{Z}_{\widehat{q}}$ should be done before taking the sum.

  With all these data being `BigInteger`, some of these difficulties might be too much hidden.

  On the other hand, having all these additional types would make the code heavier. The current compromise is certainly acceptable.
- It is a really good idea to have the names of the variables and of the parameters to be strictly identical to the ones in the specification, with `_hat`, `bold_`, etc, including in classes like `IdenticationGroup`. This allows to avoid typos where one would take p instead of $\hat{p}$ or the `IdentificationGroup` instead of the `EncryptionGroup`.

  The source code is perfectly consistent with this convention (as far as we could tell), and this is of great help for the reader (and no doubt, for the programmer as well).
- There are unavoidable mismatches between indices starting with 0 (Java convention) and 1 (convention in mathematics for row and column indices of a matrix, for instance). Each time this occurs, the code includes a comment with a warning. Again, this is very much appreciated.

Part IV.
Appendix

12. Appendix

12.1. Fiat-Shamir-Schnorr proofs over multiple fields

The standard theory for Fiat-Shamir-Schnorr (FSS) proofs assumes that the function of which a preimage is being proven is linear over a particular field (or at least, a module homomorphism over a ring). The proof used in CHVote however inolves two separate fields of different prime orders q and $\widehat{q}$. This construction is still sound, but this needs to be proven properly.

Definition 12.1 (FSS over multiple fields) *Let k and τ be positive integers and let $\mathbb{F}_1, \ldots, \mathbb{F}_k$ be finite fields with $\mathrm{char}(\mathbb{F}_i) > 2^\tau$ for all i. Let $W_1, \ldots, W_k$ and $V_1, \ldots, V_k$ be such that W_i and V_i are nontrivial, finite-dimensional vector spaces over $\mathbb{F}_i$.*

We write W for $\prod_{i=1}^k W_i$ and V for $\prod_{i=1}^k V_i$ where the product is in the category of sets (the product is not *a vector space as there is no common base field).*

Let ϕ_i for $i = 1, \ldots, k$ be $\mathbb{F}_i$-linear functions with signature $W_i \to V_i$. Let $\phi = \prod_{i=1}^k \phi_i$, that is $\phi : W \to V, (w_1, \ldots, w_k) \mapsto (\phi_1(w_1), \ldots, \phi_k(w_k))$.

Let H be a function with codomain $\{0,1\}^\tau$. The Fiat-Shamir-Schnorr protocol for proving a preimage of ϕ consists of the following algorithms.

- *Prove takes as input $(w, v) \in W \times V$, picks a random $\omega \in W$ and sets $t \leftarrow \phi(\omega), c \leftarrow H(v, t), s \leftarrow \omega + cw$. The proof is $\pi = (t, s) \in V \times W$.*
- *Verify takes as input $(v, \pi) \in V \times (V \times W)$ and computes $t' \leftarrow \phi(s) - H(v, t) \cdot v$. It returns 1 if $t = t'$ and 0 otherwise.*

Lemma 12.2 (Correctness.) *For any $w \in W$, if we set $\pi \leftarrow \mathsf{Prove}(w, \phi(w))$ then we have $\mathsf{Verify}(\phi(w), \pi) = 1$.*

In the case of a single field, correctness follows immediately from the linearity of ϕ. However, the product of fields is not necessarily a field so we cannot talk of linearity of ϕ. The product of modules however is another module, so we just need to take a step back in terminology.

Proposition 12.3 *The element $R = \prod_{i=1}^k \mathbb{F}_i$ is a finite, commutative ring; the spaces W and V are R-modules and ϕ is an R-module homomorphism.*

This follows from writing out the homomorphism properties and using linearity in each component.

Proof (correctness). Let ω be the random value chosen for the proof. Then

$$t' = \phi(\omega + H(\phi(w), \phi(\omega)) \cdot w) - H(\phi(w), \phi(\omega)) \cdot \phi(w) = \phi(\omega)$$

which is exactly the t-component of π. We used the homomorphism property of ϕ to get this result. q.e.d.

Lemma 12.4 (Special Soundness) *Given two tuples (v, t, c, s) and (v, t, c', s') both in $V \times V \times \{0,1\}^\tau \times W$ and with $c \neq c'$, if $\phi(s) = t + cv$ and $\phi(s') = t + c'v$ then the vector $\left(\frac{s_1 - s'_1}{c - c'}, \ldots, \frac{s_k - s'_k}{c - c'}\right)$ is a preimage of v under ϕ.*

The only difficulty here is a notational one: we would like to write our preimage as $\frac{s-s'}{c-c'}$ but W is an R-module not a vector space, so the inverse of the integer $c - c'$ needs to be shown to exist first. We can however write $\phi(s - s') = (c - c') \cdot v$ and write out the individual components.

Proof. W.l.o.g. assume that $c > c'$ (as integers). For field i, since $\text{char}(\mathbb{F}_i) > 2^\tau$ and $c \neq c'$ we can conclude that $\phi_i\left(\frac{s_i - s'_i}{c - c'}\right) = v_i$. This gets us our preimage of v. q.e.d.

The proof of special soundness is the only step in the theory of Sigma protocols where it is required that the underlying structure is a field — the rest of the theory applies equally well to modules. This gets us the usual soundness property:

Corollary 12.5 (soundness) *In the random oracle model, the probability of any adversary (even computationally unbounded) creating a pair (t, s) that verifies on an incorrect statement, while making at most q oracle queries, is at most $q/2^\tau$ where τ is the number of bits of entropy in the challenge space.*

12.2. Simulation-Sound Extractability

To work with Fiat-Shamir-Schnorr proofs, we propose the following theorem.

Theorem 12.6 *Let a Fiat-Shamir-Schnorr proof system be given based on a linear function ϕ and with a challenge space $\mathcal{R}$.*

Let $\mathcal{A}$ be any algorithm that may make random oracle queries and $\mathcal{G}$ be a game that can make random oracle queries and queries to the following algorithm:

$$\mathsf{Prove}(v \in V, w \in W) : r \twoheadleftarrow W;\ t \leftarrow \phi(r);\ c \leftarrow H(v, t);\ s \leftarrow r + cw;\ \text{return } (v, t, c, s)$$

Suppose that in an execution of $\mathcal{A}$ with $\mathcal{G}$ and a random oracle, $\mathcal{A}$ ends the execution by returning a valid proof (w.r.t. the oracle) (v, t, c, s) with probability α. Then, if one replaces all Prove *queries with simulated queries as described in the proof below, there exists an extractor K that runs $\mathcal{A}$ and in addition returns a witness $w \in W$ such that $\phi(w) = v$, as long as $\mathcal{A}$ does not return one of the simulated proofs, with probability at least*

$$\frac{\alpha^2}{\nu}\left(1 - \frac{\nu\sigma}{|\mathcal{R}|}\right)^2 - \frac{\alpha}{|\mathcal{R}|}\left(1 - \frac{\nu\sigma}{|\mathcal{R}|}\right)$$

for any upper bound ν on the total number of random oracle queries and σ on the total number of simulation queries in the execution.

For our first hop, we rewrite the game $\mathcal{G}$ as a game $\mathcal{G}_1$ that runs its own random oracle. Whenever the adversary makes a random oracle query, the game answers it with its own oracle. This hop

clearly does not change the distribution of any inputs or outputs — we are just moving around artificial subsytem boundaries — so the adversary still outputs a valid proof with probability α.

Next, we change the random oracle as follows. Let it store a list Q (for "queries"), initially empty. On a random oracle query $H(x)$, if there is a pair (x, y) in Q then we return y, otherwise we delegate to the existing random oracle to get an answer y and store (x, y) in the list Q. Further, we introduce a new procedure

$$\mathsf{patch}(x, y): \text{ if } \exists z : (x, z) \in Q \text{ then abort else add } (x, y) \text{ to } Q$$

As long as no-one calls patch, we have simply memoized the oracle so we have not made any changes to the semantics of the game and the probability of $\mathcal{A}$ returning a valid proof is still α. Whenever we use patch we will need to argue that its use is unlikely to cause an abort. Call this game $\mathcal{G}_2$.

Next, we pick the calls to Prove and replace them with calls to the following algorithm:

$$\mathsf{Sim}(v \in V): s \twoheadleftarrow W;\ c \twoheadleftarrow \mathcal{R};\ t \leftarrow \phi(s) - c \cdot v;\ \mathsf{patch}(t, c);\ \text{return } (v, t, c, s)$$

Call this game $\mathcal{G}_3$. We have to check two properties of this hop.

- The point t is chosen such that it has at least as much entropy as a random point in $\mathcal{R}$ (assuming $|\mathcal{R}| \leq |V|$). Therefore, for an execution where at most ν random oracle queries happen (from both the game and the adversary) and where at most σ simulation queries happen, the probability of aborting in patch is at most $\nu\sigma/|\mathcal{R}|$ as each simulation query has a $1/|\mathcal{R}|$ probability of colliding with each previously made random oracle query.
- If the patch does not abort, then the distribution of the returned simulated proofs (v, t, c, s) is identical to the distribution of Prove. Indeed, in both cases the distribution is uniform on $(v', t', c', s') \in V \times T \times \mathcal{R} \times W$ subject to the conditions $v = v', H(v', t') = c'$ and $\phi(s') = t' + c' \cdot v'$.

As long as $\mathcal{G}_3$ does not abort, the probability of $\mathcal{A}$ creating its proof therefore cannot have changed. (A separate argument for each specific game will be needed to show that $\mathcal{A}$ cannot return one of the simulated proofs itself.) The probablity of $\mathcal{A}$ creating a proof is therefore $\alpha' = \alpha(1 - \nu\sigma/|\mathcal{R}|)$.

Finally, we consider the adversary $\mathcal{A}$, the game $\mathcal{G}_3$ and the random oracle memoiser (but not the random oracle itself) as an algorithm $\mathcal{B}$ that returns whatever $\mathcal{A}$ returns. We can now apply the Forking Lemma of Bellare and Neven [1] to this algorithm $\mathcal{B}$ to get the result that if $\mathcal{B}$ returns a proof with probability α', which is valid w.r.t. the external oracle (i.e. not one of the simulated proofs) then there is an extractor K that also returns the associated witness with probability $(\alpha')^2/\nu - \alpha'/|\mathcal{R}|$. Reordering terms gives us the desired result. q.e.d.

We will use this theorem in the proof of Ballot Verifiability and Confirmed as Intended.

References

[1] M. Bellare and G. Neven. Multi-signatures in the plain public-key model and a general forking lemma. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 390–399, New York, NY, USA, 2006. ACM.

[2] D. Bernhard, V. Cortier, D. Galindo, O. Pereira, and B. Warinschi. Sok: A comprehensive analysis of game-based ballot privacy definitions. In *2015 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 499–516, May 2015.

[3] D. Bernhard, O. Pereira, and B. Warinschi. How not to prove yourself: Pitfalls of the fiat-shamir heuristic and applications to helios. In X. Wang and K. Sako, editors, *ASIACRYPT*, volume 7658 of *Lecture Notes in Computer Science*, pages 626–643. Springer, 2012.

[4] D. Bernhard and B. Warinschi. *Cryptographic Voting — A Gentle Introduction*, pages 167–211. Springer International Publishing, Cham, 2014.

[5] B. Blanchet. Automatic verification of security protocols in the symbolic model: The verifier ProVerif. In *Foundations of Security Analysis and Design (FOSAD'13)*, volume 8604 of *LNCS*, pages 54–87. Springer, 2013.

[6] B. Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, Oct. 2016.

[7] D. Boneh and R. Venkatesan. Hardness of computing the most significant bits of secret keys in diffie-hellman and related schemes. In N. Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 129–142, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[8] V. Cheval, V. Cortier, and M. Turuani. A little more conversation, a little less action, a lot more satisfaction: Global states in proverif. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF'18)*, 2018.

[9] V. Cortier, D. Galindo, R. Kusters, J. Muller, and T. Truderung. Sok: Verifiability notions for e-voting protocols. In *2016 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 779–798, May 2016.

[10] V. Cortier, D. Galindo, and M. Turuani. A formal analysis of the neuchâtel e-voting protocol. In *3rd IEEE European Symposium on Security and Privacy (EuroSP'18)*, London, UK, April 2018.

[11] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. *J. Cryptol.*, 20(1):51–83, Jan. 2007.

[12] R. Haenni, R. E. Koenig, P. Locher, and E. Dubuis. Chvote system specification. Cryptology ePrint Archive, Report 2017/325, 2017. `https://eprint.iacr.org/2017/325`.

[13] G. Lowe. A hierarchy of authentication specifications. In *10th IEEE Computer Security Foundations Workshop (CSFW'97)*. IEEE Computer Society Press, 1997.

[14] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '91, pages 129–140, London, UK, UK, 1992. Springer-Verlag.

[15] B. Schmidt, S. Meier, C. Cremers, and D. Basin. Automated analysis of Diffie-Hellman

protocols and advanced security properties. In *25th IEEE Computer Security Foundations Symposium (CSF'12)*, pages 78–94, 2012.

[16] V. Shoup. Sequences of games: a tool for taming complexity in security proofs, 2004. shoup@cs.nyu.edu 13166 received 30 Nov 2004, last revised 18 Jan 2006.