An Internet Voting Protocol with Distributed Verification Receipt Generation

Kristjan Krips[1,4], Ivo Kubjas[2,4], and Jan Willemson[1,3]

[1] Cybernetica AS
Ülikooli 2, 51003 Tartu, Estonia
{kristjan.krips,jan.willemson}@cyber.ee
[2] Smartmatic-Cybernetica Centre of Excellence for Internet Voting
Ülikooli 2, 51003 Tartu, Estonia
ivo@ivotingcentre.ee
[3] Software Technology and Applications Competence Center
Ülikooli 2, 51003 Tartu, Estonia
[4] Institute of Computer Science, University of Tartu
Ülikooli 18, 50090 Tartu, Estonia

Abstract. This paper introduces an Internet voting scheme using a verification receipt design that splits the receipt generation between the voting client and vote collecting server. The aim of the protocol is to provide provable integrity properties even in the presence of a malicious component (most notably, client or server side malware). The protocol is designed to be used in low-coercion environments. We provide full description of the protocol, formally verifiable integrity proofs using EasyCrypt, and a discussion concerning other security requirements. The protocol was used in March 2016 for the Republican party caucus voting in the state of Utah, USA.

1 Introduction

The idea of Internet voting can be both appealing and terrifying. On one hand, it offers high accessibility to elections, arguably expected by the contemporary humans who are used to all kinds of online services. On the other hand, unlike in case of paper voting, all of its risks are not yet fully understood.

What the international community seems to agree upon is that a digital remote voting solution should provide clear verification and auditing mechanisms on both individual and system level. The exact nature of these mechanisms is determined by the threat model and available infrastructure.

In this paper we are going to present an Internet voting protocol that is meant to provide the voter with a post-election verification mechanism to detect vote manipulation attacks by both a malicious client application and voting server.

To achieve this, we propose a protocol design where the receipt generation is distributed between the two. An additional goal is formatting the receipt into the same data structure as the vote itself to utilize the existing functionality of multiple-choice ballots provided by the underlying technological platform.

Our design goals and choices have been motivated by the end user requirements. The protocol was designed to be used in March 2016 by the US Republican party for the option of online ballot submission during their caucus voting in the state of Utah.

As coercion was not regarded as a high risk in the environment, then no measures were implemented to allow for the voter to vote under coercion. Namely, given the receipt value, it is possible to uniquely determine the choice the voter voted for.

The protocol is built as an instance of TIVI platform[5], which in turn has foundations in the lessons learned while developing the Estonian Internet voting system. In particular, it shares several common characteristics (mix-net and provable decryption) with the IVXV protocol used in 2017 Estonian elections [16]. Still, the authentication mechanism was determined by the environment and the receipt generation routine was developed to match the verifiability properties required by the end user.

The voters had to pre-register to take part in online voting. Of 27490 pre-registered voters 24486 cast their vote using the online voting system [26].

To the best of our knowledge, this protocol has not yet been described in full detail in public sources (except for the JavaScript code that was implementing it). We believe that this description is of independent interest to the international community.

Our final contribution is modelling the protocol in EasyCrypt and providing formal verification of some target properties of the protocol. As integrity was stated as the most important goal by the end user, we concentrated on the integrity properties on both the client and server level.

2 Requirements set by the end user

Before designing an Internet voting protocol, the properties required from the protocol must be defined. The requirements depend on the specific environment and define the available methods for a particular event.

For example, coercion-resistance and post-election verification are somehow contradicting requirements – if the voter would be able to somehow verify that his/her particular vote was counted in the tally, then it would be possible to also prove to the coercer how the voter voted. Thus, there needs to be a clear decision between these two properties.

Thus, during the event preparation, the election organizers and Internet voting channel provider need to discuss and specify the expectations. Furthermore, the process needs to be transparent and possible risks should be made known to the parties and their possible effects estimated.

[5] `https://tivi.io/`

2.1 Integrity requirements

The main goal of the protocol is to provide an exact tally. To obtain this, we require that only eligible voters are able to cast a vote, votes can not be modified nor removed, and that the ballot is correctly encoded.

Requirement 1 (Eligibility). For every ballot in the final tally, there must exists a valid and registered voter who has cast a vote for a specific choice.

Requirement 2 (Tally integrity). After the voter has received a confirmation from the server that the submitted ballot has been stored, the stored ballot must be included in the final tally.

Requirement 3 (Ballot well-formedness). A ballot is a unique one-to-one representation of the voter's choice. The algorithms for encoding and decoding the vote are well-defined and correct.

2.2 Verifiability requirements

Due to environment settings, it was not possible to use pre-existing approach for individual verification using independent verification device [17] or return codes [4]. Hence, a slightly different concept of individual verifiability had to be defined.

The approach for individual verifiability was to have a receipt after casting a ballot which could be used for verifying that the ballot was correctly cast, stored and later tallied. The property is described as Requirement 4.

Furthermore, as the receipt was handed to the voter during the ballot casting, then the receipt had to include information which would allow for confidently claim any incorrect behaviour. The property is described as Requirement 5.

Finally, we consider the voter's privacy in this verifiability setting. We require that the cast ballot stays secret even for the election organizers unless the voter decides to challenge the receipt. On one side, this desists voters from applying illegitimate appeals. On the other side, this increases the trust in the appeal if it is successful, as different ballot content would be apparent. The property is described as Requirement 6.

Requirement 4 (Inclusion verifiability). Each voter can verify that her ballot has been included in the final tally.

As the final tally may be published long after the vote was cast, the information for verifying the ballot has to be stored. We will call both the stored information and the item it is stored on a *receipt*.

Requirement 5 (Liability provability). The receipt must include information which would allow to determine liability in the case when receipt verification fails.

Otherwise, it would be possible for a malicious voter (or a group of voters) to fake the receipts, claim that the tally is incorrectly computed and decrease the trust in the system. Eventually, this could even lead to Denial of Service when publishing results.

This requirement means that the receipt gives a strong proof of correctness. If the verification fails then either it is a case of election fraud, or a case of a dishonest voter or voting application. For the former it should be possible to prove the intent to cheat by election organizers, and for the latter it should be possible to prove the invalidity of the receipt.

A similar property, called "collection accountability" was described in [8].

We note that existence of verification receipts introduces the threat of coercion [13]. Hence, our protocol is only applicable in low-coercion environments.

Requirement 6 (Voter's privacy). Voter's choice must not become known to the election organizers unless the voter challenges the receipt.

2.3 Auditability requirements

The actions of election organizers must be auditable by an independent third party. More precisely, we require the following auditability properties.

Requirement 7 (Eligibility auditability). The auditor must be able to verify that only eligible voters have been able to vote and that no additional ballots have been added to the tally.

Requirement 8 (Decryption auditability). The auditor must be able to verify that the decryption operation is performed correctly on the encrypted ballots.

Requirement 9 (Privacy-preserving auditing). Auditing the election process must not threaten voters' privacy.

3 Protocol description

This section gives an overview of the protocol. We start by introducing the protocol participants and the used notation. Then we describe the assumptions that have to be fulfilled. Finally, we give an overview of the following four main phases of the protocol: distribution of keys, vote submission and receipt generation, vote decryption, post-election vote verification.

3.1 Participants

There are a number of parties involved in the protocol. As the primary party, we have Voter (VTR) as a physical participant. To participate in the elections, she needs the VotingClient (VC) software published by the election organizers. One voter is allowed to cast one vote and revoting is not possible.

There are several central functions that are under the control of the election organizers. However, there is a clear separation of duties.

First, there is the VotingServer (VS) that interacts with the VotingClient.

Secondly, the TallyServer (TS) decrypts the votes after the voting period ends. Due to Requirement 9, the ballots are shuffled using a mix-net before handing them over to TallyServer.

Thirdly, there is a KeyHolder (KH) that generates the encryption and signing keys, stores them securely and exports them at the correct protocol stage. The latter property is crucial to prevent revealing partial tally results. It can be implemented using different methods, e.g. hardware-based approach (using smart cards or HSMs) or distributed approach.

The read-append bulletin board (RABB) receives the cast ballots and stores them until the end of the election. The read-only bulletin board (ROBB) is initialized with a list of values and serves the values to the public.

Finally, there is a CertificationAuthority (CA) which provides confirmation of the voters' identities.

Correct operation of KH, CA and RABB are achieved by using the appropriate organizational measures and auditing. For example, the CA must conform to the standard requirements set to trust service providers, whereas the hash chain based RABB can be constantly monitored and its internal consistency can be verified by independent auditors.

3.2 Notation

The signature scheme that is used in the protocol is defined by three functions $(\mathsf{KGen}_{\mathsf{Sig}}, \mathsf{Sig}, \mathsf{Vf})$. The first of them is a key generation function that generates a key pair $(\mathsf{sk}, \mathsf{vk})$, which consists of a signing key and a verification key. The signing function Sig takes the signing key sk and some plaintext message pt, and returns the corresponding signature $\mathsf{s} = \mathsf{Sig}(\mathsf{sk}, \mathsf{pt})$. The verification function Vf takes the verification key vk, signature s and a message pt, and returns true if and only if the signature is obtained by using the corresponding signing key and plaintext. The function GetVer takes as input the signing key sk and returns the corresponding verification key vk.

In the implementation of our protocol, ECDSA was used as a signature scheme [18].

The encryption scheme is defined as a triplet of functions $(\mathsf{KGen}_{\mathsf{Enc}}, \mathsf{Enc}, \mathsf{Dec})$. The key generation function $\mathsf{KGen}_{\mathsf{Enc}}$ generates a pair of encryption and decryption keys $(\mathsf{ek}, \mathsf{dk})$. The encryption function Enc takes as input the encryption key ek, some random value ω and the message pt, and returns the ciphertext $\mathsf{ct} = \mathsf{Enc}(\mathsf{ek}, \omega, \mathsf{pt})$. The decryption function takes as inputs the decryption key dk and the ciphertext ct, and returns the corresponding plaintext $\mathsf{pt} = \mathsf{Dec}(\mathsf{dk}, \mathsf{ct})$. Additionally, there are functions Encode and Decode which map the voter's vote to an element of the selected representation set and back. Finally, there is a function Prove for providing the proof of correct decryption. If the decryption key dk is stored within a hardware module or is secret-shared, then only access to the functions $\mathsf{Dec}(\mathsf{dk}, \cdot)$ and $\mathsf{Prove}(\mathsf{dk}, \cdot, \cdot)$ are provided.

In the implementation of our protocol we are using ElGamal encryption scheme [15]. ElGamal works in a public group $\mathbb{G}$ of prime order p with generator g. The decryption key dk is used for finding the encryption key $\mathsf{ek} = g^{\mathsf{dk}}$. The encryption function Enc is defined as $\mathsf{Enc}(\mathsf{ek}, \omega, \mathsf{pt}) = (g^{\omega}, \mathsf{pt} \cdot \mathsf{ek}^{\omega})$ and the decryption function Dec is specified as $\mathsf{Dec}(\mathsf{dk}, \mathsf{ct}) = \mathsf{ct}_2 \cdot \mathsf{ct}_1^{-\mathsf{dk}}$.

ElGamal encryption scheme is homomorphic. Let us have two ciphertexts ct and ct' corresponding to the plaintexts pt and pt'. Then we have

$$\mathsf{ct} \cdot \mathsf{ct}' = (g^{\omega} \cdot g^{\omega'}, \mathsf{pt} \cdot \mathsf{ek}^{\omega} \cdot \mathsf{pt}' \cdot \mathsf{ek}^{\omega'}) = (g^{\omega+\omega'}, \mathsf{pt} \cdot \mathsf{pt}' \cdot \mathsf{ek}^{\omega+\omega'}),$$

the latter being an encryption of $\mathsf{pt} \cdot \mathsf{pt}'$.

The proof of correct decryption is denoted as $\mathsf{pf} = \mathsf{Prove}(\mathsf{dk}, \mathsf{pt}, \mathsf{ct})$. The proof is based on proof of discrete logarithm equality [1] and made non-interactive using Fiat-Shamir heuristic.

Ballot bt is a data structure representing the list $(\mathsf{ct}_1, \ldots, \mathsf{ct}_n)$ of ciphertexts, which typically are encrypted votes. In the described protocol, two-element ballots are used where the first element is an encrypted vote and second element represents the encrypted receipt. The function $\mathsf{Split}(\mathsf{bt})$ returns the corresponding ciphertexts as individually accessible elements $\mathsf{ct}_1, \ldots, \mathsf{ct}_n$.

For a receipt we will use the notation R, and an encrypted receipt will be denoted as rt.

In case a random value is needed, we use an appropriate space Ω to sample the value from.

3.3 Assumptions on the operating environment of the protocol

Assumption 1 (Existence of trust base). We assume that the participants belonging to the trust base behave according to the protocol and do not leak their private information.

The three trusted parties in the protocol are KeyHolder, RABB and CA. The protocol is designed such that the trust base would contain parties whose correctness can be audited using organizational measures and secured against external attacks using technical means.

Assumption 2 (Existence of a read-append bulletin board (RABB)). We assume the existence of a bulletin board which allows reading its current state and adding entries to the end of the bulletin board. Added entries are irreversible and unmodifiable, and the entries are strictly ordered with respect to the addition.

There have been several recent proposals to achieve such a primitive, based on e.g. threshold schemes [12] or Bitcoin-like block chain technology [23].

In our implementation, we are using hash chaining, where appending to the bulletin board is only allowed by the request signed by the voting server. The public interface allows queries for the chain elements using the chain index and the auditor interface allows retrieving a range of chain elements.

Assumption 3 (Existence of a read-only bulletin board (ROBB)). We assume the existence of a read-only bulletin board which is essentially a static key-value store that can be queried by the key.

In our deployment, the ROBB was implemented via a simple web-based query interface with a static CSV file containing the receipts and decrypted votes in the back-end. Note that no Private Information Retrieval mechanism was used, and hence a malicious ROBB could link verification IP addresses to the votes [9]. Additionally, the auditor could obtain the whole static CSV file.

The main difference between the ROBB and RABB is that the former is created in one action, but the latter allows continuous addition of items.

In our protocol, the two are used in different stages. The read-append bulletin board is used to commit the voting actions during the election period, whereas the read-only bulletin board is used for post-election verification queries.

Assumption 4 (Existence of voter certification). We assume that the election organizer has prior knowledge of every eligible voter and that there is an independent certification authority which provides confirmation of their identities.

This assumption can be implemented in several ways depending on the specific environment. Some jurisdictions may have a constantly updated voter registry, some may rely on the voters registering themselves at the election organizers some time before the elections are set up. In our application scenario, the latter was the case. Since the voting period was only one day, the voters' list was kept static throughout that period.

Assumption 5 (Existence of pre-channel to voters). We assume the existence of an authenticated and secure pre-channel between every voter and the election organizer.

The pre-channel could be initialized using a national PKI if it exists. In the alternate case other methods like email and SMS channel could be used. It is possible to add additional technical measures to increase the privacy of the channel. For example, it is possible to use email over encrypted channel, mutually authenticating the servers and signing emails using DKIM [20].

3.4 Assumptions on the attack model

To simplify the modelling of the protocol in EasyCrypt, we need to define additional assumptions which restrict the capabilities of the adversaries.

Assumption 6 (Perfectly binding signature scheme). We define an abstract signature scheme that is assumed to be perfectly binding. This means that for any generated signature s on a message pt there does not exist another message pt' such that verification succeeds.

Otherwise, it would be possible to construct another message for which some signature is valid. This assumption is required for achieving auditability and post-election verifiability.

Assumption 7 (Perfectly hiding encryption scheme). We assume that the defined encryption scheme is perfectly hiding. This means that given a ciphertext ct on a message pt, no adversary can learn the encrypted message.

In practice, this assumption does not hold. For example, in case of ElGamal encryption scheme, the naïve success probability of polynomial-time adversary would be $\frac{1}{|\Omega|}$. The success probability could be increased by using systematic attacks against the encryption scheme.

However, as currently known solutions for breaking ElGamal security over safe group parameters give the adversary only negligible success probability, then for the ease of modelling the protocol, we use the aforementioned assumption.

Assumption 8 (No cooperation between adversarial parties). We assume that no two adversarial parties cooperate.

As currently the approach for proving the security of the protocol is to prove the security of the protocol for different adversaries, then allowing cooperating adversaries would exponentially grow the amount of required proofs. Thus, this assumption keeps the number of different security proofs manageable.

As a result, the corresponding security claims do not hold in the case where malicious VotingClient and malicious VotingServer cooperate.

Assumption 9 (Perfect randomness). We assume that if any honest party queries for a random value, then the value is sampled from a uniform distribution.

The assumption ensures that the probability of colliding values between different parties in the protocol is negligible.

3.5 Protocol phases

To simplify the presentation of the protocol, we will divide it into four interconnected subprotocols. The first subprotocol defines the distribution of keys between different parties. The second subprotocol describes the voting process and voting receipt generation. The third subprotocol describes how votes are decrypted and published on the ROBB. Finally, the fourth subprotocol describes vote verification by the voter.

Distribution of keys: The key generation and distribution of the encryption key pair is done by KeyHolder. Before the election period starts, it generates the key pair $(\mathsf{ek}, \mathsf{dk})$ and publishes the public part ek to the VotingClient and VotingServer. After the voting period has closed, it sends the private part dk of the key to the TallyServer.

The CertificationAuthority generates its own signing key pair $(\mathsf{sk}_{\mathrm{CA}}, \mathsf{vk}_{\mathrm{CA}})$ and distributes the verification key $\mathsf{vk}_{\mathrm{CA}}$ to the VotingServer and TallyServer.

Every voter needs her own key for signing the ballot. Thus, a signing key pair $(\mathsf{sk}_{\mathrm{VTR}}, \mathsf{vk}_{\mathrm{VTR}})$ has to be generated for every voter. To prove the identities of the voters, CertificationAuthority signs the corresponding verification key to obtain the certificate c_{VTR}. The signing keys are delivered to the voters over an encrypted and authenticated channel. The certification authority stores only the voter's verification key $\mathsf{vk}_{\mathrm{VTR}}$ and certificate c_{VTR}.

VotingServer generates a signing key pair $(\mathsf{sk}_{\mathrm{VS}}, \mathsf{vk}_{\mathrm{VS}})$ that is used for signing the receipts. The public part $\mathsf{vk}_{\mathrm{VS}}$ is sent to the VotingClient to allow for verifying the validity of the receipts.

Vote submission and receipt generation: In order to satisfy Requirement 4, we need to provide the voter with a way to verify that the ballot actually has been included in the final tally. Note that the voter was not required to obtain cast-as-intended assurance during the online protocol phase. Rather the requirements stated in Section 2 aim at generating a receipt that can be later used for post-election verification. Essentially, the receipt will be the query key to request the voter's decrypted choice from the ROBB.

There are two parties that could generate the receipt – VotingClient and VotingServer. However, a malicious VotingServer or several collaborating instances of VotingClient could then manipulate the votes and generate receipts that would leave the voter with impression that everything is fine (see below in this Section for more detailed attack descriptions).

Hence, we decided to introduce a design where both the VotingClient and VotingServer would be generating a part of the receipt. As the protocol will be using a mix-net and provable decryption based on ElGamal encryption, our goal was to also make the receipt to be an ElGamal cryptogram so that it would go through the mix-net and would allow for a decryption proof.

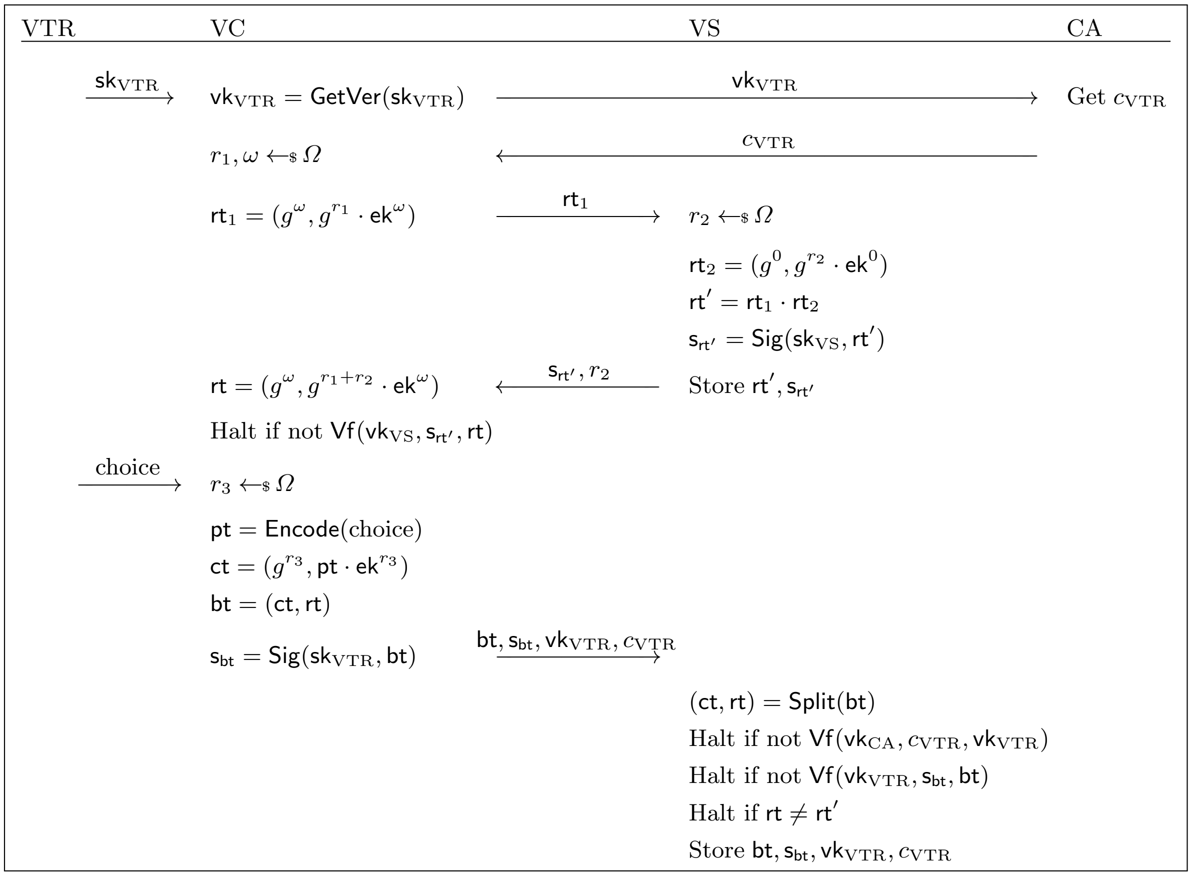
These goals are achieved by our vote submission and receipt generation protocol in Figure 1. The VotingClient encodes voter's choice as a plaintext pt and encrypts it to get the ElGamal cryptogram ct.

To form the receipt, the VotingClient and VotingServer will generate random exponents r_1 and r_2, respectively. The receipt to be used during the verification will be $\mathsf{R} = g^{r_1+r_2}$. However, there are still potential attacks to be taken into account while doing so.

If the VotingServer is malicious then by knowing the input r_1 from the VotingClient, it could provide its part r_2 in the receipt in a way that the verifying voter would accept the receipt even if a wrong ballot was counted in the tally. For mitigation, we encrypt the VotingClient's part of the receipt as rt_1 and only send this to the VotingServer. As our aim is to produce an ElGamal-encrypted receipt rt anyway, sending a homomorphic partial encryption is sufficient. The VotingServer then deterministically encrypts its value r_2 to obtain rt_2 and combines the encrypted parts to obtain the encryption $\mathsf{rt}' = \mathsf{rt}_1 \cdot \mathsf{rt}_2$.

On the other hand, a malicious set of VotingClient instances could cooperate during the receipt generation. The instances could construct a choice-receipt database, where for every possible choice there is a valid receipt R. In this case, the voter would be presented with R, although the VotingClient has cast a ballot for another choice with different receipt R', which is then discarded. Thus, verifying the vote by the voter would incorrectly succeed during post-election verification period. To mitigate this attack, we need to provide input from the VotingServer such that VotingClient could not modify its part. Thus, VotingServer signs the combined information rt' (which should equal rt) to prevent modifications by VotingClient. The resulting protocol is depicted in Figure 1.

The VotingServer checks that the voter has not already cast a vote before forwarding the ballot to the RABB. After submitting the ballot, it is stored on the RABB and the VotingClient is sent the storage index. The protocol is illustrated in Figure 2.



VTR
VC
VS
CA
$\xrightarrow{\mathsf{sk}_{\mathrm{VTR}}}$
$\mathsf{vk}_{\mathrm{VTR}} = \mathsf{GetVer}(\mathsf{sk}_{\mathrm{VTR}})$
$\xrightarrow{\mathsf{vk}_{\mathrm{VTR}}}$
Get c_{VTR}
$\xleftarrow{c_{\mathrm{VTR}}}$
$r_1, \omega \leftarrow_{\$} \Omega$
$\mathsf{rt}_1 = (g^{\omega}, g^{r_1} \cdot \mathsf{ek}^{\omega})$
$\xrightarrow{\mathsf{rt}_1}$
$r_2 \leftarrow_{\$} \Omega$
$\mathsf{rt}_2 = (g^0, g^{r_2} \cdot \mathsf{ek}^0)$
$\mathsf{rt}' = \mathsf{rt}_1 \cdot \mathsf{rt}_2$
$\mathsf{s}_{\mathsf{rt}'} = \mathsf{Sig}(\mathsf{sk}_{\mathrm{VS}}, \mathsf{rt}')$
$\xleftarrow{\mathsf{s}_{\mathsf{rt}'}, r_2}$
$\mathsf{rt} = (g^{\omega}, g^{r_1+r_2} \cdot \mathsf{ek}^{\omega})$
Store $\mathsf{rt}', \mathsf{s}_{\mathsf{rt}'}$
Halt if not $\mathsf{Vf}(\mathsf{vk}_{\mathrm{VS}}, \mathsf{s}_{\mathsf{rt}'}, \mathsf{rt})$
$\xrightarrow{\text{choice}}$
$r_3 \leftarrow_{\$} \Omega$
$\mathsf{pt} = \mathsf{Encode}(\text{choice})$
$\mathsf{ct} = (g^{r_3}, \mathsf{pt} \cdot \mathsf{ek}^{r_3})$
$\mathsf{bt} = (\mathsf{ct}, \mathsf{rt})$
$\mathsf{s}_{\mathsf{bt}} = \mathsf{Sig}(\mathsf{sk}_{\mathrm{VTR}}, \mathsf{bt})$
$\xrightarrow{\mathsf{bt}, \mathsf{s}_{\mathsf{bt}}, \mathsf{vk}_{\mathrm{VTR}}, c_{\mathrm{VTR}}}$
$(\mathsf{ct}, \mathsf{rt}) = \mathsf{Split}(\mathsf{bt})$
Halt if not $\mathsf{Vf}(\mathsf{vk}_{\mathrm{CA}}, c_{\mathrm{VTR}}, \mathsf{vk}_{\mathrm{VTR}})$
Halt if not $\mathsf{Vf}(\mathsf{vk}_{\mathrm{VTR}}, \mathsf{s}_{\mathsf{bt}}, \mathsf{bt})$
Halt if $\mathsf{rt} \neq \mathsf{rt}'$
Store $\mathsf{bt}, \mathsf{s}_{\mathsf{bt}}, \mathsf{vk}_{\mathrm{VTR}}, c_{\mathrm{VTR}}$

Fig. 1. Vote submission and receipt generation protocol.

As noted above, the receipt R is the value $g^{r_1+r_2}$ where r_1 is generated by the VotingClient and r_2 is generated by the VotingServer. For verification, R is encoded using ASCII printable characters and displayed to the voter. For auditing, the tuple $(i, \mathsf{bt}, \mathsf{s}_{\mathsf{bt}}, r_1, r_2, \mathsf{s}_{\mathsf{rt}'})$ is provided to the voter in the form of a QR-code.

Vote decryption: Once the voting period has ended, the ballots are stripped of identifying information and sent to the TallyServer. To further protect privacy during auditing, the ballots are shuffled using a mix-net and then provably decrypted. In that particular election, a mix-net based on a proof of a shuffle by Terelius and Wikström [27] was used.

The decryption process and result publishing are shown in Figure 3.

Post-election vote verification: The voter can use any modern web browser to query the ROBB by the receipt R to obtain the choice and compare the result to her cast choice. We note that for providing information-theoretic privacy

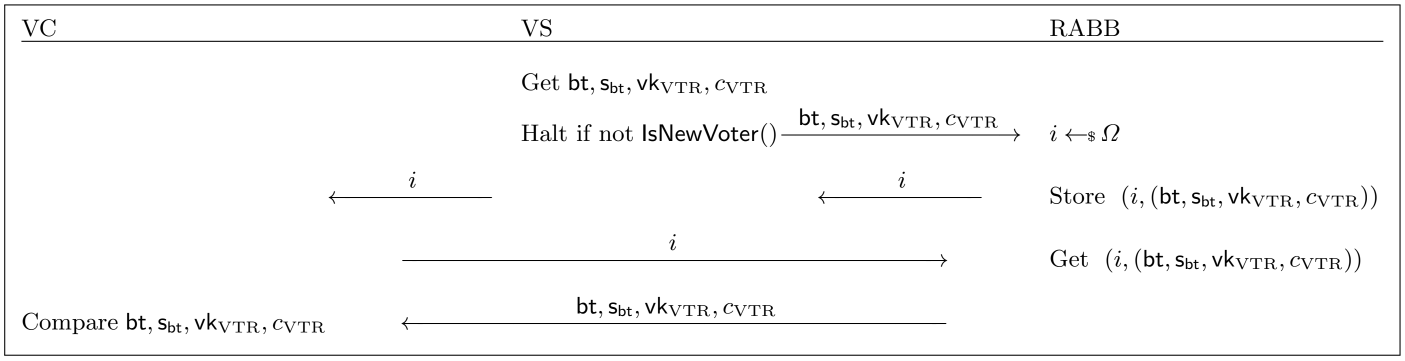

VC
VS
RABB
Get $\mathsf{bt}, \mathsf{s}_{\mathsf{bt}}, \mathsf{vk}_{\mathrm{VTR}}, c_{\mathrm{VTR}}$
Halt if not $\mathsf{IsNewVoter}()$
$\mathsf{bt}, \mathsf{s}_{\mathsf{bt}}, \mathsf{vk}_{\mathrm{VTR}}, c_{\mathrm{VTR}}$
$i \leftarrow_{\$} \Omega$
i
i
Store $(i, (\mathsf{bt}, \mathsf{s}_{\mathsf{bt}}, \mathsf{vk}_{\mathrm{VTR}}, c_{\mathrm{VTR}}))$
i
Get $(i, (\mathsf{bt}, \mathsf{s}_{\mathsf{bt}}, \mathsf{vk}_{\mathrm{VTR}}, c_{\mathrm{VTR}}))$
$\mathsf{bt}, \mathsf{s}_{\mathsf{bt}}, \mathsf{vk}_{\mathrm{VTR}}, c_{\mathrm{VTR}}$
Compare $\mathsf{bt}, \mathsf{s}_{\mathsf{bt}}, \mathsf{vk}_{\mathrm{VTR}}, c_{\mathrm{VTR}}$

Fig. 2. Ballot storage protocol

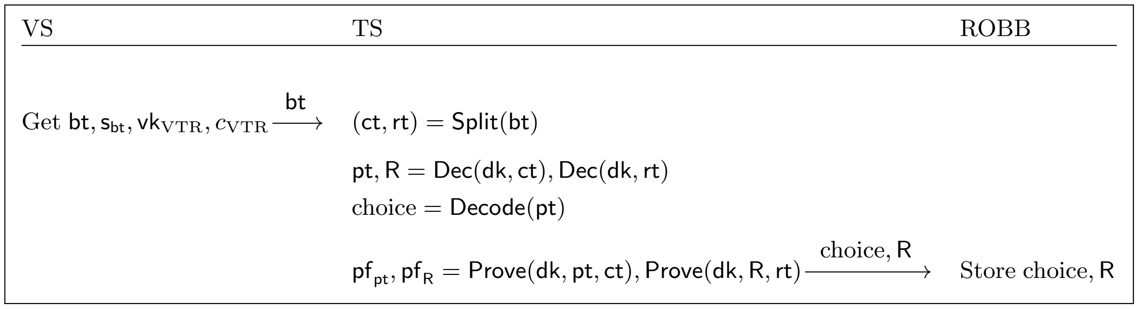

VS
TS
ROBB
Get $\mathsf{bt}, \mathsf{s}_{\mathsf{bt}}, \mathsf{vk}_{\mathrm{VTR}}, c_{\mathrm{VTR}}$
bt
$(\mathsf{ct}, \mathsf{rt}) = \mathsf{Split}(\mathsf{bt})$
$\mathsf{pt}, \mathsf{R} = \mathsf{Dec}(\mathsf{dk}, \mathsf{ct}), \mathsf{Dec}(\mathsf{dk}, \mathsf{rt})$
$\text{choice} = \mathsf{Decode}(\mathsf{pt})$
$\mathsf{pf}_{\mathsf{pt}}, \mathsf{pf}_{\mathsf{R}} = \mathsf{Prove}(\mathsf{dk}, \mathsf{pt}, \mathsf{ct}), \mathsf{Prove}(\mathsf{dk}, \mathsf{R}, \mathsf{rt})$
choice, R
Store choice, R

Fig. 3. Ballot decryption and publishing protocol

of the vote, we would need to transmit the whole content of ROBB to the VotingClient [9].

4 Requirement implementation and verification

Requirement 1 (Eligibility). This requirement is fulfilled by organizational measures. The trusted CertificationAuthority makes sure that only eligible voters are given access to the signing keys, and auditors verify the signed votes on RABB to make sure that only the votes of eligible voters are on the bulletin board.

Requirement 2 (Tally integrity). Tally integrity is assured by auditing the entries on the bulletin boards. Due to the properties of the bulletin boards, it is not possible to remove entries, and all the cast ballots must be stored there.

All of the tallied votes are published on the ROBB along with the receipts. Voters are able to use any Internet connected device that contains a browser to query the ROBB and check if their vote was tallied. Thus, if the ballot was not transferred to the TallyServer or not tallied, the voter would detect the absence of the ballot. On the other hand, if a server would have acted maliciously by not sending a vote to the bulletin board, the VotingClient would have detected this during the ballot storage protocol.

Requirement 3 (Ballot well-formedness). The encoding function Encode takes the ASCII-encoded choice, considers its byte form and interprets it as an integer. If the length of the choice is fixed, there exists an ElGamal group with large enough subgroup such that there is an injective mapping. The decoding function Decode applies the inverse operations. If the decoding function outputs an error message on an incorrectly encoded ballot, the requirement is satisfied.

Requirement 4 (Inclusion verifiability). While casting the vote the VotingClient and VotingServer generate a receipt and the VotingClient displays it to the voter. Additional cryptographic information used to generate the receipt is given to the voter in the form of a QR-code.

The encrypted receipt is added to the ballot and sent to the voting system. The tallied vote with the corresponding receipt is published in the ROBB. The voter can use her receipt to query ROBB to check if her vote was tallied. In case of a mismatch or a missing vote the voter can appeal by using the info on the stored QR-code.

Requirement 5 (Liability provability). According to the receipt storage protocol, the voter would have access to $\mathsf{R}, (i, \mathsf{bt}, \mathsf{s_{bt}}, r_1, r_2, \mathsf{s_{rt'}})$. The voter obtains the pair $(\text{choice}, \mathsf{R})$ from the ROBB using R as the key. If the value choice differs from the voter's choice, then the voter may appeal the result.

During the appeal process, the following steps are performed together by the election organizer and voter to detect the liable party:

1. The values $\mathsf{bt}, \mathsf{s_{bt}}, \mathsf{vk}_{\mathrm{VTR}}, c_{\mathrm{VTR}}$ are fetched from RABB using i. The fetched values bt and $\mathsf{s_{bt}}$ are compared to the corresponding values on the receipt.
2. The voter's signing key is verified using $\mathsf{vk}_{\mathrm{CA}}$, c_{VTR} and $\mathsf{vk}_{\mathrm{VTR}}$.
3. The ballot signature is verified using $\mathsf{vk}_{\mathrm{VTR}}$, $\mathsf{s_{bt}}$ and bt.
4. The ballot is split into ciphertexts $\mathsf{Split}(\mathsf{bt}) = (\mathsf{ct}, \mathsf{rt})$.
5. The signature on the receipt is verified using $\mathsf{vk}_{\mathrm{VS}}$, $\mathsf{s_{rt'}}$ and rt.
6. The receipt is decrypted as $\mathsf{R}' = \mathsf{Dec}(\mathsf{dk}, \mathsf{rt})$. The receipt is compared to $g^{r_1+r_2}$.
7. The ciphertext is decrypted as $\mathsf{pt} = \mathsf{Dec}(\mathsf{dk}, \mathsf{ct})$ and decoded as $\text{choice} = \mathsf{Decode}(\mathsf{pt})$. The choice is compared to one stored on ROBB.

We consider how every step allows to determine the cheating party:

1. As illustrated in Figure 2, the VotingClient obtains the values $\mathsf{bt}, \mathsf{s_{bt}}, \mathsf{vk}_{\mathrm{VTR}}, c_{\mathrm{VTR}}$ from RABB using the index i during ballot storage.
   If the step fails, then either VotingClient skipped obtaining the values from RABB or one of VotingClient and voter must have modified the receipt.
2. As illustrated in Figure 1, VotingServer halts if $\mathsf{Vf}(\mathsf{vk}_{\mathrm{CA}}, c_{\mathrm{VTR}}, \mathsf{vk}_{\mathrm{VTR}})$ fails. Thus, if the same operation fails during appeal then the election organizer cheated.

3. Similarly to previous step, the values have been verified during vote submission stage and if the step fails, then the election organizer has been cheating.
4. The ballot was split during the vote submission stage and if the step fails, then the VotingServer has behaved incorrectly. Thus, the election organizer cheated.
5. The signature is also verified by the VotingClient during ballot submission and if the step fails, then the signature $\mathsf{s}_{\mathsf{rt}'}$ is invalid on the receipt. This means that either the VotingClient or voter have modified the receipt.
6. As the value r_1 is provided by the VotingClient and it verified the correctness of the signature $\mathsf{s}_{\mathsf{rt}'}$ during the vote submission stage, then the failed comparison indicates that either VotingClient or voter modified the values r_1 or r_2 on the receipt.
7. If the comparison fails, then the result of decrypting the same ciphertext ct twice is inconsistent. It is possible to verify the proof $\mathsf{pf}_{\mathsf{pt}}$ to confirm that the election organizer has cheated.

In conclusion, if the Steps 2, 3, 4 or 7 fail, the election organizer cheats. If the Steps 1, 5 or 6 fail, then either VotingClient or the voter is cheating.

Requirement 6 (Voter's privacy). The KeyHolder sends the decryption key to TallyServer after the election period has ended. Furthermore, the encrypted ballots are stripped of identifying information and additionally shuffled. Thus, there is no preliminary access to the encrypted ballots, and after the election the voter privacy is preserved.

In the previous description we saw that in case the voter challenges the receipt, then her ballot is decrypted and the privacy is void.

Requirement 7 (Eligibility auditability). The auditor has a view of the ballots stored on the RABB and the ballots sent to the TallyServer for decryption. He checks that the signature s_{bt} for every ballot bt verifies using $\mathsf{vk}_{\mathrm{VTR}}$, and that the certificate c_{VTR} for $\mathsf{vk}_{\mathrm{VTR}}$ verifies using $\mathsf{vk}_{\mathrm{CA}}$. He then checks that the ballots sent for decryption correspond to the ballots stored on RABB.

Requirement 8 (Decryption auditability). As defined in Subsection 3.2, ElGamal encryption scheme is used. ElGamal decryption can be implemented in a provable way [1]. The auditor can retrieve the proofs $\mathsf{pf}_{\mathsf{pt}}$ and pf_{R} of correct decryption generated during the protocol illustrated in Figure 3. As the auditor can verify these proofs, this requirement is satisfied.

Requirement 9 (Privacy-preserving auditing). Before transmitting the ballots to the TallyServer, a shuffling mix-net is applied. The auditor sees shuffled ballots and the corresponding decryptions, but can not map shuffled ballots to unshuffled ballots, thus keeping the privacy of the voters.

5 Modelling the protocol with EasyCrypt

5.1 Formal verification

Creating secure cryptographic primitives is difficult. When primitives are combined to create protocols, the complexity increases even further, which may introduce subtle vulnerabilities. A classic example is the Needham–Schroeder protocol, where a severe vulnerability was found 15 years after the original paper was published [22,7].

One way to avoid such problems is to state formal security criteria and use computing tools to verify them. Unfortunately, such tools still have a long way to go before they can be used to verify all the desired properties of the protocols. Hence in practice we usually take a combined approach, modelling the protocol, formally checking as many properties as possible, and presenting heuristic arguments for the others. This paper follows the same path.

This far, majority of the formal proof attempts for cryptographic protocols have been made in the symbolic model, using the ProVerif software suite (e.g. [19], [14], [3]). Computational model is arguably closer to reality, but also more complex. This is why the tools supporting the computational approach (like EasyCrypt [5]) have matured more recently. Nevertheless, first proofs of privacy properties have already been given using EasyCrypt by Cortier *et al.* [11]. In this paper, we will be using EasyCrypt to model the protocol and to check a few integrity and verifiability properties.

EasyCrypt uses a probabilistic While language for modelling the primitives and protocols [6]. The use of an imperative language supports the choice of modelling the protocol with EasyCrypt.

5.2 Formal model of the protocol

Before modelling the voting protocol we had to decide which abstraction level to use. A detailed protocol description would allow to verify the details of the desired security properties. However, proving the security of implementation details could be complex or even impossible with EasyCrypt. For example, the current EasyCrypt version does not have a framework to support mix-nets and therefore we did not include mix-nets in our protocol model [11]. Hence, in order to focus on the desired security claims we decided to abstract away several details when describing the protocol in EasyCrypt. Thus, we modelled the protocol run with one voter that did not include mix-net and did not include provable decryption. The resulting EasyCrypt code is posted to a GitHub repository.[6]

After modelling the protocol we did a sanity check to make sure that the protocol runs as intended. We refer to the EasyCrypt lemma `votingCorrectness` to show that vote verification succeeds with probability 1 in case all parties follow the protocol. Then we showed that a malicious VotingClient who randomly changes the vote of the voter will remain undetected with probability $\frac{1}{q}$, where q is the number of candidates. This holds in case the voter uses the receipt for verification. Thus, the malicious client is not detected by vote verification when the randomly sampled vote matches the vote that was cast by the voter. Finally, we tested that neither the VotingServer nor TallyServer are able to remove votes without getting caught. This was formalized with the lemmas `voteDeletingVServer` and `voteDeletingTally`. For more details about

[6] `https://github.com/krips/uvoting`

the formalization of the lemmas, see the linked EasyCrypt source code in the gist repository.

5.3 Clash attack

While modelling the receipt protocol, we found that in the described form it was susceptible to a clash attack. Clash attack allows the malicious voting software to modify votes by reusing receipts such that different voters who vote for the same candidate get the same receipt [21]. The current voting protocol was designed to avoid clash attacks, but the problem occurs due to the way how the user interaction is implemented. Namely, the receipt is given to the voter *after* she has already made the choice in the voting software.

Thus, the malicious VotingClient could first collect the receipts of different voters and then check if a receipt for the voter's current choice is available. In case the receipt for the current choice is available, the malicious software could change the vote and display a receipt that corresponds to a vote that was cast by another voter.

A proof of concept adversarial voting client was observed during the event [2]. The site notified the user against the dangers of Internet voting and did not perform a coordinated attack.

We modelled the proof of concept attack in EasyCrypt and showed that in case of two voters who vote for the same candidate the malicious VotingClient can change the vote of the second voter by reusing the receipt that was given to the first voter. The description of the attack is given in the attached EasyCrypt code, more specifically in the module `ReceiptForgery` and in lemma named `clashAttack`.

The problem can be resolved by slightly modifying the way how the messages are shown in the user interface of the VotingClient. The voter should receive the receipt before she enters the vote to the VotingClient as then the malicious software has to pick the receipt before the voter has selected the candidate.

5.4 Experiences with EasyCrypt

We learned quite a lot by using EasyCrypt. In order to share the experience, we will discuss some of the observations and hope that they are valuable to the developers and other users of the tool.

Proving security in the computational model gives more precise results compared to the symbolic model, but the additional price to pay is increased complexity. EasyCrypt allows to specify complex algebraic protocols relatively easily, but proving the desired security properties is not straightforward. When applying a proof tactic, multiple subgoals may appear and all of them have to be proven in order to move on with the proof. It is often the case that proving simple subgoals takes more time than proving the security aspects.

Goals can be proven in EasyCrypt either by manually combining existing proof tactics or by using an automated SMT solver Alt-Ergo [10]. A subgoal may seem trivial, but if the SMT solver is not able to prove it, one has to start looking through the extensive library of theory files to find lemmas and axioms that can be used to close the subgoal. One way to simplify this process is to use the built-in *search* command which helps to find all lemmas and axioms that correspond to a given pattern. For example, if we would need to find properties about accessing map elements, then we could use the command $search(\_.[\_])$, which would return all lemmas and axioms that use the corresponding operator.

As noted above, specifying the voting protocol in EasyCrypt language was relatively easy. The main question was on how to model communication between the parties and for that we created additional functions for the parties and a separate module which modelled the execution of the protocol with a single voter. The first difficulties emerged when we had to choose which of the existing theories to use. EasyCrypt is a work in progress which means that its theories and documentation are changing. For some theories there were multiple versions available in order to provide backward compatibility. Thus, we came across a situation where we had to choose if we would like to use a new theory or an old one to model our protocol. Both options seemed to work, but we decided to use the older version as it had helpful proof examples.

The next difficulty appeared when proving a simple property about the success probability of a malicious VotingClient. It appeared that a specific version of the *seq* tactic was not documented in the reference manual, and thus we had to find other means to understand how the probability parameters could be used in connection with the *seq* tactic.

We found a small code dependency issue when setting up a fresh EasyCrypt install from the main branch. Namely, two weeks after starting to model the protocol it appeared that an old EasyCrypt theory had been removed from the main branch. Thus, we had to either rewrite some of the code or to pin EasyCrypt to a specific commit. Rewriting the code was easy, but modifying the underlying modules will usually break the proofs.

This brings us to the second lesson. We had to expand the protocol description once some of the proofs were already finished, and this meant that a large part of the proofs had to be updated. Changes to the underlying modules may break the proofs due to change of invariants, shifting of line numbers and added conditions. Thus, updating proofs might not be difficult, but it can still be time consuming when multiple proofs have to be modified.

6 Related work

The novelty of our protocol comes from the way how the receipts are generated. The receipt generation requires the voting client and server to work together and check that the other party would adhere to the protocol. If the voter uses the receipt to verify the vote, she can be sure that both the voting client and the server did not deviate from the receipt generation protocol. In case the receipt verification fails, it is possible to determine which of the two parties acted maliciously.

There are multiple voting schemes which use receipts, but the most similar to our scheme is the Selene voting protocol created by Peter Ryan *et al.* [25]. Selene aims to provide transparent verifiability and coercion-mitigation at the same time. It uses tracking numbers for voters, posted to a public bulletin board along with the votes. Thus, a voter can take her tracking number and check from the public bulletin board if the vote corresponding to the tracking number is the one that was cast. Pedersen commitments are used for generating a trapdoor for the tracking numbers. This allows the voter to use the trapdoor to lie to a coercer. In addition, to prevent coercion, the tracking numbers are revealed after the vote has already been cast.

Comparing Selene to our protocol, we clearly see that Selene was designed to be coercion resistant while our protocol was designed for elections where coercion is not an issue. However, resolving disputes involving the receipts is non-trivial in Selene, while in our protocol shared cryptographic information helps to reveal who is to blame.

When the voter is not participating in the receipt generation process, it is much harder to find out the misbehaving party. This is illustrated in the paper by Küsters *et al.* [21], which described how it is possible to attack the verifiability of multiple e-voting systems by reusing the receipts. The paper showed that this kind of an attack was possible against ThreeBallot and VAV [24], but also against a version of Helios [1] using aliases. However, if the voter and the voting server jointly generate the receipt as in our scheme, clash attacks can be mitigated by displaying the receipt value to the voter before the voter makes her choice. After the receipt value has been shown to the voter, the attacker is not able to use someone else's vote and receipt to replace the vote without getting caught.

7 Conclusion

In this paper we presented a new Internet voting protocol that supports post-election vote verification using a receipt that is jointly generated by mutually distrusting voting application and voting server. We showed how the proposed cryptographic scheme in conjunction with organizational measures can be used to meet all the required integrity, verifiability and auditability requirements. In addition, we modelled the protocol in EasyCrypt and formally proved some of the integrity properties.

The protocol was used to run a Republican caucus vote in the state of Utah in March 2016. Altogether, 24486 votes were cast using our system. A certain amount of politically emotional discussions concerning security issues was, of course, expected, but this far the discussion has been lacking technical details and precise security claims. We hope that our paper will help to fill some gaps and clear some of the misunderstandings.

Acknowledgements

The research leading to these results has received funding from the Estonian Research Council under Institutional Research Grant IUT27-1 and the European Regional Development Fund through the Estonian Centre of Excellence in ICT Research (EXCITE) and the grant number EU48684.

References

1. Ben Adida. Helios: Web-based Open-Audit Voting. In *USENIX security symposium*, volume 17, pages 335–348, 2008.
2. Andrew Appel. Internet Voting, Utah GOP Primary Election. `https://freedom-to-tinker.com/2016/03/22/internet-voting-utah-gop-primary-election`, 2016.
3. Michael Backes, Catalin Hritcu, and Matteo Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *CSF'08*, pages 195–209. IEEE, 2008.
4. Jordi Barrat, Ben Goldsmith, David Jandura, John Turner, and Rakesh Sharma. Internet voting and individual verifiability: the Norwegian return codes. *Electronic Voting*, pages 274–283, 2012.
5. Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *CRYPTO'11*, volume 6841 of *LNCS*, pages 71–90. Springer, 2011.
6. Gilles Barthe, Benjamin Grégoire, César Kunz, Yassine Lakhnech, and Santiago Zanella Béguelin. Automation in computer-aided cryptography: Proofs, attacks and designs. In *Certified Programs and Proofs*, pages 7–8, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
7. R. K. Bauer, T. A. Berson, and R. J. Feiertag. A key distribution protocol using event markers. *ACM Trans. Comput. Syst.*, 1(3):249–255, August 1983.
8. M. Bernhard, J. Benaloh, J. A. Halderman, R. L. Rivest, P. Y. A. Ryan, P. B. Stark, V. Teague, P. L. Vora, and D. S. Wallach. Public Evidence from Secret Ballots. *ArXiv e-prints*, July 2017.
9. B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50, Oct 1995.
10. Sylvain Conchon, Évelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. CC(X): Semantical combination of congruence closure with solvable theories. In *Post-proceedings of SMT 2007*, volume 198(2) of *Electronic Notes in Computer Science*, pages 51–69. Elsevier Science Publishers, 2008.
11. Véronique Cortier, Benedikt Schmidt, Constantin Catalin Dragan, Pierre-Yves Strub, Francois Dupressoir, and Bogdan Warinschi. Machine-checked proofs of privacy for electronic voting protocols. In *IEEE S&P'17*. IEEE Computer Society Press, 2017.
12. Chris Culnane and Steve Schneider. A peered bulletin board for robust use in verifiable voting systems. In *Computer Security Foundations Symposium (CSF), 2014 IEEE 27th*, pages 169–183. IEEE, 2014.
13. Stéphanie Delaune, Steve Kremer, and Mark D. Ryan. Coercion-resistance and receipt-freeness in electronic voting. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 28–42, 2006.

14. Stéphanie Delaune, Steve Kremer, and Mark D. Ryan. Verifying properties of electronic voting protocols. In *Proceedings of the IAVoSS Workshop On Trustworthy Elections (WOTE'06)*, pages 45–52, June 2006.
15. Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.
16. Sven Heiberg, Tarvi Martens, Priit Vinkel, and Jan Willemson. Improving the Verifiability of the Estonian Internet Voting Scheme. In *E-Vote-ID*, volume 10141 of *LNCS*. Springer.
17. Sven Heiberg and Jan Willemson. Verifiable Internet Voting in Estonia. In *6th International Conference on Electronic Voting: Verifying the Vote, EVOTE 2014, Lochau / Bregenz, Austria, October 29-31, 2014*, pages 1–8, 2014.
18. Cameron F. Kerry and Patrick D. Gallagher. Digital Signature Standard (DSS), 2013.
19. Steve Kremer and Mark Ryan. Analysis of an electronic voting protocol in the applied pi calculus. In *ETAPS 2005*, volume 3444 of *LNCS*, pages 186–200. Springer, 2005.
20. Murray Kucherawy, Dave Crocker, and Tony Hansen. DomainKeys Identified Mail (DKIM) Signatures. RFC 6376, September 2011.
21. Ralf Kusters, Tomasz Truderung, and Andreas Vogt. Clash Attacks on the Verifiability of E-Voting Systems. In *IEEE S&P'12*, pages 395–409. IEEE Computer Society, 2012.
22. Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, December 1978.
23. Pierre Noizat. Blockchain electronic vote. In David Lee Kuo Chuen, editor, *Handbook of Digital Currency*. Elsevier, 2015. Chapter 22.
24. Ronald L. Rivest and Warren D. Smith. Three Voting Protocols: ThreeBallot, VAV, and Twin. In *Proceedings of USENIX/ACCURATE Electronic Voting Technology (EVT)*, 2007.
25. Peter Y. A. Ryan, Peter B. Rønne, and Vincenzo Iovino. Selene: Voting with transparent verifiability and coercion-mitigation. In *Financial Cryptography and Data Security*, volume 9604 of *LNCS*, pages 176–192. Springer, 2016.
26. Smartmatic. Utah Republican Party 2016 Preference Caucus - case study. `http://www.smartmatic.com/uploads/tx_news/CS_UTAH_2016_ENG.pdf`, 2016.
27. Björn Terelius and Douglas Wikström. Proofs of restricted shuffles. In *AFRICACRYPT 2010*, volume 6055 of *LNCS*, pages 100–113. Springer, 2010.