# Creating a Decryption Proof Verifier for the Estonian Internet Voting System

Jan Willemson
jan.willemson@cyber.ee
Cybernetica
Tartu, Estonia

## ABSTRACT

This paper describes the efforts made for and lessons learnt from creating a decryption proof verifier for the Estonian IVXV Internet voting system. Our main conclusion is that cryptographic protocols aiming at providing transparency through verifiability should also take into account a non-functional requirement of low implementation complexity. We identify several steps of the verification protocol that could be made easier to implement without sacrificing security. A side-product of our effort is a fully functional IVXV decryption proof verifier written in Go that we used during the latest Estonian parliamentary elections of March 2023.

## CCS CONCEPTS

• **Security and privacy** → *Cryptography*; *Software and application security*; • **Applied computing** → **Voting / election technologies**.

**ACM Reference Format:**
Jan Willemson. 2023. Creating a Decryption Proof Verifier for the Estonian Internet Voting System. In *The 18th International Conference on Availability, Reliability and Security (ARES 2023), August 29–September 01, 2023, Benevento, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3600160.3605467

## 1 INTRODUCTION

Casting a legally binding vote over Internet has been possible in Estonia since 2005 [10]. Originally, a simple double-envelope protocol was in use, where the role of the inner, anonymous envelope was played by encryption under the system's public key. The outer, identifiable envelope was in turn implemented by signing the cryptogram with a voter's electronic identity token [6]. The signed container was sent to the Vote Collector server (VC) by the voting application, and this was the end of the voting process from the voter's perspective.

This simple protocol served its purpose well until the parliamentary elections of 2011 when a student demonstrated a proof-of concept ballot-manipulating software [6]. The attack relied on the fact that up to 2011, there was no way for the voter to establish whether or how her vote was actually received by the VC. By the next elections in 2013, this issue was addressed by adding the individual vote verification mechanism using an independent mobile device [8].

In 2014, Springall *et al.* presented an analysis of Estonian Internet voting, criticizing its poor system-side auditability properties [14]. In 2017, a new version of the protocol suite (code-named IVXV) was released, enabling much better independent verifiability of several critical system components [7].

A vote registration service was introduced, making it impossible for the VC to drop votes without detection. At the other end of the process, non-interactive zero-knowledge proofs of correct decryption were introduced to the vote decryption service. Also, a shuffling re-encryption mix-net was added before decryption in order to facilitate privacy-preserving auditing of the whole tallying process. As a result, a significant part of the process of certifying correct operation of the critical components was transformed from physical observation to data analysis.

The mix-net and the vote decryption server are the two major components in IVXV that produce independently verifiable cryptographic proofs. For both of them, there are vendor-supplied verifiers that are run by the election organizer to validate the results. However, a wider goal targeted by introducing third-party auditability mechanisms is to allow third parties to perform this data analysis using their own tools.

For the mix-net, IVXV uses Douglas Wikström's Verificatum[1]. Being one of the best established products in its field, there are several projects to produce proof verifiers for it. Of course, the Verificatum software package itself provides a verification application, and so does IVXV.

Additionally, there are several vendor-independent projects. By far the most mature of them is the formally verified verifier by Haines *et al.* [5]. There are also a few student projects with varying degree of maturity[2][3]. None of the vendor-independent verification applications has yet been used in the Estonian elections, but, from the technical point of view, deploying e.g. the verifier by Haines *et al.* should be relatively straightforward.

The proofs of correct decryption, however, are implemented locally in Estonia, and the only current verifier implementation is supplied by the vendor as a part of the IVXV software package.

Even though all of the server-side software of IVXV is open-sourced[4], the accompanying documentation has varying level of

*ARES 2023, August 29–September 01, 2023, Benevento, Italy*
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0772-8/23/08.
https://doi.org/10.1145/3600160.3605467

[1]https://www.verificatum.org/
[2]https://github.com/akels/Verificatum.jl
[3]https://github.com/ZetaTwo/sa104x-kexjobb
[4]https://github.com/vvk-ehk/ivxv

Verification of the Decryption Proof

Let us have ciphertext `c = (c_0, c_1)`, which is decrypted into the value `d` with the given public key `pk` over the parameters `(p,g)` and with the decryption proof `(a,b,s)`.

To check that the decryption was correct, the challenge `k=H("DECRYPTION"||pk||c||d||a||b)` is calculated, and where `H` is the `SHA-256` hash function. Then it is verified that `c_0^s = a * (c_1/d)^k` and `g^s = b * y^k`.

**Figure 1: Official specification of the IVXV decryption proof**

detail. Among other components, the specification for the verification of the decryption proof is literally just four lines long (see Figure 1).[5]

We contacted the development team of IVXV asking for more detailed specifications. It appears that at the time of this writing (late 2021 – early 2023), an updated version of the documentation has already been in preparation. We requested a preliminary version of this documentation and used it as a basis for our implementation of the first independent decryption proof verifier for IVXV.

This paper summarizes our main lessons learned in the process. We will also propose several improvements to the decryption proof generation that would simplify development of the verification application. Essentially, we will be adding a new non-functional requirement to the development of e-voting systems – *the cryptographic proofs must be as easy to verify as possible.* The rationale behind such a requirement is that verification simplicity would hopefully encourage a larger number of independent verifications, which would in turn hopefully increase public trust in the correctness of the election result. Also, conceptually simple verification makes it easier to sort out possible discrepancies between independently developed verification applications.

## 2 IVXV VOTE ENCRYPTION AND ZERO KNOWLEDGE PROOF OF CORRECT DECRYPTION

In order to allow for efficient mixing and decryption proofs, Estonian installation of IVXV uses ElGamal encryption in the group of residues over a prime modulus $p$. To achieve at least 10 years of confidentiality horizon, the recommended length for $p$ is 3072 bits [2]. Thus, Group 15 from RFC3526 was selected [9]. RFC3526 sets

$$p = 2^{3072} - 2^{3008} - 1 + 2^{64} \cdot ([2^{2942} \cdot \pi] + 1690314)\ .$$

The way $p$ has been chosen guarantees that it is a safe prime, i.e. $q = \frac{p-1}{2}$ is a prime as well (we will discuss the details of this choice further in Section 4.1). RFC3526 also sets a generator $g = 2$ which yields a subgroup of quadratic residues of order $q$. All the algebraic computations will take place in this subgroup.

Another observation useful for our analysis is that $p$ is relatively close to $2^{3072}$. More precisely, $2^{3072} > p > 2^{3072} - 2^{3008}$, and we obviously also have $2^{3071} > q > 2^{3071} - 2^{3007}$.

In the setup phase, the election organizer selects a random private exponent $x \in \mathbb{Z}_q$ and publishes the corresponding public key $h = g^x$ by bundling it with the voting client application.[6]

The vote is represented as a mod-$p$ quadratic residue $m$. The voting client generates the encryption randomness $r \in \mathbb{Z}_q$ and computes the corresponding ElGamal cryptogram as

$$c = (u, v) = (g^r, m \cdot h^r)\ .$$

The decryption server, possessing the private key $x$, decrypts the message as $v \cdot u^{-x}$. In order to prove that the decryption was performed correctly and using the correct private key, the decryption server essentially presents a Chaum-Pedersen proof of equality of two discrete logarithms made non-interactive using the Fiat-Shamir construction.

More concretely, the decryption server picks a random $t \in \mathbb{Z}_q$ and computes a message commitment $a = u^t$ together with a key commitment $b = g^t$.

Next, the transcript of the protocol this far is serialized as a seed for a custom hash function (implementing the oracle call), and its output is used as a challenge $k$. The decryption server then computes $s = k \cdot x + t \bmod q$ as a response, and outputs $(a, b, s)$ as the complete proof. The verifier accepts by checking the equations

$$u^s = a \cdot (v/m)^k\ ,$$
$$g^s = b \cdot h^k\ .$$

## 3 IMPLEMENTING AN INDEPENDENT DECRYPTION PROOF VERIFIER

The idea of enabling independent cryptographic proof verifiers does not serve its purpose well if it is done just for satisfying a formal requirement. We argue that an important non-functional requirement is the ease of implementation of such a tool. This includes both the time spent on development and clarity of the resulting code so that it could be easily inspected by external parties.

Of course, the simplicity of verification must be balanced out against other, possibly conflicting requirements. For example, if there were no decryption proofs, no effort would need to go into verifying them either, but such a solution would have considerably weaker integrity guarantees. Thus, our aim is to make the proofs as easily verifiable as possible, while still retaining the full cryptographic strength of the solution.

We decided to validate the decryption proof specification of IVXV and its ease of implementation by developing a proof-checking tool

[5]https://github.com/vvk-ehk/ivxv/blob/master/Documentation/en/protocols/11-audit.rst

[6]We are omitting most of the technical details not directly relevant to the topic of this paper, for example private key management techniques.

in Go.[7] Go was selected because of its speed, easy multi-threading, relatively good readability and excellent standard library including all the necessary tools e.g. for ASN.1 parsing and long integer arithmetic. Also, the vendor-supplied proof checker was developed in Java, and we learned that a proof-of-concept implementation in Python was underway as well by the vendor as a part of the specification update effort. Hence, we did not want to use either of those languages.

During the development process, we found several aspects of the verification protocol that could be improved in order to make the verifier implementation simpler and more robust.

The most fragile component of the whole specification is generation of the challenge $k$ from the seed. Formally, the seed is defined to be the DER encoding of the following ASN.1 structure:

```
SEQUENCE ::= {
    NIPROOFDOMAIN   GENERAL STRING,
    pubkey          SubjectPublicKeyInfo,
    ciphertext      encryptedBallot,
    decrypted       OCTET STRING,
    msgCommitment   INTEGER,
    keyCommitment   INTEGER
}
```

where the value of the field NIPROOFDOMAIN is defined to be the string "DECRYPTION" (without quotes).

The next step is to convert this seed into a challenge value $k \in \mathbb{Z}_q$. IVXV uses a custom challenge function for that, targeting cryptographic strength and a uniform distribution of $k$ in $\mathbb{Z}_q$ (even though this is not really necessary for the Fiat-Shamir heuristic when used in zero-knowledge proofs [3]). A central building block in the challenge function implementation is a component that can be viewed as a deterministic pseudo-random number generator (PRNG).

Just running the seed through a PRNG and taking the output modulo $q$ would introduce a modular bias. Hence, IVXV implements a rejection sampling step. First, a pseudo-random value $y$ from $\mathbb{Z}_{2^\ell}$ is generated (where $\ell$ is the bit length of $q$), and it is accepted if $y < q$. Otherwise, the process starts over, requesting new bits from the PRNG.

A good candidate for the PRNG would be an extendable output function from the SHA3 family; say, Shake256. However, back in 2016-2017 when the design decisions for the current IVXV system were taken, SHA3 was still a fresh standard. The libraries implementing all its parts (including Shake256) were not yet readily available for the major software development platforms. Thus, a decision was taken by the software architects of IVXV to develop a customized PRNG based on the SHA2-256 hash function.

More precisely, the output stream of the IVXV PRNG is defined to be

$$H(0x0000000000000001||seed)||H(0x0000000000000002||seed)||\ldots,$$

where $H$ is the SHA2-256 hash function and the seed is prepended with 64-bit big-endian sequential numbers $1, 2, 3, \ldots$.

This PRNG outputs bits in batches of 256. Since, in general, $q$ can not be expected to have length multiple to 256, there also has to be a masking step. As we saw earlier, the length of $q$ in the current implementation of IVXV is 3071 bits, so $12 \cdot 256 = 3072$ bits are queried at a time, and the top bit is masked off to produce the candidate $y$.

As a result, we get Algorithm 1.

**Algorithm 1:** Rejection sampling for the challenge

```
Data: $q > 0$, $seed$, hash function block length $b$, the bit
      length $\ell$ of $q$
Result: $y$ uniformly distributed in $\mathbb{Z}_q$
1 PRNGInit($seed$)
2 while $true$ do
3     $y' \leftarrow$ PRNGRead($b \cdot \lceil \frac{\ell}{b} \rceil$ bits)
4     $y \leftarrow \ell$ lower bits of $y'$
5     if $y < q$ then
6         return $y$
```

We have $2^{\ell-1} < q < 2^\ell$, so the probability that the inequality $y < q$ holds on line 5 of Algorithm 1 is

$$\frac{q}{2^\ell} > \frac{2^{\ell-1}}{2^\ell} = \frac{1}{2} .$$

Thus, the expected number of rejections is no more than 2.

For the current choice of $q$ in IVXV, we have $\ell = 3071$, and as noted in Section 2, inequality $q > 2^\ell - 2^{\ell-64}$ actually holds. Hence, for this particular $q$ we get that the inequality $y < q$ in Algorithm 1 holds with probability

$$\frac{q}{2^\ell} > \frac{2^\ell - 2^{\ell-64}}{2^\ell} = 1 - 2^{-64} .$$

Thus, in the current protocol, rejection almost never happens.

This has unexpected consequences from the software development point of view. If the PRNG produces good pseudo-random output, it becomes impossible to prepare test vectors that would trigger rejection. Of course, tests for the PRNG component in isolation can be written, but in the integration phase, all the rejection steps will become "dead functionality". This includes behaviour of the non-standard PRNG beyond the first call.

Severity of this problem in practice is a subject of debate. On one hand, in order to fully correspond to the specification, the logic of rejection should be implemented, including the PRNG. On the other hand, since rejection can almost never happen, the developer may skip it, or implement it incorrectly not realizing that this functionality is not covered by the test vectors.

Of course, with the current choice of parameters, potential implementation issues are unlikely to manifest themselves. But on the other hand, the flexibility to support other parameters (e.g. different values for $q$) was designed into the Algorithm 1.

If a new $q$ is selected some time in the future and an independent auditor comes to run checks with the same version of the tool he used previously, there is a risk of incompatible output. Of course, this risk can be detected with updated tests, and the problem can be attributed to programming errors. However, verifying the proofs is a public event meant to increase trust in the final tally. Any issues during this process have a potential to undermine this trust. It would be best if the protocol would be designed in a way that its implementation would leave as few ambiguous points as possible.

[7] https://go.dev/

## 4 SIMPLIFYING PROOF VERIFICATION

There are a few relatively simple adjustments that would make the specification easier to implement and test.

### 4.1 Rejection sampling

First of all, as uniform distribution is not really needed for the Fiat-Shamir challenges, the whole rejection sampling process is not necessary and can be removed.

The second adjustment is also straightforward – use a standard extensible output hash function like Shake256 instead of a hand-tailored PRNG. As of 2022, stateful implementations of Shake256 are available for a variety of development platforms, including Go[8], Python[9], Java[10], etc. This means that requiring independent verification applications to implement non-standard PRNG-s is no longer justified.

If rejection sampling is still targeted, the interplay between rejection sampling and the primes $p$ and $q$ should be addressed. Masking off the excess bits on the line 4 of Algorithm 1 is introduced with a good idea – to make the check on the line 5 more efficient. However, when $q$ is very close to $2^\ell$, this check becomes *too* efficient so that it almost never triggers, resulting in a chunk of dead, hard-to-test code.

There are several possible approaches to this problem.

First, we may say that the failure probability of $2^{-64}$ is small enough to be ignored, and that we will only be using values of $q$ very close to $2^\ell$. This means that we can ignore all rejection sampling and masking, and simply read $\ell$ first bits from the PRNG.

The main question here is whether suitable alternative values for $q$ can be selected in case e.g. its length needs to be extended in order to meet updated security requirements in the future. Luckily, RFC3526 has also standardized 4096-, 6144- and 8192-bit safe primes that all have the same structure, i.e. that their 64 highest bits are equal to 1. In case even 8192-bit modulus becomes insecure, we probably anyway have to consider changing the whole cryptosystem (e.g. to resist attacks by quantum computers).

We are actually not limited to the primes given in RFC3526, but can generate our own ones. All the MODP groups in RFC3526 follow the same pattern. A prime modulus of bit length $b$ is defined for in the form

$$p = 2^b - 2^{b-64} - 1 + 2^{64} \cdot ([2^{b-130} \cdot \pi] + i) , \tag{1}$$

where $i$ is the smallest positive integer such that $p$ is a safe prime (i.e. $\frac{p-1}{2}$ is also a prime). The 64 highest and 64 lowest bits of $p$ are set to 1 to allow for efficient modular computations (see [12, Appendix E] for the rationale of these choices).

The number $i$ can be determined by inspecting $i = 1, 2, \ldots$, and it is typically in the order of magnitude from a few hundred thousands to a few millions.

Say, we want to account for the fact that Shake256 outputs only full bytes, so in order to get 3071 bits we would still need to use some masking. In order to get rid of this, we may want to have $q$ of length 3072 and $p$ accordingly of length 3073 bits. It is straightforward to verify that 265656 is the smallest value for $i$ gives a prime number with the desired properties.

If the failure probability of $2^{-64}$ is still too large, we can replace 64 by 128 (and 130 by 258) in (1) to find that $i = 2655344$ gives rise to a suitable 3073-bit safe prime with the probability of rejection less than $2^{-128}$.

An alternative solution to the dead functionality problem is not to mask off the highest bit(s) in line 4 of Algorithm 1. As we saw in Section 3, this would lead to about two rounds of rejection computation in case of the current parameters of IVXV. This would enable creating test vectors for rejection sampling, whereas the computational overhead to the whole process would be negligible since the computation time is still mostly determined by the modular exponentiation.

### 4.2 Encoding the cryptographic material

In order to process the algebraic values defined in the protocol specification, they have to be encoded somehow on the bit level. There are a few alternatives for such an encoding. Verificatum mix-net for example uses its own byte tree data structure [1]. However, this is a non-standard structure, with no implementation used outside Verificatum.

IVXV uses JSON to wrap a list of proofs into one file, and ASN.1 DER to represent the actual cryptographic values. The choice of ASN.1 is pragmatic as it is one of the oldest and best-established byte-level serialization methods, being supported by a number of existing tools and libraries. This included the tool of our choice, Go, that provided all the required functions for base64 decoding and ASN.1 DER parsing as parts of standard library.

On the other hand, ASN.1 does add another layer of complexity, and it should only be used when really necessary, i.e. when the data needs to travel between loosely connected components.

One example of such a need is distribution of the system's public encryption key. This key is used e.g. by the voting client, individual verification application, mix-net and decryption server (together with the proof verifiers). It makes sense to distribute the key in a standard format, and ASN.1 is a good choice for that.

However, the actual cryptographic values that the decryption server outputs (i.e. $u$, $v$, $a$, $b$ and $s$) are only meant to be used by the proof verification applications. Out of these, $u$ and $v$ are actually obtained from the mix-net output in the byte tree format, re-packaged into ASN.1, and the corresponding ASN.1 structure is encapsulated into the JSON file. Similarly, the $a$, $b$ and $s$ values obtained during the proof generation are packaged into an ASN.1 structure which is then saved into a field in the JSON file.

We argue that such extra packaging is redundant. There is no reason not to export the $u, v, a, b$ and $s$ directly as fields of the JSON structure, removing the need for ASN.1 parsing as a part of the decryption proof verification altogether.

Of course, the need to parse the public key still remains. However, note that since the group used in IVXV is standard, the modulus $p$ and the generator $g$ are fixed anyway, and the only varying part is the value $h = g^x$. But $h$ is also just a number which can be extracted from the ASN.1 structure using any available tool like `asn1parse` or `openssl`, and hard-coded into the source.

[8]https://pkg.go.dev/golang.org/x/crypto/sha3#ShakeHash
[9]https://pycryptodome.readthedocs.io/en/latest/src/hash/shake256.html
[10]https://javadoc.iaik.tugraz.at/iaik_jce/current/iaik/security/md/SHAKE256InputStream.html

A problem with the IVXV public encryption key is that it is not distributed in a well-communicated manner. It is compiled into the voting application, but the latter is not open source. The public key is also needed during individual verification, and the verification application obtains it from a respective configuration file. The file is actually accessible to anyone online[11], but the link is not clearly communicated to the potential auditors. On the positive side, note that such a distribution mechanism helps to convince the auditor that the public key used during the elections and the public key provided for decryption proof auditing are actually the same.

## 5 SUPPORTING PROCESSES AND PRACTICAL DEPLOYMENT

Even though the IVXV infrastructure has been built with independent auditing in mind, the existing processes around it do not yet enable auditing to be set up and run smoothly.

The first issue is availability of the specification for the IVXV decryption proof verification. Even though we were able to obtain a version of it directly from the IVXV development team for the purposes of pur project, it has not yet been officially released by the time of this writing (March 2023). In order for the zero knowledge proofs to verify successfully, the application has to work correctly to the last bit. Achieving this is impossible without a comprehensive and up-to-date specification.

Another major problem is the lack of processes for invoking independently developed auditing applications. On one hand we want such applications to be put forward as this helps to increase transparency of the whole system. On the other hand, these applications should be subjected to strict quality control as incorrectly reporting a verification failure may cause a lot of unjustified distrust.

Thus, the first step in engaging independent auditing solutions should be developing a set of acceptance criteria and test vectors. These should cover all the possible errors and exceptional situations, as well as provide enough data for stress testing. For our development, we got one file with 4 zero knowledge proofs from the developer of IVXV and one file with 69 proofs that were obtained while testing the system for March 2023 parliamentary elections of Estonia. Based on the latter, we built a file copying each of the proofs 4,000 times in order to have 276,000 proofs for stress testing our application. However, we had no guarantee that these test files would cover all the possible situations that might occur while checking the proofs.

Currently there is also no clear process for actually invoking the independent auditing applications. There are a few possible options for this. First, it is possible to hand over the decryption proofs on a removable media during the tallying or tally verification stages. The auditors could then run their applications on their own computers and report back the results. As the votes are re-encrypted and mixed before decryption, the proof files should contain no personal information that would leak voters' preferences and thus enable coercion attacks.

However, this only holds true for the correctly formatted votes. An incorrect vote may still carry information useful e.g. in the coercion scenarios. In the simplest case, the attacker can force a victim to submit an invalid vote (e.g. using a custom voting client [4]) in an attempt to disenfranchise them. Detecting the respective invalid vote in the decrypted output assures the attacher that his attack was successful. It is also possible to use the invalid vote side channel to leak secret information [15]. In 2022, Mueller proposed an attack that allows encoding several votes into what appears as an incorrectly encoded ballot [11].

Currently, incorrect ballots are not included in the zero knowledge proof file produced by the decryption application, but this comes with a power for the decryption application to declare some votes as invalid without proving this statement. As pointed out in [11], one possible solution would be including proofs of plaintext correctness into the vote submission protocol of IVXV. In any case, invalid ballot attacks need to be mitigated before the decryption proof file can be handed over to the independent auditors without any restrictions. On the other hand, once mitigating measures are put in place, the zero knowledge proofs of correct decryption can in principle made available to everyone online.

Another possible approach would be requiring the authors of the independent verification applications to put the source code up for prior inspection. First, this would allow for standardized quality control procedures as both the election organizers and general public would be able to look at the code and test the applications beforehand. Second, such an approach would enable automated execution of the verification applications without much human involvement. Third, a public repository like GitHub would also retain publicly visible update log, helping to ensure that the publicly audited version was actually the one that was executed during the auditing procedure.

The last process that is also currently missing in Estonia is that of dispute resolution. Even after careful testing, it is still possible that different auditing applications give different outputs. A simple programming error is the most likely cause for this, but such a situation should nevertheless be carefully studied. Clear guidelines for resolving the mismatch should be established beforehand.

We tested our application during the 2023 Estonian parliamentary elections. We released the first version of the software publicly on February 23rd, i.e. 4 days before the period for electronic vote casting started.[12]

Even though all the above-mentioned procedures were not in place, we approached the Estonian Election Management Body (EMB) with a request to receive the actual decryption proof file, and our request was granted.

At the first run, our application successfully loaded proofs for all the 312181 electronic votes given. As an output, the application reported 312180 successful and 0 failed proof verifications. The mismatch of one was easily traced back to incorrect mutex handling during the multi-threaded processing of the proofs. After this bug was fixed, the application consistently reported 312181 successful verifications. The whole verification process took about 50 minutes on an AMD Ryzen 7 5700U processor having 8 cores and supporting 16 threads.

In retrospect, we feel that the mismatch of one during the initial run underlines our point made above. The independent auditing and verification applications have to undergo stringent quality control before they can be run on the real input during something

[11]https://www.valimised.ee/verify/config.json

[12]https://github.com/janwil123/IVXVDecryptionProofVerifier

as important as parliamentary elections. The respective quality control processes must be established and clearly communicated by the EMB in advance.

Compared to the original four-line specification, our Go implementation of the decryption proof verifier is 333 lines long (including empty lines and comments for better readability). A significant amount of this length is due to packing and parsing various values to and from ASN.1 DER encoding. While being relatively straightforward to read, this length could be reduced if the values required for proof verification would be included in the JSON file directly (see Section 4.2).

According to our subjective assessment, the least readable step of the whole verifier source is the seed generation using the non-standard SHA256-based PRNG, followed by the rejection sampling. As discussed in Section 4.1, neither of these complications is really necessary and can be avoided by simplifying the specification.

## 6 CONCLUSIONS AND FUTURE WORK

In this paper, we reviewed the specification of verification of zero-knowledge proofs of correct decryption implemented in the IVXV Internet voting system. In order to validate completeness of the specification, we created an implementation of the verifier in the Go programming language.

In this process, we received support from the IVXV development team both in the form of in-progress specification and answers to specific technical questions. In retrospect, we can say that completing the implementation would not have been possible without access to these answers. On one hand this means that even the in-progress specification was not yet detailed enough for a complete implementation. On the other hand, we hope that our effort also gave feedback to the IVXV development team helping to improve the specification.

We also evaluated the decryption proof protocol from the viewpoint of verifier implementation simplicity. We argue that this simplicity is an important (even though often overlooked) non-functional requirement when designing zero-knowledge proofs. Recall that the ultimate goal of all the voting-related ceremonies and protocols is to increase public trust in the election result. Cryptographic proofs, while being instrumental in achieving advanced security properties, are at the same time also some of the most complex components in the electronic voting schemes.

This makes the role of cryptographic proofs somewhat controversial. On one hand they are designed to reduce the need to blindly trust the key components like the vote decryption service. On the other hand, if the complexity of verification routines required to assess the correctness of these components reaches beyond certain threshold, the number of people who are actually able to perform this verification becomes too low to support public trust.

Whereas good knowledge of algebra and programming is unavoidable, there are still a few details in the current zero-knowledge proof protocols that can be tuned to make implementing the proof verifiers easier or more robust. In case of IVXV, for example, a part of complexity can be avoided if the unnecessary property of uniform distribution of Fiat-Shamir challenges is not targeted in the first place.

If this property is still desired, a possible trade-off can be accepting a little bit lower performance. In case of IVXV, for example, we can change the parameters so that the rejection sampling step would actually trigger with 50% probability, making it possible to cover the implementation with complete integration test vectors. The price to pay would be 12 hash function evaluations per rejection. This performance penalty can be lowered even further if Shake256 would be used, replacing separate hash function calls with one sponge squeeze.

Another layer of potential complexity is added by the need to encode algebraic values to bit-level representations. There exist standard encoding schemes (like ASN.1 DER plus base64) that are commonly used to provide interoperability between different components or even systems, e.g. for encapsulating public keys. However, there is no need to utilize extra encoding for values used only within one protocol. Dropping unneeded encoding and decoding steps can also make the code easier to write, test and audit.

We argue that the requirement of easy comprehension and verification should be taken into account already when starting to design cryptographic zero-knowledge proof protocols. On one hand, zero-knowledge is not an overly complicated concept [13]. The challenge, however seems to be doing it efficiently so that the proofs would scale to, say, millions of instances – an order of magnitude required by the voting protocols. Still, continuous advances in computer hardware should make it possible to find trade-offs and allow better explainable protocols for the price of some performance drop. This is an open question for future research and development.

As a by-product of our evaluation effort, we created a fully functional decryption proof verification utility which we used in practice during the Estonian parliamentary elections of 2023. We were able to independently verify all the zero-knowledge proofs, but a minor bug in the initial version of our tool underlines the need to have clear and reliable procedures for quality control of the independently developed verification applications.

We conclude that it is not sufficient just to create an option for external data auditors to enter the scene. One must also prepare for the scenario when the results of competing auditing tools do not agree. Several options are conceivable here, ranging from prior testing to on-site code review. In the latter case, the review will be easier to conduct if the algorithm to be implemented for verification is as simple as possible. This observation once more confirms the main point of the current paper and motivates future zero-knowledge proof designers to pay more attention to the simplicity of proof verification.

## ACKNOWLEDGMENTS

This paper has been supported by the Estonian Research Council under the grant number PRG920. The author is grateful to Sven Heiberg and Ivo Kubjas for helpful discussions and feedback during the development process.

## REFERENCES

[1] 2022. User Manual for the Verificatum Mix-Net. Version 3.0.4, https://www.verificatum.org/files/vmnum-3.1.0.pdf.

[2] Michel Abdalla, Tor Erling Bjørstad, Carlos Cid, Benedikt Gierlichs, Andreas Hülsing, Atul Luykx, Kenneth G. Paterson, Bart Preneel, Ahmad-Reza Sadeghi, Terence Spies, Martijn Stam, Michael Ward, Bogdan Warinschi, and Gaven Watson. 2018. Algorithms, Key Size and Protocols Report. https://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf.

[3] Yilei Chen, Alex Lombardi, Fermi Ma, and Willy Quach. 2021. Does Fiat-Shamir Require a Cryptographic Hash Function?. In *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 12828)*, Tal Malkin and Chris Peikert (Eds.). Springer, 334–363. https://doi.org/10.1007/978-3-030-84259-8_12

[4] Valeh Farzaliyev, Kristjan Krips, and Jan Willemson. 2021. Developing a Personal Voting Machine for the Estonian Internet Voting System. In *SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, Republic of Korea, March 22-26, 2021*, Chih-Cheng Hung, Jiman Hong, Alessio Bechini, and Eunjee Song (Eds.). ACM, 1607–1616. https://doi.org/10.1145/3412841.3442034

[5] Thomas Haines, Rajeev Goré, and Bhavesh Sharma. 2021. Did you mix me? Formally Verifying Verifiable Mix Nets in Electronic Voting. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1748–1765. https://doi.org/10.1109/SP40001.2021.00033

[6] Sven Heiberg, Peeter Laud, and Jan Willemson. 2011. The Application of I-Voting for Estonian Parliamentary Elections of 2011. In *E-Voting and Identity - Third International Conference, VoteID 2011, Tallinn, Estonia, September 28-30, 2011, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7187)*, Aggelos Kiayias and Helger Lipmaa (Eds.). Springer, 208–223. https://doi.org/10.1007/978-3-642-32747-6_13

[7] Sven Heiberg, Tarvi Martens, Priit Vinkel, and Jan Willemson. 2016. Improving the Verifiability of the Estonian Internet Voting Scheme. In *Electronic Voting - First International Joint Conference, E-Vote-ID 2016, Bregenz, Austria, October 18-21, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10141)*, Robert Krimmer, Melanie Volkamer, Jordi Barrat, Josh Benaloh, Nicole J. Goodman, Peter Y. A. Ryan, and Vanessa Teague (Eds.). Springer, 92–107. https://doi.org/10.1007/978-3-319-52240-1_6

[8] Sven Heiberg and Jan Willemson. 2014. Verifiable internet voting in Estonia. In *6th International Conference on Electronic Voting: Verifying the Vote, EVOTE 2014, Lochau / Bregenz, Austria, October 29-31, 2014*, Robert Krimmer and Melanie Volkamer (Eds.). IEEE, 1–8. https://doi.org/10.1109/EVOTE.2014.7001135

[9] Mika Kojo and Tero Kivinen. 2003. More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE). RFC 3526. https://doi.org/10.17487/RFC3526

[10] Ülle Madise and Tarvi Martens. 2006. E-voting in Estonia 2005. The first Practice of Country-wide binding Internet Voting in the World. In *Electronic Voting 2006: 2nd International Workshop, Co-organized by Council of Europe, ESF TED, IFIP WG 8.6 and E-Voting.CC, August, 2nd - 4th, 2006 in Castle Hofen, Bregenz, Austria (LNI, Vol. P-86)*, Robert Krimmer (Ed.). GI, 15–26.

[11] Johannes Mueller. 2022. Breaking and Fixing Vote Privacy of the Estonian E-Voting Protocol IVXV. In *Workshop on Advances in Secure Electronic Voting 2022*. http://hdl.handle.net/10993/49442.

[12] Hilarie K. Orman. 1998. The OAKLEY Key Determination Protocol. RFC 2412. https://doi.org/10.17487/RFC2412

[13] Jean-Jacques Quisquater, Myriam Quisquater, Muriel Quisquater, Michaël Quisquater, Louis Guillou, Marie Annick Guillou, Gaïd Guillou, Anna Guillou, Gwenolé Guillou, and Soazig Guillou. 1990. How to Explain Zero-Knowledge Protocols to Your Children. In *Advances in Cryptology — CRYPTO' 89 Proceedings*, Gilles Brassard (Ed.). Springer New York, New York, NY, 628–631.

[14] Drew Springall, Travis Finkenauer, Zakir Durumeric, Jason Kitcat, Harri Hursti, Margaret MacAlpine, and J. Alex Halderman. 2014. Security Analysis of the Estonian Internet Voting System. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 703–715. https://doi.org/10.1145/2660267.2660315

[15] Douglas Wikström, Jordi Barrat, Sven Heiberg, Robert Krimmer, and Carsten Schürmann. 2017. How Could Snowden Attack an Election?. In *Electronic Voting - Second International Joint Conference, E-Vote-ID 2017, Bregenz, Austria, October 24-27, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10615)*, Robert Krimmer, Melanie Volkamer, Nadja Braun Binder, Norbert Kersting, Olivier Pereira, and Carsten Schürmann (Eds.). Springer, 280–291. https://doi.org/10.1007/978-3-319-68687-5_17