Faking an iVote decryption proof

Why the decryption proof flaw identified in the SwissPost system affects the iVote system too

Vanessa Teague

The University of Melbourne
Parkville, Australia
vjteague@unimelb.edu.au

November 14, 2019

The proof of correct decryption specified in the iVote protocol description is susceptible to the same attack we identified already in the equivalent part of the SwissPost/Scytl system.

This could, for instance, be used by a cheating decryption service to change valid votes into nonsense that would not be counted.

1. Introduction

In earlier work with Sarah Jamie Lewis and Olivier Pereira [LPT19], we showed that an error in the implementation of the Fiat-Shamir heuristic allowed a cheating decryption service in the SwissPost-Scytl system to fake a decryption proof. The proof could pass verification even though a valid vote had been incorrectly decrypted as nonsense.

At the time, the NSWEC issued a press release in which they stated "Based on its assessment of the information supplied by these academics, the NSW Electoral Commission is confident that the new issue they describe in the Swiss Post system is not relevant to the iVote system."[1]

In this note I show that that assessment was incorrect. iVote's decryption proof and verification specification are slightly different from those of the SwissPost system, but the same attack can still be performed after a slight modification. This is demonstrated below.

[1] `https://www.elections.nsw.gov.au/About-us/Media-centre/News-media-releases/NSW-Electoral-Commission-iVote-and-Swiss-Post`

This analysis is based on the protocol description [Scy19]. I have not (yet) inspected the code. Since the error affects the verification process, it is the specification rather than the current implementation that defines the system's security anyway—if iVote had independent verification, it would be based on an independent implementation of the verification spec, not on running Scytl's code.

Nor have I examined the rest of the protocol to understand how this possibility for cheating fits with the rest of the process. Like the equivalent cheat for the SwissPost system, it assumes that the cheating decryption service can generate its own ciphertexts. I have not checked whether this is the case for the iVote mixing and decryption service. This note simply takes the proof of correct decryption given in Section 2.6.3.2 of the protocol description, and shows how it can be faked to produce a proof that passes verification but is not true.

2. iVote background

Like the SwissPost system, iVote runs El Gamal in a subgroup of $\mathbb{Z}_p^*$ of order $q = (p-1)/2$, with generator g. Remember that a standard ElGamal encryption of message m with public key pk is a pair $(C_0, C_1) = (g^r, m(pk)^r)$. In iVote, the public key is a series of values: $PK = (pk_1, pk_2, \ldots, pk_n) = (g^{sk_1}, g^{sk_2}, \ldots, g^{sk_n})$, where $sk = (sk_1, sk_2, \ldots, sk_n)$ is the (multi-valued) secret key. A ciphertext is an encryption of an n-tuple $M = (m_1, m_2, \ldots, m_n)$, computed as

$$C = (g^r, m_1.pk_1^r, m_2.pk_2^r, \ldots, m_n.pk_n^r)$$

where r is chosen randomly from $[0, q-1]$.

3. Producing a false decryption proof—iVote version

A proof of proper decryption—that the ciphertext $C = (c_0, c_1, \ldots, c_n)$ decrypts to message $M = (m_1, m_2, \ldots, m_n)$—can be constructed by anyone who knows the secret key sk. The proof generation and verification are specified in Section 2.6.3.2. of the iVote protocol description. The Prover specification is called ProveDec and is shown in Figure 1; the Verification specification is called VerifyDec and is shown in Figure 2.

3.1. An obvious inconsistency between the hash inputs

First observe that the inputs to the hash function, in Step 2 of ProveDec and Step 3 of VerifyDec, are inconsistent. Even if we take pk (in VerifyDec) to match $pk_1, pk_2, \ldots, pk_n$ (in ProveDec), there is a major discrepancy in the hashing of the ciphertexts: ProveDec hashes all n divided ciphertexts $c_1/m_1, c_2/m_2, \ldots, c_n/m_n$, while VerifyDec hashes only c_2/m—it is not clear what m is. The failure to hash c_1/m_1 and $c_3/m_3 \ldots c_n/m_n$ means that VerifyDec would not accept the output of a prover that was implemented according to this spec.

- ProveDec($(C,M),SK$) receives an ElGamal ciphertext $C = (c_0, c_1,\dots, c_n)$, the decrypted value $M=(m_1, m_2,\dots,m_n)$, and the private key $SK=(sk_1,sk_2,\dots,sk_n)$ used to obtain M from C. Given this input and the public values: g, $PK = g^{sk1}, g^{sk2},\dots, g^{skn}$ (mod p), and $q = (p - 1)/2$; the algorithm proceeds as follows:
    1) Takes as many random values as the number of private keys ($s_1,\dots,s_n$) from $1,\dots,q-1$.
    2) Computes $h=H(pk_1,pk_2,\dots,pk_N|c_1/m_1|c_2/m_2|\dots|c_n/m_n|g^\wedge(s_1)|\dots|g^\wedge(s_n)|c_0{}^\wedge(s_1)|c_0{}^\wedge(s_2)|\dots|c_0{}^\wedge(s_n)|\text{"DecryptionProof"})$
    3) Computes $z_1=s_1+sk_1\cdot h$; $z_2=s_2+sk_2\cdot h$;....; $z_n=s_n+sk_n\cdot h$ (mod q)
    4) Outputs h and $Z=z_1,\dots,z_n$.

Figure 1: ProveDec: Decryption Proof Specification

- VerifyDec($c,m,(h,z)$) receives as input an ElGamal ciphertext $C = (c_0, c_1,\dots, c_n)$, a plaintext $M=(m_0, m_1,\dots,m_n)$ and the output of a ProveDec() algorithm, (h,Z). To validate the proof, the algorithm performs the following operations:
    1) Computes $a_1 = g^{z1} \cdot pk_1{}^{-h}$; $a_2 = g^{z2} \cdot pk_2{}^{-h}$;…; $a_n = g^{zn} \cdot pk_n{}^{-h}$ (mod p).
    2) Computes $b_1 = (c_0)^{z1} \cdot (c_1/m_1)^{-h}$; $b_2 = (c_0)^{z2} \cdot (c_2/m_2)^{-h}$;…; $b_n = (c_0)^{zn} \cdot (c_n/m_n)^{-h}$ (mod p).
    3) Checks that $h = H(pk|c_2/m|a_1|a_2|\dots|a_n|b_1|b_2|\dots|b_n|\text{“DecryptionProof”})$.
    4) If the validation is successful, the algorithm outputs 1. Otherwise it outputs 0.

Figure 2: VerifyDec: Decryption Proof Verification Specification. Note the highlighted inconstency in hash inputs.

The obvious patch is to change Step 3 of VerifyDec to match Step 2 of ProveDec.[2] This would make the proof complete, that is, faithfully-implemented verification would accept anything produced by an honest prover.

However, the proof would still not be sound. Even with this correction, the verifier can be tricked into accepting a proof of a decryption statement that is not true. This is derived below.

3.2. Even with the same hash inputs, verification can be tricked

In this section we assume that VerifyDec is corrected, as described above, so that it hashes the same inputs as ProveDec. This section describes the attack for the simple case where $n = 1$, *i.e.* standard El Gamal. The extension to multi-element El Gamal is in Section 3.2.2.

ProveDec$((C, M), SK)$ is very similar to the analogous function in the SwissPost system. It proceeds as follows:

1. picks a random s_1 from $[0, q-1]$,
2. computes $h = H(pk_1|c_1/m_1|g^{s_1}|c_0^{s_1}|\text{“}\textit{DecryptionProof}\text{"})$,
3. sets $z_1 = s_1 + sk_1.h$,
4. outputs h and z_1.

H is a cryptographic hash function.

The main problem here is exactly the same as the problem we identified in the SwissPost system: the hash does not constitute a commitment to c_0. Although c_0 is intended to be an input, the value $c_0^{s_1}$ does not in fact bind a particular c_0—a cheating prover can claim a different value for c_0 later by effectively (and undetectably) claiming a different s_1 from that used to construct g^{s_1}. To see how this works, we first review the Verification spec.

VerifyDec$(c, m, (h, z)))$ receives as input an ElGamal ciphertext $C = (c_0, c_1)$, a plaintext m_1, and a proof (h, z_1). It proceeds as follows:

1. sets $a_1 = g^{z_1}.pk_1^{-h}$,
2. sets $b_1 = c_0^{z_1}.(c_1/m_1)^{-h}$,
3. checks that $h = H(pk_1|c_1/m_1|a_1|b_1|\text{“}\textit{DecryptionProof}\text{"})$.

If the check in Step 3 passes, it accepts; otherwise it rejects.

[2]There is also another typo in that M's first element is written as m_0, but there is no m_0—the ciphertext has a zero-th element but the message does not.

3.2.1. Exploiting the problem

We show deviations from the specification in bold. A cheating prover can construct a fake proof as follows:

1. pick a random s_1 **and a random x** from $[0, q-1]$,
2. compute $h = H(pk_1|c_1/m_1|g^{s_1}|\mathbf{g^x}|\text{“}\mathit{DecryptionProof}''),$
3. set $z_1 = s_1 + sk_1.h$,
4. output h and z_1.

The verifier's computation of a_1 will be correct by construction. In order to ensure that the hashes match, the cheating prover needs to generate c_0 so that the verifier in Step 2 computes the value of b_1 matching the prover's last input to the hash, that is $g^x = c_0^{z_1}.(c_1/m_1)^{-h}$. Hence

$$c_0 = (c_1/m_1)^{h/z_1}.g^{x/z_1}. \tag{1}$$

Now (h, z_1) passes verification as a decryption proof that $(g, pk_1, c_0, c_1/m_1)$ is an El Gamal encryption of 1.

However, it is highly unlikely that this is truthful, *i.e.* that $c_1/m_1 = c_0^{sk_1}$. Taking logarithms base g, and letting $s = \mathsf{dlog}_g(c_1/m_1)$, this equation would be satisfied only if $s = sk_1.(x + sh)/z_1 \pmod q$, *i.e.*, if $s(z_1 - hsk_1) = x.sk_1 \bmod q$ or, using the definition of z_1, if $s.s_1 = x.sk_1 \pmod q$. But x and s_1 are independent values chosen from $\mathbb{Z}_q$ (where q is the size of the ElGamal group, around 2^{2047} for the proposed parameters), so this coincidence occurs with negligible probability. Hence we have a valid proof for a fact that is not true.

This allows for a large set of faked proofs, because it commits only to c_1/m_1, not to c_1 or m_1. Hence it can be used as a proof that a ciphertext (c_0, c_1) decrypts to m_1, for any values for which c_1/m_1 equals the value input to the hash.

3.2.2. Extending to multi-element El Gamal

It is straightforward to extend this attack to the multi-element version of El Gamal that iVote uses—this consists simply of choosing values for the messages $m_2, m_3, \ldots, m_n$ that are consistent with the (cheating) value of c_0 derived above.

To explain the details, it helps first to write the prover and verifier in their multi-element versions, which are almost exactly the same as the single-element versions except that the first few steps are repeated n times.

Multi-element ProveDec$((C, M), SK)$ is as follows. The prover:

1. picks a sequence of random values $s_1, s_2, \ldots, s_n$, from $[0, q-1]$,
2. computes
   $h = H(pk_1|c_1/m_1|c_2/m_2|\ldots|c_n/m_n|g^{s_1}|g^{s_2}|\ldots|g^{s_n}|c_0^{s_1}|c_0^{s_2}|\ldots|c_0^{s_n}|\text{“}\mathit{DecryptionProof}''),$

3. sets $z_i = s_i + sk_i.h$, for $i = 1 \dots n$,
4. outputs h and $Z = (z_1, z_2, \dots, z_n)$.

Multi-element VerifyDec$(c, m, (h, z)))$ (corrected as described above) receives as input an ElGamal ciphertext $C = (c_0, c_1, c_2, \dots, c_n)$, a plaintext $M = (m_1, m_2, \dots, m_n)$, and a proof (h, Z). It proceeds as follows:

1. sets $a_i = g^{z_i}.pk_i^{-h}$, for $i = 1 \dots n$,
2. sets $b_i = c_0^{z_i}.(c_i/m_i)^{-h}$, for $i = 1 \dots n$,
3. checks that $h = H(pk_1|c_1/m_1|c_2/m_2| \dots |c_n/m_n|a_1|a_2| \dots a_n|b_1|b_2| \dots b_n|\text{``}\mathit{DecryptionProof}\text{''})$.

If the check in Step 3 passes, it accepts; otherwise it rejects.

The cheating prover sets up the inputs to the hash as follows. It chooses arbitrary constants α, β, γ and sets:

- $s_i = \alpha sk_i$,
- $x_i = \beta sk_i$, and
- $c_i/m_i = \gamma^{sk_i}$, for $i = 1 \dots n$.

It constructs the cheating proof as follows:

1. computes
   $h = H(pk_1| \dots |pk_n|c_1/m_1| \dots |c_n/m_n|g^{s_1}| \dots |g^{s_n}|g^{x_1}| \dots |g^{x_n}|\text{``}\mathit{DecryptionProof}\text{''})$,
2. (honestly) sets $z_i = s_i + sk_i.h$, for $i = 1 \dots n$,
3. outputs h and $Z = z_1, z_2, \dots, z_n$.

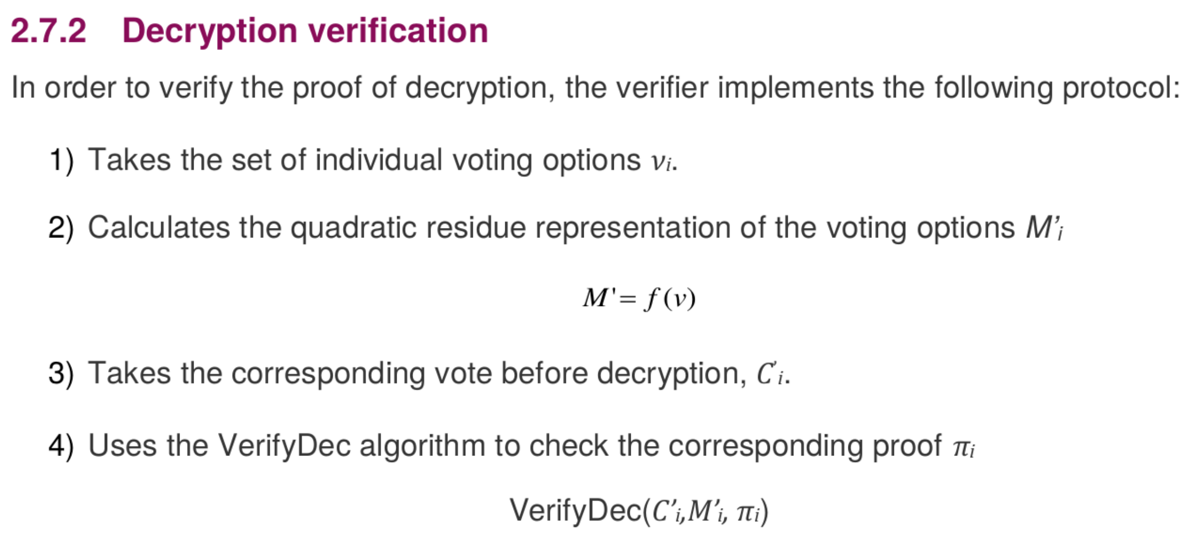
It then computes c_0 as given in Equation 1.

Since each z_i was computed honestly, the first equation in the verification check, $a_i = g^{z_i}.pk_i^{-h}$, will pass for all $i = 1 \dots n$. Step 2 is the challenge. We need to show that the value of c_0 computed in Equation 1 makes the equation $b_i = c_0^{z_i}.(c_i/m_i)^{-h}$ pass for all i, where b_i is the value we've called g^{x_i} in the cheating prover's construction. To check:

$$\begin{aligned}
c_0^{z_i}.(c_i/m_i)^{-h} &= \\
&= ((c_1/m_1)^{h/z_1}.g^{x_1/z_1})^{z_i}.(c_i/m_i)^{-h} \text{ by Equation 1} \\
&= (\tfrac{(c_1/m_1)^{z_i/z_1}}{c_i/m_i})^h.g^{x_1.z_i/z_1} \\
&= (\tfrac{\gamma^{sk_1(sk_i(\alpha+h))/(sk_1(\alpha+h))}}{\gamma^{sk_i}})^h.g^{x_1(sk_i(\alpha+h))/(sk_1(\alpha+h))} \text{ by construction} \\
&= 1^h.g^{\beta sk_i} \text{ by construction} \\
&= g^{x_i}.
\end{aligned}$$

So the value of c_0 computed in Equation 1 allows the verification equations to pass for all i.

Again the prover has committed only to c_i/m_i, not to c_i or m_i individually, so it can choose any message M as long as the ratios c_i/m_i match what was hashed.



2.7.2 Decryption verification

In order to verify the proof of decryption, the verifier implements the following protocol:

1) Takes the set of individual voting options v_i.

2) Calculates the quadratic residue representation of the voting options M'_i

$$M' = f(v)$$

3) Takes the corresponding vote before decryption, C'_i.

4) Uses the VerifyDec algorithm to check the corresponding proof π_i

$$\text{VerifyDec}(C'_i, M'_i, \pi_i)$$

Figure 3: ProveDec: Full Decryption Verification Specification

4. Discussion

The problem can be corrected by implementing the strong version of the Fiat-Shamir Heuristic. In particular, c_0 should be included in the hash (and also g, p and q unless these are chosen in a verifiably random way).

However, the iVote protocol probably contains many other opportunities for trusted components to alter votes. The full decryption verification is shown in Figure 3. It is not clear where the input ciphertexts come from, and does not say that the verifier should check that they match the mixnet's output. This introduces the risk that votes could be altered by simply subsituting them in transit among the various components of the process.

It has been suggested that the decryption proof flaw is mitigated because the mix server signs its output, which is the input to the decryption proof. If the verification spec were amended (and I believe it has been) to check that signature, then this slightly strengthens the attacker model—the mix server and the decryption service would have to collude to produce a fake proof. However, it does not solve the problem—I believe that the mixer and decryption service are both running Scytl software and both administered by the NSWEC, so the assumption that they cannot be simultaneously compromised by the same entity does not seem justified. In any case, it does not address the possibility of vote substitution before or after these components.

A much better solution would be to post votes on a public bulletin board so that independent observers could verify the mixing and decryption process. Public verifiability of that process should be achievable even with NSW's very complex ballots.

However, this does not address the verification of the voting-casting process: how could voters verify that their encrypted vote accurately reflected their intentions? This

I believe is an unsolved problem in practice, and is made substantially more difficult by the complexity of NSW votes. We have explained elsewhere why the current verification app does not provide genuine evidence [CEL+19].

5. Conclusion

iVote suffers from the same error in its decryption proof verification that SwissPosts's system did. A cheating prover can produce a decryption proof that passes verification even though the claimed decryption is false. This is still possible even after the obvious inconsistency in hash inputs between iVote's decryption prover and verifier is corrected.

If the source code and documentation had been made openly available for analysis before the election, as the Swiss system was, then these errors might have been accurately understood and mitigated before the election.

iVote is not a verifiable election system and does not provide meaningful evidence that its output accurately represents the will of voters.

References

[CEL+19] Chris Culnane, Aleksander Essex, Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. Knights and knaves run elections: Internet voting and undetectable electoral fraud. *IEEE Security & Privacy*, 17(4):62–70, 2019.

[LPT19] Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. How not to prove your election outcome. 2019. `https://people.eng.unimelb.edu.au/vjteague/HowNotToProveElectionOutcome.pdf`.

[Scy19] Scytl. NSW electoral commission iVote voting system voting protocol description, May 2019.

A. Notes on disclosure

An earlier version of this report was sent to NSWEC and Scytl on 30 Sep 2019, 45 days before its public release, in keeping with the terms of the source code access agreement.