An Attack on the the Encryption Scheme of the Moscow Internet Voting System

Alexander Golovnev*

August 24, 2019

Abstract

The next Moscow City Duma elections will be held on September 8th with an option of Internet voting. Some source code of the voting system is posted online for public testing. Pierrick Gaudry recently showed that due to the relatively small length of the key, the encryption scheme could be easily broken. This issue has been fixed in the current version of the voting system. In this note we show that the new implementation of the ElGamal encryption system is not semantically secure. We also demonstrate how this newly found security vulnerability can be potentially used for counting the number of votes cast for a candidate.

1 Introduction

The Internet voting system developed for the Moscow City Duma elections had a public testing during the month of July. Some of the system's code was posted online [git19], and the organizers asked the public to test several attack scenarios [Pub19]. The system is poorly documented, but from the source code and brief descriptions of the system [Kri19], we know that it uses the Etherium blockchain and ElGamal encryption. In one of the attack scenarios, the organizers publish a challenge consisting of the public key and some encrypted messages.[1]

Recently Pierrick Gaudry [Gau19a] discovered that the implemented encryption system used a concatenation of three ElGamal encryptions each with a key of length 256 bits. Gaudry showed that since the keys were too short, each private key could be retrieved in minutes. Moreover, Gaudry provided a very short and beautiful script computing the secret keys from the public keys. In the same note, Gaudry remarked that the implemented version of ElGamal worked in groups of even order, which means that it leaked a bit of the message. Both of these issues have been fixed in the current version of the Internet voting system. Gaudry [Gau19b] acknowledged that these specific issues have been fixed, but he is still doubtful about the security of the system. (In particular, he says [Gau19b] that he cannot properly test the system due to the

*Harvard University

[1] `https://github.com/moscow-technologies/blockchain-voting/blob/1d4f348681e961420a28e8a3a3d64719c4ddc628/encryption-keys/keys/public-key.json#L1-L5` and `https://github.com/moscow-technologies/blockchain-voting/blob/1d4f348681e961420a28e8a3a3d64719c4ddc628/encryption-keys/data/encrypted-datums.csv#L1-L10`

lack of documentation, and is concerned about potential flaws caused by the recent big changes fixing the key length issue.)

In this note we show that the new implementation of the encryption system also leaks a bit of the message. This is caused by the usage of ElGamal where the message is not mapped to the cyclic group under consideration. We show that this flaw potentially can be used for counting the number of votes cast for a candidate, which is illegal (until the end of the election period).

After the publication of this note, Pierrick Gaudry told us that he did warn the developers of the system that they should make sure that the system design does not allow attackers to encrypt elements outside of the group. As we show in this note, even the messages generated by the system (let alone an attacker) do not necessarily belong to the group.

2 Attack

The current implementation of the electronic voting system uses the ElGamal public key encryption scheme. The source code is available in [git19], and the encoding procedure is provided in Appendix A. Below we summarize the current implementation.

Let $p = 2q + 1$ and q be primes of length about 1024 bits, and let Q_p be the group of quadratic residues modulo p. Note that $|Q_p| = (p-1)/2 = q$. The system picks a $g \in Q_p$ generating Q_p. Let $\mathrm{pk} \in Q_p$ and $\mathrm{sk} \in \{1, \ldots, q\}$ be the public and secret key respectively, and $\mathrm{pk} = g^{\mathrm{sk}}$. A message $m \in \mathbb{Z}_p^*$ is encoded as a pair

$$E_{\mathrm{pk,sk}}(m) = (g^r, \mathrm{pk}^r \cdot m)\,,$$

where $r \in \{1, \ldots, q\}$ is a random number.

The problem with the current implementation is that m is allowed to be any integer from $\{1, \ldots, q-1\}$ which is naturally mapped to an element of $\mathbb{Z}_p^*$. For semantic security (under the Decisional Diffie-Hellman assumption), the message m should instead be hashed to one of the q elements of the group Q_p generated by g. In the case when m is not necessarily picked from the group of quadratic residues, the Decisional Diffie-Hellman assumption does not hold. Moreover, such a system is not semantically secure.

If $m \in Q_p$ is a quadratic residue, then for *every* choice of randomness of the encryption algorithm $E_{\mathrm{pk,sk}}(m) = (c_1, c_2)$, the second component of its encoding c_2 is a quadratic residue. Indeed, if $g, m \in Q_p$, then $g = x^2, m = y^2$ for $x, y \in \mathbb{Z}_p^*$, which implies that

$$c_2 = \mathrm{pk}^r \cdot m = g^{r \cdot \mathrm{sk}} \cdot y^2 = (x^{r \cdot \mathrm{sk}} y)^2 \in Q_p\,.$$

Similarly, if m is not a quadratic residue, then $\mathrm{pk}^r \cdot m$ is not a quadratic residue either.

Due to Euler's criterion, there is an efficient way to distinguish the two cases: $x \in \mathbb{Z}_p^*$ for a prime $p > 2$ is a quadratic residue if and only if $x^{(p-1)/2} = 1$. Since this exponentiation can be performed in time $\mathrm{poly}(\log p)$, this gives an efficient distinguishing attack.

In order to see that the implemented code indeed applies ElGamal to elements outside of the group generated by g, one can apply Euler's criterion to the published encrypted messages. In Appendix B we provide Python code showing that exactly five out of the ten published messages are quadratic residues modulo p.

3 Example

Due to the lack of documentation and only some code available online, we cannot say for sure which parts of ballots are encrypted. The following lines of code suggest that only the "deputy id" field might be encrypted[2]:

```
var dataToEncrypt = parseInt($(this).data('value'));
...
return signer.getSignedTransaction(votingId, dataToEncrypt, entropy,
encryptor);
```

and

```
<input class="bulletin__radio" type="radio" name="deputy"
value="{$deputy.id}" />
```

The "deputy id" field stores the id of the voter's chosen candidate. While the attack distinguishes two identical ballots with different votes with high probability regardless of which parts of the ballot are encrypted, for simplicity in this example we will assume that only the "deputy id" field is encrypted. Also, for this example we take the currently published group Z_p^* where

$$
\begin{aligned}
p = & 100627590814506256180379036786188261966005912425008608027910859704550882961591418803872072305745904601913015245097812875886798212712694662445323678201384359740027439588690880234391145675099291004487668846511981135309331094869021425403957856145722681330313515482620918593602329299394441379077427748866822254003\,.
\end{aligned}
$$

If the number of candidates is $k \leq 4$, and the deputy ids are integers from 1 to k, then the adversary can easily count the number of votes cast for candidate number 2. (In particular, if there are only two candidates, then the adversary can count the total number of votes for each of the candidates).

Indeed, since $1, 3$ and 4 are quadratic residues modulo p, and 2 is not, it suffices to raise the encrypted message to the power $(p-1)/2$ to check whether the vote is cast for candidate number 2. In Appendix C we give Python code emulating the implemented encryption system, and give an example of how to count the number of encryptions of the message 2 among encryptions of messages from 1 to 4.

We emphasize that this attack will distinguish (with high probability) two messages that differ only in their vote, regardless of the values of the deputy ids and which fields of the ballots are encrypted. Let k be the number of candidates. Note that exactly half of the elements of $\mathbb{Z}_p^*$ are quadratic residues. Assuming the uniform distribution of the plain messages we have that the probability of distinguishing a vote for one candidate from the votes for other candidates is 0.5 for $k = 2$, and is $\frac{k}{2^{k-1}}$ for $k > 2$.

[2]See the following links for the code on github: `https://github.com/moscow-technologies/blockchain-voting/blob/1d4f348681e961420a28e8a3a3d64719c4ddc628/voting-form/src/ballot/static/js/forms/mgik/election.js#L108-L123` and `https://github.com/moscow-technologies/blockchain-voting/blob/1d4f348681e961420a28e8a3a3d64719c4ddc628/voting-form/src/ballot/common/module_tpl/election/default/show.tpl#L59`

4 Suggested Fix

The described security vulnerability is a major issue of the implemented cryptographic scheme. We note that the provided attack does not recover the secret key as required by the public testing scenario [Pub19], but rather breaks the system without recovering the secret key.

On the other hand, we believe that there might be an easy way to fix this issue. Since the current implementation already assumes that the message $0 < m < q = (p-1)/2$ (see line 13 in the code block in Appendix A), one can use a trivial hashing procedure to map each such message m to the group of quadratic residues.

In the encoding procedure, we suggest to check whether $m^{(p-1)/2} = -1 \mod p$. If the equality holds, then m is not a quadratic residue, but $-m$ is. Therefore, one can set $m \leftarrow p - m$ and continue the encoding procedure.

In the decoding procedure, an analogous step will be required. If the decoded message $m \geq q$, then it also should be negated ($m \leftarrow p - m$).

Acknowledgements

I would like to thank Pierrick Gaudry and Noah Stephens-Davidowitz for their comments on an earlier version of this note.

References

[Gau19a] Pierrick Gaudry. Breaking the encryption scheme of the Moscow internet voting system. *arXiv preprint arXiv:1908.05127*, 2019.

[Gau19b] Pierrick Gaudry. Breaking the encryption scheme of the Moscow internet voting system. `https://members.loria.fr/PGaudry/moscow/`, 2019.

[git19] Public source code of the Moscow internet voting system. `https://github.com/moscow-technologies/blockchain-voting`, 2019.

[Kri19] Julia Krivonosova. Internet voting in Russia: how? `https://medium.com/@juliakrivonosova/internet-voting-in-russia-how-9382db4da71f`, 2019.

[Pub19] Public testing of the Internet voting system. `https://www.mos.ru/upload/documents/files/5381/Formal_Offer.pdf`, 2019.

A Current Implementation of ElGamal

Below we provide the current implementation of the ElGamal encryption in the Internet voting system.[3]

```
async encrypt(data, entropy) {
    if (!data) {
      throw new Error('Data must present!');
    }

    if (!entropy) {
      throw new Error('Entropy must present!');
    }

    const dataAsBI = new BigInt(data.toString());
    const entropyAsBI = new BigInt(entropy.toString());

    if (dataAsBI.compareTo(this.Q) >= 0) {
      throw new Error('Data to encrypt can not be bigger or equal that
      (P-1)/2!');
    }

    if (entropyAsBI.compareTo(BigInt.ONE) <= 0) {
      throw new Error('Entropy for Session Key must be Integer bigger than
      1!');
    }

    if (entropyAsBI.compareTo(this.moduleP) >= 0) {
      throw new Error('Entropy for Session Key can not be bigger or equal
      Basis Module P!');
    }

    const randomBigInt = await getRandomBigInt(BigInt.ONE,
    this.moduleP.subtract(BigInt.ONE));
    const xoredRandomBigInt = randomBigInt.xor(entropyAsBI);
    const sessionKey = trimBigInt(xoredRandomBigInt, this.moduleP.bitLength()
    - 1);

    const sharedKey = this.publicKey.modPow(sessionKey, this.moduleP);

    const a = this.generatorG.modPow(sessionKey, this.moduleP).toString();
    const b = sharedKey
      .multiply(dataAsBI)
      .remainder(this.moduleP)
      .toString();


    return { a, b };
  }
```

[3] https://github.com/moscow-technologies/blockchain-voting/blob/1d4f348681e961420a28e8a3a3d64719c4ddc628/smart-contracts/packages/crypto-lib/src/elGamal.js#L84-L125

B Published Encrypted Messages are Not Quadratic Residues

In this Appendix we use the provided public key and encrypted messages[4] to show that not all messages are quadratic residues in Z_p^*. Here $c2$ is the set of the second components of the encrypted messages (that is, each element of the set is $\text{pk}^r \cdot m$ where m is some plain message, pk is the public key, and r is a random number). The code shows that only five out of ten elements are quadratic residues (give 1 when raised to the power $(p-1)/2$).

```
p =
100627590814506256180379036786188261966005912425008608027910859704550882961591418803872072305745904601913015245097812875886798212712694662445323678201384359740027439588690880234391145675099291004487668846511981135309331094869021425403957856145722681330313515482620918593602329299394441379077427748866822254003

q = (p-1)//2

c2 = [
    869110015064974622517826385673193618336889788136649464373338293547389094044397448192792926328348698723340632646650502502743467906058388168970623263052860581382950559847777412555501704989450676046755496358356631412743565509639941737973454893064171740725143098561757549081224362414215648591783263203132049456491,
    329945787158463156253342824653891281130151930844449944715831357721279264495189216142745357056676629897986418517052061640312479742701073070752016109483404053598174999416617877699551805519137361275456656546769123076444375224889357541488942667685714188203805416972085863674686599803137288027861639262227344813980,
    256054513991066206766528731020219646413624546241484093114597729584964406701684357831590854518407777279459383097915161681981096625570956792081413077819709806694723689969137957383923170349530451483441188337477065322871518389975095982992061479564793810225632159787641001956296637123881826475110897878623324832483202,
    309361975515672696858470423522408342871717565418623822958588525166667621175580597972987902300728528688073267489198900774102263333080055036874256346941237089009381794632389798562078456796442958644789501357076108208779625474707032687737761473361742706781012217551529249331750729529106903054039467085120113440651,
    901892276593656973550635009417605368364785375514617599456318233190916831313053994704341622298458027052615259375645755548559901874024322932422684960561239260442729637671756134870576696053584273031857981168518983390538640849290557062400553071519187369524566082107009379533632083366956053084145043637897147823551,
    917647149158344453102657171361954468459150205108547086348288077416429086508880523401650934200991380942879591972292661384753907905599781678818799170526245002211336442034207826902363786376681934271623388852857592304132784015338462608883982538779159812545205628726986176857059796124483464704139139941
```

[4] https://github.com/moscow-technologies/blockchain-voting/blob/1d4f348681e961420a28e8a3a3d64719c4ddc628/encryption-keys/keys/public-key.json#L1-L5 and https://github.com/moscow-technologies/blockchain-voting/blob/1d4f348681e961420a28e8a3a3d64719c4ddc628/encryption-keys/data/encrypted-datums.csv#L1-L10

```
    244174120780,
    53180133541691920877303393106622876213880557470163604793597655634027675133
    60685116768376758300338878651961955633191844125587620057500524945640239322
    77996165942274611488630312874402187304375485303772307277867299568052321426
    13661312171461386140429576621530845469410809123204273518058446975266361694
    186911940244,
    96389110287648758509344773386657594488132549702589565012028823522666392603
    23174326871289534690190117827254235251942037419181816826781045590593293711
    55623633657479236340811419693309298082823008055773940379928788914612436976
    30183068655120651685499248763092459000930306871431366198968873609555301941
    599393034947,
    68868518968718401961947565883286957678496859516081208645391394051517430601
    54089569868014396600078685718742310976349636761884312463762214119090170143
    67814111630789237262689248078371187306393398854088463937893954685309796570
    18007065848405280697276892839194542147616119874097494557367533448036396670
    81357573332,
    59049331935932409191703521981449178033897833739363938803374780496048381081
    67852649116009537459386386032599267182731855221804003545963016545542412314
    67392800236514010370577555635998585837533974218865577533874244033450031333
    65685878245562130520111649077186632157205095851334912141011894784614717824
    328145876601
]

for i in range(len(c2)):
    print(pow(c2[i], q, p))
```

C Example

Here we show that with high probability one can distinguish encryptions of one message from the encryptions of other messages. In this example we use the currently provided public key for the ElGamal system.[5] We generate a hundred of random encryptions of the messages from 1 to 4. Then without the knowledge of the secret key, we correctly recover the number of encryptions of the message 2.

```
import random

p =
100627590814506256180379036786188261966005912425008608027910859704550882961591
141880387207230574590460191301524509781287588679821271269466244532367820138435
974002743958869088023439114567509929100448766884651198113530933109486902142 5
403957856145722681330313515482620918593602329299394441379077427748866822254003

q = (p-1)//2

g =
209381626636347175920501508716048119654501851478046536862103413700488755264656
519259880960814989775316937110581943338856604846551324970517112893162325748 2
209498106119027079663421722827363853173072102442359153533950239270315897468706
882583658777828108854618608942401856950013974374239277975257526286888685710 10
```

[5] https://github.com/moscow-technologies/blockchain-voting/blob/1d4f348681e961420a28e8a3a3d64719c4ddc628/encryption-keys/keys/public-key.json#L1-L5

```
pk =
69869939556699578412481990448414288405398435179055114398103563207158851026044
17268760023814098454244173381151935563527447828022461894488632630303059380694
21286985815829955355067923635563794223005965324347558568201679805005654774823
09964287055106318673831750120449183092965628809034498878410377075732012129326

def encrypt(data):
    if (data >= q):
      print('Data to encrypt can not be bigger or equal that (P-1)/2!')

    sessionKey = random.randint(1, p-1)
    sharedKey = pow(pk, sessionKey, p)
    a = pow(g, sessionKey, p)
    b = sharedKey * data % p
    return (a, b)


#number of candidates is 4, their ids are {1, 2, 3, 4}.
count = [0] * 5
encryptions = []
for i in range(100):
    candidate = random.randint(1, 4)
    count[candidate] += 1
    encryptions.append(encrypt(candidate))


for i in range(1, len(count)):
    print('Number of votes for candidate %d is %d' % (i, count[i]))

notQResidues = 0
for (a, b) in encryptions:
    if pow(b, q, p) != 1:
        notQResidues += 1

print('\nNumber of encrypted votes for candidate 2 is %d' % notQResidues)
```