# Breaking the encryption scheme of the Moscow internet voting system

Pierrick Gaudry

# Breaking the encryption scheme of the Moscow internet voting system

Pierrick Gaudry

CNRS, Inria, Université de Lorraine

August 2019

**Abstract**

In September 2019, voters for the election at the Parliament of the city of Moscow will be allowed to use an internet voting system. The source code of it has been made available for public testing. The encryption used in this system is a variant of ElGamal with key sizes that are too small. We explain how to retrieve the private keys from the public keys in a matter of minutes with easily available resources.

## 1 Context

**Disclaimer***: most of the context information here relies on secondary sources or automatic translation of texts written in Russian. There is a possibility of misunderstanding by the author.*

In September 2019 (one month in the future, at the date of the writing of this note), there will be elections in Moscow for the representatives at the Parliament of the city (the Moscow City Douma). During this election, up to half a million of voters will be allowed to use internet voting. The voting system that will be used has been the subject of a public test at the end of July, during which some of the source code was made public on Github [1].

Although we did not find (yet?) a public specification of the protocol in English, we understand that it uses the Ethereum blockchain, with its smart contract capabilities.

During the public test, the public code was updated every day, proposing new public keys and new encrypted data, and revealing the private keys and the original data of the day before. According to the README.md file, the goal was to decrypt the data in less than 12 hours, since this will be the duration of the election to be held in September.

## 2 Result

We will show in this note that the encryption scheme used in this part of the code is completely insecure. It can be broken in about 20 minutes using a standard personal computer, and using only free software that is publicly available. More precisely, it is possible to compute the private keys from the public keys. Once these are known, any encrypted data can be decrypted as quickly as they are created.

Without having read the protocol, it is hard to tell precisely the consequences, because, although we believe that this weak encryption scheme is used to encrypt the ballots, it is unclear how easy it is for an attacker to have the correspondence between the ballots and the voters. In the worst case scenario, the votes of all the voters using this system would be revealed to anyone as soon as they cast their vote.

After we contacted the people in charge in Russia before making this note public, they made a reply[1] on Iuliia Krivonosova's blog. They acknowledge that the keys are too small and they plan to upgrade to 1024 bits. They didn't provide a proper specification of the system yet.

## 3 The encryption scheme

We describe here what we have understood from the directory `smart-contracts/packages/crypto-lib/src/` of the public code [1].

The encryption is based on the ElGamal scheme. For a cyclic group $G$ of generator $g$, given a public key $h$ and a message $m$, the scheme produces a pair $\mathsf{Enc}_{g,h}(m) = [a, b]$. And knowing the secret key $x$ (i.e. the discrete logarithm of $h$ relative to $g$), it is possible to decrypt: $\mathsf{Dec}_{g,x}(a, b) = m$. The description in Wikipedia[2] is what is implemented, so we refer to it for further details. Let us just mention that if one part of the ciphertext ($a$ or $b$) is missing, then even knowing the secret key, there is no way to decrypt and get the original message.

In the Moscow system, things are a bit more complicated, because a variant using 3 levels of encryption is implemented. Let $G_1$, $G_2$, $G_3$, be three cyclic groups of generators $g_1$, $g_2$, $g_3$. There are three public keys $h_1$, $h_2$, $h_3$, one for each group. In order to encrypt a message $m$, they first compute the following:

- $[a_1, b_1] := \mathsf{Enc}_{g_1,h_1}(m)$
- $[a_2, b_2] := \mathsf{Enc}_{g_2,h_2}(a_1)$
- $[a_3, b_3] := \mathsf{Enc}_{g_3,h_3}(a_2)$

and then the ciphertext is the quadruple $[b_1, b_2, a_3, b_3]$. The values $a_1$ and $a_2$ are forgotten, but someone knowing the private keys $x_1$, $x_2$, $x_3$ corresponding to $h_1$, $h_2$, $h_3$, will be able to deduce $m$ as follows:

- $a_2 := \mathsf{Dec}_{g_3,x_3}(a_3, b_3)$
- $a_1 := \mathsf{Dec}_{g_2,x_2}(a_2, b_2)$
- $m := \mathsf{Dec}_{g_1,x_1}(a_1, b_1)$

The purpose of this multi-level encryption is not known to us. It can not be security, because this will be only marginally more secure that using only the group which is the most secure.

The main problem is due to the choice of $G_1$, $G_2$, $G_3$. In [1], in the `encryption-keys/keys` subdirectory, there is a file called public-key.json that contains a description of them. They are multiplicative groups of finite fields of prime orders $p_1$, $p_2$, $p_3$, each of them being Sophie Germain primes. There is nothing wrong with this apart from the

[1] `https://medium.com/p/9382db4da71f/responses/show`
[2] `https://en.wikipedia.org/wiki/ElGamal_encryption`

sizes of $p_1$, $p_2$, $p_3$, which are all less than 256-bit long. This is way, way too short to guarantee any security.

We will see below that discrete logarithms in such a small setting can be computed in a matter of minutes, thus revealing the secret keys, and then all the encrypted data becomes easy to decrypt.

Before going to this, we mention quickly another mistake in the design that could in itself have led to devastating attacks. The generators that are given in public-key.json are generators of the whole multiplicative group. However, due to the Chinese remainder theorem, it is a good practice to use a generator of prime order. In the present case, the generators will have their order divisible by 2, and therefore there is a huge risk that one bit of information leaks from a ciphertext. In a context like e-voting where a ballot can have a very simple form, this bit of information could reveal a lot of the vote (or even all of it in the case of a yes/no question). In principle, we would have had to investigate more in this direction. But due to the main attack that is much easier and far more powerful, we keep this as a remark.

# 4 Computing discrete logarithms

We now explain the state of the art of computing discrete logarithms in finite fields.

The best known algorithm for this task is the Number Field Sieve that was invented in the early 90's [2]. It is a complicated algorithm that uses heavy notions of number theory. Also, it is not suitable for computing discrete logarithms in very small finite fields, for which better algorithms are known. However, when the prime $p$ has more than a few dozens of digits, like the primes we are interested in, it becomes clearly the best.

For our attack, we used CADO-NFS [3] which is a free software implementation of the Number Field Sieve, the author of this note being one of the major developers of this piece of software. The last official release is rather old, so we used the development version, that is available from the public git repository.

CADO-NFS takes as input $p$ and $h$ and computes the discrete logarithm of $h$ in $GF(p)$ relative to an arbitrary generator. In order to get the discrete logarithm of $h$ relative to the given generator $g$, one has to run CADO-NFS again to get the logarithm of $g$, so that we can divide those two values (this is similar to what we do to compute the log in base 2 of a number when the pocket calculator provides only the log in base $e$). Hence, if $\log_{\text{cado}}(h)$ and $\log_{\text{cado}}(g)$ are the values returned by CADO-NFS as discrete logarithms of $h$ and $g$ respectively, then the discrete logarithm of $h$ relative to $g$ is just

$$\log_g(h) = \frac{\log_{\text{cado}}(h)}{\log_{\text{cado}}(g)}.$$

Fortunately, many parts of the computation can be shared between the two executions of CADO-NFS modulo the same $p$, and the software does indeed share them automatically.

Another complication is due to the fact that CADO-NFS assumes that we are in the prime order setting (as explained above, this should be what people do!), while here the order is twice a prime. This means that we have some additional easy operations to do in order to deduce the private keys corresponding to what is expected in the Moscow system. In more details, if we denote the order of the generator $g$ by $2\ell$, where

$\ell = (p-1)/2$ is a prime, then CADO-NFS gives only the result modulo $\ell$, while we need it modulo $2\ell$. Thanks to the Chinese remainder theorem, there are only two possibilities left for the private key, and it is easy to check which one corresponds to the public key.

All of these operations to be done in order to get the private key once CADO-NFS has outputted the two results are elementary number theory (just modular arithmetic) and take no time. This is included in the shell script given in Appendix, for reproducibility.

Once all the required software is downloaded and compiled, we measured the time to compute each of the 3 private keys, corresponding to the last public keys published in the repo [1] on August 1st. The computation was done on a personal computer equipped with a 4-core Intel i5-4590 processor at 3.3 GHz and 16 GB of RAM. It is running a standard Debian distribution.

The running times to retrieve the private keys are as follows:

| key number | time |
|---|---|
| 1 | 425 sec |
| 2 | 507 sec |
| 3 | 314 sec |

Note that the variation in the running time from one key to the other is not unusual for computations with moderately small primes.

Of course, for a real attack, the three private keys can be computed simultaneously on 3 machines in parallel. Indeed, the chaining involved in the multi-level ElGamal is not relevant for the keys, it occurs only during the encryption / decryption of messages.

Additionally to this immediate 3-fold parallelism for the attack, CADO-NFS also has some parallelism capabilities so that machines with more cores can reduce the time for a single key. However, there is some limit to it with the current implementation. For instance, the private key number 1 could be retrieved in 160 seconds on a machine with 64 cores.

## 5 Breaking the scheme

Now that we have everything in place, the scenario of the attack is the following. Before the day of elections, machines are prepared with all the required software downloaded, compiled and tested. As soon as the election opens, the public keys must be available, so that voters can encrypt their ballot. Then with the strategy we have just described and explicitly applied in the shell script given in Appendix, the attacker can retrieve all the private keys in less than 10 minutes using 3 standard personal computers, or even less if they have access to a more powerful machine. They can then decrypt all the data that is supposed to be secured by this encryption. In particular, if the ballots are protected with this, they must be considered as being in cleartext as soon as they are cast.

## 6 Remaining questions

- Why such small keys?

A possible explanation is a confusion with the key size that can be used when using elliptic curves for which the Number Field Sieve algorithm does not apply.

Another less excusable but still possible explanation might be related to the use of the Ethereum blockchain. In the Solidity programming language that is used to write smart contracts, the bit size of the largest supported integers is 256. Maybe the authors did not want to write a multiprecision arithmetic library that would have been required to deal with larger key sizes. This hypothesis is supported by frequent tests in the source code, checking that the big integers they manipulate are not bigger than `SOLIDITY_MAX_INT`.

- Where does the multi-level ElGamal comes from?

  This is a mystery. The only possible explanation we can think of is that the designers thought this would compensate for the too small key sizes of the primes involved. But 3 primes of 256 bits are really not the same as one prime of 768 bits.

- Are the ballots really encrypted with this scheme?

  From the source code, in particular in the subdirectory named `smart-contracts/packages/voting-lib/src/contracts`, our guess is that yes. However, reading a proper specification of the voting scheme would be the only way to know.

- Is there an easy fix?

  In principle, yes: increase the key size. The current general recommendation is at least 2048 bits. However, if the hypothesis that writing a multiprecision library for Solidity is a problem appears to be correct, there is no easy workaround. Using elliptic curves is probably a good idea, but again this is not a change that can be done in just a few hours of coding.

  Maybe there is an easier fix by changing the protocol, so that the smart contracts do not compute anything with the encrypted ballots.

## 7 Acknowledgements

Thanks to Iuliia Krivonosova and Robert Krimmer, for sharing some information about the Moscow internet voting. In particular Iuliia's blog post `https://medium.com/@juliakrivonosova/internet-voting-in-russia-how-9382db4da71f` was quite useful.

## References

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

[2] A. K. Lenstra and H. W. Lenstra, Jr., editors. *The development of the number field sieve*, volume 1554 of *Lecture Notes in Math.* Springer-Verlag, 1993.

[3] The CADO-NFS Development Team. CADO-NFS, an implementation of the number field sieve algorithm, 2019. Development version fdae0f9f382c, available at `http://cado-nfs.gforge.inria.fr/`.

# 8 Appendix: a shell script for the attack

```
## These are commands to be run on a Linux machine (Debian or Ubuntu).
## The main workhorse for the discrete logarithm computations is CADO-NFS,
## and we use GP-Pari as a 'pocket calculator' for modular arithmetic.

# install some packages
sudo apt install pari-gp jq
sudo apt install libgmp3-dev gcc g++ cmake libhwloc-dev
alias gpnoc="gp -q --default colors=\"no\""

# download and compile cado-nfs
cd /tmp
git clone https://scm.gforge.inria.fr/anonscm/git/cado-nfs/cado-nfs.git
cd cado-nfs
git checkout fdae0f9f382c  # most recent version at the time of writing
make cmake
make -j 4

# download blockchain-voting and extract public keys
cd /tmp
git clone https://github.com/moscow-technologies/blockchain-voting.git
cd blockchain-voting
git checkout d70986b2c4da  # most recent version at the time of writing
cd /tmp

# loop on the 3 public keys; could be done in parallel on 3 machines.
for i in {0,1,2}; do

  start=`date +%s`
  # extract the public key information
  p=`jq .modulos[$i] /tmp/blockchain-voting/encryption-keys/keys/public-key.json | tr -d \"`
  g=`jq .generators[$i] /tmp/blockchain-voting/encryption-keys/keys/public-key.json | tr -d \"`
  h=`jq .publicKeys[$i] /tmp/blockchain-voting/encryption-keys/keys/public-key.json | tr -d \"`
  ell=`echo "($p-1)/2" | gpnoc`

  # run cado-nfs to get log of h (takes a few minutes)
  wdir=`mktemp -d /tmp/cadorunXXXXXX`
  log_h=`/tmp/cado-nfs/cado-nfs.py -dlp -ell $ell workdir=$wdir target=$h $p`

  # run again to get log of generator (faster, since it reuses precomputed data)
  log_g=`/tmp/cado-nfs/cado-nfs.py $wdir/p75.parameters_snapshot.0 target=$g`

  # deduce private key
  x=`gpnoc <<EOF
xell=lift(Mod($log_h,$ell)/Mod($log_g,$ell)); half=lift(1/Mod(2,$ell));
x0=lift(Mod(2*half*xell, 2*$ell)); h0=lift(Mod($g,$p)^x0);
if (h0 != $h, x0=lift(Mod(2*half*xell+$ell, 2*$ell)));
x0
EOF`

  stop=`date +%s`
  echo "Private key number $((i+1)) is $x, computed in $((stop-start)) seconds."
done
```