On the (in)security of ElGamal in OpenPGP

Luca De Feo, Bertram Poettering and Alessandro Sorniotti
IBM Research Zürich

April 14, 2022, Real World Crypto, Amsterdam
How to hang a picture? (ISO 3103$\frac{1}{2}$)
How to hang a picture? (ISO 3103½)

1. Take hammer,
Details matter

How to hang a picture? (ISO 3103½)

1. Take hammer,
2. Strike nail...
Cryptographic standards, what’s the worse that could happen?

- Theoretical break.
- Side-channel leakage.
- Implementations secure in isolation, do not interoperate.
- **Implementations secure in isolation, insecure when interoperating.**
OpenPGP

- One of two standards for end-to-end email encryption (along with S/MIME).
- Many implementations:
  GnuPG, Botan (rnp/Thunderbird), Go (Protonmail), Libcrypto++, ...
- IETF RFCs:
  RFC 4880  OpenPGP Message Format
  RFC 3156  MIME Security with OpenPGP
  RFC 5581  The Camellia Cipher in OpenPGP
  RFC 6637  Elliptic Curve Cryptography in OpenPGP
### OpenPGP algorithms

**Hash Functions:** MD5, RIPE-MD, SHA-1, SHA-2.

**Symmetric Ciphers:** IDEA, TripleDES, CAST5, Blowfish, AES, Twofish, Camellia.

**Public Key Encryption:** RSA, ElGamal, ECDH.

**Signature Algorithms:** RSA, DSA, ECDSA.

---

**RFC 4880 (dated November 2007)**

"Implementations MUST implement DSA for signatures, and **ElGamal** for encryption. Implementations SHOULD implement RSA […]"
Public key algorithms specifications in OpenPGP

- **RSA**  PKCS #1
- **ECDH** NIST SP 800-56A  +  RFC 6637
- **DSA**  FIPS 186-2
- **ECDSA**  FIPS 186-3
- **ElGamal**  *El Gamal ’85*  /  *Handbook of Applied Cryptography ’97*
ElGamal according to the OpenPGP standard?

### 8.4.1 Basic ElGamal encryption

#### 8.17 Algorithm Key generation for ElGamal public-key encryption

SUMMARY: each entity creates a public key and a corresponding private key. Each entity $A$ should do the following:

1. Generate a large random prime $p$ and a generator $\alpha$ of the multiplicative group $\mathbb{Z}_p^*$ of the integers modulo $p$ (using Algorithm 4.84).
2. Select a random integer $a$, $1 \leq a \leq p - 2$, and compute $\alpha^a \mod p$ (using Algorithm 2.143).
3. $A$’s public key is $(p, \alpha, \alpha^a)$; $A$’s private key is $a$.

©1997 by CRC Press, Inc. — See accompanying notice at front of chapter.

#### 8.18 Algorithm ElGamal public-key encryption

SUMMARY: $B$ encrypts a message $m$ for $A$, which $A$ decrypts.

1. **Encryption.** $B$ should do the following:
   
   (a) Obtain $A$’s authentic public key $(p, \alpha, \alpha^a)$.
   
   (b) Represent the message as an integer $m$ in the range $\{0, 1, \ldots, p - 1\}$.
   
   (c) Select a random integer $k$, $1 \leq k \leq p - 2$.
   
   (d) Compute $\gamma = \alpha^k \mod p$ and $\delta = m \cdot (\alpha^a)^k \mod p$.
   
   (e) Send the ciphertext $c = (\gamma, \delta)$ to $A$.

2. **Decryption.** To recover plaintext $m$ from $c$, $A$ should do the following:

   (a) Use the private key $a$ to compute $\gamma^{p-1-a} \mod p$ (note: $\gamma^{p-1-a} = \gamma^{-a} = \alpha^{-ak}$).

   (b) Recover $m$ by computing $(\gamma^{-a} \cdot \delta) \mod p$.

---

## II. The Public Key System

First, the Diffie–Hellman key distribution scheme is reviewed. Suppose that $A$ and $B$ want to share a secret $K_{AB}$, where $A$ has a secret $x_A$ and $B$ has a secret $x_B$. Let $p$ be a large prime and $\alpha$ be a primitive element mod $p$, both known. $A$ computes $y_A = \alpha^{x_A} \mod p$, and sends $y_A$. Similarly, $B$ computes $y_B = \alpha^{x_B} \mod p$ and sends $y_B$. Then the secret $K_{AB}$ is computed as

$$K_{AB} = \alpha^{x_A \cdot x_B} \mod p = y_A^{x_B} \mod p = y_B^{x_A} \mod p.$$ 

In any of the cryptographic systems based on discrete logarithms, $p$ must be chosen such that $p - 1$ has at least one large prime factor. If $p - 1$ has only small prime factors, then computing discrete logarithms is easy (see [8]).

Now suppose that $A$ wants to send $B$ a message $m$, where $0 \leq m \leq p - 1$. First $A$ chooses a number $k$ uniformly between 0 and $p - 1$. Note that $k$ will serve as the secret $x_A$ in the key distribution scheme. Then $A$ computes the “key”

$$K = y_B^k \mod p,$$

where $y_B = \alpha^{x_B} \mod p$ is either in a public file or is sent by $B$. The encrypted message (or ciphertext) is then the pair $(c_1, c_2)$, where

$$c_1 = \alpha^k \mod p \quad c_2 = Km \mod p$$

and $K$ is computed in (1).
ElGamal in the wild (OpenPGP ecosystem)

Large prime $p$  Safe prime  “Schnorr” prime  “Lim-Lee” prime  other

Generator $\alpha$  primitive element  generates subgroup

Private key  $0 < a < p$  “short exponent” optimisation

Ephemeral key  $0 < k < p$  “short exponent” optimisation
What could possibly go wrong?

Our results

- Each of GnuPG, Botan and Libcrypto++ implements ElGamal in a different, non-RFC-4880-compliant way:
  - Each is *secure taken in isolation*.
  - They are interoperable: functionally and securely.
Our results

- Each of GnuPG, Botan and Libcrypto++ implements ElGamal in a different, non-RFC-4880-compliant way:
  - Each is secure taken in isolation.
  - They are interoperable: functionally and securely.

- Go does not implement ElGamal key generation and is the least offender.
Our results

- Each of GnuPG, Botan and Libcrypto++ implements ElGamal in a different, non-RFC-4880-compliant way:
  - Each is **secure taken in isolation**.
  - They are interoperable: functionally and securely.

- We analyse 800K registered PGP ElGamal public keys:
  - 2K of them are exposed to **practical plaintext recovery** when GnuPG, Botan, Libcrypto++ (or any other library using the “short exponent” optimisation) encrypts to them. We call these **cross-configuration** attacks.

- Go does not implement ElGamal key generation and is the **least offender**.
Our results

- Each of **GnuPG**, **Botan** and **Libcrypto++** implements **ElGamal** in a different, non-RFC-4880-compliant way:
  - Each is **secure taken in isolation**.
  - They are interoperable: functionally and securely.

- We analyse 800K registered PGP ElGamal public keys:
  - 2K of them are exposed to **practical plaintext recovery** when **GnuPG**, **Botan**, **Libcrypto++** (or any other library using the “short exponent” optimisation) encrypts to them. We call these **cross-configuration** attacks.

- **Go** does not implement ElGamal key generation and is the **least offender**.

- We find side channels leaking ElGamal secret keys in **GnuPG, Go** and **Libcrypto++**:
  - **GnuPG** claimed to be side-channel resistant.
  - Our attack against **GnuPG** becomes more powerful in the **cross-configuration** scenario.
Prime generation

Goal: prime $p$ with at least one large prime factor $q | (p - 1)$.

Safe primes: $p = 2q + 1$:
- Considered kind of expensive, back in the ’90s.

“Lim-Lee” primes: $p = 2q_1 q_2 \cdots q_r$, all $q_i$ large:
- Cheaper than safe primes,
- Protecting against the same attacks.

“Schnorr” primes: $p = 2qf + 1$, with $f$ arbitrary:
- Cheapest,
- Popularized by Schnorr signatures, DSA, FIPS-186-2.

Random primes: risky, don’t do it!

Other: your imagination is the only limitation!
800K registered OpenPGP ElGamal public keys

Safe primes

- 16 “standardized” primes

- generated (Libcrypto++/Botan?)

Lim–Lee?

GnuPG?

Schnorr / Other

“quasi-safe”
800K registered OpenPGP ElGamal public keys

Safe primes
- 16 “standardized” primes
- “quasi-safe”

Lim–Lee?
- GnuPG?
- Schnorr / Other

plaintext recovery attack

De Feo, Poettering, Sorniotti (IBM Research) On the (in)security of ElGamal in OpenPGP Apr 14, 2022, RWC Amsterdam
Discrete log: when $\alpha$ is primitive

$$p - 1 = 2 \cdot q \cdot \ell_3 \cdot \ell_4 \cdots$$

$$\alpha^x$$
Discrete log: when $\alpha$ is primitive

$$ p - 1 = 2 \cdot q \cdot \ell_3 \cdot \ell_4 \cdots $$

Pohlig–Hellman

$$ \alpha^x \xrightarrow{\alpha_{\ell_2}^x} \alpha_{\ell_q}^x \xrightarrow{\alpha_{\ell_3}^x} \alpha_{\ell_4}^x $$
Discrete log: when $\alpha$ is primitive

$$p - 1 = 2 \cdot q \cdot \ell_3 \cdot \ell_4 \cdots$$

Pohlig–Hellman

$$\alpha^x \to \alpha_2^x \quad \alpha_q^x \quad \alpha_3^x \quad \alpha_4^x$$

$$x \mod 2 \quad ?? \quad x \mod \ell_3 \quad x \mod \ell_4$$
Discrete log: when $\alpha$ is primitive and $x$ is “short”

$$p - 1 = 2 \cdot q \cdot \ell_3 \cdot \ell_4 \cdots$$

Pohlig–Hellman

$$\begin{align*}
\alpha^x \\
\alpha_2^x & \quad \alpha_q^x & \quad \alpha_3^x & \quad \alpha_4^x \\
\downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow \\
x \mod 2 & \quad ?? & \quad x \mod \ell_3 & \quad x \mod \ell_4
\end{align*}$$

CRT

$$x \mod 2\ell_3\ell_4 \cdots$$
ElGamal Encryption

\[ p \quad \text{prime} \]

\[ \alpha \mod p \quad \text{generator} \]

\[ \alpha^x = X \quad \text{public key} \]
ElGamal Encryption

\( p \) prime
\( \alpha \mod p \) generator
\( \alpha^x = X \) public key

\( m \) message
\( y \) random

On the (in)security of ElGamal in OpenPGP

De Feo, Poettering, Sorniotti (IBM Research)
Apr 14, 2022, RWC Amsterdam
ElGamal Encryption

\[ p \] prime
\[ \alpha \mod p \] generator
\[ \alpha^x = X \] public key

\[ m \] message
\[ y \] random

\[( Y = \alpha^y, \quad X^y \cdot m ) \] encryption

\[ m = X^y \cdot m / Y^x \] decryption
ElGamal Encryption

\[ p \] prime
\[ \alpha \mod p \] generator
\[ \alpha^x = X \] public key

\[ m \] message
\[ y \] random

\[ (p - 1) = \]
- safe: \( 2 \cdot q \)
- Schnorr: \( 2 \cdot f \cdot q \)
- Lim-Lee: \( 2 \cdot q \cdot q_2 \cdots q_r \)

\( q \) are “large” primes

\[ \alpha \] generates all of \( \mathbb{Z}_p^* \)
\[ x \in [1, p - 1] \]
\[ y \in [1, p - 1] \]

“short”

De Feo, Poettering, Sorniotti (IBM Research)  On the (in)security of ElGamal in OpenPGP  Apr 14, 2022, RWC Amsterdam
ElGamal Encryption

Bingo 0: key recovery from public key only (van Oorschot–Wiener)

\[(p - 1) = \begin{array}{c|c|c|c}
\text{safe} & 2 \cdot q & 2 \cdot f \cdot q & 2 \cdot q \cdot q_2 \cdots q_r \\
\text{Schnorr} & & & (q \text{ are “large” primes}) \\
\text{Lim-Lee} & & & (other possible)
\end{array}\]

\[\alpha \text{ generates all of } \mathbb{Z}_p^* \quad \text{generates subgroup of order } q \]

\[x \in [1, p - 1] \quad \text{“short”} \]

\[y \in [1, p - 1] \quad \text{“short”}\]
ElGamal Encryption

Bingo 0: key recovery from public key only (van Oorschot–Wiener)

Bingo 1: message recovery from single ciphertext (this work)

\[(p - 1) = 2 \cdot q \quad \text{safes} \quad 2 \cdot f \cdot q \quad \text{Schnorr} \quad 2 \cdot q \cdot q_2 \cdots q_r \quad \text{Lim-Lee} \quad (q \text{ are “large” primes)}\]

\[\alpha \quad \text{generates all of } \mathbb{Z}_p^* \quad \text{generates subgroup of order } q \quad (\text{other possible})\]

\[x \in [1, p - 1] \quad \text{“short”}\]

\[y \in [1, p - 1] \quad \text{“short”}\]
ElGamal Encryption in OpenPGP

GnuPG: Lim-Lee, generates all $\mathbb{Z}_p^*$, short exponents.

\begin{align*}
(p - 1) &= 2 \cdot q \\
2 \cdot q &= 2 \cdot f \cdot q \\
2 \cdot q \cdot q_2 \cdots q_r &= (q \text{ are “large” primes})
\end{align*}

\begin{align*}
\alpha &= \text{generates all of } \mathbb{Z}_p^* \\
\text{generates subgroup of order } q &= \text{generates subgroup of order } q \\
\text{“short”} &= \text{“short”}
\end{align*}
ElGamal Encryption in OpenPGP

**GnuPG:** Lim-Lee, generates all $\mathbb{Z}_p^*$, short exponents.

**Libcrypto++/Botan:** safe primes, generates subgroup, short exponents.

\[
(p - 1) = \begin{cases} 
2 \cdot q & \text{safe} \\
2 \cdot f \cdot q & \text{Schnorr} \\
2 \cdot q \cdot q_2 \cdots q_r & \text{Lim-Lee}
\end{cases}
\]

$q$ are "large" primes

\[
\alpha \quad \text{generates all of } \mathbb{Z}_p^* \\
x \in \quad [1, p - 1] \\
y \in \quad [1, p - 1]
\]

"short"

(Other possible)
ElGamal Encryption in OpenPGP

GnuPG: Lim-Lee, generates all $\mathbb{Z}_p^*$, short exponents.

Libcrypted++/Botan: safe primes, generates subgroup, short exponents.

Go: no key generation, $y \in [1, p-1]$.

$$(p - 1) = 2 \cdot q \quad 2 \cdot f \cdot q \quad 2 \cdot q \cdot q_2 \cdots q_r$$

$q$ are “large” primes)

$\alpha$ generates all of $\mathbb{Z}_p^*$

generates subgroup of order $q$

$x \in [1, p-1]$ “short”

$y \in [1, p-1]$ “short”
ElGamal Encryption in OpenPGP

**GnuPG:** Lim-Lee, generates all \( \mathbb{Z}_p^* \), short exponents.

**Libcrypto++/Botan:** safe primes, generates subgroup, short exponents.

Go: no key generation, \( y \in [1, p - 1] \).

\[
(p - 1) = 2 \cdot q \quad \text{safe} \quad 2 \cdot f \cdot q \quad \text{Schnorr} \quad 2 \cdot q \cdot q_2 \cdots q_r \quad \text{Lim-Lee} \quad (q \text{ are “large” primes})
\]

\( \alpha \) generates all of \( \mathbb{Z}_p^* \) \quad \text{generates subgroup of order } q

\( x \in [1, p - 1] \) \quad \text{“short”}

\( y \in [1, p - 1] \) \quad \text{“short”}
<table>
<thead>
<tr>
<th>prime type</th>
<th>group size</th>
<th>quantity since 2016</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>total</td>
</tr>
<tr>
<td>Safe prime I</td>
<td>x</td>
<td>472,518</td>
</tr>
<tr>
<td>Safe prime II</td>
<td>x</td>
<td>107,339</td>
</tr>
<tr>
<td>Lim–Lee I</td>
<td>?</td>
<td>211,271</td>
</tr>
<tr>
<td>Lim–Lee II</td>
<td>?</td>
<td>47</td>
</tr>
<tr>
<td>Quasi-safe I</td>
<td>x</td>
<td>15,592</td>
</tr>
<tr>
<td>Quasi-safe II</td>
<td>x</td>
<td>20</td>
</tr>
<tr>
<td>Quasi-safe III</td>
<td>x</td>
<td>26,199</td>
</tr>
<tr>
<td>Schnorr I</td>
<td>?</td>
<td>828</td>
</tr>
<tr>
<td>Schnorr II</td>
<td>?</td>
<td>27</td>
</tr>
<tr>
<td>Schnorr III</td>
<td>x</td>
<td>1,304</td>
</tr>
</tbody>
</table>
Side channel vulnerabilities in exponentiation → Key recovery

Threat model
- Co-located attacker;
- Targets the exponentiation in the decryption routine;
- Must trigger decryption (e.g., email decryption).

Techniques
FLUSH+RELOAD (instruction cache), PRIME+PROBE (data cache).

Findings

<table>
<thead>
<tr>
<th>Library</th>
<th>Key</th>
<th>Libcrypto++</th>
<th>Go</th>
<th>GnuPG</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \in [1, p - 1]$</td>
<td></td>
<td></td>
<td></td>
<td>unfeasible</td>
</tr>
<tr>
<td>GnuPG</td>
<td></td>
<td>trivial</td>
<td>easy</td>
<td>unfeasible/state</td>
</tr>
<tr>
<td>Libcrypto++/Botan</td>
<td></td>
<td></td>
<td></td>
<td>state/commodity*</td>
</tr>
</tbody>
</table>

*Verified experimentally on 2048 bits key.
Our results

- Each of GnuPG, Botan and Libcrypto++ implements ElGamal in a different, non-RFC-4880-compliant way:
  - Each is secure taken in isolation.
  - They are interoperable: functionally and securely.

- We analyse 800K registered PGP ElGamal public keys:
  - 2K of them are exposed to practical plaintext recovery when GnuPG, Botan, Libcrypto++ (or any other library using the “short exponent” optimisation) encrypts to them. We call these cross-configuration attacks.

- Go does not implement ElGamal key generation and is the least offender.

- We find side channels leaking ElGamal secret keys in GnuPG, Go and Libcrypto++:
  - GnuPG claimed to be side-channel resistant.
  - Our attack against GnuPG becomes more powerful in the cross-configuration scenario.