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

Details matter

How to hang a picture? (ISO 3103½)

Details matter

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

- An IETF Encryption standard, since 1998.
- 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 α of the multiplicative group $\mathbb{Z}_p^*$ of the integers modulo p (using Algorithm 4.84).
2. Select a random integer a, $1 \le a \le p-2$, and compute $\alpha^a \bmod p$ (using Algorithm 2.143).
3. A's public key is (p, α, α^a); A's private key is a.

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, α, α^a).
   (b) Represent the message as an integer m in the range $\{0, 1, \dots, p-1\}$.
   (c) Select a random integer k, $1 \le k \le p-2$.
   (d) Compute $\gamma = \alpha^k \bmod p$ and $\delta = m \cdot (\alpha^a)^k \bmod p$.
   (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 $\gamma^{p-1-a} \bmod p$ (note: $\gamma^{p-1-a} = \gamma^{-a} = \alpha^{-ak}$).
   (b) Recover m by computing $(\gamma^{-a}) \cdot \delta \bmod p$.

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 α be a primitive element mod p, both known. A computes $y_A \equiv \alpha^{x_A} \bmod p$, and sends y_A. Similarly, B computes $y_B \equiv \alpha^{x_B} \bmod p$ and sends y_B. Then the secret K_{AB} is computed as

$$\begin{aligned} K_{AB} &\equiv \alpha^{x_A x_B} \bmod p \\ &\equiv y_A^{x_B} \bmod p \\ &\equiv y_B^{x_A} \bmod p. \end{aligned}$$

Ma
This
Gran
Cryp
Th
Univ
1501

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 $0 \le m \le p-1$. 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"

$$K \equiv y_B^k \bmod p, \tag{1}$$

where $y_B \equiv \alpha^{x_B} \bmod p$ 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

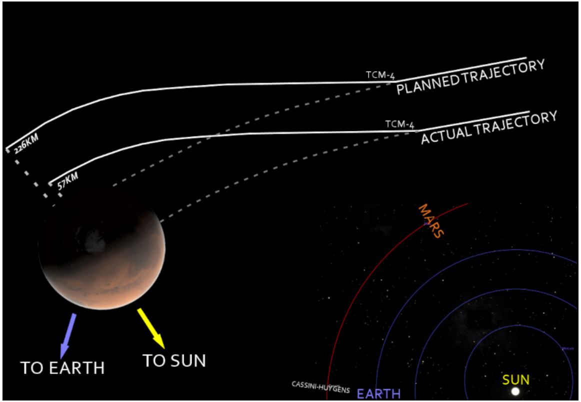
$$c_1 \equiv \alpha^k \bmod p \qquad c_2 \equiv Km \bmod p \tag{2}$$

and K is computed in (1).

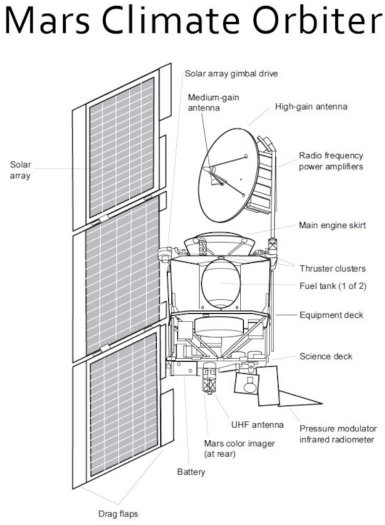
ElGamal in the wild (OpenPGP ecosystem)

Large prime p	Safe prime	"Schnorr" prime	"Lim-Lee" prime	other
Generator α	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?



credit: Wikipedia



Mars Climate Orbiter spacecraft

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.

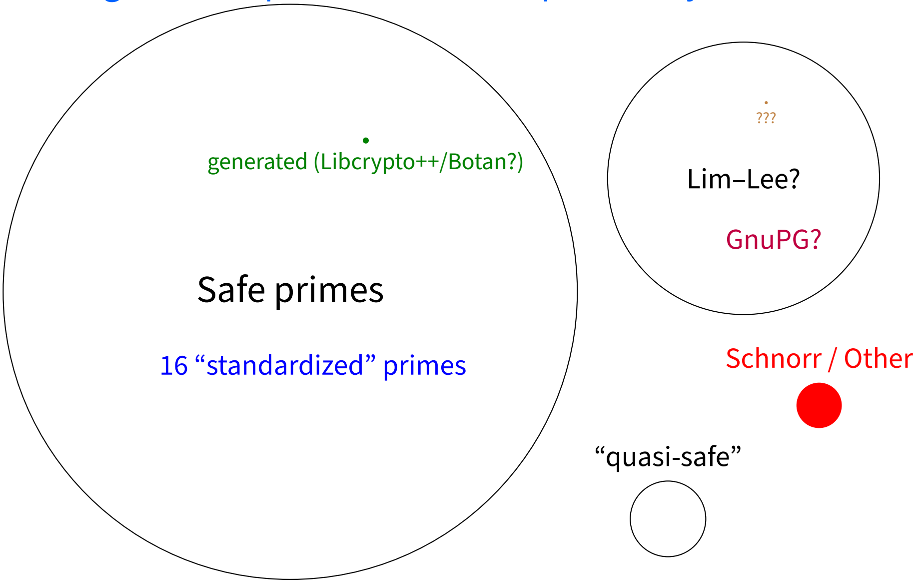
"Schnorr" primes: $p = 2qf + 1$, with f arbitrary:

- Cheapest,
- Popularized by Schnorr signatures, DSA, FIPS-186-2.

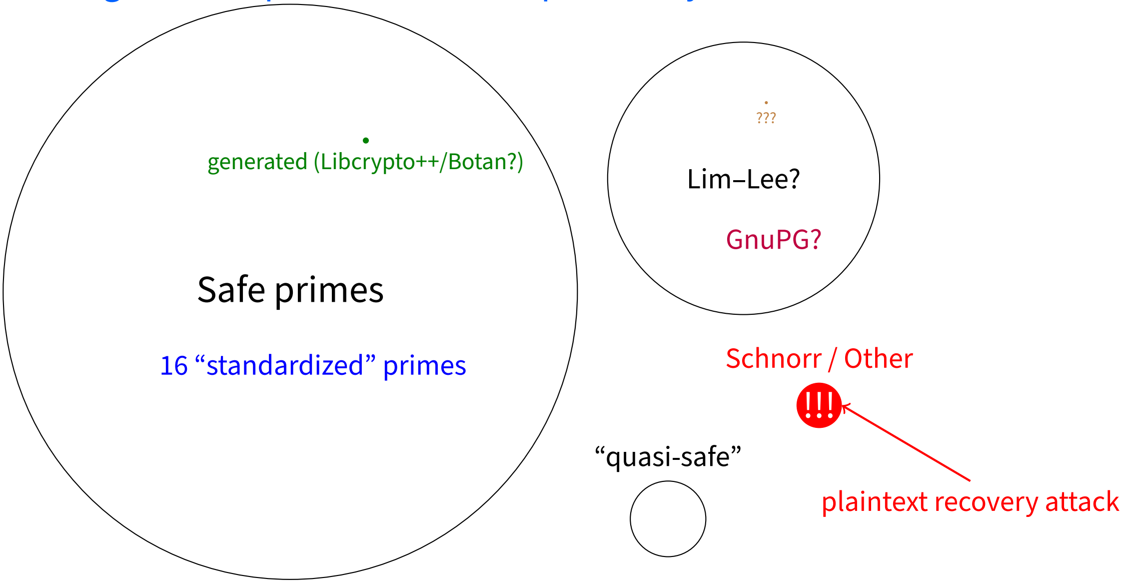
Random primes: risky, don't do it!

Other: your imagination is the only limitation!

800K registered OpenPGP ElGamal public keys



800K registered OpenPGP ElGamal public keys



Discrete log: when α is primitive

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

$$\alpha^x$$

Discrete log: when α is primitive

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

Pohlig–Hellman

$$\begin{tikzcd}
 & & \alpha^x \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] & \\
\alpha_2^x & \alpha_q^x & \alpha_3^x & \alpha_4^x
\end{tikzcd}$$

Discrete log: when α is primitive

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

Pohlig–Hellman

$$\begin{tikzcd}
 & & \alpha^x \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] & \\
\alpha_2^x \arrow[d] & \alpha_q^x & \alpha_3^x \arrow[d] & \alpha_4^x \arrow[d] \\
x \bmod 2 & ?? & x \bmod \ell_3 & x \bmod \ell_4
\end{tikzcd}$$

Discrete log: when α is primitive and x is “short”

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

Pohlig–Hellman

CRT

$$\begin{tikzcd}
 & \alpha^x \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
\alpha_2^x \arrow[d] & \alpha_q^x & \alpha_3^x \arrow[d] & \alpha_4^x \arrow[d] \\
x \bmod 2 \arrow[dr] & ?? & x \bmod \ell_3 \arrow[dl] & x \bmod \ell_4 \arrow[dll] \\
 & x \bmod 2\ell_3\ell_4\cdots & &
\end{tikzcd}$$

ElGamal Encryption

p	prime
$\alpha \bmod p$	generator
$\alpha^x = X$	public key

ElGamal Encryption

p	prime	m	message
$\alpha \bmod p$	generator	y	random
$\alpha^x = X$	public key		

ElGamal Encryption

p	prime
$\alpha \bmod p$	generator
$\alpha^x = X$	public key

m	message
y	random

$$(Y = \alpha^y, \quad X^y \cdot m) \qquad \text{encryption}$$

$$m = X^y \cdot m / Y^x \qquad \text{decryption}$$

ElGamal Encryption

p	prime	m	message
$\alpha \bmod p$	generator	y	random
$\alpha^x = X$	public key		

	safe	Schnorr	Lim-Lee	
$(p-1) =$	$2 \cdot q$	$2 \cdot f \cdot q$	$2 \cdot q \cdot q_2 \cdots q_r$	(q are "large" primes)
α	generates all of $\mathbb{Z}_p^*$		generates subgroup of order q	(other possible)
$x \in$	$[1, p-1]$		"short"	
$y \in$	$[1, p-1]$		"short"	

ElGamal Encryption

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

	safe	Schnorr	Lim-Lee	
$(p-1) =$	$2 \cdot q$	$2 \cdot f \cdot q$	$2 \cdot q \cdot q_2 \cdots q_r$	(q are "large" primes)
α	generates all of $\mathbb{Z}_p^*$		generates subgroup of order q	(other possible)
$x \in$	$[1, p-1]$		"short"	
$y \in$	$[1, p-1]$		"short"	

ElGamal Encryption

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

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

	safe	Schnorr	Lim-Lee	
$(p-1) =$	$2 \cdot q$	$2 \cdot f \cdot q$	$2 \cdot q \cdot q_2 \cdots q_r$	(q are “large” primes)
α	generates all of $\mathbb{Z}_p^*$		generates subgroup of order q	(other possible)
$x \in$		$[1, p-1]$	“short”	
$y \in$		$[1, p-1]$	“short”	

ElGamal Encryption in OpenPGP

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

	safe	Schnorr	Lim-Lee	
$(p-1) =$	$2 \cdot q$	$2 \cdot f \cdot q$	$2 \cdot q \cdot q_2 \cdots q_r$	(q are "large" primes)
α	generates all of $\mathbb{Z}_p^*$		generates subgroup of order q	(other possible)
$x \in$	$[1, p-1]$		"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.

	safe	Schnorr	Lim-Lee	
$(p-1) =$	$2 \cdot q$	$2 \cdot f \cdot q$	$2 \cdot q \cdot q_2 \cdots q_r$	(q are "large" primes)
α	generates all of $\mathbb{Z}_p^*$		generates subgroup of order q	(other possible)
$x \in$	$[1, p-1]$		"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, $y \in [1, p-1]$.

	safe	Schnorr	Lim-Lee	
$(p-1) =$	$2 \cdot q$	$2 \cdot f \cdot q$	$2 \cdot q \cdot q_2 \cdots q_r$	(q are “large” primes)
α	generates all of $\mathbb{Z}_p^*$		generates subgroup of order q	(other possible)
$x \in$	$[1, p-1]$		“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, $y \in [1, p-1]$.

	safe	Schnorr	Lim-Lee	
$(p-1) =$	$2 \cdot q$	$2 \cdot f \cdot q$	$2 \cdot q \cdot q_2 \cdots q_r$	(q are "large" primes)
α	generates all of $\mathbb{Z}_p^*$		generates subgroup of order q	(other possible)
$x \in$	$[1, p-1]$		"short"	
$y \in$	$[1, p-1]$		"short"	

800K registered OpenPGP ElGamal public keys

prime type	group size			quantity	
	$p-1$	q	other	total	since 2016
Safe prime I	x			472,518	783
Safe prime II		x		107,339	219
Lim–Lee I	?			211,271	6,003
Lim–Lee II		?		47	24
Quasi-safe I	x			15,592	89
Quasi-safe II		x		20	3
Quasi-safe III			x	26,199	125
Schnorr I	?			828	810
Schnorr II		?		27	26
Schnorr III			x	1,304	1,300

Side channel vulnerabilities in exponentiation $\rightarrow$ 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

Key \ Library	Libcrypto++	Go	GnuPG
$x \in [1, p-1]$			unfeasible
GnuPG	trivial	easy	unfeasible/state
Libcrypto++/Botan			state/commodity*

*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.