ElGamal key switching

Michael Brand

November 23, 2023

Executive Summary

We describe a method for "key switching", that is, the oblivious replacement of one cryptographic key with another, for use in the ElGamal cryptosystem, in FinTracer-like scenarios.

1 Introduction

In cryptography, *key switching* refers to the oblivious replacement of one cipher key by another.

In this document, we will describe how to perform such key switching in the ElGamal encryption scheme.

We also describe how key-switching may be beneficial in FinTracer-like scenarios, but at this point mainly from a mathematical point of view. We do not introduce a specific scenario that would make it intel-interesting.

We begin in Section 2 where we discuss the necessary mathematical background. In Section 3 we describe the actual key-switching algorithm. We then conclude in Section 4 by describing use scenarios for the technique.

Note 1.1. Key switching being a fairly elementary procedure, mathematically, I expect ElGamal key-switching is well-documented in the public literature, even though I was not immediately able to find for it a public source. Nevertheless, the availability of such an operation can potentially be important for future features in the FinTracer related suite of algorithms, for which reason it is documented here.

2 Mathematical background

2.1 Ed25519

The mathematical background regarding Ed25519, as used in the context of the FinTracer algorithm, is largely as was described in [3]. We repeat it here briefly.

Additive ElGamal encryption is an operation performed on an Abelian group. In our case, this is the Ed25519 additive group. This is a cyclic group of prime order p, so its group operation is isomorphic to addition in $\mathbb{Z}/p\mathbb{Z}$. The isomorphic mapping is one that can be explicitly calculated via an *encoding* operation which we denote Ed25519(). If a is an element of $\mathbb{Z}/p\mathbb{Z}$, then Ed25519(a) is an element of Ed25519. For any two such elements, a and b, of $\mathbb{Z}/p\mathbb{Z}$, the isomorphism guarantees

$$\text{Ed25519}(a) + \text{Ed25519}(b) = \text{Ed25519}(a + b),$$

where the first "+" is the Ed25519 group operation and the second "+" is addition in $\mathbb{Z}/p\mathbb{Z}$ (i.e., addition modulo p).

Importantly, it is not computationally feasible to calculate the inverse transform to encoding. Thus, it is not practically possible to "decode" a general element of Ed25519.

We can now define a new multiplication operation on Ed25519 that will be isomorphic, under the same mapping, to $\mathbb{Z}/p\mathbb{Z}$ multiplication, making Ed25519 a field. This operation will be defined so as to satisfy

$$\text{Ed25519}(a)\text{Ed25519}(b) = \text{Ed25519}(ab)$$

for every a and b in $\mathbb{Z}/p\mathbb{Z}$.

Moreover, the same isomorphism can be used to describe Ed25519 as a $\mathbb{Z}/p\mathbb{Z}$-module, i.e. defining a product between an element of $\mathbb{Z}/p\mathbb{Z}$ and an element of Ed25519 whose result is an element of Ed25519. This gives us

$$\text{Ed25519}(a)\text{Ed25519}(b) = \text{Ed25519}(ab) = a\text{Ed25519}(b).$$

Note that because

$$ab = \underbrace{b + \cdots + b}_{a \text{ times}},$$

the module product is practically computable: if $x = \text{Ed25519}(b)$, then

$$ax = \underbrace{x + \cdots + x}_{a \text{ times}}.$$

Thus, module product can be computed via group addition, which is a computationally feasible operation.[1]

2.2 ElGamal encryption

In previous documents, e.g. [1], we described ElGamal encryption as a black box taking a plaintext m and returning a ciphertext q. In this document we will open this black box.

The ElGamal ciphertext is a pair of Ed25519 elements: $q = (x, e_m)$. In this document, we will refer to x as the *auxiliary key* and e_m as the *encrypted message*, where m is the *message*.

The formula for ElGamal encryption is as follows:

$$e_m = sx + m.$$

Here, x, the auxiliary key, is chosen randomly and uniformly among all Ed25519 elements, and s is the *private key*, known (in $\mathbb{Z}/p\mathbb{Z}$ form) only to the party able to decrypt.

ElGamal is a *public key* encryption scheme. The *public key* is any pair (g, h), both elements in Ed25519, such that $h = sg$. When encrypting a message m using (g, h), one generates r, a uniformly-chosen random $\mathbb{Z}/p\mathbb{Z}$ element, and then uses gr as the auxiliary key x and $hr + m$ as the encrypted message. This works because $hr = (sg)r = s(gr) = sx$.

ElGamal encryption relies critically on the idea that while module products are computationally feasible, computing the product between two Ed25519 elements directly is not. It is not difficult to see that for encryption and decryption using s one only needs to perform products in the $\mathbb{Z}/p\mathbb{Z}$-module, whereas for a party knowing only the public key (g, h) decryption requires products/divisions in the field we defined over Ed25519.

ElGamal is also a *semi-homomorphic* encryption scheme, because given $(x_1, sx_1 + m_1)$ and $(x_2, sx_2 + m_2)$, which encrypt m_1 and m_2, respectively, the pair $(x_1 + x_2, sx_1 + m_1 + sx_2 + m_2) = (x_1 + x_2, s(x_1 + x_2) + (m_1 + m_2))$, which is easy to calculate from the original pair, encrypts the sum $m_1 + m_2$.

Note that the auxiliary key can always be assumed to be uniformly distributed and independent of any other element, because one can always generate a new, uniformly distributed r in $\mathbb{Z}/p\mathbb{Z}$, and from it a new *nonce* (an encryption of zero), (gr, hr). Summing this encryption of zero to any ciphertext pair results in a new pair, encrypting the same value as the original pair, but with an auxiliary key that is uniformly and independently distributed. We refer to such an action as *refreshing*.

[1] In practice, we do not use this algorithm to compute ax, as it requires $O(a)$ group summations. Instead, we use doubling, which requires only $O(\log a)$ summations to reach the same result.

3 Key-switching algorithm

Key-switching is a process by which a ciphertext using some private key s_A of a message m is converted to a ciphertext of the same message, but using a different key s_B. Importantly, no party should gain during this process any knowledge of m, so simply decrypting and re-encrypting is not an option.

We show an algorithm that performs key switching in what is perhaps the most constraining scenario. In this scenario, there are two parties, A and B. Party A knows private key s_A, party B knows private key s_B, and both parties know both public keys. We begin with party B having a ciphertext m encrypted as $s_A x_A + m$ using auxiliary key x_A and private key s_A, and we want to conclude with party A having the same ciphertext encrypted as $s_B x_B + m$ using auxiliary key x_B and private key s_B, all this with neither A nor B having any access to the value of m.

Note 3.1. If we had wanted m, in the final configuration, to be encrypted using some third party's private key s_C, we could have done that using exactly the same algorithm. Nowhere is the private key s_B used, only the corresponding public key.

The algorithm that performs key-switching in this scenario is as follows.

1. B encrypts $s_A x_A + m$ with s_B, arriving at $(x_B, s_A x_A + s_B x_B + m)$.
2. B sends $(x_A, x_B, s_A x_A + s_B x_B + m)$ to A.
3. A decrypts with its private key, arriving at $s_B x_B + m$, as desired.

Note 3.2. A typical usage scenario for key switching is in order to provide some party, in a very controlled way, with access to specific data. In the example above, this could be used as a method to provide B with the value of m without A discovering this value. In this scenario, A would ultimately send $s_B x_B + m$ back to B for B to decrypt it.

When used in this way, there is no reason for B to send x_B to A. It is not cryptographically harmful that x_B is sent to A, but its value is never used.

If the intention is for A to use $s_B x_B + m$ in any other way, then x_B is needed for the encrypted message to remain decipherable.

Note 3.3. The process described above is not restricted to just two keys. What we see is that in ElGamal encryptions and decryptions by any set of keys commute, i.e. it does not matter in which order we perform them. In this example we encrypted by s_A (before the start of the algorithm), and then encrypted by s_B and decrypted by s_A; commutativity tells us that this has the same total effect as starting from the same starting point (a value encrypted by s_A), decrypting by s_A and then encrypting by s_B. Clearly, in this case the final result is the same value encrypted by s_B. The same property ensures that we can use any number of keys, utilising them to encrypt and decrypt in any conceivable order, and will always get to the same final result.

4 Use scenarios

The starting point of the key-switching algorithm, as described in Section 3, is directly applicable to the FinTracer scenario. Here, party A is an FIU and party B is an RE: FinTracer retrains information in the form of tag values encrypted with the FIU's key. In FinTracer, as in the construction of Section 3, the value stored by B is hidden from B by the encryption and hidden from A by physical separation.

It is, however, unlikely that we will ultimately want to use the key-switching algorithm for the scenario described in Note 3.2, which is portrayed there as a "typical usage scenario".

The reason for this is that it would create a vulnerability, in the event of an RE failing to follow the protocol. An RE would be able to send any chosen tag value to the FIU, instead of the tag value that the FIU intends for it to read. The RE can thus learn the value of a chosen tag. This is a breach of privacy that we would like to prevent.

In this section we describe a different type of usage for the key-switching algorithm that does not have such a weakness.

Consider, again, the standard FinTracer scenario in which one party, B (the RE), holds data (tag values) encrypted with s_A, the private key of party A (the FIU), and consider what functions B can compute without divulging the contents of the data to either A or B.

Because we can sum ciphertexts (due to the semi-homomorphic property) and can multiply ciphertexts by $\mathbb{Z}/p\mathbb{Z}$ elements (using repeated addition), B can compute any linear function of the encrypted values e_m it has, using as coefficients any values p_B that it can compute in plaintext.

We will show that using key switching, B can also multiply encrypted message values e_m by values p_A that only A can compute in plaintext (and, by extension, B can compute any linear function of B's encrypted values whose coefficients are dot products of values p_A that A can compute in plaintext and values p_B that B can compute in plaintext, i.e. functions that can be computed as a sum of elements of the form $p_A p_B e_m$).

Computing $p_A e_m$, as desired, is not trivial to do: without key switching, if A sends p_A to B, it will have by this disclosed the information of p_A to B. If it sends only an encrypted version of p_A, B will not be able to calculate the product $p_A e_m$. Finally, if B sends e_m to A, this will divulge m to A, because A holds the decryption key for e_m.

The protocol that overcomes all these difficulties is as follows. Recall that at the start of the protocol the value e_m, held by B, equals $s_A x_A + m$, where the value of the auxiliary key x_A is also known to B.

1. B encrypts e_m with s_B, leading to $e'_m = s_A x_A + s_B x_B + m$.
2. B sends to A the values x_A, x_B and e'_m.
3. A multiplies by p_A by computing $y_A = p_A x_A$, $y_B = p_A x_B$ and $e''_m = p_A e'_m = s_A y_A + s_B y_B + p_A m$.
4. A refreshes e''_m by adding to it both zeroes encrypted by s_A and zeroes encrypted by s_B. The final result is $(y_A + x'_A, y_B + x'_B, s_A(y_A + x'_A) + s_B(y_B + x'_B) + p_A m) = (x''_A, x''_B, s_A x''_A + s_B x''_B + p_A m)$, where x''_A and x''_B are uniformly distributed.
5. A sends this final result to B as (x''_A, x''_B, e'''_m).
6. B decrypts e'''_m with s_B, leading to $(x''_A, s_A x''_A + p_A m)$, which is simply a fresh encryption, with key s_A, of $p_A m$, as desired.

Note that this is a slight variation over the key-switching protocol described in Section 3, in that A does not decrypt the message using s_A. It would have been equally possible to decrypt using s_A (leaving the message encrypted only by s_B), perform the multiplication by p_A under this encryption, and then to re-encrypt by s_A. We chose to perform the multiplication under two layers of encryption for three reasons.

1. Because we can,
2. Because this way is faster, and
3. Because it is easier to prove that this protocol cannot leak to B the plaintext of any tag value B may possess if A never uses its private key throughout the algorithm.

References

[1] Brand, M. (2020), *The Complete, Authorised and Definitive FinTracer Compendium (Volume 1, Part 1, 1st Edition): A work in progress*, AUSTRAC internal. (Available as: `FT_combined.pdf`)

[2] Brand, M. (2020), Oblivious setting of source accounts in FinTracer. (Available as: `oblivious_a.pdf`)

[3] Brand, M. (2021), Oblivious querying of FinTracer tag values. (Available as: `oblivious_b.pdf`)