The original multi-hop proxy re-encryption algorithm

Jon Lamar and Bob Wall

March 7, 2018

1 Initial algorithm

The foundation of the IronCore Labs proxy-reencryption system is an algorithm that was first published by Wang and Cao in [WC09]. Here we present it in our notation.

1.1 Parameters

Let $params = (k, p, \mathbb{G}_1, \mathbb{G}_T, e, \mathbf{g}, \mathbf{g_1}, \mathbf{h_1}, \mathbf{h_2}, \mathbf{h_3}, H_1, \mathbf{H_2}, Sig)$ be public parameters, where:

- k is the number of bits required to store keys;
- p is a prime;
- $\mathbb{G}_1$ and $\mathbb{G}_T$ are abelian groups with $\mathbb{G}_1$ written additively and $\mathbb{G}_T$ written multiplicatively;[1]
- $e : \mathbb{G}_1 \times \mathbb{G}_1 \to \mathbb{G}_T$ is a bilinear pairing;
- $\mathbf{g}$ is an arbitrary fixed nonzero point in $\mathbb{G}_1$.
- $\mathbf{g_1}$, $\mathbf{h_1}$, $\mathbf{h_2}$, and $\mathbf{h_3}$ are random elements of $\mathbb{G}_1$ which do not lie in the cyclic subgroup generated by $\mathbf{g}$.
- $H_1 : \{0,1\}^* \to \mathbb{F}_p$ and $\mathbf{H_2} : \mathbb{G}_T \to \mathbb{G}_1$ are two one-way collision-resistant hash functions.
- $Sig = (\mathcal{G}, \mathcal{S}, \mathcal{V})$ is a strongly unforgeable signature scheme.

1.2 The scheme

The scheme consists of five algorithms: KeyGen, Enc, ReKeyGen, ReEnc, and Dec.

- $\mathsf{KeyGen}(params) \to (pk, sk)$: Generate a public/private key pair.
  1. Let the *secret key, sk* be chosen randomly from $\mathbb{Z}_p^*$.
  2. The *public key* is
  $$\mathbf{pk} \leftarrow sk \cdot \mathbf{g}$$
- $\mathsf{Enc}(params, \mathbf{pk}_j, m) \to C^{(1)}$: Encrypt a message $m \in \mathbb{G}_T$ to user j, given j's public key $\mathbf{pk}_j$.
  1. Let the *ephemeral secret key*, esk, be chosen randomly from $\mathbb{Z}_p^*$

[1] Throughout, we will use **bold** to denote elements of $\mathbb{G}_1$.

2. The *ephemeral public key* is

$$\mathbf{epk} \leftarrow esk \cdot \mathbf{g}$$

3. The *encrypted message* is

$$em \leftarrow m \cdot e(\mathbf{pk}_j, \mathbf{g_1})^{esk}$$

4. The *encryption authentication code* is

$$\mathbf{eac} \leftarrow esk \cdot (H_1(\mathbf{epk}) \cdot \mathbf{h_1} + H_1(\mathbf{epk} \| em) \cdot \mathbf{h_2} + \mathbf{h_3})$$

5. The ciphertext is

$$C^{(1)} \leftarrow (\mathbf{epk}, em, \mathbf{eac})$$

The superscript 1 indicates that $C^{(1)}$ is a first-level ciphertext.

- $\mathsf{ReKeyGen}(params, sk_i, \mathbf{pk}_j, P_i) \rightarrow rk_{i \rightarrow j}$: Generate a re-encryption key from user i (the delegator) to user j (the delegatee), given a proxy P_i associated to user i. The proxy will perform the actual re-encryption of ciphertexts encrypted to user i after that user has delegated decryption to user j.

1. Let the *re-encryption secret key*, rsk, be chosen randomly from $\mathbb{Z}_p^*$
2. Let the *temporary key* K be chosen randomly from $\mathbb{G}_T$
3. The *re-encryption public key* is

$$\mathbf{rpk} \leftarrow rsk \cdot \mathbf{g}$$

4. The *encrypted temporary key* is

$$rek \leftarrow K \cdot e(\mathbf{pk}_j, \mathbf{g_1})^{rsk}$$

5. The *re-encryption authentication code* is

$$\mathbf{rac} \leftarrow rsk \cdot (H_1(\mathbf{rpk}) \cdot \mathbf{h_1} + H_1(\mathbf{rpk} \| rek \| svk_{P_i}) \cdot \mathbf{h_2} + \mathbf{h_3})$$

6. The *re-encryption point* is

$$\mathbf{rep} \leftarrow \mathbf{H_2}(K) + (-sk_i) \cdot \mathbf{g_1}$$

7. The *re-encryption key* is

$$rk_{i \rightarrow j} \leftarrow (\mathbf{rpk}, rek, svk_{P_i}, \mathbf{rac}, \mathbf{rep})$$

8. The re-encryption key is sent to the proxy P_i via a secure channel

- $\mathsf{ReEnc}(params, ssk_{P_i}, rk_{i \rightarrow j}, C^{(l)}) \rightarrow C^{(l+1)}$: Re-encrypt a level-$l$ ciphertext encrypted to user i into a level-$(l+1)$ ciphertext encrypted to user j. Additional information is appended to the new ciphertext. Operation is performed by user i's proxy, P_i, which must have its secret signing key ssk_{P_i} available.

To re-encrypt a *first-level* ciphertext $C^{(1)}$:

1. If any parse or verify step fails, return $\perp$
2. Parse $C^{(1)}$ into $(\mathbf{epk}, em, \mathbf{eac})$
3. Parse $rk_{i \to j}$ into $(\mathbf{rpk}, rek, svk, \mathbf{rac}, \mathbf{rep})$
4. Validate the authentication code of the encrypted message, by verifying that
$$e(\mathbf{g}, \mathbf{eac}) = e(\mathbf{epk}, H_1(\mathbf{epk}) \cdot \mathbf{h_1} + H_1(\mathbf{epk} \| em) \cdot \mathbf{h_2} + \mathbf{h_3})$$
5. Validate the authentication code for the re-encryption key, by verifying that
$$e(\mathbf{g}, \mathbf{rac}) = e(\mathbf{rpk}, H_1(\mathbf{rpk}) \cdot \mathbf{h_1} + H_1(\mathbf{rpk} \| rek \| svk) \cdot \mathbf{h_2} + \mathbf{h_3})$$
6. The *re-encrypted message* is
$$em' \leftarrow em \cdot e(\mathbf{epk}, \mathbf{rep})$$
7. The *altered ciphertext* is
$$C' \leftarrow (\mathbf{epk}, em', \mathbf{eac})$$
8. The *proxy signature* is
$$S^{(2)} \leftarrow \mathcal{S}(ssk_{P_i}, (C', \mathbf{rpk}, svk_{P_i}, \mathbf{rac}))$$
9. The *re-encryption block* is
$$RB^{(2)} \leftarrow (\mathbf{rpk}, rek, svk_{P_i}, \mathbf{rac}, S^{(2)})$$
10. The final level-2 ciphertext is
$$C^{(2)} \leftarrow (C', RB^{(2)})$$

Note that the signature does not include rek, and that the ciphertext includes all of the pieces of the re-encryption key except **rep**.

To re-encrypt a *level-l* ciphertext $C^{(l)}$, where $l > 1$:

1. If any parse or verify step fails, return $\perp$
2. Parse $C^{(l)}$ into $(C', RB^{(2)}, \ldots, RB^{(l)})$
3. For each integer $k \in [2, l]$, parse $RB^{(k)}$ into $(\mathbf{rpk}^{(k)}, rek^{(k)}, svk^{(k)}, \mathbf{rac}^{(k)}, S^{(k)})$
4. Parse $rk_{i \to j}$ into $(\mathbf{rpk}^{(l+1)}, rek^{(l+1)}, svk^{(l+1)}, \mathbf{rac}^{(l+1)}, \mathbf{rep}^{(l+1)})$
5. Validate the authentication code of the last re-encryption block by verifying that
$$e(\mathbf{g}, \mathbf{rac}^{(l)}) = e(\mathbf{rpk}^{(l)}, H_1(\mathbf{rpk}^{(l)}) \cdot \mathbf{h_1} + H_1(\mathbf{rpk}^{(l)} \| rek \| svk_{P_{i_{l-1}}}) \cdot \mathbf{h_2} + \mathbf{h_3})$$
6. Validate the authentication code of the new re-encryption key by verifying that
$$e(\mathbf{g}, \mathbf{rac}^{(l+1)}) = e(\mathbf{rpk}^{(l+1)}, H_1(\mathbf{rpk}^{(l+1)}) \cdot \mathbf{h_1} + H_1(\mathbf{rpk}^{(l+1)} \| rek \| svk^{(l+1)}) \cdot \mathbf{h_2} + \mathbf{h_3})$$

7. Validate the signature of the first re-encryption block

$$\mathcal{V}(svk^{(2)}, S^{(2)}, (C', \mathbf{rpk}^{(2)}, svk^{(2)}, \mathbf{rac}^{(2)})) = 1$$

8. Validate the signature of each subsequent re-encryption block; for each integer $k \in [3, l]$, verify

$$\mathcal{V}(svk^{(k)}, S^{(k)}, (C', RB'^{(2)}, \ldots, RB'^{(k-1)}, \mathbf{rpk}^{(k)}, svk^{(k)}, \mathbf{rac}^{(k)})) = 1$$

9. The *re-encrypted temporary key* is

$$rek'^{(l)} \leftarrow rek^{(l)} \cdot e(\mathbf{rpk}^{(l)}, \mathbf{rep}^{(l+1)})$$

10. The *altered re-encryption block* is

$$RB'^{(l)} \leftarrow (\mathbf{rpk}^{(l)}, rek'^{(l)}, svk^{(l)}, \mathbf{rac}^{(l)}, S^{(l)})$$

11. Create a new proxy signature for the present level of re-encryption by computing

$$S^{(l+1)} \leftarrow \mathcal{S}(ssk_{P_i}, (C', RB'^{(2)}, \ldots, RB'^{(l)}, \mathbf{rpk}^{(l+1)}, svk_{P_i}, \mathbf{rac}^{(l+1)}))$$

12. Create a level-$(l+1)$ re-encryption block,

$$RB^{(l+1)} \leftarrow (\mathbf{rpk}^{(l+1)}, rek^{(l+1)}, svk_{P_i}, \mathbf{rac}^{(l+1)}, S^{(l+1)})$$

13. The final level-$(l+1)$ ciphertext is

$$C^{(l+1)} \leftarrow (C', RB'^{(2)}, \ldots, RB'^{(l)}, RB^{(l+1)})$$

In summary, on each round of re-encryption (after the first), the last rek value from the previous round of re-encryption is re-encrypted, then the first four pieces of the new re-encryption key as well as the new signature are appended. Note that the encrypted message em from the original ciphertext $C^{(1)}$ is only re-encrypted once, at level 2. After that, it is not changed again.

- $\mathsf{Dec}(params, sk_i, C^{(l)}) \rightarrow m$: Decrypt a ciphertext encrypted to user i. As above, we return $\perp$ if any parse or verify step fails.

  To decrypt a first-level ciphertext $C^{(1)}$:

  1. Parse $C^{(1)}$ into $(\mathbf{epk}, em, \mathbf{eac})$
  2. Validate the authentication code of the ciphertext by verifying that

$$e(\mathbf{g}, \mathbf{eac}) = e(\mathbf{epk}, H_1(\mathbf{epk}) \cdot \mathbf{h_1} + H_1(\mathbf{epk} \| em) \cdot \mathbf{h_2} + \mathbf{h_3})$$

  3. Finally, recover the plaintext m via the operation

$$m \leftarrow em \cdot e(\mathbf{epk}, (-sk_i) \cdot \mathbf{g_1})$$

  To decrypt a level-l ciphertext $C^{(l)}$:

1. Parse $C^{(l)}$ into $(C', RB'^{(2)}, \ldots, RB'^{(l-1)}, RB^{(l)})$
2. Parse C' into $(\mathbf{epk}, em', \mathbf{eac})$
3. Parse $RB^{(l)}$ into $(\mathbf{rpk}^{(l)}, rek^{(l)}, svk_{P_{i_l}}, \mathbf{rac}^{(l)}, S^{(l)})$
4. For each integer k in $[2, l-1]$, parse $RB'^{(k)}$ into $(\mathbf{rpk}^{(k)}, rek'^{(k)}, svk^{(k)}, \mathbf{rac}^{(k)}, S^{(k)})$
5. Validate the last re-encryption block (to confirm that the last encrypted temporary key is not modified) by verifying
$$e(\mathbf{g}, \mathbf{rac}^{(l)}) = e(\mathbf{rpk}^{(l)}, H_1(\mathbf{rpk}^{(l)}) \cdot \mathbf{h_1} + H_1(\mathbf{rpk}^{(l)} \| rek^{(l)} \| svk^{(l)}) \cdot \mathbf{h_2} + \mathbf{h_3})$$
6. For each integer $k \in [2, l]$, validate the signature of each re-encryption block by verifying that
$$\mathcal{V}(svk^{(k)}, S^{(k)}, (C', RB'^{(2)}, \ldots, RB'^{(k-1)}, \mathbf{rpk}^{(k)}, svk^{(k)}, \mathbf{rac}^{(k)})) = 1$$
7. Recover the $(l-1)$-st temporary key K_{l-1} via the operation
$$K_{l-1} \leftarrow rek^{(l)} \cdot e(\mathbf{rpk}^{(l)}, (-sk_i) \cdot \mathbf{g_1})$$
8. For each integer k from $l-2$ down to 1, recover the k-th temporary key K_k via the operation
$$K_k \leftarrow rek^{(k+1)} \cdot e(\mathbf{rpk}^{(k+1)}, -\mathbf{H_2}(K_{k+1}))$$
9. Finally, recover the plaintext m via the operation
$$m \leftarrow em' \cdot e(\mathbf{epk}, -\mathbf{H_2}(K_1))$$

1.3 Example

In the following example, assume users Alice, Bob, Carol, and Dave. Values are subscripted using their initials; for example, pk_A is Alice's public key, sk_D is Dave's secret key, etc. Assume that the same proxy, P_Z, is shared by all users.

For this example, suppose a user Zed encrypts a message m_A to Alice, creating the ciphertext C_A, which is composed of $(\mathbf{epk}_A, em_A, \mathbf{eac}_A)$. When Alice receives C_A, she performs these decryption steps:

1. Parse C_A into $(\mathbf{epk}_A, em_A, \mathbf{eac}_A)$
2. Compute
$$e(\mathbf{g}, \mathbf{eac}_A) = e(\mathbf{g}, H_1(\mathbf{epk}_A) \cdot \mathbf{h_1} + H_1(\mathbf{epk}_A \| em_A) \cdot \mathbf{h_2} + \mathbf{h_3})^{esk_A}$$
3. Compute
$$e(\mathbf{epk}_A, H_1(\mathbf{epk}_A) \cdot \mathbf{h_1} + H_1(\mathbf{epk}_A \| em_A) \cdot \mathbf{h_2} + \mathbf{h_3}) =$$
$$e(esk_A \cdot \mathbf{g}, H_1(\mathbf{epk}_A) \cdot \mathbf{h_1} + H_1(\mathbf{epk}_A \| em_A) \cdot \mathbf{h_2} + \mathbf{h_3})$$

Since the pairing e is bilinear, the exponent esk_A can be moved inside the pairing and multiplied by either term, so the two values should be equivalent if C_A has not been corrupted.

Now, the encrypted message was generated by

$$\begin{aligned} m \cdot e(\mathbf{pk}_A, \mathbf{g_1})^{esk_A} = \\ m \cdot e(sk_A \cdot \mathbf{g}, \mathbf{g_1})^{esk_A} = \\ m \cdot e(esk_A \cdot \mathbf{g}, sk_A \cdot \mathbf{g_1}) \end{aligned}$$

While decrypting, Alice computes

$$\begin{aligned} em/e(\mathbf{epk}_A, sk_A \cdot \mathbf{g_1}) = \\ em/e(esk_A \cdot \mathbf{g}, sk_A \cdot \mathbf{g_1}) = \\ m \cdot e(esk_A \cdot \mathbf{g}, sk_A \cdot \mathbf{g_1})/e(esk_A \cdot \mathbf{g}, sk_A \cdot \mathbf{g_1}) = \\ m \end{aligned}$$

Now, suppose that instead of decrypting C_A, Alice delegates her decryption to Bob. She creates a re-encryption key $rk_{A \to B}$, which is composed of

$$(\mathbf{rpk}_{AB}, rek_{AB}, svk_{P_Z}, \mathbf{rac}_{AB}, \mathbf{rep}_{AB})$$

and sends this to P_Z, along with the ciphertext C_A. P_Z re-encrypts the message, generating the new ciphertext C'_B, which is composed of

$$(\mathbf{epk}_A, em'_B, \mathbf{eac}_A, RB_{AB})$$

where em'_B is

$$em_A \cdot e(\mathbf{epk}_A, \mathbf{rep}_{AB})$$

and RB_{AB} is

$$(\mathbf{rpk}_{AB}, rek_{AB}, svk_{P_Z}, \mathbf{rac}_{AB}, S_{AB})$$

When Bob receives C'_B, he performs these steps:

1. Parse C'_B into (C', RB_{AB})
2. Compute

$$e(\mathbf{g}, \mathbf{rac}_{AB}) = e(\mathbf{g}, rsk_{AB} \cdot (H_1(\mathbf{rpk}_{AB}) \cdot \mathbf{h_1} + H_1(\mathbf{rpk}_{AB} \| rek_{AB} \| svk_{P_Z}) \cdot \mathbf{h_2} + \mathbf{h_3})$$

3. Compute

$$\begin{aligned} e(\mathbf{rpk}_{AB}, H_1(\mathbf{rpk}_{AB}) \cdot \mathbf{h_1} + H1(\mathbf{rpk}_{AB} \| rek_{AB} \| svk_{P_Z}) \cdot \mathbf{h_2} + \mathbf{h_3}) = \\ e(rsk_{AB} \cdot \mathbf{g}, H_1(\mathbf{rpk}_{AB}) \cdot \mathbf{h_1} + H1(\mathbf{rpk}_{AB} \| rek_{AB} \| svk_{P_Z}) \cdot \mathbf{h_2} + \mathbf{h_3}) \end{aligned}$$

Again, by the properties of bilinearity, these two should be equivalent in the absence of message corruption. Continuing,

$$\begin{aligned} \text{Compute } \mathcal{V}(svk_{P_Z}, S_{AB}, (C', \mathbf{rpk}_{AB}, svk_{P_Z}, \mathbf{rac}_{AB})) = \\ \mathcal{V}(svk_{P_Z}, \mathcal{S}(ssk_{P_Z}, (C', \mathbf{rpk}_{AB}, svk_{P_Z}, \mathbf{rac}_{AB})), (C', \mathbf{rpk}_{AB}, svk_{P_Z}, \mathbf{rac}_{AB})) \end{aligned}$$

The verification algorithm should return 1 in the absence of data corruption.

Continuing:

1. Compute

$$K'_{AB} = rek_{AB}/e(\mathbf{rpk}_{AB}, sk_B \cdot \mathbf{g_1}) =$$
$$K_{AB} \cdot e(\mathbf{pk}_B, \mathbf{g_1})^{rsk_{AB}}/e(rsk_{AB} \cdot \mathbf{g}, sk_B \cdot \mathbf{g_1}) =$$
$$K_{AB} \cdot e(sk_B \cdot \mathbf{g}, rsk_{AB} \cdot \mathbf{g_1})/e(rsk_{AB} \cdot \mathbf{g}, sk_B \cdot \mathbf{g_1}) =$$
$$K_{AB} \cdot e(rsk_{AB} \cdot \mathbf{g}, sk_B \cdot \mathbf{g_1})/e(rsk_{AB} \cdot \mathbf{g}, sk_B \cdot \mathbf{g_1}) =$$
$$K_{AB}$$

2. Compute

$$em'/e(\mathbf{epk}_A, \mathbf{H_2}(K'_{AB})) =$$
$$m \cdot e(\mathbf{pk}_A, \mathbf{g_1})^{esk_A} \cdot e(\mathbf{epk}_A, \mathbf{rep}_{AB})/e(esk_A \cdot \mathbf{g}, \mathbf{H_2}(K_{AB})) =$$
$$m \cdot e(sk_A \cdot \mathbf{g}, esk_A \cdot \mathbf{g_1}) \cdot e(esk_A \cdot \mathbf{g}, \mathbf{H_2}(K_{AB}) + (-sk_A) \cdot \mathbf{g_1})/e(esk_A \cdot \mathbf{g}, \mathbf{H_2}(K_{AB})) =$$
$$m \cdot e(esk_A \cdot \mathbf{g}, sk_A \cdot \mathbf{g_1}) \cdot e(esk_A \cdot \mathbf{g}, \mathbf{H_2}(K_{AB}) + (-sk_A) \cdot \mathbf{g_1})/e(esk_A \cdot \mathbf{g}, \mathbf{H_2}(K_{AB})) =$$
$$m \cdot e(esk_A \cdot \mathbf{g}, sk_A \cdot \mathbf{g_1} + \mathbf{H_2}(K_{AB}) + (-sk_A) \cdot \mathbf{g_1})/e(esk_A \cdot \mathbf{g}, \mathbf{H_2}(K_{AB})) =$$
$$m \cdot e(esk_A \cdot \mathbf{g}, \mathbf{H_2}(K_{AB}))/e(esk_A \cdot \mathbf{g}, \mathbf{H_2}(K_{AB})) =$$
$$m$$

Now, suppose Bob delegates his decryption to Carol. He computes the re-encryption key $rk_{B\to C}$:

1. Let rsk_{BC} be chosen randomly from $\mathbb{Z}_p^*$
2. Let K_{BC} be chosen randomly from $\mathbb{G}_T$
3. $\mathbf{rpk}_{BC} \leftarrow rsk_{BC} \cdot \mathbf{g}$
4. $rek_{BC} \leftarrow K_{BC} \cdot e(\mathbf{pk}_C, \mathbf{g_1})^{rsk_{BC}}$
5. $\mathbf{rac}_{BC} \leftarrow (H_1(\mathbf{rpk}_{BC}) \cdot \mathbf{h_1} + H_1(\mathbf{rpk}_{BC} \| rek_{BC} \| svk_{P_Z}) \cdot \mathbf{h_2} + \mathbf{h_3})^{rsk_{BC}}$
6. $\mathbf{rep}_{BC} \leftarrow \mathbf{H_2}(K_{BC}) + sk_B \cdot \mathbf{g_1}$
7. $rk_{B\to C} \leftarrow (\mathbf{rpk}_{BC}, rek_{BC}, svk_{P_Z}, \mathbf{rac}_{BC}, \mathbf{rep}_{BC})$

and shares that and C'_B with P_Z. The proxy re-encrypts it, generating the new ciphertext C'_C, using the following steps:

1. Parse C'_B into $(C', \mathbf{rpk}_{AB}, rek_{AB}, svk_{P_z}, \mathbf{rac}_{AB}, S_{AB})$
2. $rek'_{AB} \leftarrow rek_{AB} \cdot e(\mathbf{rpk}_{AB}, \mathbf{rep}_{BC})$
3. $RB'_{AB} \leftarrow (\mathbf{rpk}_{AB}, rek'_{AB}, svk_{P_z}, \mathbf{rac}_{AB}, S_{AB})$
4. $S_{BC} \leftarrow \mathcal{S}(ssk_{P_z}, (C'_B, RB'_{AB}, \mathbf{rpk}_{BC}, svk_{P_z}, \mathbf{rac}_{BC}))$
5. $RB_{BC} \leftarrow (\mathbf{rpk}_{BC}, rek_{BC}, svk_{P_z}, \mathbf{rac}_{BC}, S_{BC})$
6. $C'_C \leftarrow (C', RB'_A, RB_{BC})$

When Carol decrypts C'_C, she follows these steps:

1. Parse C'_C into (C', RB'_{AB}, RB_{BC})

2. Compute

$$e(\mathbf{g}, \mathbf{rac}_{BC}) =$$
$$e(\mathbf{g}, rsk_{BC} \cdot (H_1(\mathbf{rpk}_{BC}) \cdot \mathbf{h_1} + H_1(\mathbf{rpk}_{BC} \| rek_{BC} \| svk_{P_Z}) \cdot \mathbf{h_2} + \mathbf{h_3}))$$

3. Compute

$$e(\mathbf{rpk}_{BC}, H_1(\mathbf{rpk}_{BC}) \cdot \mathbf{h_1} + H1(\mathbf{rpk}_{BC} \| rek_{BC} \| svk_{P_Z}) \cdot \mathbf{h_2} + \mathbf{h_3}) =$$
$$e(rsk_{BC} \cdot \mathbf{g}, H_1(\mathbf{rpk}_{BC}) \cdot \mathbf{h_1} + H1(\mathbf{rpk}_{BC} \| rek_{BC} \| svk_{P_Z}) \cdot \mathbf{h_2} + \mathbf{h_3})$$

Again, by the properties of bilinearity, these two should be equivalent in the absence of message corruption. Continuing,

1. Compute

$$\mathcal{V}(svk_{P_Z}, S_{AB}, (C', \mathbf{rpk}_{AB}, svk_{P_Z}, \mathbf{rac}_{AB})) =$$
$$\mathcal{V}(svk_{P_Z}, \mathcal{S}(ssk_{P_Z}, (C', \mathbf{rpk}_{AB}, svk_{P_Z}, \mathbf{rac}_{AB})), (C', \mathbf{rpk}_{AB}, svk_{P_Z}, \mathbf{rac}_{AB}))$$

2. Repeat for RB_{BC}

The verification algorithm should return 1 for both cases in the absence of data corruption.

Continuing:

1. Compute

$$K'_{BC} = rek_{BC} / e(\mathbf{rpk}_{BC}, {\mathbf{g_1}}^{sk_C}) =$$
$$K_{BC} \cdot e(\mathbf{pk}_C, \mathbf{g_1})^{rsk_{BC}} / e(\mathbf{g}^{rsk_{BC}}, {\mathbf{g_1}}^{sk_C}) =$$
$$K_{BC} \cdot e(\mathbf{g}, {\mathbf{g_1}}^{sk_C})^{rsk_{BC}} / e(\mathbf{g}^{rsk_{BC}}, {\mathbf{g_1}}^{sk_C}) =$$
$$K_{BC} \cdot e(\mathbf{g}^{rsk_{BC}}, {\mathbf{g_1}}^{sk_C}) / e(\mathbf{g}^{rsk_{BC}}, {\mathbf{g_1}}^{sk_C}) =$$
$$K_{BC}$$

2. Compute

$$K'_{AB} = rek'_{AB} / e(\mathbf{rpk}_{AB}, \mathbf{H_2}(K_{BC})) =$$
$$rek_{AB} \cdot e(\mathbf{rpk}_{AB}, \mathbf{rep}_{BC}) / e(\mathbf{rpk}_{AB}, \mathbf{H_2}(K_{BC})) =$$
$$K_{AB} \cdot e(\mathbf{pk}_B, \mathbf{g_1})^{rsk_{AB}} \cdot e(\mathbf{rpk}_{AB}, \mathbf{H_2}(K_{BC}) + -sk_B \cdot \mathbf{g_1}) / e(rsk_{AB} \cdot \mathbf{g}, \mathbf{H_2}(K_{BC})) =$$
$$K_{AB} \cdot e(\mathbf{g}, sk_B \cdot \mathbf{g_1})^{rsk_{AB}} \cdot e(rsk_{AB} \cdot \mathbf{g}, \mathbf{H_2}(K_{BC}) + -sk_B \cdot \mathbf{g_1}) / e(rsk_{AB} \cdot \mathbf{g}, \mathbf{H_2}(K_{BC})) =$$
$$K_{AB} \cdot e(rsk_{AB} \cdot \mathbf{g}, sk_B \cdot \mathbf{g_1}) \cdot e(rsk_{AB} \cdot \mathbf{g}, \mathbf{H_2}(K_{BC}) + -sk_B \cdot \mathbf{g_1}) / e(rsk_{AB} \cdot \mathbf{g}, \mathbf{H_2}(K_{BC})) =$$
$$K_{AB} \cdot e(rsk_{AB} \cdot \mathbf{g}, sk_B \cdot \mathbf{g_1} + \mathbf{H_2}(K_{BC}) + -sk_B \cdot \mathbf{g_1}) / e(rsk_{AB} \cdot \mathbf{g}, \mathbf{H_2}(K_{BC})) =$$
$$K_{AB} \cdot e(rsk_{AB} \cdot \mathbf{g}, \mathbf{H_2}(K_{BC})) / e(rsk_{AB} \cdot \mathbf{g}, \mathbf{H_2}(K_{BC})) =$$
$$K_{AB}$$

Then actually decrypt the message:

1. Compute

$$em'/e(\mathbf{epk}_A, \mathbf{H_2}(K'_{AB})) =$$
$$m \cdot e(\mathbf{pk}_A, \mathbf{g_1})^{esk_A} \cdot e(\mathbf{epk}_A, \mathbf{rep}_{AB})/e(esk_A \cdot \mathbf{g}, \mathbf{H_2}(K_{AB})) =$$
$$m \cdot e(sk_A \cdot \mathbf{g}, \mathbf{g_1})^{esk_A} \cdot e(esk_A \cdot \mathbf{g}, \mathbf{H_2}(K_{AB}) + -sk_A \cdot \mathbf{g_1})/e(esk_A \cdot \mathbf{g}, \mathbf{H_2}(K_{AB})) =$$
$$m \cdot e(esk_A \cdot \mathbf{g}, sk_A \cdot \mathbf{g_1}) \cdot e(esk_A \cdot \mathbf{g}, \mathbf{H_2}(K_{AB}) + -sk_A \cdot \mathbf{g_1})/e(esk_A \cdot \mathbf{g}, \mathbf{H_2}(K_{AB})) =$$
$$m \cdot e(esk_A \cdot \mathbf{g}, sk_A \cdot \mathbf{g_1} + \mathbf{H_2}(K_{AB}) + -sk_A \cdot \mathbf{g_1})/e(esk_A \cdot \mathbf{g}, \mathbf{H_2}(K_{AB})) =$$
$$m \cdot e(esk_A \cdot \mathbf{g}, \mathbf{H_2}(K_{AB}))/e(esk_A \cdot \mathbf{g}, \mathbf{H_2}(K_{AB})) =$$
$$m$$

2 Simplifying and enhancing authentication

The algorithms as proposed were designed to support multiple proxies, and to allow for the reencryptions of the message to be done in separate steps, with intermediate results potentially stored for periods of time. In the IronCore environment, there is only a single proxy, and the reencryption hops are always all applied at the same time. Because of this, we made a simplification of the algorithm to replace the authentication codes, which required a relatively high-overhead pairing computation, with an authentication hash combined with a single signature, computed over the entire encrypted or re-encrypted message. Both approach support the same authenticated encryption capability, but our solution adds significantly less time to the computation of the encrypted message and the reencrypted message, and adds less to the size of the encrypted message.

2.1 Parameters

Let $params = (k, p, \mathbb{G}_1, \mathbb{G}_T, e, \mathbf{g}, \mathbf{g_1}, \mathsf{SHA256}, \mathbf{H_2}, Sig)$ be public parameters, where:

- k is the number of bits required to store keys;
- p is a prime;
- $\mathbb{G}_1$ and $\mathbb{G}_T$ are abelian groups with $\mathbb{G}_1$ written additively and $\mathbb{G}_T$ written multiplicatively;[2]
- $e : \mathbb{G}_1 \times \mathbb{G}_1 \to \mathbb{G}_T$ is a bilinear pairing;
- $\mathbf{g}$ is an arbitrary fixed nonzero point in $\mathbb{G}_1$.
- $\mathbf{g_1}$ is a random element of $\mathbb{G}_1$ which does not lie in the cyclic subgroup generated by $\mathbf{g}$.
- $\mathsf{SHA256} : \{0,1\}^* \to$ 256-bit hash and $\mathbf{H_2} : \mathbb{G}_T \to \mathbb{G}_1$ are two one-way collision-resistant hash functions.
- $Sig = (\mathcal{G}, \mathcal{S}, \mathcal{V})$ is the Ed25519 strongly unforgeable signature scheme, with a key generation algorithm, a signing algorithm, and a verification algorithm.

[2]Throughout, we will use **bold** to denote elements of $\mathbb{G}_1$.

2.2 The scheme

The scheme consists of five algorithms: KeyGen, Enc, ReKeyGen, ReEnc, and Dec.

- $\mathsf{KeyGen}(params) \rightarrow (\mathbf{pk}, sk, psk, ssk)$: Generate a public/private key pair and a public/private signing key pair.
  1. Let the *secret key, sk* be chosen randomly from $\mathbb{Z}_p^*$.
  2. The *public key* is
  $$\mathbf{pk} \leftarrow sk \cdot \mathbf{g}$$
  3. The *signing key pair* is
  $$(psk, ssk) \leftarrow \mathcal{G}$$

- $\mathsf{Enc}(params, \mathbf{pk}_j, psk_i, ssk_i, m) \rightarrow C$: Encrypt a message $m \in \mathbb{G}_T$ to user j, given j's public key $\mathbf{pk}_j$ and i's signing key pair (psk_i, ssk_i).
  1. Let the *ephemeral secret key*, esk, be chosen randomly from $\mathbb{Z}_p^*$
  2. The *ephemeral public key* is
  $$\mathbf{epk} \leftarrow esk \cdot \mathbf{g}$$
  3. The *encrypted message* is
  $$em \leftarrow m \cdot e(\mathbf{pk}_j, \mathbf{g_1})^{esk}$$
  4. The *authentication hash* is
  $$ah \leftarrow \mathsf{SHA256}(epk||m)$$
  5. The *signature* is
  $$sig \leftarrow \mathcal{S}(\mathbf{epk}||em||ah||psk_i, ssk_i)$$
  6. The ciphertext is
  $$C \leftarrow (\mathbf{epk}, em, ah, psk_i, sig)$$

- $\mathsf{ReKeyGen}(params, sk_i, \mathbf{pk}_j, psk_i, ssk_i) \rightarrow rk_{i \rightarrow j}$: Generate a re-encryption key from user i (the delegator) to user j (the delegatee).
  1. Let the *re-encryption secret key*, rsk, be chosen randomly from $\mathbb{Z}_p^*$
  2. Let the *temporary key* K be chosen randomly from $\mathbb{G}_T$
  3. The *re-encryption public key* is
  $$\mathbf{rpk} \leftarrow rsk \cdot \mathbf{g}$$
  4. The *encrypted temporary key* is
  $$rek \leftarrow K \cdot e(\mathbf{pk}_j, \mathbf{g_1})^{rsk}$$
  5. The *re-encryption point* is
  $$\mathbf{rep} \leftarrow \mathbf{H_2}(K) + (-sk_i) \cdot \mathbf{g_1}$$

6. The *signature* is
$$sig \leftarrow \mathcal{S}(\mathbf{rpk}||rek||\mathbf{rep}||psk_i, ssk_i)$$

7. The *re-encryption key* is
$$rk_{i\rightarrow j} \leftarrow (\mathbf{rpk}, rek, \mathbf{rep}, psk_i, sig)$$

8. The re-encryption key is sent to the proxy via a secure channel

- $\mathsf{ReEnc}(params, rk_{i\rightarrow z}, C) \rightarrow RC$ – or – $\mathsf{ReEnc}(params, rk_{i\rightarrow a}, [...rk_{y\rightarrow z}], C) \rightarrow RC$: Re-encrypt a ciphertext C encrypted to user i into a ciphertext encrypted to user z. There must be at least one re-encryption key, but there can be a chain of multiple keys. The *from* user of the first re-encryption key must be i, and the *to* user of the last re-encryption key must be z. The chain must match the *to* user of each re-encryption key with the *from* user of the succeeding re-encryption key to form the chain. An additional *re-encryption block* is appended to the new ciphertext for each re-encryption. This operation is performed by the proxy, which must have its own signing key pair $(psk_{proxy}, ssk_{proxy})$ available.

  For the first re-encryption key in the chain, re-encrypt C as follows:

  1. If any parse or verify step fails, return $\perp$
  2. Parse C into $(\mathbf{epk}, em, ah, psk_m, sig_m)$
  3. Parse $rk_{i\rightarrow z}$ into $(\mathbf{rpk}, rek, \mathbf{rep}, psk_i, sig_{rk})$
  4. Validate the signature of the encrypted message, by verifying that
  $$\mathcal{V}(\mathbf{epk}||em||ah||psk_m, psk_m, sig_m) = true$$
  5. Validate the signature for the re-encryption key, by verifying that
  $$\mathcal{V}(\mathbf{rpk}||rek||\mathbf{rep}||psk_i, psk_i, sig_{rk}) = true$$
  6. The *re-encrypted message* is
  $$em' \leftarrow em \cdot e(\mathbf{epk}, \mathbf{rep})$$
  7. The *altered ciphertext* is
  $$C' \leftarrow (\mathbf{epk}, em', ah)$$
  8. The *re-encryption block* is
  $$RB \leftarrow (\mathbf{rpk}, rek)$$
  9. The re-encrypted ciphertext RC', which is not complete, is
  $$RC' \leftarrow (C', RB)$$

  For each subsequent re-encryption key in the chain, $rk_{a\rightarrow b}$, perform the following:

  1. Parse the last *re-encryption block* of RC' into $(\mathbf{rpk}_{last}, rek_{last})$

2. Parse $rk_{a\to b}$ into $(\mathbf{rpk}, rek, \mathbf{rep}, psk_a, sig_{rk})$
3. Validate the signature for the re-encryption key, by verifying that

$$\mathcal{V}(\mathbf{rpk}||rek||\mathbf{rep}||psk_a, psk_a, sig_{rk}) = true$$

4. Replace rek_{last} in the last re-encryption block with

$$rek'_{last} \leftarrow rek_{last} \cdot e(\mathbf{rpk}_{last}, \mathbf{rep})$$

5. Append a new re-encryption block RB_{new} to RC'

$$RB_{new} \leftarrow (\mathbf{rpk}, rek)$$

Finally, append the signature:

$$sig \leftarrow \mathcal{S}(RC'||psk_{proxy}, ssk_{proxy})$$

$$RC \leftarrow (RC', psk_{proxy}, sig)$$

In summary, on each round of re-encryption (after the first), the rek value from the previous round is re-encrypted, then the first two pieces of the new re-encryption key are concatenated as a re-encryption block*.t. Note that the encrypted message em from the original ciphertext C is only re-encrypted once, at level 2. After that, it is not changed again.

- $\mathsf{Dec}(params, sk_i, CT) \to m$: Decrypt a ciphertext encrypted to user i. As above, we return $\perp$ if any parse or verify step fails.

  For any ciphertext, the first step is to validate the signature:

  1. Parse CT into (C, psk_m, sig_m)
  2. Validate signature, by verifying that

$$\mathcal{V}(C||psk_m, psk_m, sig_m) = true$$

  To decrypt a first-level ciphertext $C^{(1)}$:

  1. Parse C into $(\mathbf{epk}, em, ah)$
  2. Recover the plaintext m via the operation

$$m \leftarrow em \cdot e(\mathbf{epk}, (-sk_i) \cdot \mathbf{g_1})$$

  To decrypt a level-l ciphertext $C^{(l)}$:

  1. Parse $C^{(l)}$ into $(C', RB'^{(2)}, \ldots, RB'^{(l-1)}, RB^{(l)})$
  2. Parse C' into $(\mathbf{epk}, em', ah)$
  3. Parse $RB^{(l)}$ into $(\mathbf{rpk}^{(l)}, rek^{(l)})$
  4. For each integer k in $[2, l-1]$, parse $RB'^{(k)}$ into $(\mathbf{rpk}^{(k)}, rek'^{(k)})$

5. Recover the $(l-1)$-st temporary key K_{l-1} via the operation
$$K_{l-1} \leftarrow rek^{(l)} \cdot e(\mathbf{rpk}^{(l)}, (-sk_i) \cdot \mathbf{g_1})$$

6. For each integer k from $l-2$ down to 1, recover the k-th temporary key K_k via the operation
$$K_k \leftarrow rek^{(k+1)} \cdot e(\mathbf{rpk}^{(k+1)}, -\mathbf{H_2}(K_{k+1}))$$

7. Finally, recover the plaintext m via the operation
$$m \leftarrow em' \cdot e(\mathbf{epk}, -\mathbf{H_2}(K_1))$$

After m has been recovered, validate the authentication hash of the ciphertext by verifying that $\mathsf{SHA256}(epk||m) = ah$

3 Security problems

In [ZW13], the authors showed that the original scheme was not in fact CCA-secure. Zhang and Wang presented a formal security model for CCA-security, then provided an attack where the proxy P acts as the adversary.

They showed that the proxy can take a first-level ciphertext, multiply the original **epk** by a new factor $t \cdot \mathbf{g}$, where t is randomly chosen from $\mathbb{Z}_p^*$, yielding a new $\mathbf{epk}' = (esk + t) \cdot \mathbf{g}$. The proxy then computes a re-encryption of the message to user j, using the modified $\mathbf{epk}'$ to compute em'. This re-encryption is valid and satisfies the authentication checks.

By the rules of the game-based definition of CCA-security, the adversary can ask the decryption oracle to decrypt this ciphertext, since it is not the original encrypted message or a re-encryption of that message. Since the ciphertext is still valid, it can be decrypted by the oracle acting as user j, so the adversary knows exactly what plaintext was used to generate the ciphertext. The algorithm of [WC09] is thus not CCA-secure. It is not clear whether there are any usable exploits that can be produced by this weakness, or if it just taking advantage of the way the CCA-security game is structured.

A subsequent paper, [CL14], presents another weakness of the scheme by Wang and Cao. They describe a "proxy bypass attack." The fundamental problem is shown by our previous example. When Carol decrypts a ciphertext that was re-encrypted from Alice to Bob and then from Bob to Carol, she learns not only the value of the temporary key $K_{B\to C}$ that is part of the re-encryption key Bob generated to delegate decryption to Carol, but also the value of the temporary key $K_{A\to B}$, which Alice generated to delegate decryption to Bob. Now, if Carol is able to intercept any subsequent ciphertext that was encrypted to Alice and then re-encrypted to Bob, she simply computes
$$m_{new} = em'_{new} \cdot e(\mathbf{epk}_{new}, -\mathbf{H_2}(K_{A\to B})).$$

4 Fixing security flaws by randomizing re-encryption

In [CL14], the authors provide a workaround to the security flaw. In this workaround, the algorithms KeyGen, ReKeyGen, and Enc are unchanged. The ReEnc and Dec algorithms are changed as follows: (additions and modifications are highlighted in red.)

In ReEnc:

For the first re-encryption key in the chain, re-encrypt C as follows:

1. If any parse or verify step fails, return $\perp$
2. Parse C into $(\mathbf{epk}, em, ah, psk_m, sig_m)$
3. Parse $rk_{i \to j}$ into $(\mathbf{rpk}, rek, \mathbf{rep}, psk_i, sig_{rk})$
4. Validate the signature of the encrypted message, by verifying that
$$\mathcal{V}(\mathbf{epk}||em||ah||psk_m, psk_m, sig_m) = true$$
5. Validate the signature for the re-encryption key, by verifying that
$$\mathcal{V}(\mathbf{rpk}||rek||\mathbf{rep}||psk_i, psk_i, sig_{rk}) = true$$
6. The *randomized re-encryption secret key*, $rrsk$, is chosen randomly from $\mathbb{Z}_p^*$
7. The *randomized re-encryption temporary key*, rrK, is chosen randomly from $\mathbb{G}_T$
8. The *randomized re-encryption public key* is
$$\mathbf{rrpk} \leftarrow rrsk \cdot \mathbf{g}$$
9. The *randomized re-encryption encrypted temporary key* is
$$rrek \leftarrow rrK \cdot e(\mathbf{pk}_j, \mathbf{g_1})^{rrsk}$$
10. The re-encrypted message is
$$em' \leftarrow em \cdot e(\mathbf{epk}, \mathbf{rep}+\mathbf{H_2}(rrK))$$
11. The *altered ciphertext* is
$$C' \leftarrow (\mathbf{epk}, em', ah)$$
12. The re-encryption block is
$$RB \leftarrow (\mathbf{rpk}, rek, \mathbf{rrpk}, rrek)$$
13. The re-encrypted ciphertext RC', which is not complete, is
$$RC' \leftarrow (C', RB)$$

For each subsequent re-encryption key in the chain, $rk_{a \to b}$, perform the following:

1. Parse the last *re-encryption block* of RC' into $(\mathbf{rpk}_{last}, rek_{last}, \mathbf{rrpk}_{last}, rrek_{last})$
2. Parse $rk_{a \to b}$ into $(\mathbf{rpk}, rek, \mathbf{rep}, psk_a, sig_{rk})$
3. Validate the signature for the re-encryption key, by verifying that
$$\mathcal{V}(\mathbf{rpk}||rek||\mathbf{rep}||psk_a, psk_a, sig_{rk}) = true$$

4. The *randomized re-encryption secret key*, $rrsk$, is chosen randomly from $\mathbb{Z}_p^*$
5. The *randomized re-encryption temporary key*, rrK, is chosen randomly from $\mathbb{G}_T$
6. The *randomized re-encryption public key* is
$$\mathbf{rrpk} \leftarrow rrsk \cdot \mathbf{g}$$
7. The new *randomized re-encryption encrypted temporary key* is
$$rrek \leftarrow rrK \cdot e(\mathbf{pk}_j, \mathbf{g_1})^{rrsk}$$
8. Replace $rrek_{last}$ in the last re-encryption block with
$$rrek'_{last} \leftarrow rrek_{last} \cdot e(\mathbf{rrpk}_{last}, \mathbf{rep} + \mathbf{H_2}(rrK))$$
9. Replace rek_{last} in the last re-encryption block with
$$rek'_{last} \leftarrow rek_{last} \cdot e(\mathbf{rpk}_{last}, \mathbf{rep} + \mathbf{H_2}(rrK))$$
10. Append a new re-encryption block RB_{new} to RC'
$$RB_{new} \leftarrow (\mathbf{rpk}, rek, \mathbf{rrpk}, rrek)$$

Finally, append the signature:

$$sig \leftarrow \mathcal{S}(RC'||psk_{proxy}, ssk_{proxy})$$
$$RC \leftarrow (RC', psk_{proxy}, sig)$$

In Decrypt:

To decrypt a level-l ciphertext $C^{(l)}$:

1. Parse $C^{(l)}$ into $(C', RB'^{(2)}, \ldots, RB'^{(l-1)}, RB^{(l)})$
2. Parse C' into $(\mathbf{epk}, em', ah)$
3. Parse $RB^{(l)}$ into $(\mathbf{rpk}^{(l)}, rek^{(l)}, \mathbf{rrpk}^{(l)}, rreK^{(l)})$
4. For each integer k in $[2, l-1]$, parse $RB'^{(k)}$ into $(\mathbf{rpk}^{(k)}, rek'^{(k)}, \mathbf{rrpk}^{(l)}, rreK'^{(l)})$
5. Recover the $(l-1)$-st temporary key K_{l-1} via the operation
$$K^{(l-1)} \leftarrow rek^{(l)} \cdot e(\mathbf{rpk}^{(l)}, (-sk_i) \cdot \mathbf{g_1})$$
6. Recover the $(l-1)$-st random re-encryption temporary key
$$rrK^{(l-1)} \leftarrow rrK^{(l)} \cdot e(\mathbf{rrpk}^{(l)}, (-sk_i) \cdot \mathbf{g_1})$$
7. For each integer k from $l-2$ down to 1, recover the k-th temporary key K_k and random re-encryption temporary key rrK_k via the operation
$$K^k \leftarrow rek'^{(k+1)} \cdot e(\mathbf{rpk}^{(k+1)}, -\mathbf{H_2}(K^{(k+1)}))$$
$$rrK^k \leftarrow rrek'^{(k+1)} \cdot e(\mathbf{rrpk}^{(k+1)}, -\mathbf{H_2}(K^{(k+1)} - \mathbf{H_2}(rrK^{(k+1)}))$$
8. Finally, recover the plaintext m via the operation
$$m \leftarrow em' \cdot e(\mathbf{epk}, -\mathbf{H_2}(K^{(1)}) - \mathbf{H_2}(rrK^{(1)}))$$

References

[CL14] Y. Cai and X. Liu. A multi-use CCA-secure proxy re-encryption scheme. *IEEE 12th International Conference on Dependable, Autonomic, and Secure Computing*, 7, 2014.

[WC09] Hongbing Wang and Zhenfu Cao. A fully secure unidirectional and multi-use proxy re-encryption scheme. *ACM CCS Poster Session*, 2009.

[ZW13] J. Zhang and X. A. Wang. On the security of two multi-use CCA-secure proxy re-encryption schemes. *Int. J. Intelligent Information and Database Systems*, 7(5):422–440, 2013.