Data Storage By Secure Crumbling With Signing Trusted Third Parties

Cyril Dever
Edgewhere

September 18, 2019

Abstract

We define a secure data storage solution based on the presence of one (or more) trusted third parties necessary to perform encryption and decryption operations on a message split in crumbs. This secure storage method is particularly safe since the encryption elements are distributed among the different participants and can't be discovered by a single procedure which would allow breaking a unique encryption code. We show that this distribution of crumbs and their separate encryption considerably increases the security of the storage since, in the absence of a participant, the message can't be recovered. Furthermore, the algorithm doesn't allow anyone other than the rightful owner of the original message to know in clear all or part of the data at any time whatsoever. This technique is pending patent[*].

I. Introduction

There are already multiple available ways to store data after encrypting it. However, the current techniques of data encryption for the storage and recovery of stored data and their decryption are operations all the more complex as the security must be high.

This complexity comes with the added burden of the risk that the encryption key is always susceptible to being broken and/or hacked.

The goal of our new algorithm, called the `crumbl`™ technology, is to develop simple yet particularly effective means for securing data storage.

Our procedure describes a method of secure storage of a source data, owned by one (or more) *holder(s)*, using already proven techniques of asymmetric encryption with the participation of so-called trusted third parties, each having a pair of private and public keys.

II. Basic Definitions

Definition 1 (Source Data)**.** The source data d is the data that has to be protected by the `crumbl` encryption protocol.

Definition 2 (Crumbl)**.** A *crumbl* (or crumbled string) is the final result of the encryption of a source data through the `crumbl` process. Among other elements, it uses crumbs which come from slices of the source data.

Definition 3 (Crumb)**.** A crumb ς is an encrypted portion of data of size n in its binary form:

$$\varsigma := \sum_{j=0}^{n-1} x_j \mid x_j \in \{0,1\} \tag{1}$$

It could be the byte array itself or any string representation of it (hexadecimal, binary, base-64, ...).

When presented with a lower index (eg. ς_8), it indicates the order (starting at 0) in which to eventually concatenate it with the others. With an added upper index (eg. ς^π), it indicates its signer (π) during encryption. Finally, an upper left index refers to the current operation (eg. ${}^{O_1}\varsigma$ for operation O_1).

A set of crumbs can only be assigned to one source data. In other words, it is obvious that one can't mix a crumb ς_1 from a data d_1 with a crumb ς_2 from a data d_2.

Definition 4 (Slice)**.** A slice σ is a padded plaintext portion of the source data.

Let $\zeta()$ be a slicing function, $\mu()$ a padding function and $\mu^{-1}()$ its inverse. For t slices made out of a source data d, we have:

$$\begin{cases} \sigma_i := \mu\left[\zeta_i(d,t)\right] \\ d := \mu^{-1}(\sigma_0) \parallel \mu^{-1}(\sigma_1) \parallel \ldots \parallel \mu^{-1}(\sigma_{t-1}) \end{cases} \tag{2}$$

[*] filed under registration number `FR1908258` at INPI on July 19, 2019

III. The Protocol

Definition 5 (Operation)**.** An operation takes a source data and encrypts it with the `crumbl`, or back.

Definition 6 (Participant)**.** A participant $\pi \in P$ (or signer) is defined by his pair of public (PK) and private (SK) keys unique to an `crumbl` operation he is taking part along with other participants/signers.

$$\begin{aligned} \pi : P &\to (\mathcal{K} \times \mathcal{K}) \\ \pi_i &\mapsto (\pi_i^{SK}, \pi_i^{PK}) \end{aligned} \tag{3}$$

There are two kinds of participants involved in the process:

- The holders who wish to protect their asset, ie. the source data;
- The trusted third parties, generally being corporations and the main sponsors of the system, who only participate in data encryption/decryption as signers and are paid for it.

Definition 7 (Holder)**.** The holder is the only participant able to have access to the data in clear, ie. the source data. He could be the rightful owner of the data or anyone to whom the latter delegates its use.

He is (or they are, should there be more than one holder involved in an operation) the signer(s) of a special crumb: ς_0, ie. the one with index 0.

There must be at least one holder and one trusted third party in the list of participants[1].

1. Encryption

Algorithm 1 presents the encryption protocol of the `crumbl`™ technology.

Let p be the number of participants forming the set $P \leftarrow \{\pi_p\}$ of signers, $P_0 \in P$ the subset of holders, and $P_\tau \in P$ the subset of trusted third parties with $P_0 \cup P_\tau = P$.

And let H be the holder of the source data d.

Finally, let $\mathfrak{c}()$ be the encryption function of the `crumbl`[2]:

$$\begin{aligned} \mathfrak{c} : \quad \omega \times \mathcal{K} &\to \omega \\ (msg, pubkey) &\mapsto \mathfrak{c}(msg, pubkey) \end{aligned} \tag{4}$$

Algorithm 1: Encryption protocol

Input: d, P
Output: the crumbled string Cr or an error

if $|d| = 0 \vee |P| < 2$ **then**
  throw *invalid input*
initialize a new set of crumbs: $\mathcal{C} \leftarrow \emptyset$;
ask all participants $\pi_i \in P \setminus \pi_H$ for their new public key;
each participant π_i creates a new pair of keys along with a request ID π_i^{RID}, this tuple being stored for future use in the decryption process;
d is prepared and split into a set of t slices $\{\sigma_0, \ldots, \sigma_{t-1}\}$ with: $t = |P_\tau| + 1$;
H encrypts σ_0 with his own new public key:

$$\mathcal{C} \leftarrow \varsigma_0^{\pi_H} := \mathfrak{c}_{\pi_H}(\sigma_0, \pi_H^{PK}) \tag{5}$$

while *H receives each participant's public key (π_i^{PK})* **do**
  if $\pi_i \in P_0$ **then**
    H encrypts σ_0:

$$\mathcal{C} \leftarrow \varsigma_0^{\pi_i} := \mathfrak{c}_{\pi_i}(\sigma_0, \pi_i^{PK})$$

  else
    all other slices are encrypted by H with the received public key:

$$\begin{aligned} &\forall j \in \{\sigma_1, \ldots, \sigma_{t-1}\} : \\ &\mathcal{C} \leftarrow \varsigma_j^{\pi_i} := \mathfrak{c}_{\pi_i}(\sigma_j, \pi_i^{PK}) \end{aligned} \tag{6}$$

Cr is finalized by H using d and the set of all crumbs $\mathcal{C}$;
return Cr

[1] We shall see that maximum security starts with at least four participants: one holder and three trusted third parties.

[2] It could be any asymmetric protocol as long as it is available for H in the *words* space ω. Thus, we assume that H knows which protocol uses each participant π_i; therefore, he'd be using the appropriate $\mathfrak{c}_{\pi_i}()$ function.

As shown, everything takes place in H environment which guarantees that the source data is never sent, let alone known, by any other stakeholder.

If no error is raised, the output crumbl Cr can be stored anywhere, by any of the stakeholders and/or some outsourcer (eg. a hosting service). In any case, H should store a tuple of references to Cr (or d) and the keypair used for the operation. He may also store its verification hash and use it for that purpose.

Definition 8 (Verification Hash)**.** A verification hash V is made of the concatenation of the 64 first characters of a crumbled string Cr:

$$V := \|_{i=1}^{64} Cr[i] \tag{7}$$

where $Cr[i]$ is the i-th character of Cr.

By design, the verification hash is unique to an operation — see Definition 10 and (12).

It is generally used for search or storage purposes[3].

By definition, it is verified that[4]: $V \subset Cr$.

2. Decryption

Algorithm 2 presents the decryption protocol.

Let P'_τ a subset of P_τ of size $1 \leq n \leq |P_\tau| - 1$, and H_1 one of the signing holders that wishes to recover the data.

And let $v(d)$ be the hashered function to build a verification hash for a data d — see (12), and `MIN_LENGTH` > 64 the minimum required length of a crumbled string.

Finally, let $\mathfrak{D}()$ be the decryption function of the `crumbl`:

$$\begin{aligned} \mathfrak{D}: \quad \mathbb{N} \times \omega \times \mathcal{K} &\to \omega \\ (j, Cr, privkey) &\mapsto \mathfrak{D}(j, Cr, privkey) \end{aligned} \tag{8}$$

with j being the j-th crumb.

If there's no error, the returned item is a copy of the original source data as a string.

[3]For example, our latest implementation requires that we ask a hosting service for a crumbl by sending its verification hash.

[4]We will use the notation $V \subset Cr$ in the rest of the document when we want to assert that a passed V is Cr's appropriate verification hash.

Algorithm 2: Decryption protocol

Input: Cr, a decrypter $H_1 \in P_0$, P'_τ, an optional verification hash V
Output: the data d or an error

if $V \neq \emptyset \wedge V \not\subset Cr$ **then**
    throw *invalid verification hash*
else if $|Cr| <$ `MIN_LENGTH` **then**
    throw *invalid crumbled string*
from Cr, get the number t of slices;
initialize a new set of slices $\mathcal{S} \leftarrow \emptyset$;
set the timeout limit θ;
initialize R the map of received messages by $H1$ with cardinality t: $\forall i \in [1..t] : R_i \leftarrow \emptyset$;
for $i \leftarrow 1$ **to** p **by** 1 **do**
    H_1 requests his decrypted crumbs to trusted third party π_i;
while *current time* $\leq \theta$ **do**
    foreach $\pi_i \in P'_\tau$ **do**
        for $j \leftarrow 1$ **to** t **by** 1 **do**
            if $\exists \sigma_j := \mathfrak{D}(j, Cr, \pi_i^{SK})$ **then**
                π_i sends his slice σ_j to $H1$: $R_{j,i} \leftarrow \sigma_j^{\pi_i}$;
for $j \leftarrow 1$ **to** $|R|$ **by** 1 **do**
    process received $\sigma_j^{\pi_i}$: $\sigma_j \leftarrow \sigma_j^{\pi_i}$;
    if $\sigma_j \notin \mathcal{S}$ **then**
        $\mathcal{S} \leftarrow \sigma_j$;
if $|\mathcal{S}| \neq t$ **then**
    throw *missing* $(t - |\mathcal{S}|)$ *slices*
H_1 decrypts crumb 0: **if** $\exists \sigma_0 := \mathfrak{D}(0, Cr, H_1{}^{SK})$ **then**
    $S \leftarrow \sigma_0$
else
    throw H_1 *is not a holder*
use (2) on $\mathcal{S} := \{\sigma_t\}$ to recover d:

$$d \leftarrow \mu^{-1}(\sigma_0) \parallel \mu^{-1}(\sigma_1) \parallel \ldots \parallel \mu^{-1}(\sigma_{t-1})$$

if $V \neq \emptyset \wedge v(d) \neq V$ **then**
    throw *invalid recovered data d against verification hash V*
return d

IV. The Process

This section describes the detailed encryption process used in the $\mathfrak{c}()$ and $\mathcal{D}()$ functions in the above protocol.

Let us first give a more precise definition of a crumbl through its actual representation as a crumbled string.

Definition 9 (Crumbled string)**.** The final crumbled string Cr is made of the concatenation of a so-called hashered prefix with the concatenation of the base-64 string representation of all the crumbs:

$$Cr := v(d) \parallel \left(\sum_{i=0}^{t} \sum_{j=0}^{p} (\varsigma_i^{\pi_j})_{64} \right) \tag{9}$$

where we use the symbol $\sum$ in the end part of (9) for concatenation[5].

1. A unique prefix

Let $sort(items)$ be the function that returns a lexicographically sorted set of items[6], and $cut(word, at)$ the function that splits the passed $word$ in two after the at-th character.

And let $\mathfrak{h}()$ be a secure cryptographic hashing function returning a 256-bits hash[7].

Definition 10 (Hashered prefix)**.** We build the hashered prefix by concatenating two parts:

- The 32 first characters of the hexadecimal string representation of the hash of the source data (using $\mathfrak{h}()$): h^+;
- The 32 last characters of this hash (h^-) XORed with the padded lexicographically sorted owners' crumbs concatenation in hexadecimal.

$$\Rightarrow h^+, h^- := cut\left(\mathfrak{h}(d), 32\right) \tag{10}$$

Let $HR()$ be the hashering function that takes h^- and the set of crumbs $\mathcal{C}$, and returns the second part of the hashered prefix [8].

$$\begin{aligned} HR : \omega \times \omega^t &\to \omega \\ (h^-, \mathcal{C}) &\mapsto h^- \oplus \left(\textstyle\sum^{\parallel} sort(\mathcal{C}) \right) \end{aligned} \tag{11}$$

The full hashering function $v()$ takes the source data d and its associated set of crumbs to return the hashered prefix using (10) in the process to build h^+ and h^-:

$$\begin{aligned} v : \quad \omega \times \omega^t &\to \omega \\ (d, \mathcal{C}) &\mapsto v(d, \mathcal{C}) := h^+ \parallel HR(h^-, \mathcal{C}) \end{aligned} \tag{12}$$

The use of the hashering function ($HR()$) ensures the hashered prefix uniqueness.

Proof. Let d be a source data owned by π_h, and ${}^{O_1}Cr$ and ${}^{O_2}Cr$ the results of the `crumbl` process from two operations O_1 and O_2 initiated by π_h.

We are trying to prove that ${}^{O_1}Cr$ and ${}^{O_2}Cr$ will be different as long as at least π_h respects the process (even though some or all other participants don't play fair).

Thanks to Definition 6, we know that π_h would use different keypairs for O_1 and 0_2 such as:

$${}^{O_1}\varsigma_0^{\pi_h} \neq {}^{O_2}\varsigma_0^{\pi_h}$$

Because (11) uses $sort()$ in $HR()$, we know that ς_0 will always be the first crumb used to build the hashered prefix.

Therefore, let x_1 and x_2 be the hashered prefixes during O_1 and O_2, through (12) we have:

$$\begin{cases} x_1 \leftarrow v(d, \mathcal{C}_1) & \iff \mathcal{C}_1 := \{{}^{O_1}\varsigma_0^{\pi_h}, \ldots\} \\ x_2 \leftarrow v(d, \mathcal{C}_2) & \iff \mathcal{C}_2 := \{{}^{O_2}\varsigma_0^{\pi_h}, \ldots\} \end{cases}$$

$$\Rightarrow x_1 \neq x_2$$

Using (9), we see that for the same $d \in \omega$:

$$\begin{cases} {}^{O_1}Cr := x1 \parallel \left(\sum_{i=0}^{t} \sum_{j=0}^{p} ({}^{O_1}\varsigma_i^{\pi_j})_{64} \right) \\ {}^{O_2}Cr := x2 \parallel \left(\sum_{i=0}^{t} \sum_{j=0}^{p} ({}^{O_2}\varsigma_i^{\pi_j})_{64} \right) \end{cases}$$

$$\Rightarrow \text{if } \pi_{h_{O_1}} \neq \pi_{h_{O_2}} \text{ then } {}^{O_1}Cr \neq {}^{O_2}Cr$$

By construction, each Cr is unique at least thanks to the uniqueness of its hashered prefix if one participant (and even more so a holder) respects the protocol. □

[5]From now on, we may use this notation whenever it's clear in the explanation, or the $\sum^{\parallel}$ alternative when applied on a full set/array.

[6]If the items are presented as a map, they are first lexicographically sorted on their keys, then their values are also lexicographically sorted.

[7]We use the `SHA-256` algorithm in our implementation because of its native availability in most browsers.

[8]Recall that when using a map, like $\mathcal{C}$, the first sorting is made on the keys which are the crumb numbers ranging from 0 to $t \in \mathbb{N}$.

Prepended to the crumbs, not only does this prefix gives the crumbled string absolute uniqueness, but it also allows easy indexing.

2. A step-by-step construct

2.1 Tools

2.1.1 Obfuscation

Let $\mathfrak{F}()$ be an implementation of an almost Format-Preserving Encryption scheme based on a Feistel[1] cipher[9].

$\mathfrak{F}()$ is used as our obfuscation tool in the process. It returns an obfuscated word of even length when applied on a source data, and the exact copy of the source data when applied on its obfuscated version.

2.1.2 Padding

Algorithm 3: Padding function μ

```
Input: a message m, the minimum size s for the output
Output: the appropriately padded message if necessary
let l be the length of m: l ← |m|;
δ ← l − s;
if δ ≥ 0 then
    return m
else
    if m[0] = PAD_0 then
        if m[l − 1] = PAD_1 then
            return (PAD_2 × δ) || m
        else
            return (PAD_1 × δ) || m
    else
        if m[l − 1] = PAD_0 then
            if m[0] = PAD_1 then
                return (PAD_2 × δ) || m
            else
                return (PAD_1 × δ) || m
        else
            return (PAD_0 × δ) || m
```

We herein define some padding (and respective unpadding) features such as an input returns unmodified if its length is greater than the wanted minimum size (in case of padding), or if it wasn't padded in the first place (in case of unpadding).

Let $\texttt{PAD}_0$, $\texttt{PAD}_1$ and $\texttt{PAD}_2$ be three special padding characters[10], and _ the symbol used when a variable should be discarded.

The reason why we use these three different kinds of padding characters is because of the way the Feistel cipher works during obfuscation: by switching left and right parts of the input data, we might take the risk to be mistaken at the end of the unpadding process should we choose padding characters that are the same than one of the two ends of the input, hence the different options for the padding character.

Algorithm 3 describes this special padding function $\mu()$.

In our implementation, we also add an optional parameter to μ to be sure to return a padded input of even length, therefore modifying a bit the flow of operations in the final function [11].

And Algorithm 4 shows $\mu^{-1}()$, the reverse unpadding function.

Algorithm 4: Unpadding function μ^{-1}

```
Input: an eventually padded message m
Output: the unpadded message
if m[0] ≠ PAD_0 ∧ m[0] ≠ PAD_1 ∧ m[0] ≠ PAD_2 then
    return m
else
    _, m' ← cut(m, 1);
    while m'[0] ∈ {PAD_0, PAD_1, PAD_2} do
        _, m' ← cut(m', 1);
    return m'
```

[9] We use our own implementation described in [2].

[10] In our implementation, we use the following UTF-8 characters:

- $\texttt{PAD}_0 \leftarrow$ `U+0002` (start-of-text) character;
- $\texttt{PAD}_1 \leftarrow$ `U+0004` (end-of-transmission) character;
- $\texttt{PAD}_2 \leftarrow$ `U+0005` (enquiry) character.

[11] Check out the code in Go (`https://github.com/cyrildever/feistel` or in TypeScript (`https://github.com/cyrildever/feistel-cipher`).

Obviously, we have:

$$\forall d \in \omega : d' := \mu(d) \iff d := \mu^{-1}(d')$$

2.1.3 Slicing

The last tool needed before encryption is the *slicer*. It takes a string s to slice as well as the number of desired slices t and returns a set of padded slices $\mathcal{S} := \{\sigma_0, \ldots, \sigma_{t-1}\}$.

For maximum security, the slices are not supposed to be of the same length; therefore, they run through the padding function $\mu()$ at the end of the process.

Let Δ_{max} be the maximum standard deviation allowed by the system between slices, and $Rnd()$ a Pseudo-Random Number Generator (PRNG) function[3].

Algorithm 5: Slicer ζ

Input: s, t
Output: $\mathcal{S}$

initialize a set of split masks: $\mathcal{M} \leftarrow \emptyset$;
seed $Rnd()$;
set variables $pos \leftarrow 0$ and $\bar{x} \leftarrow \left\lfloor \frac{|s|}{t} \right\rfloor$;
while $|\mathcal{M}| \neq t \wedge (\sum_{j=1}^{t} \mathcal{M}_j) \neq |s|$ **do**
  for $i \leftarrow 0$ **to** $t-1$ **by** 1 **do**
    find the mask length l_i such as:

$$l_i := \bar{x} - \Delta_{max} < Rnd() \leq \bar{x} + \Delta_{max}$$

    $\mathcal{M} \leftarrow l_i$;
    $pos \leftarrow pos + l_i + 1$;
initialize $\mathcal{S} \leftarrow \emptyset$;
$(\sigma_0, r) \leftarrow cut(s, \mathcal{M}_0)$;
$\mathcal{S} \leftarrow \sigma_0$;
for $i \leftarrow 1$ **to** $t-1$ **by** 1 **do**
  $(\sigma_i, r) \leftarrow cut(r, \mathcal{M}_i)$;
  $\mathcal{S} \leftarrow \sigma_i$;
return $\mathcal{S}$

Note. The number of desired slices depends on the size of the input string s and the system's minimum length of slice `MIN_SLICE_SIZE`> 2.

2.2 Construct

Algorithm 6 describes the steps that leads from the source data d to the crumbled string Cr, provided we have all the necessary material at our disposal (participants' keys, tools, system settings configured, ...).

Let K be a secret key and N the number of rounds to use for obfuscation.

Algorithm 6: From data to crumbl

Input: d, P
Output: Cr, V

obfuscate input: $d' \leftarrow \mathfrak{F}(d, K, N)$;
eventually pad it: $d' \leftarrow \mu(d')$;
determine the number of slices t and therefore the Δ_{max};
build the slices: $\mathcal{S} \leftarrow \zeta_{\Delta_{max}}(d', t)$;
initialize the set of crumbs: $\mathcal{C} := \{\varsigma_{t \times |P| - 1}\}$;
allocate crumbs to participants in a map $\mathcal{A}_{\mathcal{P}}$ where each holder is associated with crumb ς_0 and each trusted third party with $t-1$ crumbs from $\mathcal{S}' := \mathcal{S} \setminus \varsigma_0$;
for $i \leftarrow 0$ **to** $t-1$ **by** 1 **do**
  for $j \leftarrow 0$ **to** $|A_P| - 1$ **by** 1 **do**
    $\mathcal{C} \leftarrow \mathfrak{c}(\sigma_i \in \mathcal{S}, \pi_j^{PK} \in P)$;
use (9) to build the crumbled string: $Cr := v(d) \parallel \left(\sum^{\parallel} \mathcal{C} \right)$;
return Cr, $v(d)$

The allocation method may differ depending on the security and continuity preferences of the system. The more trusted third parties encrypt crumbs, the heavier the result, the longer the Cr, but the more resilient the system can be (see section V).

In any case, the allocation rule must respect the following Proposition:

Proposition 1. Let $\mathcal{C}' := \mathcal{C} \setminus \varsigma_0$ be the set of crumbs to allocate to $\pi_i \in P_\tau \forall i := [1, \ldots, t-1]$. Each π_i must have at most $t-2$ crumbs to encrypt and the allocation should be evenly distributed among the participants such as

$$\sum_{i=1}^{t-1} P(\varsigma_i) = 1 \iff P(\varsigma_i) = \frac{1}{|P_\tau|} \tag{13}$$

ensuring maximum resilience to the system.

For example, with three participants, we have

$$|P_\tau| \leftarrow 3 \Rightarrow P(\varsigma_i) = \frac{1}{3}$$

which necessarily leads to the following distribution of crumbs:

$$\begin{aligned} \mathcal{C}^{\pi_{\tau_1}} &:= \{\varsigma_1, \varsigma_2\} \\ \mathcal{C}^{\pi_{\tau_2}} &:= \{\varsigma_1, \varsigma_3\} \\ \mathcal{C}^{\pi_{\tau_3}} &:= \{\varsigma_2, \varsigma_3\} \end{aligned}$$

$$\Rightarrow \begin{cases} P(\varsigma_1) = \frac{1}{6} + \frac{1}{6} = \frac{1}{3} \\ P(\varsigma_2) = \frac{1}{3} \\ P(\varsigma_3) = \frac{1}{3} \end{cases}$$

We will see in the proof of Proposition 3 that this case corresponds to a secure and resilient system.

2.3 Deconstruct

The inverse method (to recover a source data from a crumbl Cr) involves the respect of the decryption protocol (see Algorithm 2). Therefore, the holder π_H that wants to recover the data must first gather the decrypted crumbs from the trusted third parties.

As seen above, the allocation of crumbs at encryption respects the following formula: each trusted third party may only encrypt at most $t-1$ crumbs.

It follows that at least two trusted third parties are necessary (and could be enough) to get all the needed decrypted crumbs. As soon as they are in possession of the holder π_H, he can run Algorithm 7 that indeed details the last steps of the decryption protocol — see Algorithm 2.

Algorithm 7: From slices to data

Input: Cr, π_H^{SK}, $\mathcal{S}' := \mathcal{S} \setminus \varsigma_0$, an optional verification hash V
Output: the recovered data or an error

```
from Cr, get the number t of slices;
if |S'| ≠ t − 1 then
    throw not enough third-party slices
initialize S ← S';
π_H decrypts his own crumb ς_0:
    S ← ς_0 := 𝔇(0, Cr, π_H^SK);
use 2 to build the recovered data d:
    d ← Σ_{i=0}^{t−1} μ^{−1}(S_i)
if V ≠ ∅ ∧ v(d) ≠ V then
    throw invalid verification hash
return d
```

$$d \leftarrow \sum_{i=0}^{t-1} \mu^{-1}(\mathcal{S}_i)$$

V. Security and Resilience

It is easy to prove that the `crumbl`™ technology can be implemented in a secure and resilient way.

1. A secure system

Definition 11 (Secure)**.** We define the system to be fully *secure* if it doesn't only rely on a "classic" asymmetric or symmetric encryption scheme with one keypair eventually shared.

Proposition 2. If there is at least one holder π_H and one trusted third party π_τ, then the system is *secure*.

Proof. We start by proving that the system can't be secure if only one actor is present.

By design, only π_H can encrypt the crumb ς_0.

Therefore, if π_τ is alone, the system is not working, let alone secure.

And if π_τ is encrypting ς_0, he is acting as a holder. Now this implies that the lonely holder is encrypting all the crumbs, which is the very definition of a non-secure "classic" encryption scheme.

Secondly, if the two participants are present and $\pi_H \neq \pi_\tau$, then neither of them will have access to the other's keypair to decrypt the other's crumb(s).

Moreover, π_τ doesn't have access to ς_0; thus, he can't finalize the decryption process at all. This also means that, should π_H disappear or loses his secret key for the operation ${}^{O}\pi_H^{PK}$, the data won't ever be recovered. Thus, the system is secure in that case.

If π_H respects the encryption protocol, he wouldn't encrypt all the crumbs, leaving at least one crumb $\varsigma_i \mid i \neq 0$ only encrypted by π_τ; thus, he will need the latter to eventually finalize the decryption process, proving the system is secure in the absence of one necessary participant.

Finally, by construction, it is not a "classic" encryption scheme anymore. Therefore, the system is *secure*. □

2. A resilient system

Definition 12 (Resilient)**.** We define the system to be *resilient* if the failure of a trusted third party doesn't prevent a holder to recover his data.

Proposition 3. Let $\mathcal{G}_\tau$ be the total possible set of trusted third parties, and ${}^{O}P_\tau \subset {}^{O}P \subset \mathcal{G}_\tau$ the actual trusted third parties participating in an operation O, if its cardinality $|{}^{O}P_\tau| \geq 3$ and $|\mathcal{G}_\tau| > |{}^{O}P_\tau|$ then the `crumbl`™ protocol is both secure and resilient as long as the system enforces a replacement mechanism for any failing trusted third party $\in \mathcal{G}_\tau$.

Proof. We first prove that, although the system using one trusted third party is secure (cf. Definition 11), it is not *resilient*. Indeed, if the one and only trusted third party fails, then the holder is left with at least one missing crumb, and therefore can't recover the data anymore.

In the same way, if there are two trusted third parties (π_{τ_1} and π_{τ_2}) involved, as by construction each trusted third party encrypts $t-2$ crumbs, there is at least one crumb that only π_{τ_1} doesn't encrypt while π_{τ_2} does (and vice-versa).

So, if either one of them fails, the system is still not resilient.

Finally, if there are 3 participating trusted third parties $\pi_{\tau_1}, \pi_{\tau_2}, \pi_{\tau_3} \in P_\tau$, from (5) and (6), we have: *(for instance with one holder $\pi_H \in P_0$)*

$$
\begin{aligned}
&\mathcal{C}^{\pi_H} := \{\varsigma_0\}\\
&\mathcal{C}^{\pi_{\tau_1}} := \{\varsigma_1, \varsigma_2\}\\
&\mathcal{C}^{\pi_{\tau_2}} := \{\varsigma_1, \varsigma_3\}\\
&\mathcal{C}^{\pi_{\tau_3}} := \{\varsigma_2, \varsigma_3\}\\
&\iff \mathcal{C} := \mathcal{C}^{\pi_H} \cup \mathcal{C}^{\pi_{\tau_1}} \cup \mathcal{C}^{\pi_{\tau_2}} \cup \mathcal{C}^{\pi_{\tau_3}}
\end{aligned}
$$

It is easy to see that if any of the trusted third party fails, the other two still have the full set of crumbs under their control, eg. if π_{τ_1} fails, then $\mathcal{C}^{\pi_{\tau_2}} \cup \mathcal{C}^{\pi_{\tau_3}} := \{\varsigma_1, \varsigma_2, \varsigma_3\}$ which will allow π_H with his own ς_0 to rebuild the complete set $\mathcal{C}$ of the operation, and therefore recover the data.

And so on and so forth the more you add trusted third party participants and respects the allocation rule (cf. Proposition 1). □

References

[1] Horst Feistel. *Cryptography and Computer Privacy*, Scientific American, 1973.

[2] Cyril Dever. *Feistel Cipher with Hash Round Function*, 2021. `https://github.com/cyrildever/feistel`

[3] E. Barker, W. Barker, W. Burr, W. Polk, M. Smid. *Recommendation for Key Management*, NIST, 2012.