# Data Storage By Secure Crumbling With Signing Trusted Third Parties

Cyril Dever
Edgewhere

September 18, 2020

**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 $\varsigma$ 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. $\varsigma_8$), it indicates the order (starting at 0) in which to eventually concatenate it with the others. With an added upper index (eg. $\varsigma^{\pi}$), it indicates its signer ($\pi$) during encryption.

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 $c1$ from a data $d1$ with a crumb $c2$ from a data $d2$.

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

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

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

[*] 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.

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

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: $\varsigma_0$, 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]:

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

**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 $\pi_i$ creates a new pair of keys along with a request ID $\pi_i^{RID}$, this tuple being stored for future use in the decryption process;

$d$ is prepared and split into a set of $t$ slices $\{\sigma_0, \dots, \sigma_{t-1}\}$ with: $t = |P_\tau| + 1$;

$H$ encrypts $\sigma_0$ 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 ($\pi_i^{PK}$)* **do**
  **if** $\pi_i \in P_0$ **then**
    $H$ encrypts $\sigma_0$:

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

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

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

$Cr$ 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 $\omega$. Thus, we assume that $H$ knows which protocol uses each participant $\pi_i$; 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 $Cr$:

$$V := \|_{i=1}^{64} Cr[i] \tag{6}$$

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

By design, the verification hash is unique to an operation — see Definition 10 and (11).

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 $P'_\tau$ a subset of $P_\tau$ 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 (11), and `MIN_LENGTH` $> 64$ the minimum required length of a crumbled string.

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

$$\begin{array}{rll} \mathfrak{D} : & \mathbb{N} \times \omega \times \mathcal{K} & \to \omega \\ & (j, Cr, privkey) & \mapsto \mathfrak{D}(j, Cr, privkey) \end{array} \tag{7}$$

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 $H_1 \in P_0$, $P'_\tau$, 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 $\theta$;
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 $\pi_i$;
**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**
        $\pi_i$ sends his slice $\sigma_j$ 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 $d$:

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

**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.

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

$$Cr := v(d) \parallel \left( \sum_{i=0}^{t} \sum_{j=0}^{p} (\varsigma_i^{\pi_j})_{64} \right) \tag{8}$$

where we use the symbol $\sum$ in the end part of (8) 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.

$$\Rightarrow h^+, h^- := cut\,(\mathfrak{h}(d), 32) \tag{9}$$

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.

$$\begin{aligned} HR : \omega \times \omega^t &\to \omega \\ (h^-, \mathcal{C}) &\mapsto h^- \oplus \left( \sum\nolimits^{\parallel} sort(\mathcal{C}) \right) \end{aligned} \tag{10}$$

[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.

The full hashering function $v()$ takes the source data $d$ and its associated set of crumbs to return the hashered prefix using (9) in the process to build $h^+$ and $h^-$:

$$\begin{aligned} v : \quad \omega \times \omega^t &\to \omega \\ (d, \mathcal{C}) &\mapsto v(d, \mathcal{C}) := h^+ \parallel HR(h^-, \mathcal{C}) \end{aligned} \tag{11}$$

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

*Proof.* Let $d$ be a source data owned by $\pi_h$, and $Cr_1$ and $Cr_2$ the results of the `crumbl` process from two operations $O_1$ and $O_2$ (respectively initiated by $\pi_{h_{O_1}}$ and $\pi_{h_{O_2}}$).

We are trying to prove that, for $O_1$ and $O_2$, their respective crumbled string ($Cr^{O_1}$ and $Cr^{O_2}$) will be different as long as at least $\pi_h$ respects the process (even if some or all other participants don't play fair).

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

$$\varsigma_0^{\pi_{h_{O_1}}} \neq \varsigma_0^{\pi_{h_{O_2}}}$$

Because (10) uses $sort()$, we known that $\varsigma_0$ will always be the first crumb used to build the hashered prefix.

Therefore, through (11), let $x_1$ and $x_2$ be the hashered prefixes during $O_1$ and $O_2$, we have:

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

$$\Rightarrow x_1 \neq x_2$$

With (8), we can conclude that:

$$\forall \pi_h, d : \text{if } \pi_{h_{O_1}} \neq \pi_{h_{O_2}} \text{ then } Cr^{O_1} \neq Cr^{O_2}$$

□

Prepended to the crumbs, this prefix gives the crumbled string absolute uniqueness, in particular between two different operations on the same source data.

Furthermore, it allows easy indexing.

## 2. Crumb and uncrumb

Lorem ipsum ...