

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 `crumb1`® 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 `crumb1` encryption protocol.

Definition 2 (Crumb). A *crumb* (or crumbled string) is the final result of the encryption

of a source data through the `crumb1` 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:

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

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:

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

*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 `crumb1`, 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 `crumb1` operation he is taking part along with other participants/signers.

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

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: ζ_0 , ie. the one with index 0.

There must be at least one holder and one trusted third party in the list of participants¹.

1. ENCRYPTION

Algorithm 1 presents the encryption protocol of the `crumb1`® 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 .

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

Finally, let $\mathfrak{c}()$ be the encryption function of the `crumb1`²:

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

Algorithm 1: Encryption protocol

Input: d, P

Output: the crumbled string Cr or an error

- 1 **if** $|d| = 0 \vee |P| < 2$ **then**
- 2 **throw** *invalid input*
- 3 initialize a new set of crumbs: $\mathcal{C} \leftarrow \emptyset$;
- 4 ask all participants $\pi_i \in P \setminus \pi_H$ for their new public key;
- 5 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;
- 6 d is prepared and split into a set of t slices $\{\sigma_0, \dots, \sigma_{t-1}\}$ with: $t = |P_\tau| + 1$;
- 7 H encrypts σ_0 with his own new public key:

$$\mathcal{C} \leftarrow \zeta_0^{\pi_H} := \mathfrak{c}_{\pi_H}(\sigma_0, \pi_H^{PK}) \quad (5)$$

- 8 **while** H receives each participant's public key (π_i^{PK}) **do**
 - 9 **if** $\pi_i \in P_0$ **then**
 - 10 H encrypts σ_0 :
 - 11 $\mathcal{C} \leftarrow \zeta_0^{\pi_i} := \mathfrak{c}_{\pi_i}(\sigma_0, \pi_i^{PK})$
 - 12 **else**
 - 13 all other slices are encrypted by H with the received public key:
 - 14 $\forall j \in \{\sigma_1, \dots, \sigma_{t-1}\} :$
 - 15 $\mathcal{C} \leftarrow \zeta_j^{\pi_i} := \mathfrak{c}_{\pi_i}(\sigma_j, \pi_i^{PK})$
 - 16 **end while**
 - 17 Cr is finalized by H using d and the set of all crumbs \mathcal{C} ;
 - 18 **return** Cr
-

²It 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 :

$$V := \parallel_{i=1}^{64} Cr[i] \quad (6)$$

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 purposes³.

By definition, it is verified that⁴: $V \subset Cr$.

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 hashed function to build a verification hash for a data d — see (11), and $\text{MIN_LENGTH} > 64$ the minimum required length of a crumbled string.

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

$$\begin{aligned} \mathcal{D} : \quad \mathbb{N} \times \omega \times \mathcal{K} &\rightarrow \omega \\ (j, Cr, \text{privkey}) &\mapsto \mathcal{D}(j, Cr, \text{privkey}) \end{aligned} \quad (7)$$

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.

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

⁴We 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

```

1 if  $V \neq \emptyset \wedge V \not\subset Cr$  then
2   throw invalid verification hash
3 else if  $|Cr| < \text{MIN\_LENGTH}$  then
4   throw invalid crumbled string
5 from  $Cr$ , get the number  $t$  of slices;
6 initialize a new set of slices  $\mathcal{S} \leftarrow \emptyset$ ;
7 set the timeout limit  $\theta$ ;
8 initialize  $R$  the map of received
  messages by  $H1$  with cardinality  $t$ :
   $\forall i \in [1..t] : R_i \leftarrow \emptyset$ ;
9 for  $i \leftarrow 1$  to  $p$  by 1 do
10    $H_1$  requests his decrypted crumbs to
    trusted third party  $\pi_i$ ;
11 while current time  $\leq \theta$  do
12   foreach  $\pi_i \in P'_\tau$  do
13     for  $j \leftarrow 1$  to  $t$  by 1 do
14       if  $\exists \sigma_j := \mathcal{D}(j, Cr, \pi_i^{SK})$  then
15          $\pi_i$  sends his slice  $\sigma_j$  to  $H1$ :
          $R_{j,i} \leftarrow \sigma_j^{\pi_i}$ ;
16 for  $j \leftarrow 1$  to  $|R|$  by 1 do
17   process received  $\sigma_j^{\pi_i}$ :  $\sigma_j \leftarrow \sigma_j^{\pi_i}$ ;
18   if  $\sigma_j \notin \mathcal{S}$  then
19      $\mathcal{S} \leftarrow \sigma_j$ ;
20 if  $|\mathcal{S}| \neq t$  then
21   throw missing  $(t - |\mathcal{S}|)$  slices
22  $H_1$  decrypts crumb 0: if
   $\exists \sigma_0 := \mathcal{D}(0, Cr, H_1^{SK})$  then
23    $\mathcal{S} \leftarrow \sigma_0$ 
24 else
25   throw  $H_1$  is not a holder
26 use (2) on  $\mathcal{S} := \{\sigma_i\}$  to recover  $d$ :
   $d \leftarrow \mu^{-1}(\sigma_0) \parallel \mu^{-1}(\sigma_1) \parallel \dots \parallel \mu^{-1}(\sigma_{t-1})$ 
27 if  $V \neq \emptyset \wedge v(d) \neq V$  then
28   throw invalid recovered data  $d$  against
    verification hash  $V$ 
29 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 hashed prefix with the concatenation of the base-64 string representation of all the crumbs:

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

where we use the symbol Σ in the end part of (8) for concatenation⁵.

1. A UNIQUE PREFIX

Let $sort(items)$ be the function that returns a lexicographically sorted set of items⁶, 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 hash⁷.

Definition 10 (Hashed prefix). We build the hashed 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.

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

Let $HR()$ be the hashing function that takes h^- and the set of crumbs \mathcal{C} , and returns

the second part of the hashed prefix⁸.

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

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

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

The use of the hashing function ($HR()$) ensures the hashed prefix uniqueness.

Proof. Let d be a source data owned by π_h , and O_1Cr and O_2Cr 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_1Cr and O_2Cr 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 O_2 such as:

$$O_1 \zeta_0^{\pi_h} \neq O_2 \zeta_0^{\pi_h}$$

Because (10) uses $sort()$ in $HR()$, we know that ζ_0 will always be the first crumb used to build the hashed prefix.

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

$$\begin{aligned} \begin{cases} x_1 \leftarrow v(d, \mathcal{C}_1) \\ x_2 \leftarrow v(d, \mathcal{C}_2) \end{cases} &\iff \begin{cases} \mathcal{C}_1 := \{O_1 \zeta_0^{\pi_h}, \dots\} \\ \mathcal{C}_2 := \{O_2 \zeta_0^{\pi_h}, \dots\} \end{cases} \\ &\Rightarrow x_1 \neq x_2 \end{aligned}$$

Using (8), we see that for the same $d \in \omega$:

$$\begin{cases} O_1Cr := x1 \parallel \left(\sum_{i=0}^t \sum_{j=0}^p (O_1 \zeta_i^{\pi_j})_{64} \right) \\ O_2Cr := x2 \parallel \left(\sum_{i=0}^t \sum_{j=0}^p (O_2 \zeta_i^{\pi_j})_{64} \right) \end{cases}$$

$$\Rightarrow \text{if } \pi_{h_{O_1}} \neq \pi_{h_{O_2}} \text{ then } O_1Cr \neq O_2Cr$$

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

⁸Recall 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 $t \in \mathbb{N}$.

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

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

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

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] cipher⁹.

$\mathfrak{F}()$ 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 \preceq

Input: a message m , the minimum size s for the output

Output: the appropriately padded message if necessary

```

1 let  $l$  be the length of  $m$ :  $l \leftarrow |m|$ ;
2  $\delta \leftarrow l - s$ ;
3 if  $\delta \geq 0$  then
4   return  $m$ 
5 else
6   if  $m[0] = \text{PAD}_0$  then
7     if  $m[l-1] = \text{PAD}_1$  then
8       return  $(\text{PAD}_2 \times \delta) \parallel m$ 
9     else
10      return  $(\text{PAD}_1 \times \delta) \parallel m$ 
11  else
12    if  $m[l-1] = \text{PAD}_0$  then
13      if  $m[0] = \text{PAD}_1$  then
14        return  $(\text{PAD}_2 \times \delta) \parallel m$ 
15      else
16        return  $(\text{PAD}_1 \times \delta) \parallel m$ 
17  else
18    return  $(\text{PAD}_0 \times \delta) \parallel m$ 

```

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

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 PAD_0 , PAD_1 and PAD_2 be three special padding characters¹⁰, 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 $\preceq()$, this special padding function.

In our implementation, we also add an optional parameter to \preceq to be sure to return a padded input of even length, therefore modifying a bit the flow of operations in the final function¹¹.

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

Algorithm 4: Unpadding function \preceq^{-1}

Input: an eventually padded message m

Output: the unpadded message

```

1 if  $m[0] \neq \text{PAD}_0 \wedge m[0] \neq \text{PAD}_1 \wedge m[0] \neq \text{PAD}_2$  then
2   return  $m$ 
3 else
4    $\_, m' \leftarrow \text{cut}(m, 1)$ ;
5   while  $m'[0] \in \{\text{PAD}_0, \text{PAD}_1, \text{PAD}_2\}$  do
6      $\_, m' \leftarrow \text{cut}(m', 1)$ ;
7   return  $m'$ 

```

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

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

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

Obviously, we have:

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

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 $\mathcal{S} := \{\sigma_0, \dots, \sigma_{t-1}\}$.

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}

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

$$l_i := \bar{x} - \Delta_{max} < Rnd() \leq \bar{x} + \Delta_{max}$$
- 7 $\mathcal{M} \leftarrow l_i$;
- 8 $pos \leftarrow pos + l_i + 1$;
- 9 initialize $\mathcal{S} \leftarrow \emptyset$;
- 10 $(\sigma_0, r) \leftarrow cut(s, \mathcal{M}_0)$;
- 11 $\mathcal{S} \leftarrow \sigma_0$;
- 12 **for** $i \leftarrow 1$ **to** $t - 1$ **by** 1 **do**
- 13 $(\sigma_i, r) \leftarrow cut(r, \mathcal{M}_i)$;
- 14 $\mathcal{S} \leftarrow \sigma_i$;
- 15 **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

- 1 obfuscate input: $d' \leftarrow \mathfrak{F}(d, K, N)$;
- 2 eventually pad it: $d' \leftarrow \mu(d')$;
- 3 determine the number of slices t and therefore the Δ_{max} ;
- 4 build the slices: $\mathcal{S} \leftarrow \zeta_{\Delta_{max}}(d', t)$;
- 5 initialize the set of crumbs:

$$\mathcal{C} := \{\zeta_{t \times |P| - 1}\};$$
- 6 allocate crumbs to participants in a map \mathcal{A}_P where each holder is associated with crumb ζ_0 and each trusted third party with $t - 1$ crumbs from $\mathcal{S}' := \mathcal{S} \setminus \zeta_0$;
- 7 **for** $i \leftarrow 0$ **to** $t - 1$ **by** 1 **do**
- 8 **for** $j \leftarrow 0$ **to** $|\mathcal{A}_P| - 1$ **by** 1 **do**
- 9 $\mathcal{C} \leftarrow \mathfrak{c}(\sigma_i \in \mathcal{S}, \pi_j^{PK} \in P)$;
- 10 use (8) to build the crumbled string:

$$Cr := v(d) \parallel \left(\sum^{\parallel} \mathcal{C} \right);$$
- 11 **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 heaviest the result, but the more resilient the system can be.

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.