77th European Study Group with Industry

27th September – 1st October 2010

Stefan Banach International Mathematical Center, Warsaw, Poland

Organizers:

Systems Research Institute
Polish Academy of Sciences

Institute of Mathematics
Polish Academy of Sciences

OCCAM
Oxford Centre for Collaborative Applied Mathematics

RB/3/2011

Cryptographic techniques used to provide integrity of digital content in long-term storage

REPORT ON THE PROBLEM

Honorary patronage:

British Embassy

Sponsors:

Sygnity
competence on

INDUSTRIAL DEVELOPMENT AGENCY
JOINT STOCK COMPANY

Problem presented by

Martin Šimka

Paweł Wojciechowski

Polish Security Printing Works (PWPW)

Report authors

Małgorzata Bladoszewska (University of Warsaw)

Tomasz Brożek (Warsaw School of Information Technology)

Michał Zając (University of Warsaw)

Contributors

Lucyna Cieślik (Polish Academy of Sciences)

Maria Donten-Bury (University of Warsaw)

Kamil Kulesza (Polish Academy of Science)

John Ockendon (University of Oxford)

Łukasz Stettner (Polish Academy of Sciences)

Piotr Wojdyłło (Polish Academy of Sciences)

Wladimir Zubkow (University of Oxford)

ESGI77 was jointly organised by

System Research Institute of the Polish Academy of Sciences

Institute of Mathematics of the Polish Academy of Sciences

Oxford Centre for Collaborative Applied Mathematics

and it was supported by

Sygnity S.A.

Industrial Development Agency Joint Stock Company

under the honorary patronage of

The British Embassy in Poland

Executive Summary

The main objective of the project was to obtain advanced mathematical methods to guarantee the verification that a required level of data integrity is maintained in long-term storage. The secondary objective was to provide methods for the evaluation of data loss and recovery. Additionally, we have provided the following initial constraints for the problem: a limitation of additional storage space, a minimal threshold for desired level of data integrity and a defined probability of a single-bit corruption.

With regard to the main objective, the study group focused on the exploration methods based on hash values. It has been indicated that in the case of tight constraints, suggested by PWPW, it is not possible to provide any method based only on the hash values. This observation stems from the fact that the high probability of bit corruption leads to unacceptably large number of broken hashes, which in turn stands in contradiction with the limitation for additional storage space.

However, having loosened the initial constraints to some extent, the study group has proposed two methods that use only the hash values. The first method, based on a simple scheme of data subdivision in disjoint subsets, has been provided as a benchmark for other methods discussed in this report. The second method ("hypercube" method), introduced as a type of the wider class of clever-subdivision methods, is built on the concept of rewriting data-stream into a n-dimensional hypercube and calculating hash values for some particular (overlapping) sections of the cube.

We have obtained interesting results by combining hash value methods with error-correction techniques. The proposed framework, based on the BCH codes, appears to have promising properties, hence further research in this field is strongly recommended.

As a part of the report we have also presented features of secret sharing methods for the benefit of novel distributed data-storage scenarios. We have provided an overview of some interesting aspects of secret sharing techniques and several examples of possible applications.

Table of contents

1 Introduction

1.1 Problem description

(1.1.1) The increase in the amount of data, both created and stored electronically, entails the necessity to construct various data storage systems. In the view of different requested storage periods, we divide systems into:

- short-term – storage period not longer than 3 years,
- medium-term – storage period between 3 and 10 years,
- long-term – storage period longer than 10 years, but with a specified end-date,
- unlimited – storage period longer than 10 years with no specified end-date.

(1.1.2) The unlimited storage is sometimes called “eternal”. In this case we have to pay special attention to the integrity of the stored digital content. For this reason, various digital marking techniques are used, so that even after a long time one should be able to verify the integrity of stored data.

(1.1.3) The main objective is to use advanced mathematical methods, especially cryptographic techniques applied in the process of digital marking of the content. These techniques ought to guarantee verification and integrity of the long-term-stored digital content.

(1.1.4) Proposed methods should take into account mainly:

- different kinds (classes) of stored content, e.g. cultural heritage, court documentation, accounting documentation etc.,
- limitations of database size,
- anticipated frequency of access to stored resources.

(1.1.5) Another very important aspect of the problem consists of finding the limits on applications of advanced mathematical methods, especially those based on cryptographic techniques and checking their applicability in the evaluation of data losses (e.g. due to the "corrosion" of media) as well as in a potential data recovery. Original data is marked as data in time t_0, while data that might be corrupted (because of “corrosion”) as data in time t_1.

(1.1.6) Special attention should be paid to:

- Systems and schemes of coding, which allow for a detection and correction of write errors
- Cryptographic techniques, such as:
  - ♦ public-key and asymmetric encryption,
  - ♦ secret sharing methods,
  - ♦ secure multiparty computations.

1.2 Problem breakdown

(1.2.1) A few assumptions and constrains have been proposed by the PWPW Representative when discussing the problem:

- T - amount of stored data
- R - amount of additional disk space we can use, in order to provide proof of data correctness we assume that $R \approx 0.1T$
- r - bit error rate (BER), we assume that r is about 0.01, i.e. at time of testing integrity of data, 1% of all bits is corrupted.

- g - accuracy of given proof of correctness, we assume that g is about 0.01, that is our proof should show that at least $\frac{T - Tr - Tg}{T}$ of data is correct.
- t_0 - time of storing the original data
- t_1 - time of testing the integrity of data

(1.2.2) The problem lies in finding such a method that can be used to determine at time t_1, with given accuracy, the ratio of the correct data, stored at time t_0 to the all data available. Furthermore, it is expected that the method allows assessing whether data is false.

(1.2.3) It would be useful if the method proved that data is not corrupted above a certain threshold value of BER .

(1.2.4) During the talk with the PWPW Representative we made the following assumptions and remarks:

- Errors cannot be avoided. A carrier which stores our data is imperfect, so we can be sure that there will be errors in data over long time horizon.
- The bit error rate, amounting up to 1% of data, is very high. For example, let us assume that we have a book in which every single letter is coded with 8 bits. Due to the error ratio, we anticipate about 1 error in every sequence of 100 bits, so in every sequence of 12 letters we shall expect a wrong letter. Therefore, in this paper we would like to present some solutions in which our initial assumptions were less constrained and this ratio is assumed to be smaller.
- Stored data is organized in files and we know the type of every file, like document, video, audio, archive files. Nevertheless, we can treat data as a sequence of bits (raw data approach).

(1.2.5) The PWPW Representative presented an idea of using hash functions so as to provide a proof of correctness of particular parts of data. Our work shows that the use of hash function only is not sufficient to complete our task, so hash functions with additional methods of correcting errors have been considered. We have also taken into account another way of dealing with hashes. It is based on the idea of computing a number of hashes from different divisions of data into blocks (a.k.a. hypercube method).

(1.2.6) We do not have any information about physical properties of data carriers, so we added an assumption of uniformly distribution of errors. If further details about distribution are available, our methods can be calibrated to deal with it without any loss of usability.

(1.2.7) Having dealt with hash functions, we focused on a solution based on secret sharing method. Due to time limitations, however, it could not have been completed during the workshop. Therefore, we have presented a helicopter view of the functionalities provided by secret sharing schemes.

2 Hash functions

The starting point for our research was a method based on hash functions. In this chapter we will show restrictions of using hash functions and describe the main ideas of extension of such approach.

2.1 Basics

The terms and notation used in this section comes, unless stated otherwise, from *Handbook of Applied Cryptography* ([1]).

(2.1.1) The hash function is a well known cryptographic tool, widely applied in providing data integrity check. The idea behind its use in the discussed problem is very simple – we compute the value of the hash function for given data twice: at the beginning and at the end of the storage process. If data is changed (loss of integrity), then these two hash values would most likely differ, otherwise both values remain the same.

(2.1.2) In a more formal way, we can say that hash function h is a function from $\{0,1\}^k$ to $\{0,1\}^l$, which has the following properties[1]:

- A minor change of the input string alters the output in at least $l/2$ bits.
- Probability of finding a bit-stream of the same hash value as another given bit-stream is negligible[2].
- Probability of finding a bit-stream of a given hash value is negligible.

(2.1.3) In order to introduce the notation used in further sections, we shall describe the integrity check process in a more formal way:

- We describe data by s and hash value by v, where $v = h(s)$ at the beginning of storing period (time t_0).
- After storage (time t_1) data may differ a little (e.g. due to the corrosion), so we describe it by s' and the corresponding hash value by $v' = h(s')$.
- If $v = v'$ then the data is correct with probability almost 1, otherwise we conclude that data is corrupted. Unfortunately, we do not know the percentage of corrupted bits – even if only one bit changes, the whole block is corrupted.

2.2 Restrictions on using hash functions

(2.2.1) The use of hash functions in a way presented above has some limitations. We shall present them with the following algorithm: let us assume that we have divided data into k blocks of the same length. For every block we compute the hash value at the beginning and at the end of the storage period. Let us say that we have detected l corrupted blocks (the appropriate hash values differ). Then the ratio l/k describes the upper bound of the bit-corruption ratio.

(2.2.2) More formally, we can describe the procedure above in this way:

- We divide a sequence of bits s into blocks $a_1, a_2, \ldots, a_k$ of the same length.
- For $i = 1,2,...,k$ we calculate and save the value $v_i = h(a_i)$.
- Next, we compare it with the value $v'_i = h(a'_i)$, that is with the hash function value computed on the block after storage period. If $v_i = v'_i$, we know with probability almost 1, that there were no corrupted bits in the part a_i. In other cases we have to assume that every bit might be corrupted.

---

[1] Where $l < k$. Secure hash functions should have the length of output l >= 256 bits.

[2] That is expected time needed to obtain two bit-streams with the same has value is exponential to the length of output of considered function.

- We compute $l = |\{i \in \{1,2,..k\} : v_i \neq v'_i\}|$ and denote the value l/k as b.

(2.2.3) In order to determine the usefulness of hash functions we calculate the expected value of the ratio l/k under constraints given in (1.1.2). Since bits are corrupted independently (as we have assumed above), we have the following probability that *i*-th bit is not corrupted:

$$P(v_i = v'_i) = (1-r)^{|a_i|}. \quad (1)$$

The expected value of the l/k reads:

$$E(b) = \frac{1}{k}\sum_{i=1}^{k}(1 - P(v_i = v'_i)) = 1-(1-r)^{|a_i|}. \quad (2)$$

As mentioned in (2.1.2), the length of hash function output is about 256 bits. Since the additional space for hash codes is $R = 0.1T$, the length of a_i for $i = 1,2,...,k$ should be at least 2560 bits long. Therefore, in this case the expected value of bit error rate is:

$$E(b) = 1-(0.99)^{2560} = 1-6.7\cdot 10^{-12} \approx 1, \quad (3)$$

which is unacceptably high.

(2.2.4) In conclusion, dividing data into disjoint blocks and computing a hash value for each of them to check the integrity of data is not particularly useful when assuming the constraints given in (1.2.1). Such constrains require blocks to be quite big, which makes the probability of block corruption equal almost 1. However, removing some of constrains and using smaller output blocks (e.g. 100 bits long) results in lower performance of the hash function.

2.3 Alternative division method

Introduction

(2.3.1) In paragraph 2.2, we have discussed the scheme of dividing data into disjoint blocks of equal length. Naturally, this is not the only possible approach to the given problem: a single bit needs not to be only in one block and blocks may have different length. It transpires that dividing data into blocks in a clever way leads to a better estimation of a corruption ratio, so that the upper bound for the ratio is closer to real value of ratio.

(2.3.2) Our first step to construct such a clever division was to arrange bits in a square (as in the **Figure 1**. In this case the blocks for which we calculate hash codes are composed as rows and columns in a square. Therefore, each bit is included in 2 blocks (1 row and 1 column, cf. **Figure 1**).

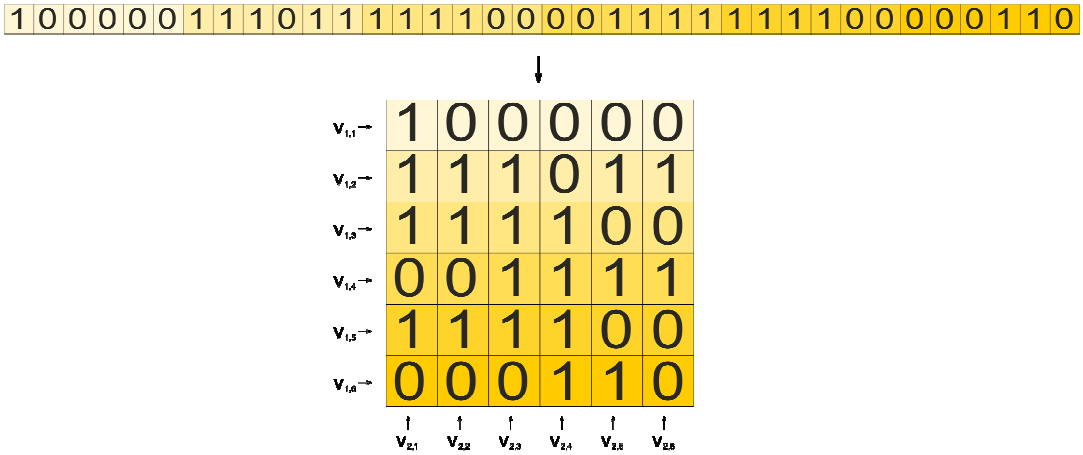

Figure 1 Example of arranging data in a square, 12 hash values are calculated and saved – 6 for rows ($v_{1,1}, v_{1,2}, \ldots, v_{1,6}$) and 6 for columns ($v_{2,1}, v_{2,2}, \ldots, v_{2,6}$).

(2.3.3) At first glance, it seems to make no sense to divide data into overlapping blocks – blocks would need to be longer so as to satisfy the requirement $R = 0.1T$, which increases the probability of corrupting a hash value. Nevertheless, such an approach may provide a very good upper limit of errors in the data if certain assumptions are fulfilled. Consider the situation presented in the **Figure 2**.

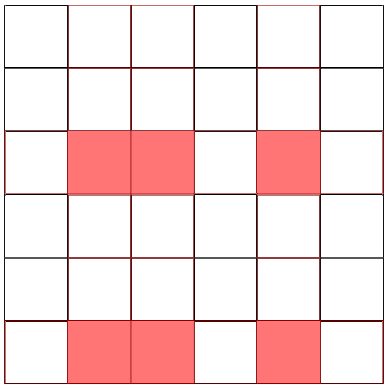

Figure 2 Visualization of the given data after storage. Blocks, for which hash value has changed, are highlighted. As every corrupted bit changes the hash value either for row and column, all potentially corrupted bits are located on the intersections of the highlighted rows and columns.

Let p_1, p_2 be the percentages of corrupted hash values in respectively rows and columns. The percentage of corrupted hash values for all blocks equals

$$p_h = \frac{p_1 + p_2}{2} \tag{4}$$

However, the upper bound of corrupted bits is generally smaller and reads:

$$b = p_1 \cdot p_2 . \tag{5}$$

In the case presented in the Figure 2 $p_h = 0.417, b = 0.167, r \geq 0.083$.

Generalization – the hypercube method

(2.3.4) A generalization of the method presented in (2.3.2) can be obtained by arranging data in a d-dimensional hypercube (cf. **Figure 3** for 3-dimensional case).

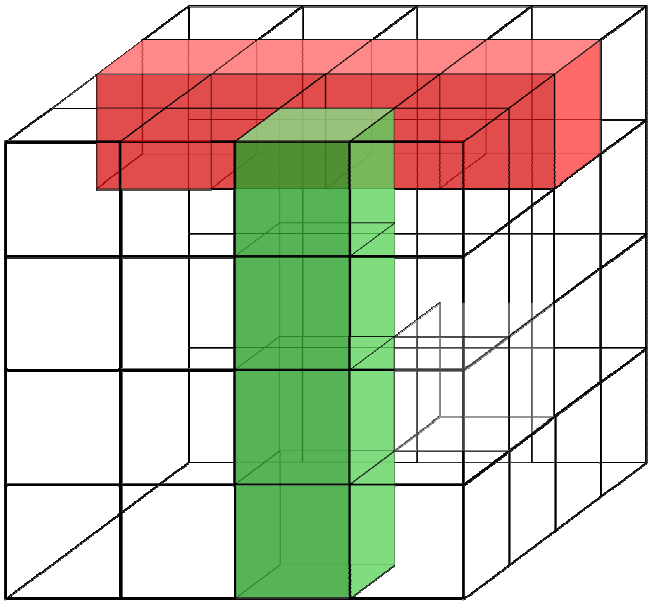

Figure 3 3-dimensional hypercube consisting of 64 cells (grey). Every cell represents 1 bit of data. Block is a subset of bits forming a section (red, green). In this case there are $3 \cdot 16 = 48$ blocks.

In this general case, if the percentages of corrupted hash values in dimensions $1,2,...,d$ equal $p_1, p_2, ..., p_d$ respectively, then we have the following upper bound of bit error rate:

$$b = \sqrt[d-1]{p_1 \cdot p_2 \cdot ... \cdot p_d} \,. \tag{6}$$

Calculations and optimization for hypercube method

(2.3.5) It was our aim to find such parameters of the hypercube that the obtained upper bound of corrupted bits is the best (the lowest). At the same time, we optimized the number of dimensions d and the size (number of cells) S of the hypercube (detailed calculations are presented in the Appendix). We decided to divide the data into parts, each consisting of S bits and make a hypercube for each of them separately.

(2.3.6) For the purpose of calculating the expected value of b, we assumed that the distribution of errors in the data is uniform and bits are corrupted.

(2.3.7) We calculated that this method does not work for initial constraints: $r = 0.01, R = 0.1T$ and 256-bit-long hash codes. The reason for this discovery is analogous to the one described in Section 2.2: for large blocks the probability of corrupting its hash value is very high. Therefore, we decided to change some of our assumptions. Firstly, we chose $r = 0.0001$. Secondly, we decided to use 100-bit-long hash codes.

Results

(2.3.8) For the assumptions made in 2.3.7, the optimal dimension of a hypercube is equal to 2 and each part consists of $4 \cdot 10^6$ bits (see Appendix). In this method the expected value of the upper bound of corrupted bits is smaller than 0.058. We would like to emphasize that dividing data into disjoint blocks of bits (method described in 2.2.1) under the same assumptions would give the expected upper bound of corrupted bits equal to 0.095. The hypercube method lowers the upper bound of errors more than 1.6 times.

Remarks & further research

(2.3.9) As indicated in (2.3.6), we assumed uniform distribution of errors in data. We suspect that this method may work very badly for a specific distribution of errors. However, we believe that this problem is manageable.

(2.3.10) The method is based on arranging data in a hypercube and calculating hash values for sections. Generally, there might be other acceptable ways of dividing data into blocks, giving lower expected value of b under the same assumptions. Firstly, we can arrange data in a hypercube and calculate hash values for other subsets of bits, for example hyper-planes. Secondly, we can abandon the idea of the hypercube and invent a completely different division.

(2.3.11) This method does not give a satisfying upper bound of corrupted bits for $r = 0.01$, which makes it useless in some real-world applications. On the other hand, if we had a method of measuring the level of the integrity of the data based on dividing data into blocks, we might improve the expected value of b by dividing data e.g. in a way presented in 2.3.4. We recommend it as a supporting tool.

2.4 Hash codes with error correction

Introduction

(2.4.1) The major limitation of the hash functions in solving the problem is a very high probability of hash value corruption for long blocks. As mentioned before, the analysis of the hash values enables us only to say whether there are any corrupted bits in a block. To calculate the upper bound, we have to assume that all bits from the block marked as corrupted may have changed. If we recognized which blocks are corrupted “only a bit” and which are more corrupted, then we would be able to measure the level of the integrity of data more precisely.

(2.4.2) For the purpose of such recognition, we decided to use error correcting codes. Generally, error correcting codes are bits added to original data (or part of data), with the aim of correcting a predetermined number of errors. In our problem, we use them in the following way (example in the **Figure 4**):

1. We divide data into blocks and calculate hash values for each of them.
2. We add codes correcting up to d errors to each block.
3. After storage we correct errors using error correcting codes.
4. After correction we compare saved and new hash values. If there are more than d errors in a particular block, then stored and calculated hash values differ. Otherwise, they will remain the same. Based on the information of how many hash values are changed, we can calculate the upper limit of corrupted bits.

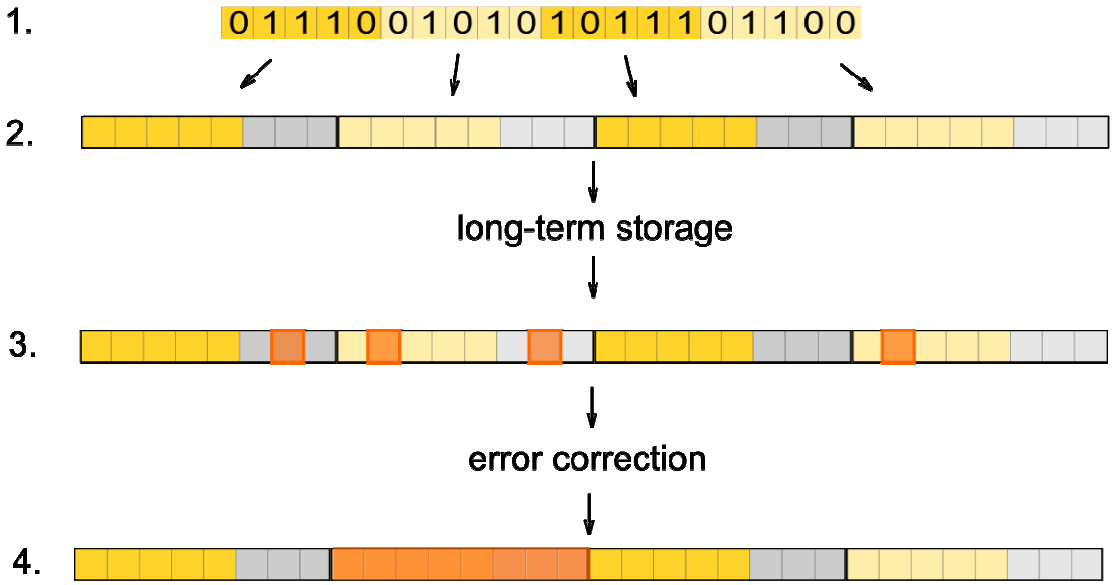

Figure 4 Exemplary visualisation of the proposed method. There are 4 blocks of 5 bits each. We add codes correcting 1 error (grey). Some errors occur in data after storage (3.) (red). We use codes to correct them. Finally, we calculate hash values again. If a number of errors in a block was bigger than 1 (like in a second block), then not all errors were corrected and the hash value is changed.

(2.4.3) In order to grasp the significance of this method, consider a situation presented in the **Figure 4**. After the storage there are 4 errors: 2 in original data and 2 in the added bits. If we used method presented in 2.2, two blocks would be corrupted. Since we use error correcting codes, we can recognize that in the fourth block only 1 bit has changed. Moreover, we can correct this error, which is an added value of the method.

Furthermore, the errors which occur in added bits are also corrected. It means that we do not need to deal with them additionally.

Theory

(2.4.4) During our research we focused only on the BCH error correcting codes. We would like to quote the theorem, which enabled us to make some calculations.

We will use the following terms:

- word – sequence of bits;
- coded word – a word which we would like to correct;
- control symbols – additional symbols (bits) used to correct errors in a coded word;
- coding word – a coded word with control symbols.

(2.4.5) Below we shall present the theorem of the BCH codes (proof in [2]):

For each $d,m \in Z_+$, $d < \frac{2^m - 2}{m}$ there exists such a BCH code that all following statements are true:

- Coding words are $2^m - 1$ long.
- This code corrects d errors in a coding word.
- The number of control symbols is $d \cdot m$.

This means that the length of a coded word is $2^m - dm - 1$.

(2.4.6) One of the most important conclusions of this theorem is that we do not need a lot of additional space for control symbols. If the size of a block is A, we need approximately $d \cdot \log A$ of additional space to correct d errors.

Results

(2.4.7) We were interested whether we overcame the major limitation of using hash functions, so we calculated the probability of corrupting hash value. It transpired that using error correcting codes combined with hash functions would give satisfying upper bound of corrupted bits for $r \approx 0.5$.

We made calculations for different values of r and R and tried to choose the best parameters m, d for them. We assumed that we know the BCH code correcting d errors in the $2^m - 1$ bits long coding word. We decided to use 100-bit-long hash codes. Results are presented in the **Table 1**.

Table 1. Probability of corrupting hash code depending on the values of m, d, r, R.

m	d	r	R	Probability of corrupting hash code
16	600	1%	$0.185T$	< 0.1%
16	357	0.5%	$0.1T$	< 0.01%
17	678	0.53%	$0.1T$	< 0.7%
16	357	0.53%	$0.1T$	< 0.4%
15	187	0.53%	$0.1T$	< 0.5%
14	96	0.53%	$0.1T$	< 1.6%

Remarks & further research

(2.4.8) For the purpose of calculations we assumed that the distribution of errors in the data is uniform (like in the paragraph 2.3). Once more information of error distribution is available (e.g. the specific storage hardware is selected), obtained results can be adapted accordingly, possibly with improved performance.

(2.4.9) The main advantage of this method is that it not only measures the level of the integrity of stored data, but also improves it. It can also be combined with error codes that are already used by PWPW with the aim of enhancing performance.

(2.4.10) We would like to emphasize that the theorem (2.4.5) guarantees only the existence of the BCH code satisfying some requirements. We do not know whether effective algorithms of constructing such a code or coding and decoding words exist. Moreover, it cannot be ruled out that there are some other error correcting codes which might be more useful in a real world application. This area is open for further study.

3 Secure Secret Sharing Method

Having investigated hash functions, we shall now check a different approach. Apart from ordinary verification of the integrity of long-term-stored digital content, it might provide some additional features, namely:

- extended capabilities in: verification of the integrity, recovery of corrupted bits, design of the access structure to the stored digital content;
- an opportunity to optimize the PWPW's requirements concerning storage.

All the features listed above and many others can be provided by secret sharing protocols. In cryptography, Secure Secret Sharing (SSS) scheme [4] is understood as a method of the distribution of a secret among a group of participants, all of them having their own share in the secret. The secret can be reconstructed only when authorized participants combine their shares.

3.1 Basic capabilities

(3.1.1) By using Secret Sharing Schemes one can store data distributed in some insecure locations in a secure[3] way ([3]).

(3.1.2) Threshold secret sharing. A threshold is a minimal number of participants which have to co-operate to reconstruct the secret. A scheme, where at least t out of n players is necessary to reveal the secret is described as a (t,n) threshold scheme. It allows placing securely $t-1$ shares outside secure locations (e.g. own trusted systems), say, literally distribute $t-1$ shares over the Internet.

(3.1.3) Schemes for which we can provide verification of the integrity of secrets are called Verifiable Secret Sharing (VSS).

(3.1.4) A proper design of the access structure improves the functionality of secret sharing.

- One of the simplest access structures was presented above – every set of at least t out of n participants is allowed to reconstruct the secret.
- More advanced structures can be implemented as follows:
  - $P=\{P_1,...,P_n\}$ is a set of participants taking part in sharing.
  - Every family R of subsets of P can be an access structure.
- We can provide different levels of access for different participants. For example, the main participant (PWPW) has more rights than a trusted outsider (e.g. governmental institutions), which in turn has more rights than a not trusted participant (e.g. ones using shares from the Internet).

Example 1 (*generalised access structure*) Our task is to guarantee verification of the integrity of long-term-stored digital content. For example, let us consider recordings of speeches of famous politicians. One can distribute a secret among some governmental institutions and set the condition under which the secret can be revealed, e.g. at least 5 institutions from 5 different ministries have to collaborate in order to reconstruct the secret and so on.

By using additional participants with different levels of privileges we can minimise the probability of leaking or losing the data.

[3] In the secret sharing, there are at least two notions of security: information-theoretical and computational security. There are significant differences between the two types, yet, it is rather beyond the scope of this paper. In order to simplify further discussion without losing its generality, we will simply discuss secure or perfectly secure secret sharing schemes.

3.2 Extended capabilities

(3.2.1) Now extended capabilities of secret sharing schemes shall be presented.

(3.2.2) Pre-positioned secret sharing. A pre-positioned secret sharing is an example of an access structure where all data requested to reconstruct the secret is known except for a single crucial share which has to be given later. For example, the PWPW can distribute the whole data over the Internet by a pre-positioned secret sharing scheme with a short, crucial share kept locally. Let us explore a difference between secret sharing and simple encryption of data in this model. The advantages will be clear once more extended capabilities are outlined.

Example 2 (*scheme with an activating share*) Let us assume that we have a situation described in the Example 1 – data is stored locally on the servers of PWPW and in a few places all over the world – in the United States, China, Russia etc. By means of a pre-positioned scheme foreign institutions can partake in given shares (a share made out of a share is called a subshare) beyond unauthorised participants who cannot reconstruct institutional shares until foreign and trusted parties cooperate, because their shares are crucial.

(3.2.3) Proactive Secret Sharing (PSS) has the following features:

- One can change (periodically renew) participants' shares in a secret without revealing or changing it.
- One can recover corrupted shares (these shares correspond to dishonest participants). In our case – we can periodically check the consistency of shares and recover corrupted ones. Another reason to use pro-active secret sharing is the fact that if we find a corrupted share during the verification process (by e.g. VSS scheme), we can easily replace a broken share with a correct one. So there is a simple way to “maintain” integrity of shares periodically, which implies integrity of data.

(3.2.4) Multi-secret shares have the following features:

- A scheme where any subset of set of participants shares another secret is available.
- It seems that a single share can be used in a few secrets, optimizing storage space.

Example 3 (*multi-secret scheme*) In the presented case we can use a multi-secret scheme. We do not need to create a separate shares and secrets for all files. We can make just a single sharing scheme with such a property that different subsets of foreign institutions can reconstruct speeches of different politicians and all speeches reconstructed in this way make up a collection.

3.3 Combining properties

(3.3.1) One of the most desired properties of secret sharing schemes is its flexibility in combining functionalities described above. Further research is required to describe which properties can be combined with each other.

(3.3.2) In our case – we might conduct research aimed at developing a scheme which is e.g.

- perfectly secure, pro-active, integrity-providing and activated by a share from the PWPW.
- a multi-secret scheme where any subset of participants has its own secret that cannot be revealed without the share from the PWPW.

(3.3.3) A scheme with combined properties is not necessarily a textbook material available right away, but has to be carefully engineered instead. Hence, some additional work might be required before implementation.

3.4 Additional Considerations

(3.4.1) Reconstructing original data from shares might occasionally need some computational effort and data may not always be available in real-time. Still, task complexity is polynomial in time, yet, usually feasible in practice.

(3.4.2) It seems that once a perfectly secure secret sharing scheme is applied its users should be protected against future developments in cryptoanalysis, which would affect the cryptography based on computational complexity (e.g. most of the employed public-key cryptosystems like RSA).

3.5 Open questions

(3.5.1) As described above, the secret sharing schemes provide many tools to deal with the PWPW problem. Still, there are some open questions that definitely need further investigation:

- Which verification techniques are optimal in solving the problem of corrupted shares and data in the case of digital content of the PWPW interest? One should remember that by using secret sharing scheme, we can provide some verification based on secure multi-party computations.
- Further effort should be expended to design an optimal access structure for particular types of stored files.
- Various types of files have different data that is crucial to their consistency. It seems that we do not necessarily need to protect the whole data, but only some crucial parts. It is worth considering which fragments are really important for each type of files. If we made this classification, we would be able to protect crucial parts only by means of secret sharing.

4 Conclusion and proposals for further research

4.1 Hash functions

(4.1.1) Let us recall the method proposed in section 2.2. We divided data into disjoint blocks of the same length and compared two hash values for each of them – the first value was computed before the storage, the second – after it. This method was not considered as a good way of dealing with the given problem. The probability that in any block of data for which we compute the value of hash function will remain the same after some time is negligible.

(4.1.2) With some additional assumptions, like limited data size and smaller bit error rate, we have shown that dividing data into blocks in a clever way may improve the estimation of corrupted bits ratio. However, due to its limitations, this method should be applied only as a supporting tool.

(4.1.3) Hash functions combined with error correction methods may provide very good error estimation. Under given constraints concerning the bit error rate and the maximal amount of additional data, there is a probability of 0.0001 that not every error in a single code word will be corrected.

(4.1.4) One of the most convenient cases, for which we can prove that the upper bound of the number of corrupted bits is small, is when errors are uniformly distributed.

Nevertheless, we believe that errors occur rather in blocks, in particular parts of carrier of data etc., but not uniformly. The question is – can we reorganize the bits to make the distribution of errors uniform?

4.2 Secret sharing method

(4.2.1) Secret sharing method is an alternative way of thinking about data storing. It provides a number of new functionalities which allow storing of data divided among some local (trusted) participants and some untrusted parties (like public FTP servers or in general 'the Internet') in a secure way.

(4.2.2) Different participants taking part in data sharing can enjoy a different level of privileges in data access and recovery. It is important to determine how many levels of privileges should be designed and how many participants should be on each level. It seems that almost any access structure can be implemented by using the secret sharing.

(4.2.3) In many secret sharing methods we assume that shares stored locally (trusted participants) are at least of the size of secret. It is worth investigating whether it is possible to deliver a method which would be both: secure and space-saving (i.e. local shares are smaller than a secret). We believe that such schemes can be obtained.

(4.2.4) An important property of secret sharing schemes is verifiability of shares. It is especially crucial in our problem, in which we deal with corrupted data, as verification protocols can play a role of correcting codes. It is worth exploring which of them would be optimal in our problem.

(4.2.5) Different secret sharing schemes have various properties. We have described properties of schemes which are: pro-active (we can periodically change participants' shares), pre-positioned (there is a crucial share without which a secret cannot be revealed), multi-secret (a few different secrets are shared) or verifiable (we can determine which shares were corrupted and reveal a secret without them). The question is: which of the mentioned properties can be combined?

4.3 Other possibilities

(4.3.1) In this section, we will outline an additional approach, which was discussed after the 77^{th} ESGI, nevertheless it is worth further research. There are check-digit schemes that allow determining whether a bit-stream was corrupted over a certain threshold, say, 1% of bits were changed. Should this be a case, the check-digit scheme provides information that corruption has occurred. Usually the threshold can be set individually for a particular application. Furthermore, since the main task of the scheme is error detection not error correction, usually less additional information is stored (shorter checksum) than in error correction codes. In general, the length of the checksum can even decreased further, should statistical reasoning be introduced, for instance it is allowed that in a small number of cases scheme sensitivity is different from the set threshold (not necessarily lower). In such a situation, it is even possible to decrease the ration of checksum's size to the size of information stored with the increasing volume of information. A good example of such construction is graph coloring based on the check-digit scheme described in [5]. It is recommended to research applications of check-digit schemes with the characteristics outlined above for the purpose of the problem presented by PWPW and to revaluate results already obtained for hash functions as well as to investigate a joined use of check-digit schemes with secret sharing methods.

Bibliography

[1] Alfred J. Menzes, Paul van Oorschot, Scott A. Vanstone, *Handbook of Applied Cryptography*, http://www.cacr.math.uwaterloo.ca/hac/ (link active: 2010/10/17)

[2] Witold Lipski, Wiktor Marek. *Analiza kombinatoryczna*. Biblioteka Matematyczna PWN, Warszawa 1986.

[3] Ronald Cramer, Ivan Damgård, Jesper Buus Nielsen, Multiparty Computation, an Introduction, Contemporary Cryptology (Catalano/ Cramer/ Damgaard/ DiCrescenzo/ Pointcheval/ Takagi), *Advanced Courses in Mathematics CRM Barcelona*, Birkhauser, 2005.

[4] Adi Shamir, *How to share a secret*, Communications of the ACM 22 (11), pages 612–613, 1979

[5] Kamil Kulesza, Zbigniew Kotulski, *On a Check-Digit Method Based On Graph Coloring*, Proceeding of IEEE International Conference on "Computer as a Tool", EUROCON 2007, Warsaw, September 9-12, pp. 214 - 217, IEEE eXpolre.

5 Appendix

5.1 Hypercube model

Introduction

(5.1.1) In this paragraph we will present detailed calculations for the specific clever division based on a hypercube (the respective idea and visualization are presented in paragraph 2.3).

Specification

(5.1.2) We will use the following notation: the dimension of hypercube H is d. Every bit (cell) has d coordinates $(x_1, x_2, x_3, ..., x_d)$. The size of data $S = 10^t$, so the side length of hypercube is $10^{\frac{t}{d}}$.

(5.1.3) A *section* in H is a set of $10^{\frac{t}{d}}$ cells such that $(d-1)$ of their coordinates are the same. Sections are parallel to axes. Section parallel to i-th axis and meeting point $(\bar{x}_1, \bar{x}_2, ..., \bar{x}_d)$ is defined below:

$$S_i(\bar{x}_1, \bar{x}_2, ..., \bar{x}_d) = \left\{ \begin{array}{l} (x_1, x_2, ..., x_d) \in H : (x_1 = \bar{x}_1) \wedge (x_2 = \bar{x}_2) \wedge ... \\ ... \wedge (x_{i-1} = \bar{x}_{i-1}) \wedge (x_{i+1} = \bar{x}_{i+1}) \wedge ... \wedge (x_d = \bar{x}_d) \end{array} \right\} \quad (7)$$

Dependences

(5.1.4) Before stating anything about the hypercube method, we shall discuss the main dependences between different values describing the method, such as: additional space R to remember hash values, the size of data $S = 10^t$, the dimension of the hypercube d and so on.

(5.1.5) Firstly, we will calculate how much of additional space is necessary to remember the hash values. Every cell is in d sections and there are 10^t cells. Every section consists of $10^{t/d}$ cells. As a result, the total amount of sections in H is:

$$\frac{d \cdot 10^t}{10^{\frac{t}{d}}} = d \cdot 10^{t \cdot \frac{d-1}{d}} . \tag{8}$$

Since we use 100 bits long hash values, we need additional space equal to:

$$R = d \cdot 10^{t \cdot \frac{d-1}{d}+2}$$

(5.1.6) It is also necessary to know the maximal amount of errors in data, if there is a given amount of wrong hash codes. Suppose there are k errors in hash codes. Let $(k_1, k_2, ..., k_d)$ be the number of wrong hash codes in the first, second, … and d-th direction respectively. The following upper bound of errors in the data would be:

$$b \leq \frac{d-1\sqrt{k_1 \cdot k_2 \cdot ... \cdot k_d}}{10^t} . \tag{9}$$

Moreover, it is easy to show that:

$$k_1 \cdot k_2 \cdot ... \cdot k_d \leq \left(\frac{k}{d}\right)^d . \tag{10}$$

It means that if there are k wrong hash codes:

$$b \leq \frac{\left(\frac{k}{d}\right)^{\frac{d}{d-1}}}{10^t} . \tag{11}$$

The last thing we need is the maximal amount of wrong hash codes, if there are l corrupted bits. Every bit may corrupt d hash codes, so the maximal amount of corrupted hash codes is

$$l \cdot d . \tag{12}$$

Optimization

(5.1.7) Suppose that after storage there are 10^{t-4} errors in the data (in other words: $r = 0.0001$). Due to (12), we know that these bits are corrupted at most $d \cdot 10^{t-4}$ hash codes. With the information that at most $d \cdot 10^{t-4}$ hash codes are wrong, we may calculate (from equation #11) that:

$$b \leq \frac{\left(\frac{d \cdot 10^{t-4}}{d}\right)^{\frac{d}{d-1}}}{10^t} . \tag{13}$$

We would like to know that not all of the bits are corrupted ($b \leq 1$), so t and d must satisfy the inequality:

$$\left(\frac{d\cdot 10^{t-4}}{d}\right)^{\frac{d}{d-1}} < 10^{t} \tag{14}$$

$$t < 4\mathrm{d}\ . \tag{15}$$

Obviously, the lower t, the more precise we might be.

(5.1.8) As $R = 0.1T = 10^{t-1}$, we can create additional inequality for t and d

$$d\cdot 10^{t\frac{d-1}{d}+2} \le 10^{t-1} \tag{16}$$

$$d(\log_{10} d + 3) \le t\ . \tag{17}$$

So t and d must satisfy:

$$d(\log_{10} d + 3) \le t < 4\mathrm{d}\ . \tag{18}$$

For $t < 40$ such d can be found. As indicated before, the lower t, the more precise we are. We decided to choose

$$t = d(\log_{10} d + 3)\ . \tag{19}$$

(5.1.9) We would like to find the value of d which would make our prediction more precise.

The first step was to find d, such that interval $(d(\log_{10} d + 3), 4d)$ is as big as possible. We defined function

$$g(d) = d(1 - \log_{10} d) \tag{20}$$

and found its maximum, which is approx. 3.7. Then we checked values of d such as $2,3,4,5$ and calculated that prediction is the most precise when $d = 2$.

(5.1.10) We decided to prove that prediction is the most precise when $d = 2$

- Case $d = 2$

$$t = 2(\log_{10} 2 + 3) \approx 6.6 \tag{21}$$

$(10^{6.6-4})^{\frac{2}{2-1}} = 10^{5.2}$ – the possible number of errors.

$\frac{10^{5.2}}{10^{6.6}} \approx 0.0398 < 4\%$ - the maximal ratio of corrupted data.

- Other cases ($d \neq 2$).
  We calculated that the possible percentage of corrupted data is equal to

$$10^{\frac{t-4\mathrm{d}}{d-1}}\ . \tag{22}$$

We would like it to be as small as possible. We calculated that for $d > 1$ this function increases, so the optimum is $d = 2$.

(5.1.11) We would like to emphasize, that $b < 0.04$ only if $r = 0.0001$. We calculated (using computer), that if $E(r) = 0.0001$ and bits corrupt independently with probability 0.0001, then $E(b) < 0.058$

(5.1.12) Note that dividing data in exclusive blocks of bits would give a worse expected upper bound of corrupted bits, equal to 0.095.