Simple Resilient Bit Preservation

Brute force search for bit corrections

Dirk Roorda

2013-04-02

Abstract
In order assure the integrity of digital assets over time, a digital repository needs a fixity service that detects and corrects bit errors in its holdings. This paper describes a resilient way to overcome bit errrors, even if all accessible versions of a datasource are corrupt including checksums. The method is practical in the sense that it can be implemented without special tools. The effort needed per faulty bit is very high compared to correction methods used in data transmission, but the data redundancy needed is very little. In other words: this method is geared to long-term storage with a low occurrence of errors, which is a situation that repositories typically have to deal with.

1 Software

Before I explain the idea itself, I would like to point you to a github repository called **bit-recover** where the code that implements this idea is stored. I have experimented extensively with artifically bit-rotted files and various algorithms to recover from the errors. The results of those experiments can also be found in that project. Here is the link to the documentation, and from there you can find the github repository itself: http://bit-recover.readthedocs.org .

2 Reliability of mass data storage devices

It is difficult to assess how reliable current mass storage devices are. Several measures are around, such as Mean-Time-To-Failure (MTTF), and they have been questioned subsequently [Schroeder 2006] and convenient measures such as Mean-Time-To_Dataloss have been proposed [Gray 2005]. It is clear that errors occur, and that most of them are corrected by low-level redundancy such as RAID, and various error correction codes at a low level of communication, such as Reed-Solomon for CDs and DVDs, and Low-Density Parity Checks for information transfer through noisy channels.
These measurements and techniques are all located in the processing of data, the events that data is read or written.
However, digital repositories usually preserve quite a lot of data that is not accessed for many years. They have the obligation to guarantee the bit integrity of all their resources over many years, regardless of the frequency of access. Hence there is a felt need to independently assess and maintain the bit constancy of data resources. It is particularly important that methods to do so run efficiently and quietly without much overhead in terms of processing and storage. After all, bit errors are expected to be extremely rare, and too much cost would not be justified by the improvement. Another important design criterion is ease of implementation and maintenance. Bit error checking and correcting has to go on for a long time, so the solution should facilitate easy evolution to other computing contexts. At the same time, the frequency of running such software is not high, so again, expensive software would not be justified by the gain.

In the following sections a method will be described that has these desired characteristics.

3 Resilient error correction

It is possible to craft a practical error correction method out of generally available building blocks. There are ingenious error correction methods such as Reed-Solomon and Low-Density-Parity-Check, but they have been optimised for encoding and decoding in real-time and this comes at an expense: they use quite a bit of redundancy. We are looking for much less redundancy at the expense of the amount of work needed to correct errors. This makes sense in a situation where terabytes of data need to be protected over years and where only a handful of bit errors are expected per TB per year.

Here is the basic trick right away: use checksums to detect errors first, and then perform brute force searches to find data blocks that match given checksums. On the face of it, this might look unnecessarily cumbersome but the beauty is in the details. Checksum algorithms, especially the cryptographically secure ones, are designed in such a way that it is infeasible to find a block of data that matches a given checksum. When we do our brute force, however, we have additional information: we assume that the block we are looking for differs from what we are given in just a few bits. Within those constraints a brute force search is perfectly possible.

3.1 Overview of the method

We will use this trick twice in our method, in slightly different ways.
Here is an outline of the complete method:

(i) Generate checksums.
As an initial step: add checksums for each block of data and store them separately. Use a small block size, e.g. 1KB. We will discuss the choice of the checksum algorithm below.

(ii) Verify checksums
Upon periodical checking: recompute the checksums and compare them to the stored checksums. For every mismatch, take corrective action.

(iii) Repair: error correction without using backup versions
Corrective action is performed in one or two steps: the first one without consulting any backup data, and if that does not succeed, there is a step involving backups, which may or may not succeed. The first step goes as follows:

(iii-1) window distortion of blocks
Generate a lot of slight distortions of the actual data and compute their checksums. The distortions will be local distortions, inside a window of a limited number of bits, where the window slides over the whole data block.

(iii-2) finding hits
Compute the bit-distance between the checksums of the distorted blocks with the given checksum. Whenever that distance is small, we have a hit. A hit consists of a (small) distortion of the data block and a (small) distortion of the checksum that satisfy the condition that the distorted checksum is actually the checksum of the distorted block.

(iii-3) Evaluate hits

If there is exactly one hit then the corresponding (distorted) block is a very good candidate for being the true version of the original block.
If there are multiple hits, then something peculiar has occurred: we have found very similar bitstreams that have very similar checksums. Checksums have been designed to have the property that if you change the data only a little bit, then its checksum will change significantly. A practical choice would be to choose the hit whose block version is least distant from the version to be corrected, but we are on shaky ground here. It is better to rely on the backup. Whether cases like these occur at all is ultimately dependent on the adequacy of the chosen checksum algorithm.
If there are no hits, then the corruption is outside the search space. We could increase the search effort, but if there is a backup, this is the moment to consult it.

(iv) Restore: error correction using a backup version
This step is appropriate after the attempt to repair, in order to remedy the cases that the repair-step could not solve.
Compare the backup version of the block with the current, damaged version of the block. Again, we perform a brute force search for hits of distorted blocks and checksums, but we search for a different kind of distortion.

(iv-1) mask distortion of blocks
Now create distortions of this block in the following way: keep the bits in which both versions agree to their agreed values. Vary the bits in which they differ exhaustively. So, if there are 5 bits where the versions differ, we generate 32 versions of the block.

(iv-3) finding hits
This step works exactly the same as in the previous step.

(iv-4) Evaluate hits
We have the same possible outcomes. However, if we find no hits, the consequence is more grave: we lack the information to retrieve the original version in any easy way. The only option is to broaden the search, either by working with more liberal parameters, or by undertaking altogether different kinds of searches. Even in this case, all is not lost. By close inspection of the checksums and blocks involved we may make guesses as to what happened to the block, and test those hypotheses.

3.2 Details of the method

As we said: there is beauty in the details. Here are the details.

3.2.1 The nature of the errors

We assume that the bit errors that we have to restore are random errors of limited size and frequency. The typical error is a distortion of original bits in a region of 1 to 8 bits, and we assume that such errors may occur with a frequency of 1 in an MB. Our method is capable of handling bigger frequencies, but the empirical frequencies associated to bit rot are much, much lower, say 1 in a TB per year.

3.2.2 The checksum algorithm

The choice of checksum algorithm is crucial to the success of our method. There are two impacts: one on the reliability of this data protection method and one on the efficiency of it.

(a) Security of the the checksum algorithm

The most important property of a checksum that we need is the absence of collisions. But we know that there must be many collisions out there, they are just hard to find by deliberate effort. If the bit-size of the checksums is small, collisions are more prevalent, to the point of rendering this whole method worthless. For example, we have tested an MD5 hash reduced to 16 bits. The repair stage of our method finds very many multiple hits, among which the true hit cannot be distinguished anymore.
On the other hand, using true MD5 did not give any multiple hits during our tests. There is also another concern here: is it possible, given a block and its checksum, is it possible to find a different block with the same checksum? If so, the checksum method in question is not resistant against deliberate modification of the data it protects. In recent years MD5 turned out to be not quite up to the task. [Stevens 2007] describes a way where one can extend two freely chosen blocks of data with exactly the same bit string and obtain identical MD5 checksums.
The question is: is this weakness exploitable in our scenario? Certainly not directly, because we work with fixed size blocks of data. So this particular exploit is not possible, but of course other, future ones might be.
There is also another answer to the weakness of MD5: do we really need a cryptographically secure hash algorithm when protecting the integrity of the data? We can encrypt our data by means of state-of-the-art cryptography and guard the integrity of the encrypted data by means of checksums that are not necessarily cryptographically secure. Any deliberate exploit of the cryptographic weakness will damage the data, but cannot change the unencrypted data to a given value. If we also have checksums for the unencrypted data, this damage will be detected upon decryption.
So an attacker needs to do a lot of work simply to cause the loss of one datablock.
As we shall see in the following subsection, the most secure hashing algorithm that we consider (SHA256) is also the most expensive one in terms of overall efficiency of the method described here. So the particulars of each use case will determine how the balance between efficiency and security will be struck.

(b) Trade off between space and time

The factors that are relevant to the overall performance of this method are the following parameters: the size of the data blocks, the bit size of the checksum method.
Checksums take up space, they are a form of redundancy. If we set the amount of redundancy we are willing to accept, then the data block size and the checksum bit size become related. Suppose we go for a redundancy of one checksum bit for 32 databits. That is roughly 3 percent. If we choose 32-bit checksums, our block size should be 32 * 32 = 1024 bits or 128 bytes. That is pretty small. On the other hand, if we choose 256-bit checksums, our block size should be 256 * 32 = 8192 bits or 1024 bytes, a kilobyte.

In both cases, the checksum size is sufficient to guard the integrity of the data in such a block size. In practice, checksums guard whole large files of multiple megabytes or even gigabytes. In our scenario it is the efficiency of the brute force search that is most susceptible to the choice of block size.
Our method searches for window distortions and mask distortions. The size of the search space is exponential in the size of the window/mask and quadratic in the length of the block.

So the block size is an important indicator of the efficiency of the repair process. The smaller the block size is, the easier it is to repair bit errors. But a smaller block size increases the redundancy for a given checksum method. The higher the

redundancy is, the greater the cost is to store and maintain the checksums themselves. If we try to keep block size and redundancy down at the same time, we need a checksum method that has few bits. But then the risk of collision increases, and collisions destroy the reliability of the method.

So the question is: is there is a sweet spot where no collisions occur in practice, and correcting is still efficient and redundancy is acceptable? The answer is yes, but before we present the results of the experiments, we shall give a few more details of the repair and restore methods.

3.2.3 Window distortion

The repair method uses window distortion. Here we describe window distortion in more detail and estimate the effort needed to find hits. We also indicate how we traverse the search space.
Suppose we verify blocks against given checksums and we find a mismatch

```
cccc !~ bbbbbbbbbbbbbbbbbbbbbb
```

Here the c-s stand for the bits in the checksums, the b-s for the bits in the block.

```
cccc ~? BbBBbbbbbbbbbbbbbbbbb
cccc ~? bBbBBbbbbbbbbbbbbbbbb
cccc ~? bbBbBBbbbbbbbbbbbbbbb
cccc ~? bbbBbBBbbbbbbbbbbbbbb
…
cccc ~~ bbbbbbbbbbbbbBbBBbbbbbb
```

Here we slide a distortion window of 4 bits over the block, it is a window consisting of the bits 1011. The window will be XORed with the piece of the block it is lined up against.
In order to estimate the search effort, we must know how many distortion windows of length n there are. In order to prevent applying the same distortions twice, we demand that every distortion window starts and ends with a 1. Then it is easy to see that distortions caused by windows of different length must be different. Conversely, every distortion corresponds to a window of which the first and last bit are a 1. It follows that the number of distortion windows of length n is 1 for $n = 1$ or 2, and 2^{n-2} for $n = 2, 3, \ldots$. So the number of frames of length up to and including n is exactly 2^{n-1}.
If the block length is m then we have to apply all these frames to nearly m positions. In fact, we have to subtract the frame length for each frame, but is m is large with respect to n, we only slightly overestimate if we say that there are $m * 2^{n-1}$ possibilities to deal with.
In order to check a possibility we have to compute its checksum and compute the bit-difference between the computed checksum and the given checksum. If the checksum size is small with respect to the size of the block, we may assume that this distance comparison takes constant time. Computation of the checksum is roughly linear in the length of the block m, say $C * m$ where C is a constant. Hence the overall effort is $C * m^2 * 2^{n-1}$.
When there is no hit for a block, we will have to do this many operations.
When there is a hit, we would like to find it as quickly as possible. Therefore we first look for all 1-bit frames, then for all 2-bit frames and so on, assuming that errors with fewer bits are more frequent than errors with more bits.

3.2.4 Mask distortion

The restore method uses mask distortion. Here we skectch the algorithm, estimate the effort needed, and indicate a non-standard but efficient way to traverse the search space.
Suppose we need to restore a block from back up, and the backed up block also has errors. Suppose in fact that everything is corrupt: both data blocks and the checksum itself.

```
cccc !~ bbbbbbbbbbbbbbbbbbbbbbb != bbbBbbbbBbbbbbBbbbbbBbb
```

Then we proceed as follows: define a mask on the data block that identifies the bits that are different between the two blocks. A simple bitwise XOR between the blocks yields such a mask.
Now try out all variations of the block where the bits outside the mask are kept constant, and the bits inside the mask take all possible values. If the size of the mask is n, then there are 2^n variants to try.
The testing of variants is analogous to the case of window distortion above. We gather hits, based on the bit-distance between the computed checksum and the given distance.
This strategy will miss cases where data and backup have an error at the exactly the same position. When a restore fails, a mixture of window distortion and mask distortion might succeed after all, but this will cost considerably more effort.
It is possible to optimize the search strategy if we assume that errors in the data take the shape of burst errors and that errors in data and backup are not related. In that case, it is most probable that only one block is damaged, so the mask corresponds to the bit errors in either the data block or the backup block. So we should try first the masks that consist of purely 1s and purely 0s. If both blocks are damaged, then there is a high probability that the leftmost bits of the mask correspond to errors in the one block, and the rightmost bits of the mask correspond to errors in the other block. So we should continue with trying the masks that consists of a sequence of 0s followed by a sequence of 0s, or vice versa.
Call bitstring *a* smoother than bitstring *b* if the number of bit changes in *a* is less than those of *b*. A bit change is an occurrence of the substring 01 or 10.
We can now formulate our search strategy for masks as follows: try smoother bitstrings first.
This strategy yields a huge reduction in search actions, provided you are content with a non-exhaustive search for hits. In my experiments there were many cases where the search space was too big for the allotted time, and yet the smooth search strategy yielded correct results in all cases.

4 Correctness

The correctness of this approach cannot theoretically be proven, simply because the method is not 100% correct. The basic reason for this is that we rely on the fact that some events that are possible will not happen. This is not a problem if the probability of such events is extremely low, but it prevents any proof of 100% correctness.
The main kind of event that should not happen is that if we change a data block in only a few bits, its checksum will remain the same or change in only a few bits. Can we estimate the probability that such a situation arises? We cannot, and so we have executed a number of experiments to see what we can expect in practice.
An other concern is deliberate tampering. How resistant is our method against second preimage attacks, i.e. is it feasible for an attacker to replace a block of data by another block with the same checksum?

Nowadays it is feasible for MD5, by means of a brute force search. It is also feasible to extend two given files by a few kilobytes of carefully chosen data in such a way that the resulting files have the same MD5 hash. The reduced versions of MD5 are even more prone to this fact.
The lesson is that the integrity check that we are building here cannot be trusted to be resistant against intentional tampering. Nevertheless, the known exploits of MD5 can not be applied to blocks with a fixed and rather small size. They rely on appending data to strings. Moreover, if the data in question is strongly encrypted, all an attacker can do by means of a second pre-image attack, is to substitute an encrypted data block by another with the same checksum. He has no control over the contents of the unencrypted data he is substituting. An attacker can only damage the data, but not tamper with them. And in order to do unseen damage, he has to take a lot of trouble: find second MD5 pre-images.
There is a way to overcome this: not using MD5 but SHA256 as checksum method. This will affect performance if we keep the redundancy overhead constant, because the blocks will be twice as large, so the brute force searches take four times more time, and even more, because computing the SHA256 takes more time than the MD5 checksum.

5 Testing

In order to determine the limits of our method, we have conducted a series of experiments. The basis of the experiments is a 1MB photo (jpeg). A variety of block-wise checksums of it was computed. The photo was damaged with hundreds of random burst errors. The same was done with a backup of the photo. The checksums were also subjected to damage. Then attempts to repair and restore the photo were undertaken with increasing amounts of efforts. The success rate and time needed for each setting was recorded.
The checksum method that we use is a parameter. We have tested our method using CRC32, MD4, MD5, SHA256 and 3 reduced versions of MD5, where we compute the MD5 checksum but retain only 16, 32 or 64 bits of it (the full MD5 checksum is 128 bits). When MD5 was defined it was still infeasible to find two blocks of data with the same checksum.

6 Feasibility

7 References

[Schroeder 2006] Schroeder, Bianca and Gibson, Garth A. Disk Failures in the Real World: what does an MTTF of 1,000,000 hours mean to you? Technical Report. Carnegie Mellon University. 2006.
http://www.pdl.cmu.edu/ftp/Failure/CMU-PDL-06-111.pdf

[Stevens 2007] Stevens, Marc; Lenstra, Arjen and De Weger, Benne. Vulnerability of software integrity and code signing applications to chosen-prefix collisions for MD5. Technical Report. Technical University Eindhoven. 2007.
http://www.win.tue.nl/hashclash/SoftIntCodeSign/

[Gray 2005] Gray, Jim and Van Ingen, Catharine. Empirical Measurements of Disk Failure Rates and Error Rates. Microsoft Research Technical Report MSR-TR-2005-166. 2005.

Data Integrity
http://www.fsl.cs.sunysb.edu/docs/integrity-storagess05/integrity.html

HMAC
http://en.wikipedia.org/wiki/Hash-based_message_authentication_code

S.M.A.R.T.
http://en.wikipedia.org/wiki/S.M.A.R.T

BitRot
Research by Google
http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/archive/disk_failures.pdf

Research by Microsoft
http://research.microsoft.com/pubs/64599/tr-2005-166.pdf

Personal experience on a blog
http://jonathan.pearce.name/mohtalim.old/viewtopic.php?f=1&t=3340
http://blog.tddium.com/2011/03/23/backups-in-the-cloud/

David Rosenthal
http://www.bl.uk/ipres2008/presentations_day2/43_Rosenthal.pdf
http://www.dlib.org/dlib/november05/rosenthal/11rosenthal.html

Carnegie Mellon
http://www.pdl.cmu.edu/ftp/Failure/CMU-PDL-06-111.pdf

ServerFault
http://serverfault.com/questions/77710/is-bit-rot-on-hard-drives-a-real-problem-what-can-be-done-about-it

Error Correction
Reed Solomon
http://en.wikipedia.org/wiki/Reed–Solomon_code

CRC
http://www.repairfaq.org/filipg/LINK/F_crc_v3.html

Low density parity check = Gallager codes
http://en.wikipedia.org/wiki/Low-density_parity-check_code
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.5754 (Practical Loss Resilient Codes)

Hamming
http://en.wikipedia.org/wiki/Hamming_code
http://kipirvine.com/asm/workbook/error_correcting.htm

Xilinx
http://www.xilinx.com/support/documentation/application_notes/xapp715.pdf

Double Error SRAM
http://www.isi.edu/~draper/papers/esscirc08.pdf

McEliece page
http://www.ee.caltech.edu/EE/Faculty/rjm/
Turbo codes
http://www.ee.caltech.edu/EE/Faculty/rjm/papers/Allerton98.pdf

BurtleBurtle.net (various interesting materials about hashes and codes)
http://burtleburtle.net/bob/hash/index.html

California Digital Library
https://merritt.cdlib.org

DLIB
http://www.dlib.org/dlib/november05/rosenthal/11rosenthal.html
https://confluence.ucop.edu/download/attachments/13860983/UC3-Foundations-latest.pdf?version=1

MD5
http://en.wikipedia.org/wiki/MD5

http://www.enterprisestorageforum.com/continuity/features/article.php/3716796/When-Bits-Go-Bad.htm

http://en.wikipedia.org/wiki/Data_corruption
http://indico.cern.ch/getFile.py/access?contribId=3&sessionId=0&resId=1&materialId=paper&confId=13797
http://en.wikipedia.org/wiki/ECC_memory
http://en.wikipedia.org/wiki/Chipkill
http://en.wikipedia.org/wiki/BCH_code