Combining Chunk Boundary and Chunk Signature Calculations for Deduplication

Witold Litwin, Darrell D. E. Long, Thomas Schwarz, S.J.

***Abstract*— Many modern, large-scale storage solutions offer deduplication, which can achieve impressive compression rates for many loads, especially for backups. When accepting new data for storage, deduplication checks whether parts of the data is already stored. If this is the case, then the system does not store that part of the new data but replaces it with a reference to the location where the data already resides. A typical deduplication system breaks data into chunks, hashes each chunk, and uses an index to see whether the chunk has already been stored. Variable chunk systems offer better compression, but process data byte-for-byte twice, first to calculate the chunk boundaries and then to calculate the hash. This limits the ingress bandwidth of a system. We propose a method to reuse the chunk boundary calculations in order to strengthen the collision resistance of the hash, allowing us to use a faster hashing method with fewer bytes or a much larger (256 times by adding two bytes) storage system with the same high assurance against chunk collision and resulting data loss.**

***Keywords*— Deduplication, Algebraic Signatures**

I. Introduction

DEDUPLICATION is an increasingly popular strategy to compress data in a storage system by identifying and eliminating duplicate data. Online deduplication of backup workloads has shown impressive compression ratios (20:1 as reported by Zhu, Li and Patterson [1], and up to 30:1 as reported by Mandagere, Zhou, Smith, and Uttamchandi [2]). It has been shown to scale to petabytes [3].

Identifying online duplicate data in a system that already stores petabytes of data is difficult. File-based deduplication only checks for duplicate files, but looses deduplication opportunities in all instances where a file is stored again with only slight changes. Additionally, many systems do not store files, but streams, as arise for example from taking a full system backup. Chunk-based deduplication [4], [5], [6] divides input into chunks, characterizes these chunks by their hash, and looks for duplicate chunks using an index based on the hashes. Using fixed-sized chunks looses deduplication opportunities when data differs from already present data by small insertions or deletions. Content defined chunking [5] uses local information to define chunk boundaries in way independent of alterations in previous chunks. Small, localized changes in a large file do not alter most of the chunks of the file. The system can recognize this duplicate data and avoid storing them twice. Calculating chunk boundaries involves processing incoming data byte-for-byte. After calculating the chunk boundaries, the deduplication system calculates a hash value of the chunk and then looks up (and later stores) the value in an index of all previous chunk hashes. If we can find the same hash value, we conclude that the chunk is already stored and do not store it a second time. In order to achieve good bandwidth on ingress, deduplication typically does not verify the identity of chunks with a byte-for-byte comparison. If there is a hash collision – two different chunks have the same hash value – then the system has lost the newer data. The data loss rate due to these collisions has to be smaller than the data loss rate due to other causes such as disk failures or cooling system failures for deduplication to be acceptable. We easily control the hash collision probability by using good hash functions with more bits, replacing MD5 with 128 bits for SHA-1 (160 bits) or upgrading to a member of the SHA-2 family such as SHA-256 with 256 bits.

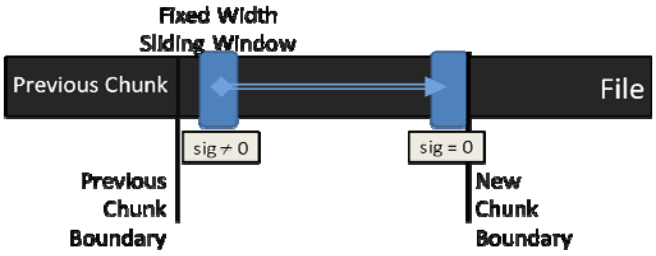

Figure 1: Sliding Window Technique

In addition to loading indices into memory from a disk or flash memory, chunk determination and chunk hash calculation form a bottleneck for which we pay in the form of better or more processors. Using a faster hash function is therefore attractive.

Chunk boundaries need to be calculated in a manner independent of alterations to the file or the data stream at far away locations. It commonly uses a sliding window technique [5] (Fig. 1). This technique calculates a function of the bytes in a small window and sets a chunk boundary if the value is equal to zero. Our contribution is to reuse this calculation to obtain additional bytes for the hash and strengthen collision resistance at no costs to performance. Alternatively, by reusing work already done, we can use a faster hashing algorithm that generates fewer bytes. Our work can be incorporated in all current deduplication techniques with one slight exception, which we will now discuss.

Witold Litwin, Centre d'Etude y Recherche en Informatique Appliqué, Université Paris Dauphine, Pl du Mal. de Lattre de Tassigny, 75016 Paris, France. Witold.Litwin@dauphine.fr

Darrell Long, *Fellow, IEEE,* Computer Science Department, University of California at Santa Cruz, 1156 High Street, Santa Cruz, CA, USA. darrell@cs.ucsc.edu

Thomas Schwarz, S.J., *Senior Member, IEEE,* Dto. Informática y Ciencias de la Computación, Av. 8 de Octubre, Universidad Católica del Uruguay, Montevideo, Uruguay. tschwarz@ucu.edu.uy

The generic method of chunk boundary determination is probabilistic and can lead to very small and to very large chunks, which has a negative impact on deduplication rates. Eshgi and Tang [7] use additional techniques to prevent chunks that are excessively small or large. To prevent excessively small chunks, possible chunk boundaries encountered shortly after the start of a new chunk are ignored. They also use a secondary condition to create alternative chunk boundaries. If a chunk becomes too big, the most appropriate alternative boundary is used. If we incorporate our proposal with their technique, we would not be able to incur the savings accruing from not evaluating the window near the chunk beginning.

The security of the system can depend on the security of the hash function used. An adversary with access to the storage system can mount a targeted-collision attack [8] by finding a collision with a chunk that is to be inserted in the future. For an example, assume that the storage systems stores system files of various workstations. A patch will change these files in a predictable manner. An adversary could create a new chunk colliding with a chunk in a modified system file and insert it in the storage system before the new system files are stored there. After the victim's system has been patched and backed up, the system would then recover not the attacked file but a file altered by the adversary. We can protect against this type of attack in various ways [8], including simply by using a keyed-Hash Message Authentication Code (HMAC) instead of a well-known, fixed hash function such as MD5. In this paper, we are concerned with preventing collisions arising statistically from the large number of chunks in the system.

In what follows, we explain the mathematics of chunk boundary calculation with an eye towards performance (Section II), then present our chunk boundary cum hashing algorithm (Section III), and finally evaluate the quality of the addition to the chunk hash and its consequences for collision resistance (Section IV).

II. Chunk Boundary Calculations

We determine chunk boundaries by calculating a function of a small window (e.g. of size 4 to 10 bytes) and setting a chunk boundary if the function has a value in a specific set. Most appropriate is the use of a rolling hash, in which the value of the window moved to the right by one byte is calculated from the previous value, the byte on the left that just entered the window, and the byte on the right that just left the window. There are several, mathematically related possibilities for defining a rolling hash. We will here use a variant of algebraic signatures [9] but present for the sake of completeness the popular Rabin Fingerprints [10] in order to show that our choice bears the same computational burden.

A. Rabin Fingerprint

Rabin's fingerprinting scheme [10], [11] associates first a polynomial to a bit string $P = (p_0, p_1, ..., p_{m-1})$ by setting

$$p_P(t) = p_0 t^{m-1} + p_1 t^{m-2} + ... + p_{m-2} t + p_{m-1}$$

and then uses a fixed irreducible polynomial *g(t)* over F_2, (i.e. a polynomial with coefficients either 0 or 1) to define the *Rabin Fingerprint* of the string as

$$R(P) = p_P \pmod g$$

where g defines this particular fingerprint. As is usual, we identify a polynomial $p_P(t)$ with the bit string P, where, if necessary, we pad on the left with zeros to fit P into bytes or words. The number of bits in the Rabin fingerprint is given by the degree of the polynomial $g(t)$. For example, a four-byte Rabin fingerprint requires a polynomial $g(t)$ of degree 32.

Assume a sliding window size of w and a byte string $P = (p_0, p_1, ...)$. We first convert the byte string P into a string of bits. We identify also a byte p_i with the polynomial

$$p_i(t) = p_{i,0} t^7 + p_{i,1} t^6 + ... + p_{i,6} t + p_{i,7}$$

where $p_i = [p_{i,0}, p_{i,1}, ..., p_{i,6}, p_{i,7}]$ is the decomposition of the byte p_i into eight bits. We then calculate modulo $g(t)$:

$$\begin{aligned} & R(p_{r+1}, p_{r+2}, ..., p_{r+w}) \cdot t^{-8} \\ &= p_{r+1}(t) t^{8(w-2)} + p_{r+2}(t) t^{8(w-3)} + ... + p_{r+w}(t) t^{-8} \\ &= p_r(t) t^{8(w-1)} + p_r(t) t^{8(w-1)} + p_{r+1}(t) t^{8(w-2)} + ... + p_{r+w}(t) t^{-8} \\ &= p_r(t) t^{8(w-1)} + R(p_r, p_{r+1}, p_{r+2}, ..., p_{r+w-1}) + p_{r+w}(t) t^{-8} \end{aligned}$$

In the second equation, we use the fact that addition of coefficients is the bitwise exclusive-or operation. We will continue to use the "+" symbol to denote this operation, which is the addition in the Galois field with two elements, 0 and 1. Consequentially, $p_r(t) t^{8(w-1)} + p_r(t) t^{8(w-1)} = 0$. As a result of our calculation, we obtain the transformation rule for sliding windows:

$$\begin{aligned} & R(p_{r+1}, p_{r+2}, ..., p_{r+w}) \\ &= p_r(t) t^{8w} + R(p_r, p_{r+1}, p_{r+2}, ..., p_{r+w-1}) + p_{r+w}(t) \end{aligned}$$

We will use this transformation rule to calculate the sliding window signature. The easiest implementation uses two tables for the multiplication with t^8 and t^{8w}. Incidentally, Broder [11] has proposed a very efficient solution that calculates the Rabin fingerprint of an object by processing several characters at a time. This technique is of course not available to us.

An alternative way of calculating the Rabin fingerprint of a sliding window maintains the fingerprint $R(p_0, ..., p_m)$ of the current chunk seen so far. We then calculate the value of a chunk up to the n^{th} byte by

$$R(p_0, p_1, ..., p_n) = R(p_0, p_1, ..., p_{n-1}) \cdot t^8 + p_n(t)$$

The value of the sliding value is then given by

$$R(p_r, ..., p_{r+w-1}) = R(p_0, ..., p_{r+w-1}) \cdot t^{8w} + R(p_0, ..., p_{r-1})$$

where again we calculate modulo $g(t)$. In both cases, the straightforward implementation uses two table lookups and two XOR operations.

The table sizes are large. If the degree of g is k, then there are 2^k possible values for the fingerprint and correspondingly there are 2^k entries of size 2^k bits. For even moderate values of k, the table becomes too big, especially to fit in the L1 or L2 cache. For example, if $k = 16$, the table size is 131 kB, so that fingerprint calculations using a single table for each multiplication limits the size of fingerprints to two bytes.

Fortunately, we can use the linearity of multiplication. To generalize, assume that we want to implement multiplication modulo $g(t)$, a fixed polynomial of degree k. Call

$$\Re_k = \{a_k t^k + a_{k-1} t^{k-1} + ... a_2 t + a_1\}$$

the set of all polynomials with binary coefficients up to degree k–1. Assume that we want to implement multiplication by a polynomial $f(t) \in \Re_k$. We break each polynomial $a(t) \in \Re_k$ into addends

$$A_0(t) + A_1(t) t^8 + A_2(t) t^{16} + ...$$

with coefficients $A_0(t) = a_8 t^7 + ... + a_2 t + a_1$, $A_1(t) = a_{16} t^7 + ... + a_{10} t + a_9$ that represent the results of breaking the k-bit string into bytes. If necessary, we buffer with zeros at the end. Assume now that we want to implement multiplication with a given polynomial $f(t)$ modulo $g(t)$. We observe that by the distributive law of multiplication

$$\begin{aligned} f(t)a(t) &= f(t)\left(A_0(t) + A_1(t) t^8 + A_2(t) t^{16} + ...\right) \\ &= f(t)A_0(t) + f(t)t^8 A_1(t) + f(t)t^{16} A_2(t) + ... \end{aligned}$$

We therefore need a table (with 2^8 entries of k bits each) to implement multiplication (mod $g(t)$) of $f(t)$ and a polynomial in $\Re_8$, a second table to implement multiplication of $f(t)t^8$ and a polynomial in $\Re_8$, etc. All in all, we need $k/8$ tables of size $2^8 k$ bits, at a considerable savings in space, but also at the costs of adding $k/8$–1 exclusive-or operations.

B. Algebraic Signatures

Algebraic signatures are hash functions with algebraic properties [9]. Just as the cryptographically secure hashes such as SHA-1, SHA-2 or MD5 [12], [13], they identify large data objects with very low probability of collision. Unlike these hashes, they have algebraic properties that for example allow us to calculate the algebraic signature of a composite object from the algebraic signatures of the components. These properties could be exploited by a hypothetical attacker if algebraic signatures were to be used in their place in a cryptographic protocol, but we do not use them here in a way that impacts security. Algebraic signatures use Galois field arithmetic, which we now briefly review.

As the better known fields of rational, real, or complex numbers, Galois fields also have an addition and a multiplication that share the same arithmetic rules. The number of elements in a Galois field is always a power of a prime, and for each such number, only one Galois field (up to isomorphy) exists. We are only concerned with Galois fields where the number of elements is a power of 2. We write $F(2^f)$ for the only Galois field with 2^f elements. Because of the byte-based nature of storage systems, we furthermore will only use f=8 in what follows. With this choice of f, we can identify each element in the Galois field with a byte. However, we point out that our results are true when using a different value for f. We implement the operations in $F(2^f)$ by using the representation of Galois field elements as bit strings of length f. The addition is given by the exclusive-or and the zero element is the bit strings with only zero digits.

For our purposes (as well as for other applications such as cryptography) so-called *primitive elements* in the Galois field are important. An element α is a primitive element if its powers α^i, i=0, 1, ... make up all non-zero elements of the field. Algebra proves that all Galois fields not only have primitive elements, but that they are plentiful. If we fix a primitive element α, then any other non-zero element β is a power α^i of α, where i is a uniquely determined number between 0 and 2^f–1. In this case, we call i the logarithm of β and write $i = \log_a(\beta)$ and $\beta = \text{antilog}_\alpha(i)$. We can multiply two non-zero elements in the Galois field by

$$\beta \cdot \gamma = \text{antilog}_\alpha(\log_\alpha(\beta) + \log_\alpha(\gamma))$$

where the addition is taken modulo 2^f–1. Galois field multiplication can be implemented in a variety of ways such as by one or several tables. The best method for Galois field multiplication depends on the processor architecture, cache sizes, speed and miss penalties, and the size of the field [14].

The definition of algebraic signatures as given in [9] is not optimal for use as a rolling hash. We therefore define an alternative algebraic signature, the s-signature, which has the same basic structure and corresponding properties as the algebraic signature and is simply the algebraic signature of the window reversed byte-by-byte. We identify the bytes in the window with elements of the Galois field with 2^8 elements.

An s-signature consists of several components each of which is a byte. A component of the s-signature is defined using an element β of the Galois field. If $P = (p_0, p_1, ..., p_{n-1})$ is a string of bytes, then we define the component s-signature as

$$s(\beta, P) := \sum_{\nu=0}^{n-1} p_\nu \beta^{n-\nu-1}$$

The complete s-signature is a vector of length m where each element is a component s-signature defined by consecutive powers of a primitive element α.

$$s_{\alpha,m}(P) := \left(s(\alpha, P), s(\alpha^2, P), ..., s(\alpha^m, P)\right)$$

When we slide a window one character to the right, we update a component of the s-signature using

$$\begin{aligned} &s\left(\beta, (p_{r+1}, p_{r+2}, ..., p_{r+w})\right) = \\ &p_{r+w} + s\left(\beta, (p_r, p_{r+1}, p_{r+2}, ..., p_{r+w-1})\right) + \beta^w p_r \end{aligned}$$

As we want to use s-signatures as hashes of the whole chunk, we combine calculating the compound s-signature of the chunk seen so far with updating the signature of the window as it slides character by character to the right. When we process one more byte in the chunk, we update all components of the s-signature using

$$s\left(\beta, (p_0, p_1, ..., p_n)\right) = \beta \cdot s\left(\beta, (p_0, p_1, ..., p_{n-1})\right) + p_n$$

and calculate the window signature from

$$\begin{aligned} &s\left(\beta, (p_{r+1}, p_{r+2}, ..., p_{r+w})\right) = \\ &s\left(\beta, (p_0, p_1, p_2, ..., p_{r+w-1})\right) + \beta^w s\left(\beta, (p_0, p_1, p_2, ..., p_{r-1})\right) \end{aligned}$$

We set a chunk boundary if the window s-signature is zero. This is done by comparing two compound signatures, beginning at the left boundary of the current chunk and ending just before or at the end of the window. The condition is

$$s(\beta,(p_0,p_1,...,p_{r+w-1})) = \beta^w \cdot s(\beta,(p_0,p_1,...,p_{r-1}))$$

Each time we process a new character, we upgrade the *s*-signature of the chunk seen so far. Per component of the *s*-signature, processing this character costs us one addition (exclusive-or) and one Galois field multiplication, which we can implement as a table lookup. We then test for a chunk boundary, which costs us also an addition (exclusive-or) and a Galois field multiplication per component.

Comparing the costs of calculating *s*-signatures and Rabin fingerprints, we obtain the same number of table look-ups and exclusive-or operations. Thus, operationally, neither method has an advantage. We prefer algebraic signatures because we can prove their suitability as good hash functions. The argument of our paper (chunk boundary calculations can be re-used for obtaining additional bytes of the hash at no costs) is valid for either algebraic signatures or Rabin fingerprints.

```
i = 0
for j in range(4):
  oc[j] = (0, 0)
while True:
  c = getNextCharacter(file)
  p1 = mult(a,p1) ^ c
  p2 = mult(a2,p2) ^ c
  (q1, q2) = oc[i]
  if mult(a4, p1), mult(a8, p2) & mask
      == (p1, p2) & mask:
       trigger_chunk_boundary()
oc[i] = (p1, p2)
i = i+1 mod 4
```

Figure 2: Pseudo-Code for Chunk Boundary Calculation

III. Implementation

We use an *s*-signature of size two bytes and use a small window of four bytes. Internally, we maintain a circular buffer in order to store the last four *s*-signatures of the chunk seen so far. Because *s*-signatures are very good hash function (as we will show in the next section) we chose the small window size, though the method carries directly over to larger windows. The two byte size of the *s*-signature implies that we will break on average after 2^{16} bytes, or equivalently, that the average chunk size is 64 kB. This value is higher than usually used, but we can modify the condition in order to break more frequently.

The properties of the *s*-signature (Section IV) guarantee that we will break if we encounter four zero bytes in a row. This allows us to check without additional costs if we have reached a longer run of zeros. We do not need to store these zeros explicitly, but can compress the file by storing the number of zeros encountered instead.

We give pseudo-code of our implementation in Fig. 2. We chose a fixed primitive element α and generate multiplication tables for multiplication by α, α^2, α^4, and α^8. These elements are the variables a, a2, a4, and a8 in the code. We store $s(\alpha, .)$ and $s(\alpha^2, .)$ in variables p1 and p2. When we process a new character c of the file, we actualize p1 and p2 using $p1 = \alpha \cdot p1 \oplus c$ and $p2 = \alpha^2 \cdot p2 \oplus c$. We then store the tuple (p1,p2) in the circular buffer of size four. However, before we overwrite the value, we retrieve the tuple (q1, q2) previously stored there, which is the *s*-signature of the chunk four characters before. These values are used to test the break condition $q1 \cdot \alpha^4 = p1$ and $q2 \cdot \alpha^8 = p2$. The multiplications mult with various constants are implemented by table look-up using a table with 256 entries.

A break is triggered whenever two values of 16 bits each are equal. Since both values behave like random numbers, this happens with probability 2^{-16}. If we want to have breaks happen more frequently, we can calculate the logical AND of both sides with a mask with *l* bits set, leading to average chunk sizes of 2^l bytes.

We give the complete algorithm for chunk boundary termination in Fig. 2. There, we maintain a circular buffer (oc) to contain the previous four chunk signatures and use *i* as an index into this buffer. Since our window size of four is a power of two, incrementing the index is particularly efficient since

```
i = (i+1) & m
```

accomplishes the increment modulo 2^k with $m = 2^k - 1$.

We verified the efficiency of our algorithm experimentally. Our results in Table I show that on our test machine (with an Intel Duo processor with clock rate 2.3 GHz), chunk boundary calculation run about twice as fast as an implementation of MD5 from RSA Security, Inc. (1991). SHA-1 was 20% slower than MD5. Calculating the complete chunk hash as an algebraic signature of 16 bytes unfortunately run 3.8 times slower than MD5. Thus, our initial hope of integrating boundary calculation and chunk hash calculation proved to be infeasible using algebraic signatures. Newer processors have new machine instructions in order to implement AES and other cryptographic algorithms faster and on these architectures, algebraic signature calculation could also run faster.

TABLE I: Throughput of MD5 and Chunk Boundary Calculation

TASK	THROUGHPUT
MD5	1.939 GB/sec
Chunk Boundary Determination	4.267 GB/sec

IV. Added Anti-Collision Assurance

A collision is a situation where chunk signatures coincide while the chunks themselves differ. The result of a collision in a deduplication system is that the incoming file is not stored correctly, rendering it essentially unusable. A chunk collision therefore always constitutes data loss. We now first calculate the added anti-collision assurance assuming a perfectly flat hash function (where each value of the hash is taken with the same probability) and then discuss the flatness of algebraic signatures.

A. A Bound for Storage System Size Based on Anti-

Collision Assurance

The problem of calculating the probability of one or more collisions in a system with N chunks is better known as the Birthday Paradox. Good approximations for the collision probability are known [15], but are strictly valid only if each

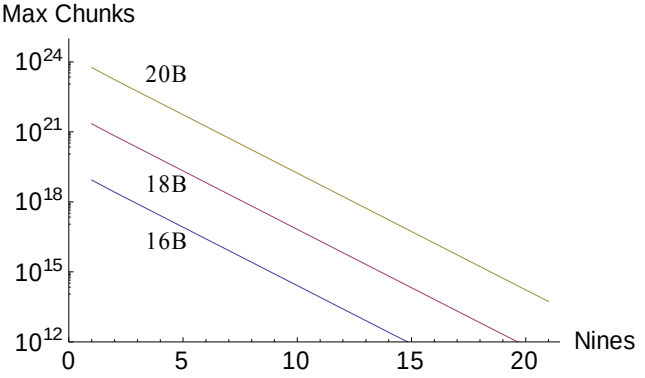

Figure 3: Maximum Number of Chunks for perfectly flash hash functions of 16, 18, and 20 bytes in dependence on the number of nines in the anti-collision assurance.

value of the signature is equally likely. In this case, a reasonable approximation for the collision probability if there are 2^m possible hash values is

$$p_{\text{col}}(m) = 1 - \exp(\frac{-N^2}{2^{m+1}})$$

This number applies to a signature of m bits. We measure resilience against the existence of even one collision with the *assurance*, which we define to be the probability that there is no collision in the system. We usually measure assurance in terms of numbers of nine. For instance, an assurance level of six nines states that the probability of no collision is 99.9999% and corresponds to a value of $\varepsilon = p_{\text{col}} = 0.000001$. To be acceptable, the assurance against even one collision in the system needs to be higher than the assurance against any other type of data loss such as a combination of disk drive failures. If n is the number of nines in the assurance level, then we have to choose $\varepsilon = 10^{-n}$ as our assurance target. Accordingly, for the probability of a collision to be less than a given assurance target ε, we solve the corresponding inequality and obtain

$$N \le \sqrt{-\ln(1\text{-}\varepsilon)2^{m+1}}$$

As we choose $\varepsilon << 1$, we can use the linear approximation of the Taylor expansion of the natural logarithm around 1 to obtain

$$N \le \sqrt{\varepsilon} \cdot 2^{(m+1)/2}$$

with an error of $O(\varepsilon)$. We specify ε in the number n of nines of confidence that a system with N chunks does not suffer a single collision, i.e. we set $\varepsilon = 10^{-n}$. If we take the decadic logarithm of this number, we obtain

$$\log_{10}(N) \le -\frac{n}{2} + (m+1)\frac{\log_{10}(2)}{2}$$

$$\approx -\frac{n}{2} + 0.150515(m+1)$$

As a result, the maximum admissible number of chunks in the storage archive increases by $10^{1.20412}$ for each additional byte. Two additional bytes increase the maximum size of the archive by a factor of 256 and three by a factor of 4096. We present numerical results in Fig. 3. The x-axis gives the numbers of nine of assurance while the y-axis gives the maximum number of chunks. We can multiply this number by the average chunk size to obtain the maximum storage capacity of the system (based solely on collision resistance).

For example, if the probability of data loss in the storage system is estimated at 10^{-9} per year (a very good value) and the life of the storage system is estimated to be 100 years (a very high value), then assurance against a collision should be better than 10^{-11}. We arbitrarily pick an anti-collision assurance level of 15 nines. If we use a hash similar to MD5 with 16 bytes, the maximum number of chunks is then 8.692×10^{11}. If we use our method to add two bytes to the hash size, we obtain a maximum of 2.225×10^{14}. If the average chunk size is 4 kB, the maximum size of the storage system changes from hundreds of petabytes to tens of exabytes, reaching the limits of current files systems.

B. Flatness of Algebraic Signatures

Our calculation used the fact that the hash function is *balanced* or *perfectly flat*, i.e. that all values are taken equally often. This is exactly the case for algebraic signatures and approximately the case for Rabin fingerprints, if we consider each bit combination equally likely for a chunk. A verdict on the flatness of cryptographically secure signatures such as MD5 and the various members of the SHA-1 and SHA-2 families is much harder to obtain, though Bellone and Kohno [16] report close to perfect flatness for signatures obtained from a few bytes of SHA-1 output.

The argument for perfect flatness of algebraic signatures (or s-signatures) for random text is simple. If all 8b-characters in a string are equally likely to appear and there is no dependence between characters, then the probability that a given signature value is taken is proportional to the inverse of the total number of possible strings that take this value.

We calculate the probability that a given large random string X has a particular value $\mathbf{c} = (c_1, c_2, ..., c_m)$ as its (compound) algebraic signature. This gives us m equations

$$s_{\alpha,m}(X) := \left(s(\alpha, X), s(\alpha^2, X), ..., s(\alpha^m, X)\right) = \left(c_1, c_2, ..., c_m\right)$$

which we can write equivalently as

$$\mathbf{A} \cdot \mathbf{X} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix}$$

with matrix

$$\mathbf{A} = \begin{pmatrix} \alpha^{n-1} & \alpha^{n-2} & \cdots & \alpha^{1} & 1 \\ \alpha^{2(n-1)} & \alpha^{2(n-2)} & \cdots & \alpha^{2} & 1 \\ \alpha^{3(n-1)} & \alpha^{3(n-2)} & \cdots & \alpha^{3} & 1 \\ \vdots & \vdots & \ddots & & \vdots \\ \alpha^{m(n-1)} & \alpha^{m(n-2)} & \cdots & \alpha^{m} & 1 \end{pmatrix}$$

Matrix **A** is a matrix of Vandermonde type, if we order the columns in reverse order and is therefore of maximal rank *m*. The solution of the system of equations forms therefore a hyperspace of co-dimension *m* and the number of solutions is equal for each choice **c** of signature value. Therefore, if the input is random, then each value of the compound algebraic signature is taken with exactly the same probability. The equivalent statement for MD5 and hashes in the SHA family is only approximately true. It is not true for Rabin fingerprints, since Rabin fingerprints are taken modulo a polynomial *g*. If we transform *g* into a binary string and transform it from there to an arbitrary precision integer, we can say that the Rabin fingerprint cannot be larger than *g*. Of course, the number of values that a Rabin fingerprint cannot take is quite small, certainly much less than half of the possible values, so that Rabin fingerprints are still approximately flat.

TABLE II: Actual and expected number of collisions using a two byte signature on a dictionary (2.259MB) of moderately sized words.

Nr of Words Colliding	Observed	Expected
0	2741	2664.6
1	8539	8533.6
2	13599	13664.5
3	14563	14587.0
4	11645	11678.9
5	7377	7480.4
6	4140	3992.7
7	1835	1826.7
8	721	731.25
9	269	260.2
10	69	83.3
11	30	24.3
12	8	6.5
≥ 13	0	2.1

TABLE III: Actual and expected number of collisions using a three byte signature on the same dictionary

Nr of Words Colliding	Observed	Expected
0	16 568 642	16 568 642.3
1	207 274	207272
2	1 293	1 296.47
3	7	5.40624
≥ 4	0	0.0169079

TABLE IV: Actual and expected number of collisions using a two byte signature on a dictionary (2.259MB) of moderately sized words.

Nr of Words Colliding	Observed	Expected
0	19092	19011
1	23370	23527.7
2	14654	14558.7
3	5943	6005.84
4	1194	1858.18
5	455	499.928
6	89	94.8663
7	18	16.7721
8	1	2.5946
9	0	0.35678
10	0	0.0441544
≥ 11	0	0

Our calculations and arguments so far assume completely random input and we know that these do not make up the workload of storage systems. We therefore used English text as input for a test of flatness. To stack the deck against us, we do not calculate chunk signatures of English text, but instead use only words.

Our raw data was a list of English words (209881 words or 2.259MB) that is used to perform a dictionary attack on a password file (in our case, by administrators checking the strength of user chosen passwords). We calculated the 2-component signature (2 bytes) of all words with more than 5 letters (65536 words). We then tabulated how often different words had the same signature. For example, we found that there were 12 signatures that were each taken by 8 different words (penultimate line). The result is in Table II. We compared this number with the expected number of collisions for a perfectly flat signature of two bytes. The expected value of collisions is then given by a Poisson distribution with sample mean 3.202530. These numbers form the right column in Table II. Just looking at the numbers, the algebraic signature behaved in this case better than expected. We evaluated the difference with the χ^2 value, which we calculated to be 0.21 using 8+1 classes. While small, this value is not so small that one could state that the algebraic signature is flatter than that of a random, perfectly flat signature.

We repeated the experiment on the same set, but now used a three component signature and obtained the values in Table III. The coincidence between observed and expected values is still remarkably good. The χ^2 value of 0.479166 confirms this.

We repeated the first experiment on another, smaller dictionary (Table IV). In this case, observations and expected values do not match quite as well and the χ^2 value of 4.79742 confirms this impression. Encouragingly enough, the tail of the distribution of actual collisions is smaller and the number of signatures taken by only one word is higher than expected. Again, the algebraic signature performs better than expected.

We would like to emphasize that the fact that there are collisions at all is simply a function of the small size of algebraic signatures. There are 216 possible values that a two-component algebraic signature can take and any collection numbering more than that number must have collisions.

V. Conclusions

We presented a method that exploits work already done for chunk boundary calculations in order to add two (or three) bytes to the chunk signature. The additional bytes, obtained at no additional operational costs, improve the resilience of deduplication against chunk signature collision. As a result, a storage system can grow by a factor of 256 or (4096 when calculating three additional bytes) to provide the same high assurance level as before the change. Alternatively, the same two (or three) bytes can be used to switch to a faster, smaller, but less collision resilient hash function for the chunk signature. Our method constitutes one of the few cases of a practically "free lunch" in Computer Science. Integrating the additional bytes provided by the algebraic signature with the chunk signature is trivial. Loading the index into memory for searches and additions is largely independent of the size of index entries. If we choose a faster, shorter hash function, we recover the collision resilience of a larger hash while benefitting from faster calculations.

VI. Future Work

The work presented here came out of an attempt to combine chunk boundary determination and chunk signature calculation in a single pass, using algebraic signatures. Unfortunately, calculating an algebraic signature of 16 or 20 bytes length turned out to be several times slower than calculating MD5 or SHA1. This is not surprising since both hash functions were designed to process several bytes in a single step, avoiding the costly isolation of a single byte in a machine word on current processor architectures. Newer chip architectures (such as the Westmere chips introduced by Intel in 2010) implement new machine instructions useful for the calculation of AES and the speed question needs to be revisited for these machines. Since costs dictate that storage systems use commodity components, we do not foresee the use of special hardware to accelerate hash and signature calculations.

References

[1] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST), 2008, pp. 269–282.

[2] N. Mandagere, P. Zhou, M. Smith, and S. Uttamchandani, "Demystifying data deduplication," in Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion. ACM, 2008, pp. 12–17.

[3] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge, "Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup," in Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2009), 2009.

[4] G. Forman, K. Eshghi, and S. Chiocchetti, "Finding similar files in large document repositories," in Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining. ACM, 2005, p. 400.

[5] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in Proceedings of the eighteenth ACM symposium on Operating systems principles. ACM, 2001, pp. 174–187.

[6] S. Quinlan and S. Dorward, "Venti: a new approach to archival storage," in Proceedings of the FAST 2002 Conference on File and Storage Technologies, vol. 4, 2002.

[7] K. Eshghi and H. Tang, "A framework for analyzing and improving content-based chunking algorithms," Hewlett-Packard Labs Technical Report TR, vol. 30, 2005.

[8] M. W. Storer, K. Greenan, D. D. Long, and E. L. Miller, "Secure data deduplication," in Proceedings of the 4th ACM international workshop on Storage security and survivability, StorageSS '08. 2008, pp. 1–10.

[9] W. Litwin and T. Schwarz, "Algebraic signatures for scalable distributed data structures," in Proceedings. 20th International Conference on Data Engineering, 2004, pp. 412–423.

[10] M. Rabin, "Fingerprinting by random polynomials," Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[11] A. Broder, "Some applications of Rabins fingerprinting method," in Sequences II: Methods in Communications, Security, and Computer Science, Springer Verlag, 1993, pp. 143–152.

[12] F. I. P. S. NIST, "FIPS-180-1: Secure Hash Standard," 1995.

[13] R. Rivest, "RFC1321: The MD5 message-digest algorithm," RFC Editor United States, 1992.

[14] K. Greenan, E. Miller, and T. Schwarz, "Optimizing Galois Field arithmetic for diverse processor architectures and applications," in Proceedings of the 16th IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2008.

[15] Z. Schnabel, "The estimation of total fish population of a lake," American Mathematical Monthly, vol. 45, no. 6, pp. 348–352, 1938.

[16] M. Bellare and T. Kohno, "Hash function balance and its impact on birthday attacks," in Advances in Cryptology-Eurocrypt 2004. Springer, 2004, pp. 401–418.