HashFusion – a method for combining cryptographic hash values

Monahan, Brian; Chen, Liqun; Haber, Stuart

Hewlett Packard Labs

HPE-2017-08

Keyword(s):
HashFusion; cryptographic hash functions; composite hash functions

Abstract:
An important technique for confirming data integrity is to construct cryptographic hash values for the data. For bulk data, this can be done by generating hash values for ordered sequences of (variable sized) data blocks. This technique is already widely used in various HPE businesses, such as networking, data storage, and software certification (code integrity). The standard hash-chaining approach to constructing hash values can involve retaining large amounts of data due to differences in the order they are received in. In this paper, we present novel solutions to allow hash values to be calculated in stages. These partial hashes are calculated as data becomes available and then opportunistically combined as needed during the computation. The significant advantage of our approach is the considerable flexibility in exactly how and when those hashes are combined, leading to increased efficiencies in managing resources.

External Posting Date: February 7, 2017 [Fulltext]
Internal Posting Date: February 7, 2017 [Fulltext]

HashFusion – a method for combining cryptographic hash values

Brian Monahan[1], Liqun Chen[2] and Stuart Haber [3]

Hewlett Packard Labs

Abstract

An important technique for confirming data integrity is to construct cryptographic hash values for the data. For bulk data, this can be done by generating hash values for ordered sequences of (variable sized) data blocks. This technique is already widely used in various HPE businesses, such as networking, data storage, and software certification (code integrity). The standard hash-chaining approach to constructing hash values can involve retaining large amounts of data due to differences in the order they are received in. In this paper, we present novel solutions to allow hash values to be calculated in stages. These partial hashes are calculated as data becomes available and then opportunistically combined as needed during the computation. The significant advantage of our approach is the considerable flexibility in exactly how and when those hashes are combined, leading to increased efficiencies in managing resources.

1. Introduction

Arguably the most important use of cryptographic hash functions such as SHA-256 and SHA-3 is to insure the integrity of data. Traditionally, a single hash value for a list or sequence of data blocks is computed by concatenating them in order and regarding the resulting concatenation as a long single input to the hash function. Clearly, the blocks need to be available in the right sequence in order to compute the overall hash value correctly.

Our technique using partial hashing will provide some mitigation of these constraints. Here are two well-known application areas where partial hashing is useful:

- **Bulk data transfer / Secure streaming**: It is inconvenient and costly to retain potentially large amounts of data for integrity hashing purposes, just because constituent packets arrive out-of-order. Partial hashing allows us to aggregate hashes in a consistent and secure manner, thus permitting storage resources to be released sooner.
- **Computing integrity of data with "gaps" filled in later:** A well-known example involves signing a large software program where parts of the program (e.g. kernel modules) need to be updated from time to time and where the length of the updated part(s) could vary. In this use-case, various gaps are permitted in the data sequence which can be efficiently filled in as data becomes available. This would be efficiently achieved by retaining the already known partial hashes and then providing a means to combine these with hashes of new pieces as and when they became available.

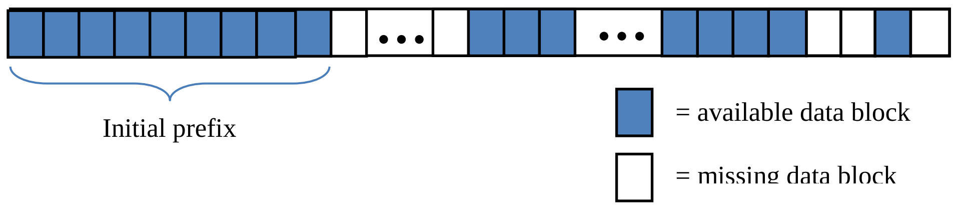


Figure 1: Partially completed bulk data with both available and currently missing data blocks

The question we ask is whether *partial hashes* for the available data blocks could be computed without first

[1] brianqmonahan@googlemail.com
[2] liqun.chen@surrey.ac.uk
[3] stuart.haber@acm.org

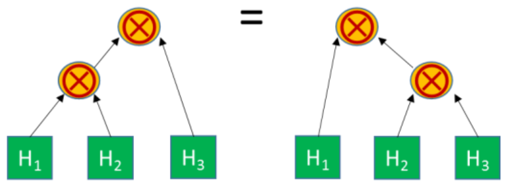
knowing the total number of data blocks and then later combined with the hash values of other parts as they become available. In this work, our target is to find solutions that allow secure partial hashes of data blocks to be correctly assembled in sequence, whilst allowing considerable flexibility in exactly how and when those hashes are combined.

2. Our Approaches

We briefly present two approaches[4], both of which satisfy the requirements of our design target. The initial approach given here provides an elegant "stepping stone" between the existing, standard approach based on Merkle hash trees and the matrix-based HashFusion approach. The key mathematical observation is for each solution to provide a *hash-combining* functional mechanism (called $\otimes$ here) that is both *associative* and *non-commutative*. In equational terms:

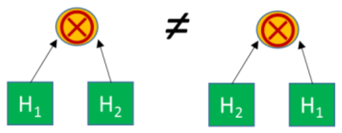
Asssociative Combination:

$$(H_1 \otimes H_2) \otimes H_3 \;=\; H_1 \otimes (H_2 \otimes H_3)$$



Non-Commutative Combination:

$$(H_1 \otimes H_2) \;\neq\; (H_2 \otimes H_1)$$



where H_1, H_2 and H_3 are arbitrary hash values. Basically, associativity means the function can be used to freely combine groups of hashes as necessary, and non-commutativity means the order of the hash values must be preserved. The security challenges posed by these hash-combination operations is discussed briefly below.

2.1. Initial approach

Our initial approach, illustrated in Figure 2, is a relatively straightforward approach and demonstrates that associative, non-commutative hash combination is at least feasible and potentially secure, albeit with some practical limitations such as scalability.

This approach involves taking independent hashes of each data block, which are then concatenated in sequence as byte vectors. The final hash value is obtained by hashing the resulting complete byte vector obtained once all data blocks have been hashed.

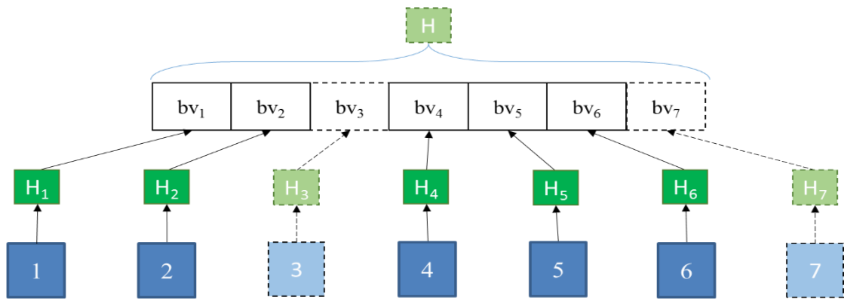


Figure 2: Using a sequence of hashes to construct the final hash value

[4] We assume here for both solutions that the ordering and adjacency info. of data blocks within the sequence is defined (as some kind of metadata) by the transmitting source and then used by the recipient(s). The final hash value is communicated out-of-band. For example, this positional information could be implemented by an index number, amongst other options.

The hash combination function here is byte vector concatenation and is well known to be both associative and non-commutative. In addition, we can prove that this technique preserves the collision resistance of the underlying primitive hash function used. Implementations of this technique need to accumulate sequence fragments that will be eventually assembled into a single, complete sequence of hashes.

2.2. HashFusion

The next approach, called here HashFusion, involves constructing partial hash values by using a binary hash combination function, $\otimes : \mathit{Hash} \times \mathit{Hash} \rightarrow \mathit{Hash}$, where this function will be *both* associative and non-commutative. This approach is described diagrammatically in Figure 3 below, for some contiguous collection of data blocks.

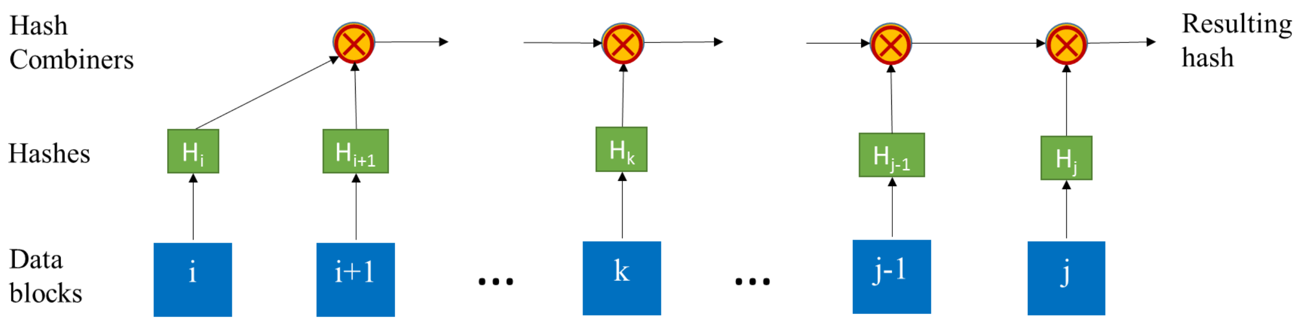
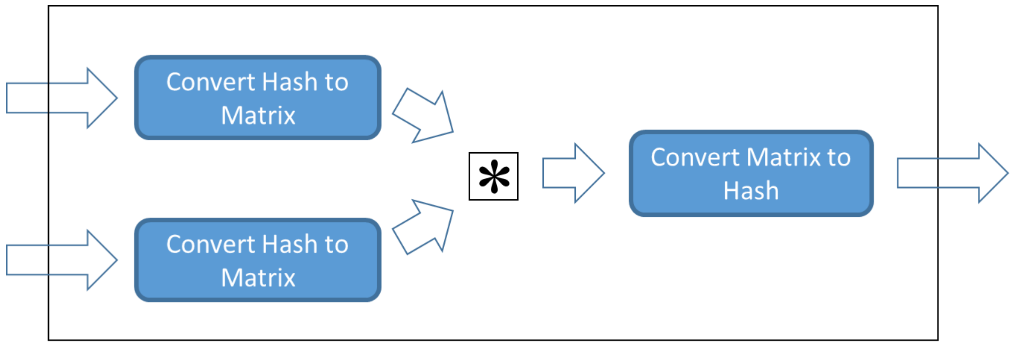


Figure 3: Combining contiguous blocks of hashes using ⊗

Because of its associativity, the function ⊗ can be used to combine *groups* of hashes (in appropriate sequence) as soon as they can be formed, thus reducing the number of hashes needing to be retained over time.

Figure 4 below describes the basic structure of the function ⊗ which takes hash values coming from standard hash functions and combines them by converting each input hash value (seen as a fixed size byte vector) into a particular kind of matrix with integer entries. These matrices are then combined via *matrix multiplication* (denoted here as $*$) using arithmetic *mod k*. Finally, the resulting matrix is converted back into a byte vector, again of standard fixed length. The process of conversion between hash-values and matrices is discussed in greater detail below.



$$\otimes : \mathit{Hash} \times \mathit{Hash} \rightarrow \mathit{Hash}$$
$$(H_1 \otimes H_2) = H_3$$

Figure 4: Internal structure of hash combiner function, ⊗

Our construction makes use of particular classes of matrices over rings of integers that turn out to form non-Abelian groups. Specifically, these classes consists of the (n x n) matrices over the integer ring $\boldsymbol{Z}_k$, each having the following upper-triangular form:

$$\begin{pmatrix} 1 & r_{1,2} & r_{1,3} & r_{1,4} & \dots & r_{1,n-1} & r_{1,n} \\ 0 & 1 & r_{2,3} & r_{2,4} & \dots & r_{2,n-1} & r_{2,n} \\ 0 & 0 & 1 & r_{3,4} & \dots & r_{3,n-1} & r_{3,n} \\ \dots & \dots & \dots & & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 1 & r_{n-1,n} \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 \end{pmatrix}$$

All of the matrices of interest have lower triangular 0 entries, leading diagonal 1, and otherwise arbitrary elements from the integer ring $\mathbf{Z}_k$. We shall call this set ***UTM***(n, k), for n > 2 and k > 1.

Here are some important mathematical facts about ***UTM***(n, k):

- Matrix multiplication is associative and, in general, non-commutative for n > 2.
- The identity matrix belongs to ***UTM***(n, k), and also, multiplication is *closed*.
- Each matrix in ***UTM***(n, k) is non-singular with determinant 1, as the leading diagonal consists entirely of 1's. A non-singular matrix has non-zero determinant – for us, the significant point is that the product of non-singular matrices over a ring is also non-singular i.e. non-singularity of these matrices is preserved by product.
- Even though all the entries within each matrix (i.e. $r_{i,j}$) are elements of rings (i.e. they don't have multiplicative inverses in general), it turns out that each matrix in ***UTM***(n, k) *does* possess an inverse of the same form, primarily because all these matrices have determinant equal to 1. That this inverse exists can be straightforwardly shown by using the adjoint form of the general matrix inverse. This implies that, for each n > 2 and k > 1, each set ***UTM***(n, k) forms a *group* under matrix multiplication.
- For each matrix in ***UTM***(n, k), there are $T_{n-1} = n(n - 1)/2$ general elements above the diagonal where T_n is the nth triangular number. This information is used when converting between hash-values and matrices.

The following matrices in ***UTM***(4, 6) illustrate both the multiplication and existence of inverses of matrix elements:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & 5 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 4 & 1 & 3 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The fact that ***UTM***(n, k) forms a group is important for our application as:

1. It shows that all possible matrices can appear as the result of an arbitrary product of other matrices from the same group. This ensures that byte vectors extracted from matrices can represent arbitrary hash-values.
2. Non-singularity implies that the product is never degenerate and could not lead to singular matrices under iterated multiplication. If we had permitted singular matrices, it is certainly possible for the matrix product of two non-zero matrices could in fact be the (degenerate) zero matrix (since matrix rings have non-trivial *divisors*

of zero). Repeated multiplication by singular matrices makes it increasingly likely that the outcome becomes degenerate at some stage. Ensuring non-singularity of the matrices we use will entirely avoid this pitfall.

As noted above, because this set of matrices forms a non-Abelian multiplicative sub-group of sufficient size within the underlying matrix ring, the corresponding hash combiner function, $\otimes$ is also guaranteed to be *surjective* – that is, capable of producing all possible hash values as outputs, by supplying appropriate input to each argument independently.

More precisely, this means that, for any matrix *A* and matrix *B*, there are (unique) matrices *X* and *Y* such that:

$(A * X) = B$ and also $(Y * A) = B$.

Because this group is non-Abelian, the matrices *X* and *Y* are generally not equal. The group properties also implies that, for any matrices *U* and *V*:

$(A * U) = (A * V) \Rightarrow U = V$ and equally: $(U * B) = (V * B) \Rightarrow U = V$

This means we can cancel on both the left and right of the $*$ operator.

2.2.1. Converting hash-values to and from matrices

As noted earlier, the hash-combiner function involves constructing upper triangular matrices from hash values and vice versa, we shall assume here that hash-values correspond directly to byte vectors of some fixed length. To obtain the matrix, we take entries from the byte vector and use these to fill-in entries in the upper-triangular part of the matrix, for some particular size of matrix. Any remaining entries in the upper triangular part are set to zero. To obtain the hash value from a given matrix following multiplication, extract the entries from the upper triangular part of the matrix and insert them into the byte vector. This mapping can be performed deterministically in any number of ways – what matters is that a specific way is chosen and then used systematically. In this manner, entries in the matrix correspond to each byte of the hash value, as represented in byte vector form.

However, in general, the number of bytes in a standard hash value does not equal the number of upper triangular entries in a matrix. Therefore, we will need to pad the byte vector with zeroes (at the end) to obtain sufficient entries for packing into the matrix. Table 1 gives the dimensions of the matrices needed to represent various standard hash values and how many padding bytes are needed.

Standard Hash Function	Hash size (bytes)	Dimension of UT matrix	Number of UT entries	Padding entries
RIPEMD 160	20	7	21	1
SHA 224	28	8	28	0
SHA 256	32	9	36	4
SHA 384	48	11	55	7
SHA 512	64	12	66	2

Table 1: Mapping of standard hash function[5] sizes to required UT[6] matrix dimension

[5] See NIST Cryptographic Hash working group: http://csrc.nist.gov/groups/ST/hash/
[6] UT = Upper Triangular

As an illustration of this process – consider this SHA256 hash value[7] (32 bytes):

```
0x0b a9 04 ea e8 77 3b 70 c7 53 33 db 4d e2 f3 ac 45 a8 ad 4d db a1 b2 42 f0 b3 cf c1 99 39 1d d8
```

We then choose matrices of dimension 9, since these have sufficiently many upper triangular entries (36) to contain the 32 values needed for this hash-value. The above vector is then padded with 4 zeroes at the end to bring the length up to 36 entries. This vector is then mapped into a 9 x 9 upper triangular matrix in ***UTM***(9, 2^8):

$$\begin{pmatrix}
1 & 0b & a9 & ea & 3b & 33 & ac & a1 & 99 \\
0 & 1 & 04 & e8 & 70 & db & 45 & b2 & 39 \\
0 & 0 & 1 & 77 & c7 & 4d & a8 & 42 & 1d \\
0 & 0 & 0 & 1 & 53 & e2 & ad & f0 & d8 \\
0 & 0 & 0 & 0 & 1 & f3 & 4d & b3 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & db & cf & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & c1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

Clearly this process can be reversed, on the proviso that any zero padding required was added in the manner illustrated and is removed again on extracting the entries. It turns out that, due to the upper triangular form used, the result of multiplying matrices with zero padding given in this way is also guaranteed to have the same zero padding as well. Thus, the finite subset of all matrices in ***UTM***(n, k) with a particular zero padding is closed under multiplication and clearly contains the identity. As a corollary of Lagrange's theorem for finite groups, this means this set also contains inverses as well – and thus forms a proper subgroup of the non-Abelian group ***UTM***(n, k).

The particular advantages of both our embodiments over more rigid partial hashing schemes such as Merkle Hash-Trees is that our use of associative hash combiners allows hashes to be combined in different ways and yet correctly compute the overall hash. This gives considerable flexibility as to how and when partial hashes can be computed which, in turn, yields reduced management overheads and allows storage resources to be released sooner. For example, once hashes are computed and combined, their storage and related data blocks can be released, thus reducing pressure on associated data transmission buffers.

2.2.2. A worked example

One way to illustrate how associative hash combiners are used is to examine a worked example. Suppose that a sequence of data blocks is sent over a communications medium. Typically, these data blocks are received in a different order than their transmission order. We can show how these blocks can then be economically accumulated into intermediate hash structures using HashFusion. So, consider the following 9 data blocks:

Now suppose that these blocks are received in the following order: 2, 4, 3, 7, 6, 1, 5, 9, and 8. The following diagram in Figure 5 illustrates how our algorithm calculates the top hash value:

[7] This value was obtained by hashing the string "Hello World!" using OpenSSL.

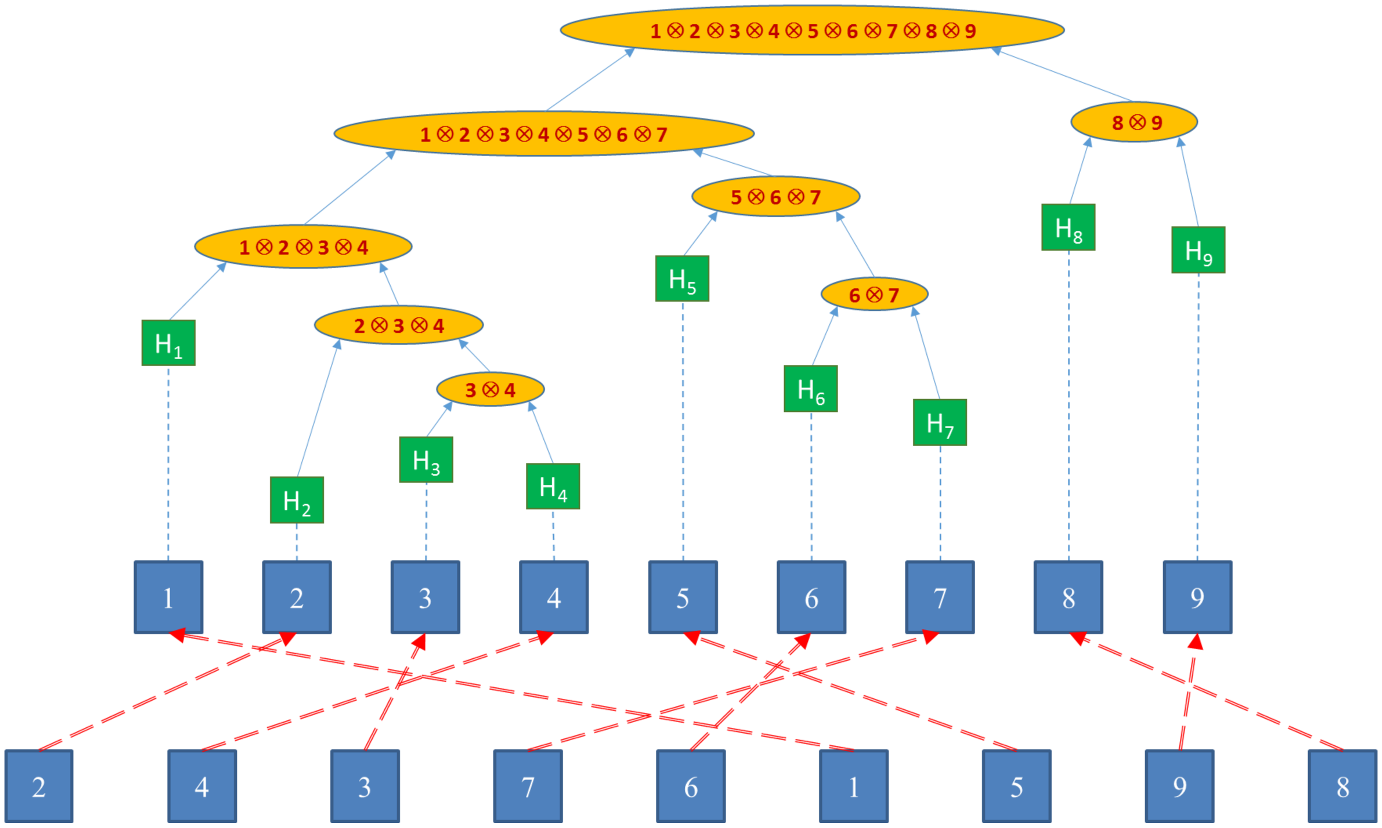


Figure 5: Using HashFusion to calculate the final hash value for the input sequence 2, 4, 3, 7, 6, 1, 5, 9, and 8.

From a memory storage point of view, we remark that the largest number of (partial) hashes needing to be stored at any one time is essentially *N/2*, where *N* is the total number of data blocks. This could arise when combination has been delayed as much as possible (for example, by receiving all the even numbered blocks followed by the remaining odd numbered). However, adjacent elements will eventually start to arrive, which then enables hash combination to reduce the number of partial hashes. Typically, arrival sequences forcing these limits are likely to be quite rare in practice. Instead, random groupings are to be expected.

2.2.3. Security challenges – towards a proof of secure combination

HashFusion involves some techniques of matrix algebra which, up till now, have received little or no attention from the cryptography community. As such, these techniques present fresh challenges to showing that HashFusion can be used securely.

The primary source of challenge here arises due to the way that our technique involves multiplying matrices in a matrix group, where the matrices are derived from the hash values themselves. Some of these questions are:

- Our technique maps hash values into elements of a matrix group – these elements therefore have inverses. How does the existence of inverses impact security?
- Any element of a finite matrix group has a *finite order* (i.e. $A^p = Id$, for some value of p). Our application concerns elements of ***UTM***(n, 2^8) – which implies, by Lagrange's theorem, that all orders of elements are powers of two. In particular, this implies that there will be elements of low order (e.g. 2, 4, 8, …).
    - What is the distribution of orders in non-Abelian groups of the form ***UTM***(n, 2^8)? How common are low order elements?
    - What is the effect of low order elements on the security of hash combiners?

- What security advice should be given to application developers wishing to use HashFusion?

The fundamental basis for the security of the hash-combining mechanism is that it must inherit its security properties of collision resistance from the standard cryptographic hash functions used. In fact, it is necessary to show that any successful attack on HashFusion would necessarily reduce to an effective attack on the cryptographic hash functions used – such an attack is widely believed to be infeasible.

While we do not know of any particular weakness that can be used to attack HashFusion itself, we still need an appropriate proof showing that a *composite hash function* using HashFusion has the well-known properties of a cryptographic hash functions: (see: e.g. https://en.wikipedia.org/wiki/Cryptographic_hash_function)

- **Pre-image resistance**

  Given a hash value *h* it should be computationally hard to find any message *m* such that *h = hashFn(m)*. This concept is related to that of *one-way functions*. Functions that lack this property are vulnerable to *preimage attacks*.

- **Second pre-image resistance**

  Given an input m_1 it should be computationally hard to find different input m_2 such that $hashFn(m_1) = hashFn(m_2)$. Functions that lack this property are vulnerable to *second-preimage attacks*.

- **Collision resistance**

  It should be computationally hard to find two different messages m_1 and m_2 where $hashFn\ (m_1) = hashFn\ (m_2)$. Such a pair is called a *cryptographic hash collision*. This property is sometimes referred to as *strong collision resistance*. It requires a hash value at least twice as long as that required for *preimage-resistance*; otherwise collisions may be found by a *birthday attack*.

It turns out that Bellare and Micciancio [3] discussed a paradigm for collision-free hashing, which has a similar target to our work. However, since all the combination algorithms considered in [3] are commutative, an index would need to be added to the hash of each block hash computation, in order to achieve non-commutativity of their hash function. Looking further at the security results given in [3], that suggests some ways for demonstrating collision resistance (in terms of Random Oracles) applicable to the approach herein.

Clearly, a thorough investigation of the security properties of matrix-based HashFusion remains to be completed. Conducting a comprehensive security analysis for HashFusion is required prior to deployment within secure contexts.

2.2.4. Computational performance of HashFusion

We are implementing a testing framework and libraries for HashFusion in both C and Python. We conducted a number of preliminary performance tests of HashFusion within our system and, rather pleasingly, some early results have shown that our matrix based algorithm can perform well, albeit under lab bench-test conditions.

The Python version, (using the well-known NumPy scientific computing extension) usefully demonstrated functional capability of the algorithm - but equally showed that there is plenty of room for improved performance as well! This prompted us to develop a C version of the hash combination library to demonstrate its performance in software.

The preliminary test results shown below resulted from implementing the basic version of HashFusion (for various sizes of standard hash) and then simply determining how many combinations could be computed per second. This information is shown in Table 2.

Standard Hash Function	Hash size (bytes)	Dimension of UT matrix	Hash Combinations per sec	Standard Error
RIPEMD 160	20	7	789k	2.1k
SHA 224	28	8	781k	2.0k
SHA 256	32	9	699k	2.1k
SHA 384	48	11	537k	1.6k
SHA 512	64	12	468k	0.9k

Table 2: Performance of hash combination for different hash sizes

These results were gathered over a large number of runs (600) for each hash size and averaged to give the number of hash combinations per second (with standard error).

What is somewhat surprising about these figures is that there aren't any substantial changes as dimension varies. Given that matrix multiplication is generally $O(n^3)$ for dimension n, this variation is not reflected in these results. A possible explanation for this is that caching effects and general software overheads have swamped much of the differences in timing due to differing dimension.

We clearly have much more work to do in investigating and improving performance further; for example, we have only explored simple configurations of the hash combination algorithm itself – and we don't yet have any figures for any end-to-end tests involving data transfers and assembly.

Although these initial results are somewhat inconclusive, they suggest that further investigation of performance aspects of HashFusion could demonstrate further performance improvements. For instance, although hardware acceleration for HashFusion has yet to be fully considered, it could offer significant advantages since integer matrix multiplication is fast on GPUs.

We admit that our performance results are somewhat moot if the security case for HashFusion turns out to be too hard to make – so our first priority has to be security, followed closely by performance.

3. Applying HashFusion to Hash Trees

The Merkle hash tree provides a way of hashing large number of data blocks by hashing adjacent elements together in a fixed binary tree-structure organization (see [1,2]). The primary advantage over standard hash chaining is that the primitive data blocks can be independently hashed and do not thereafter need to be retained. The construction proceeds by building full-size binary subtrees as soon as they arrive, and then finalizing the tree when a "no-more-input" signal is received, typically by "promoting" any rightmost subtrees as high as possible. In this way, the tree can be built online when blocks arrive out of sequence, as long as each arriving block is labelled with its place in sequence. This protocol avoids needing to know the total number of blocks, but it clearly needs to retain incomplete subtrees.

A snapshot of a partially completed Merkle hash-tree is described in Figure 6, for the same set of data blocks as used in the earlier example. At this stage, 5 blocks have arrived; blocks 2, 3, 4, 6, and 7. These have been hashed at level 0, and a single composite hash (H_B) has been computed at level 1.

The downside is that the Merkle hash tree is in general a rigidly structured full binary tree that retains any partial subtree fragments until they have been completed and the corresponding hash computed. The issue is that the structure of the Merkle tree is fixed at construction time and must be followed rigidly to recompute the hash value, unlike our use of associative hash combiners. Thus, as seen in Figure 6, the Merkle hash-tree technique imposes strict constraints on how the hashes of data blocks are combined.

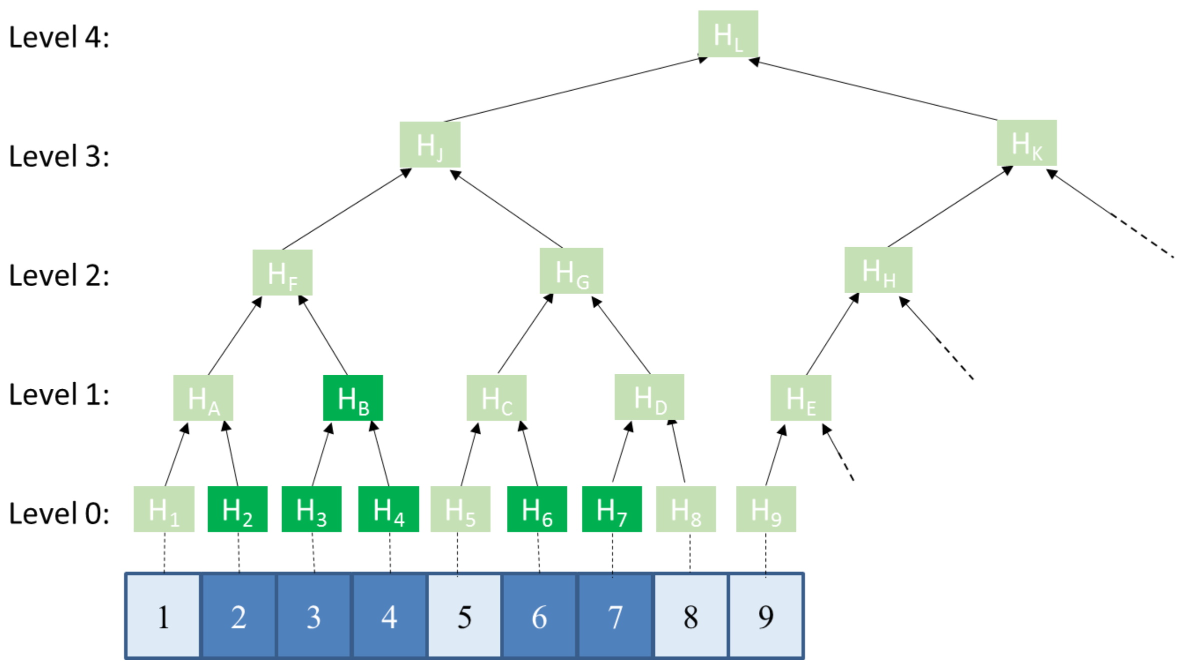
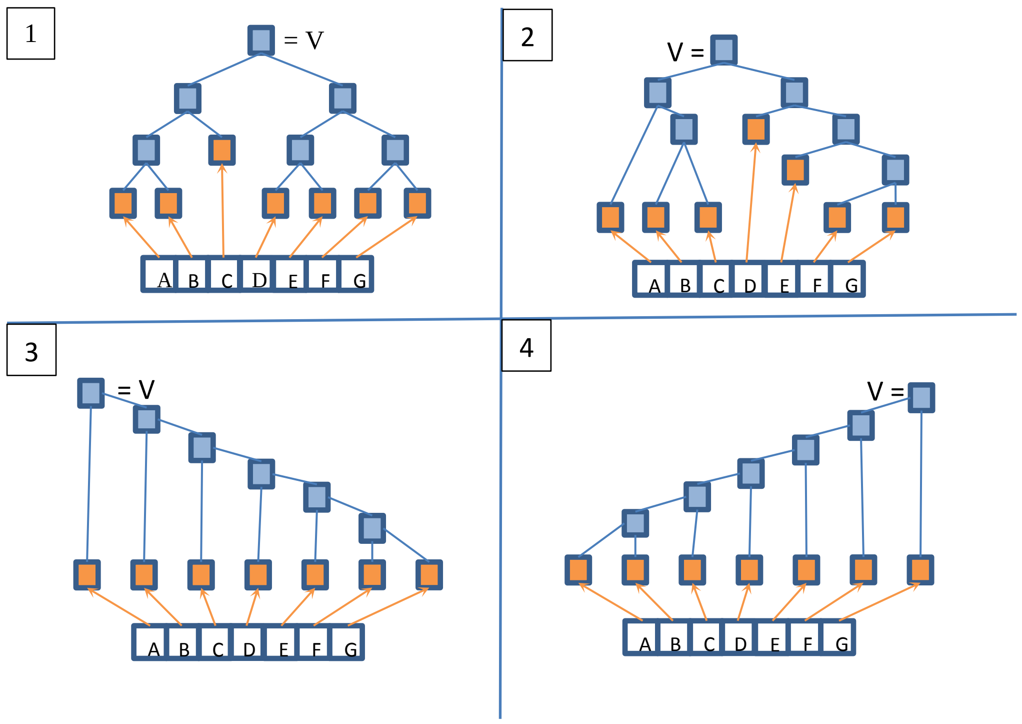


Figure 6: A partially completed Merkle hash tree, after blocks 2, 3, 4, 6, and 7 have arrived.

3.1. Tree Structures exploiting HashFusion

By using HashFusion and its associative property, we can implement a more flexible tree structure than the Merkle hash tree which is efficient for both the update of existing hashes and arbitrary insertion/deletion of entries.

The basic idea is to exploit the way that HashFusion's associativity allows data to be *regrouped* – this means that each of the following different shaped structures compute identical values, V, at root:



This is different to the Merkle hash tree case – all of these shapes would result in different values for Merkle hash trees.

However, where these topologies do differ is in average update times – the fastest update time is for panel 1 above (the most balanced), the slowest update times for panels 3 and 4 (the most skewed), and panel 2 is between these. The speed of update is related to the average distance from edge nodes to the root.

Now, the key observation is to realise that we need the structure to be near balanced for an efficient update (i.e. all hashes are (almost) at the same distance from the root) – but because HashFusion is associative, we can *tolerate* different shapes of tree. Thus, we can use a self-balancing binary tree structure (such as Red-Black trees) which has greater flexibility with providing balanced updates than the rigidly specified Merkle Tree. It is this flexibility which, where *N* is the number of nodes in the tree, allows general insertion/deletion of frontier nodes in *O(lg N).* Moreover, the updating of values follows the same statistics i.e. *O(lg N)*.

4. Further work

As described earlier, an investigation of the security properties of HashFusion is necessary for validation purposes. The objective of such an investigation would show how HashFusion can make hashing more flexible while retaining security and efficiency. Assuming the outcome of this investigation is positive, it would be necessary to externally publish in some appropriate form due to the innovation and novelty of these ideas within the cryptography context.

Finally, once security assurances concerning this technology have been satisfactorily established, one could entertain using HashFusion as an alternative to Merkle Hash Trees in a wide number of contexts – such developments might include:

- Extensions to hash chaining applications such as Distributed Ledger Technologies (e.g. blockchain).
- Data integrity for general distributed filing systems similar to e.g. ZFS.
- Data integrity within Cloud data storage context.
- Use of hash combination for distributed data integrity as part of networking protocols.

Acknowledgements

The authors thank in particular colleagues Fraser Dickin, Miranda Mowbray, and Philipp Reinecke from HP Labs, in Bristol for their advice, encouragement and support. Brian in particular also thanks both Richard Brown and Martin Sadler for their steadfast support and sage advice over the years.

References

[1] Merkle, R. C. (1988). *"A Digital Signature Based on a Conventional Encryption Function"*. In Advances in Cryptology — CRYPTO '87. LNCS **293**. p. 369.

[2] Merkle, R. C. (1979) "Secrecy, authentication, and public key systems". Stanford Ph.D. thesis.

[3] Mihir Bellare and Daniele Micciancio, (1997) *"A New Paradigm for collision-free hashing: Incrementality at reduced cost",* In Advances in Cryptology, Eurocrypt 97 Proceedings, LNCS **1233**, W. Fumy ed., Springer-Verlag. (see full paper at: `https://cseweb.ucsd.edu/~mihir/papers/inchash.pdf`)