Reviving the Idea of Incremental Cryptography for the Zettabyte era Use case: Incremental Hash Functions Based on SHA-3

Hristina Mihajloska [1], Danilo Gligoroski [2], and Simona Samardjiska [1]

Faculty of Computer Science and Engineering, UKIM, Skopje, Macedonia
hristina.mihajloska@finki.ukim.mk, simona.samardjiska@finki.ukim.mk
Department of Telematics, NTNU, Trondheim, Norway
danilog@item.ntnu.no

Abstract. One of the crucial factors for enabling fast and secure computations in the Zettabyte era is the use of incremental cryptographic primitives. For files ranging from several megabytes up to hundreds of gigabytes, incremental cryptographic primitives offer speedup factors measured in multiple orders of magnitude. In this paper we define two incremental hash functions *i*SHAKE128 and *i*SHAKE256 based on the recent NIST proposal for SHA-3 Extendable-Output Functions SHAKE128 and SHAKE256. We give two practical implementation aspects of a newly introduced hash functions and compare them with already known tree based hash scheme. We show the trends of efficiency gains as the amount of data increases in comparisons between our proposed hash functions and the standard tree based incremental schemes. Our proposals have the security levels against collision attacks of 128 and 256 bits.

Keywords: incremental hashing, SHA-3, Shake128, Shake256, iShake128, iShake256, Zettabyte era

1 Introduction

The idea of incremental hashing was introduced by Bellare, Goldreich and Goldwasser in [4] and improved later in [5]. Incremental hashing can be achieved also by using Merkle trees [14] as it is discussed for example in [7]. In a nutshell, the idea of incremental hashing is that if we have already computed the hash value of some document, and this document is modified in one part, then instead of re-computing the hash value of the whole document from scratch, we just need to update the hash value, performing computations only on the changed part of the document. In this way, incremental hashing of closely related documents offers speed gain up to several orders of magnitude compared to classical hashing. Yet, so far, the concept has not been particularly well accepted by the industry mainly due to the following two reasons: 1. The security level for the incremental hash functions is detached from the size of the produced hash value, that is usually several thousands of bits long. This is different from the ordinary cryptographic hash functions such as SHA-1, SHA-2, SHA-3, where the size of the hash value correspond to the claimed bit-security level of the hash function. 2. In order to achieve a certain level of security (for example 2^{128} or 2^{256}), the known incremental hash functions [4, 5] need to perform expensive modular operations over large prime integers. That makes them one or more orders of magnitude slower than the ordinary cryptographic hash functions.

However, it seems that our modern world has entered an era where a low computational demands for update operations of hash values finds applications in real situations. First of all, the data storage cost is no longer an issue (see for ex. [3]). Second, according to recent reports, the global IP traffic will pass the zettabyte threshold by the end of 2015, and will reach 1.4 zettabytes per year by 2017 [9], and by 2020, we can expect the size of the digital universe to reach 44 zettabytes [10].

Thus, the scale of data being processed every day by various cloud services, sensor networks, distributed storage systems and digital media services, already calls for new solutions that will use the paradigm of incrementality.

For example, let us consider the use case scenario of sensor networks where data arrives continuously and it needs to be stored. The data comes from the nodes whose data rates rapidly increase as sensor technology improves and as the number of sensors expands [11]. A typical representative for this scenario are environmental sensor networks used for natural disaster prevention or weather forecasting. In these cases, all data that is collected from different sensors should be publicly available for later analyses and computations. Nevertheless, integrity should be preserved, and usually, such dataset should be singed by a trusted party. In any case, data hashing is unavoidable, and as the dataset is being updated, so should the hash value be recomputed. Normally, the update of such datasets is done by simply appending new data from the sensors to the existing, or by changing a small part of the existing dataset. As the size of the dataset grows, (and can reach hundreds of terabytes [17]), recalculating the hash value of the entire dataset can become notoriously demanding in terms of both time and energy. Incremental update, on the other hand, can reduce the recalculation of the hash value to the minimum, and only of the parts of the dataset that have changed, or have been appended.

Another use case scenario where updates come in the form of insertions of new elements or modifications of existing data are distributed storage systems for managing structured data, such as Cloud Bigtable by Google [8]. It is designed to scale to a very large size, like petabytes of data across thousands of commodity servers. Its data model is described as persistent multidimensional sorted map, and it uses Google *SSTable* file format to internally store data [8]. Each SSTable contains a sequence of blocks typically of 64KB in size and every block has its own unique index that is used to locate the block. Using this kind of file formats where blocks have its unique numbers, incremental hashing can be successfully implemented despite the variable-size setting: In addition to the update operation, in order to perform incremental hash calculations, we introduce additional insert and delete operations.

The rest of the paper is organized as follows. In the next section we give the necessary mathematical preliminaries and definitions about incremental hash functions. In Section 3 we give an algorithmic descriptions of incremental operations for two practical settings. After that, in Section 4, we define two incremental hash functions with security levels of 128 and 256 bits. Comparison analysis between our proposals and incremental tree based hash scheme is given in the subsequent Section 5. Finally, we conclude our paper in Section 6 with recommendations on where and how to use our incremental hash functions.

2 Mathematical preliminaries

2.1 Incremental hash functions

We will use the following definition for an incremental hash function adapted from [5, Sec. 3.1]:

Definition 1.

1. *Let* $h : \{0,1\}^b \to \{0,1\}^k$ *be a compression function that maps* b *bits into* k *bits.*
2. *Let the message* M *be represented as a concatenation of* n *blocks, where* $n < N$ *for some predefined number* N *which is larger than the number of blocks in any message we plan to hash, i.e.,* $M = M_1||M_2||\ldots||M_n$.

3. *The size of each block M_i is determined by the following relation: $|M_i| = b - Length(ID_i)$, where ID_i is a unique identifier for the block M_i.*
4. *For each block M_i, $i = 1, \ldots, n$, append ID_i to get an augmented block $\overline{M_i} = M_i||ID_i$.*
5. *For each $i = 1, \ldots, n$, apply h to $\overline{M_i}$ to get a hash value $y_i = h(\overline{M_i})$.*
6. *Let $(G, \bigodot)$ be a commutative group with operation $\bigodot$ where $G \subseteq \{0,1\}^k$.*
7. *Combine $y_1, \ldots, y_n$ via a combining group operation $\bigodot$ to get the final hash value $y = y_1 \bigodot y_2 \bigodot \ldots \bigodot y_n$.*

Denote the incremental hash function as:

$$y(M) = \text{HASH}^h_{\langle G \rangle}(M_1||M_2||\ldots||M_n) = \bigodot_{i=1}^{n} h(M_i||ID_i) \tag{1}$$

Since the group $(G, \bigodot)$ is commutative, the computation is parallelizable too. In such a case, the combining group operation $\bigodot$ is commutative and invertible, and increments are done as follows. If block M_i changes to M_i', then the new hash value is computed as $y(M') = y(M) \bigodot^{-1} h(\overline{M_i}) \bigodot h(\overline{M_i'})$ where $\bigodot^{-1}$ denotes the inverse operation in the group $(G, \bigodot)$ and $y(M)$ is the old hash value. The cost of an increment operation is two hash computations and two operations in G.

The choice of good combining operation is important for both security and efficiency. The authors of the paper [5] proposed four different families of hash functions depending on the combining operation. In XHASH, the combining operation is set to bitwise XOR. The multiplicative hash, MuHASH uses multiplication in a group where the discrete logarithm problem is hard. AdHASH stands for hash function obtained by setting the combining operation to addition modulo a sufficiently large integer, and LtHASH uses vector addition. They showed that the scheme XHASH is not secure. In addition, they defined a computational problem in arbitrary groups that they called *balance problem*. In a nutshell, the balance problem is the problem of finding two disjoint subsets $I, J = 1, \ldots, n$, not both empty, for a given random sequence of elements $a_1, \ldots, a_n$ of G, such that $\bigodot_{i \in I} a_i = \bigodot_{j \in J} a_j$. The authors of [5] concluded that in order to have a collision-free hash function over G, the balance problem for the group $(G, \bigodot)$ should be hard. They estimated that a hash value of size ≈ 1024 bits would suffice for security level of 2^{80}. However, later on, Wagner in [18] showed that using a generalized birthday attack, these parameters are breakable, implying that the size of the hash values should be much bigger (for standard security levels, even up to tens of thousands of bits). Wagner also showed how to solve the k-sum problem for certain operations (a special case of the balance problem), with time and space complexity of $O(k \cdot 2^{\frac{n}{1+\lg[k]}})$ using lists of size $2^{\frac{n}{1+\lg[k]}}$ elements. More precisely, Wagner [18] showed the following:

Proposition 1. *Let $\text{HASH}^h_{\langle G \rangle}$ be an incremental hash function defined by Definition 1. For any $Y \in \{0,1\}^n$ the complexity of finding a preimage message $M = M_1||M_2||\ldots||M_k$ of length $k \leqslant N$ blocks such that $Y = \text{HASH}^h_{\langle G \rangle}(M)$ is:*

$$\min_{k \leqslant N} O(k \cdot 2^{\frac{n}{1+\lg[k]}}) \tag{2}$$

If the length of the messages is not restricted, then the minimum in equation (2) is achieved for messages of $k = 2^{\sqrt{n}-1}$ blocks.

So, 10-15 years ago, the lack of an urgent need to hash extremely big files, as well as the difference between the hash sizes of classical hash functions (160 - 512 bits) versus the hash sizes

in the incremental case (2500 - 16000 bits), killed the attractiveness of the concept of incremental hashing. However, there are new trends and new reality. In particular, the latest SHA-3 standard allows arbitrary hash sizes [16], the needs for incremental digesting of big files are increasing, and the cost of storing longer hash values is decreasing. These are the main reasons why we revive the idea of incremental hashing in this paper.

3 Incremental hashing scheme

We will instantiate the incremental hash function from Definition 1 in two practical settings: fixed-size data and variable-sized data. In the fixed-size data setting, the data that needs to be hashed has a predetermined fixed size, and thus the total number of data blocks is fixed. The real use case scenarios can be found in cloud services (Images of Virtual Machines [1,2], cloud storage [12]), digital movies distributions [15], collecting data from sensor networks and many more. In the fixed-size data scenario, the incremental operations that need to be implemented are: block replacement (*replace* operation) and block appending. The other setting is a variable-sized data, such as managing structured data, where additionally the incremental operations for insertion (*insert* operation) and deletion (*delete* operation) of a block should be supported. In order to implement these operations, we will use dynamic data structures.

For both of the aforementioned scenarios, the basic algorithmic description is given in Algorithm 1. The underlying hash primitive and combining operation in the algorithm are the following:

Underlying hash function. The concrete hash function h has to map b bits into k bits (k is a multiple of 64), $h : \{0,1\}^b \to \{0,1\}^k$. A typical cryptographic hash functions such as SHA-1 or SHA-2 outputs a short hash value of 160 or 256 or 512 bits. However, for achieving security levels of 128 or 256 bits we need the value of k to be more than 2000 bits. We use the recently proposed Extendable-Output Functions SHAKE128 and SHAKE256 defined in the NIST Draft FIPS-202 [16]. A concrete definition is given in Section 4.

Combining operation. For the compression function $h : \{0,1\}^b \to \{0,1\}^k$ where k is a multiple of 64 bits i.e. $k = 64 \cdot L$, we use word-wise addition in the commutative group $((\mathbb{Z}_{2^{64}})^{k/64}, \boxplus_{64})$, since it is a very efficient operation on the modern 64-bit CPUs. The operation $\boxplus_{64}$ represents 64-bit word-wise addition of $k/64$ words, and $\boxminus_{64}$ the inverse operation of word-wise subtraction of $k/64$ words.

Algorithm 1 - Incremental hash function
Input. A sequence of blocks $M_1, M_2, \ldots, M_n$, where each M_i has a fixed size of $b - \text{Length}(ID)$ bits.
Output. k bits of hash output.
1. For each block M_i, $i = 1, \ldots, n$, append ID_i to get an augmented block $\overline{M_i} = M_i \| ID_i$;
2. For each $i = 1, \ldots, n$, apply h to the blocks $\overline{M_i}$ to get a hash value $y_i = h(\overline{M_i})$;
3. Combine $y_1, \ldots, y_n$ via a combining group operation $\boxplus_{64}$ to get the final hash value $$y = y_1 \boxplus_{64} y_2 \boxplus_{64} \cdots \boxplus_{64} y_n.$$
4. Output y and store it.

Fig. 1. An algorithm for incremental hash function. Note that when we deal with the fixed size data $ID_i \equiv \langle i \rangle$ and for variable size setting it is $ID_i \equiv (BN_i, ptr_{BN_i})$

Using appropriate parameter values for the formulations above, we have two practical settings:

1. **Fixed-size data.** Hashing data which has a predetermine fixed size. The total number of data blocks is fixed, or can be changed by appending new blocks.
   Block indexing. The data M is virtually divided into a fixed number of blocks $M_1, M_2, \ldots, M_n$. In this case each block M_i has index i and its 64-bit binary encoding represents its unique identifier $ID_i \equiv \langle i \rangle$. This virtual division of data is shown in Figure 2.
   Incremental update operation. Once the hash function is applied on M, there is no need to repeat the same procedure for the whole M, but we can apply an incremental update operation. In this case the only update operation is the following one:
   – *Block Substitution.* This kind of update operation is applied on blocks M_i and M_i', where M_i' is the changed version of the block M_i. In total two block hash operations are applied. The hash update operation is given by Algorithm 2, and its graphical presentation in Figure 5.
   Data overhead. There is no data overhead in this case. The final hash value has a size of k bits. This is the only data necessary to store if we want to recompute the hash.

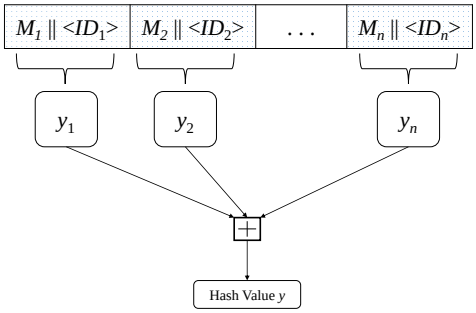

Fig. 2. Construction of incremental hash function for fixed size data.

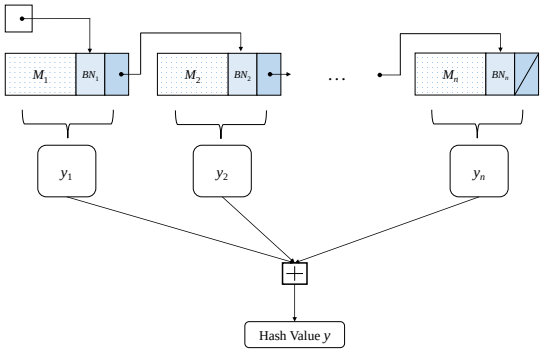

Fig. 3. Construction of incremental hash function for variable size data. The colored parts present the data overhead.

2. **Variable-size data.** Hashing structured data which can have a variable size but where the data blocks always have a unique block identifier that does not change.
   Block indexing. Data is divided into an ordered sequence of blocks $M_1, M_2, \ldots, M_n$. In this case the unique identifier consists of a nonce for that block, denoted as BN and a pointer to the nonce of the next block i.e. $ID_i \equiv (BN_i, ptr_{BN_{i+1}})$). Additionally we need a head for this data structure i.e., a pointer for the first data block M_1 and the pointer of the last block M_n, that points to NULL i.e. $ptr_{BN_{n+1}} = \text{NULL}$. This hybrid data structure is in fact a singly-linked list with direct access via unique nonces and it is shown in Figure 3.
   Incremental update operations. In this case we have the following three update operations:
   – *Block Substitution.* This kind of update operation is applied on block M_i and M_i' (the changed version of the block M_i). The hash update operation is the same as in the case of fixed size data settings, just with a difference in the presentation of ID, i.e. $ID_i = (BN_i, ptr_{BN_{i+1}})$. In total two block hash operations are applied. An algorithm is given by Algorithm 2 and its graphical presentation is given in Figure 5.

Algorithm 2 - Block Substitution
Input. The old block M_i and the new one M_i'. The old hash value y.
Output. k bits of updated hash output.
1. Calculate $y_i = h(\overline{M_i})$;
2. Calculate $y_i' = h(\overline{M_i'})$;
3. Combine y, y_i and y_i' via a combining group operation $\boxplus_{64}$ to get the new updated final hash value $y' = y \boxminus_{64} y_i \boxplus_{64} y_i'$;
4. Output y' and store it.

Fig. 4. An algorithm for incremental hash update operation: Block Substitution.

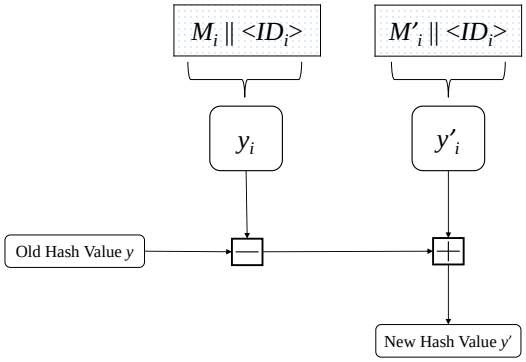

Fig. 5. An update hash operation: Block substitution. Note that when we deal with the fixed data size $ID_i \equiv \langle i \rangle$ and for variable data size it is $ID_i \equiv (BN_i, ptr_{BN_{i+1}})$.

- *Block Insertion.* An insertion of a new block M_j with nonce BN_j after block M_i is performed by changing the unique identifier ID_i. The old value of $ID_i = (BN_i, ptr_{BN_{i+1}})$ is replaced by the new value $ID_i = (BN_i, ptr_{BN_j})$. In total three block hash operations are applied. This operation is given by Algorithm 3, and its graphical presentation in Figure 6.
- *Block Deletion.* To delete a block M_i we need to change the unique identifier of the $i-1$-th block, $ID_{i-1} = (BN_{i-1}, ptr_{BN_i})$ into $ID_{i-1} = (BN_{i-1}, ptr_{BN_{i+1}})$. In total three block hash operations are applied. The hash update operation is given by Algorithm 4, and its graphical presentation in Figure 7.

Data overhead. In this case we have two sub-cases. One is (1) when the data is tightly coupled with the media where it is stored, and the other is (2) if it is flexible. In the sub-case (1) there is no data overhead, and the output is just the k bits of the final hash value. For the sub-case (2) the data overhead is the information about the hybrid singly-linked list with direct access $ID_1, ID_2, \ldots, ID_n$ that is outputed together with the final hash value of size k bits.

3.1 Incremental tree based hash scheme

Merkle was the first one who proposed the tree hashing which can be used for incremental hashing [14]. In his scheme the incrementality is implemented at the cost of storing all intermediate hash values of all tree levels. But that can significantly increase the data overhead of this incremental hash scheme.

Instead of working with full trees, if we want to focus on performance, then we need to limit the tree depth on one or two [6, 7]. More levels in the tree, means more hash stages and more data overhead. Assume for simplicity that the hash tree has depth 1. The graphical representation of the one level tree hashing mode is given in Figure 11. An algorithmic description of the one level tree hashing is given by Algorithm 5.

For this scheme the data M is divided into blocks $M_1, M_2, \ldots, M_n$ and we need the following components:

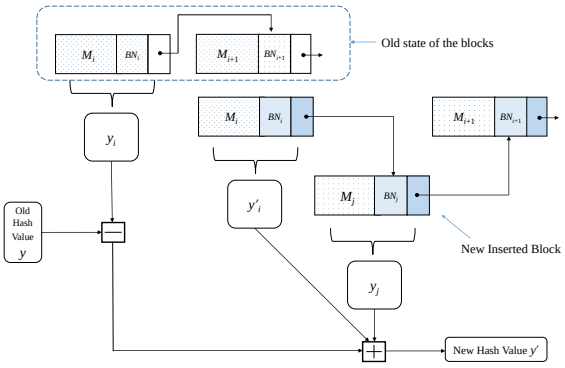

Fig. 6. An update hash operation: Block insertion. Here the block M_j is inserted after the block M_i.

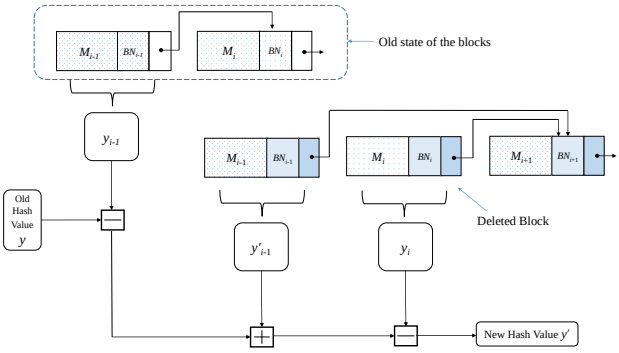

Fig. 7. An update hash operation: Block deletion. Here the block M_i is deleted.

Algorithm 3 - Block Insertion
Input. The block M_i and ID_i after which the insertion will be done; The new block M_j;
Output. k bits of hash output.
1. Calculate $y_i = h(\overline{M_i})$;
2. Calculate $y_j = h(\overline{M_j})$;
3. Transform ID_i into ID'_i i.e. $ID'_i \equiv (BN_i, ptr_{BN_j})$;
4. Calculate $y'_i = h(\overline{M'_i})$, where $\overline{M'_i} = M_i \| ID'_i$;
5. Combine y, y_i, y_j and y'_i via a combining group operation $\boxplus_{64}$ to get the new updated final hash value $y' = y \boxminus_{64} y_i \boxplus_{64} y'_i \boxplus_{64} y_j$;
4. Output y and store it.

Fig. 8. An algorithm for incremental hash update operation in the variable size setting: Block Insertion. Here the block M_j is inserted after the block M_i.

Algorithm 4 - Block Deletion
Input. The block M_i and ID_i that should be deleted; The previous block M_{i-1} and ID_{i-1} from the sequence;
Output. k bits of hash output.
1. Calculate $y_{i-1} = h(\overline{M_{i-1}})$;
2. Calculate $y_i = h(\overline{M_i})$;
3. Transform ID_{i-1} into ID'_{i-1} as $ID'_{i-1} \equiv (BN_{i-1}, ptr_{BN_{i+1}})$;
4. Calculate $y'_{i-1} = h(\overline{M'_{i-1}})$, where $\overline{M'_{i-1}} = M_{i-1} \| (BN_{i-1}, ptr_{BN_{i+1}})$;
5. Combine y, y_{i-1}, y_i and y'_{i-1} via a combining group operation $\boxplus_{64}$ to get the new updated final hash value $y' = y \boxminus_{64} y_{i-1} \boxminus_{64} y_i \boxplus_{64} y'_{i-1}$;
4. Output y and store it.

Fig. 9. An algorithm for incremental hash update operation in the variable size setting: Block Deletion. Here the block M_i is deleted.

One level tree hash function. Any cryptographic hash function h that maps data with arbitrary size into k bits can be used. It has two stages:

- *Hashing tree leaves.* The hash function h maps the leaves M_i of b bits into k bits i.e. $y_i = h(M_i)$.
- *Root hash.* The final hash value y is computed by hashing the concatenation of the hashes of the leaves, i.e. $y = h(y_1 || y_2 || \ldots || y_n)$.

Incremental update operation. Once the root hash is computed, the update operation has the following variants:

- *Block Substitution.* This kind of update operation is applied on blocks M_i and M'_i, where M'_i is a changed version of the block M_i. In total one block hash operation and one root hash computation are performed. This operation is given by Algorithm 6, and its graphical presentation in Figure 13.

Algorithm 5 - One level tree hashing
Input. A sequence of blocks $M_1, M_2, \ldots, M_n$ with fixed size of b bits.
Output. $n * k$ bits of leaves hashes and k bits of the root hash.
2. For each block $M_i, i = 1, \ldots, n$, apply h to them to get a hash value $y_i = h(M_i)$; 3. Concatenate $y_1, \ldots, y_n$ and apply h to the concatenated string to get the root hash value $$y = h(y_1 \| \| y_2, \ldots, y_n).$$ 4. Output y and store it. Store all the intermediate leaves hashes $y_1, y_2, \ldots, y_n$.

Fig. 10. An algorithm for incremental tree based hash function with depth 1.

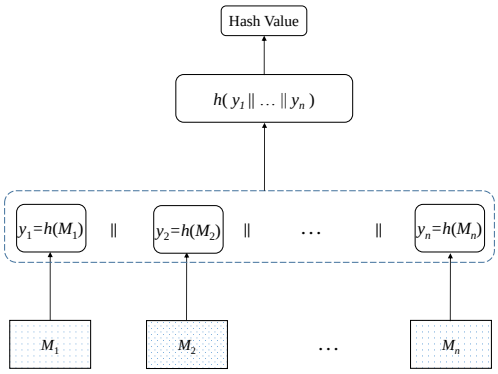

Fig. 11. Incremental hashing using one level tree structure.

– *Block Insertion.* An insertion of a new block M_j after block M_i means insertion of the new hash value $h(M_j)$ after the stored hash value $h(M_i)$ and computation of the root hash. This operation is given by Algorithm 7, and its graphical presentation in Figure 15.
– *Block Deletion.* To delete a block M_i we need to delete the stored hash value of that block and to compute the root hash. It is given by Algorithm 8, and its graphical presentation in Figure 17.

Data overhead. The data overhead is $(n+1) \times k$ bits which comes from n hashes y_i and the final root hash y.

Algorithm 6 - Block substitution in tree hashing
Input. The position i of the old block and the new one M_i'. The old hash value y and all intermediate leaves hashes $y_1, y_2, \ldots, y_n$.
Output. $n * k$ bits of leaves hashes and k bits of the root hash.
1. Calculate $y_i' = h(M_i')$; 2. Replace y_i with y_i'; 3. Concatenate $y_1, \ldots, y_n$ and apply h to the concatenated string to get the root hash value $y = h(y_1 \| \| y_2, \ldots, y_n)$. 4. Output y and store it. Store all the intermediate leaves hashes $y_1, y_2, \ldots, y_n$.

Fig. 12. An algorithm for incremental tree based hash update operation: Block Substitution.

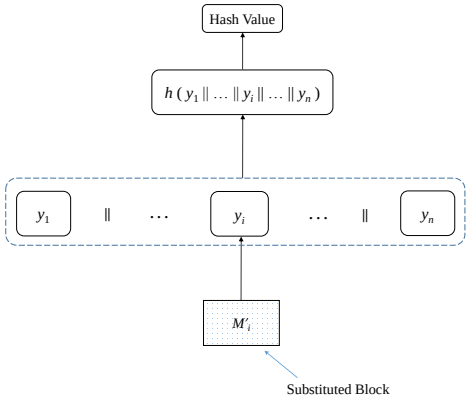

Fig. 13. An update hash operation: Block substitution. Here the block M_i is substituted with the block M_i'.

Algorithm 7 - Block insertion in tree hashing
Input. The position i where the insert should be done. The new block M_j and all intermediate leaves hashes $y_1, y_2, \ldots, y_n$.
Output. $n * k$ bits of leaves hashes and k bits of the root hash.
1. Calculate $y_j = h(M_j)$; 3. Concatenate $y_1, \ldots, y_i, y_j, y_{i+1}, \ldots y_n$ and apply h to the concatenated string to get the root hash value $y = h(y_1 \| \| y_2, \ldots, y_{n+1})$. 4. Output y and store it. Store all the intermediate leaves hashes $y_1, y_2, \ldots, y_{n+1}$.

Fig. 14. An algorithm for incremental tree based hash update operation: Block Insertion, where the block M_j is inserted after the block M_i.

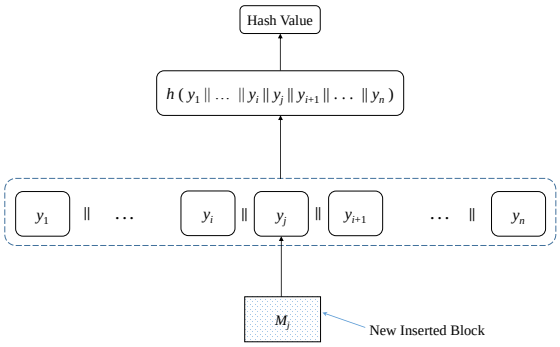

Fig. 15. An update hash operation: Block insertion. Here the block M_j is inserted.

Algorithm 8 - Block deletion in tree hashing
Input. The position i of the block that should be deleted. All intermediate leaves hashes $y_1, y_2, \ldots, y_n$.
Output. $n * k$ bits of leaves hashes and k bits of the root hash.
1. Delete $y_i = h(M_i)$; 3. Concatenate $y_1, \ldots, y_{i-1}, y_{i+1}, \ldots y_n$ and apply h to the concatenated string to get the root hash value $y = h(y_1 \| \| y_2, \ldots, y_{n-1})$. 4. Output y and store it. Store all the intermediate leaves hashes $y_1, y_2, \ldots, y_{n-1}$.

Fig. 16. An algorithm for incremental tree based hash update operation: Block Deletion. Here the block with index i is deleted.

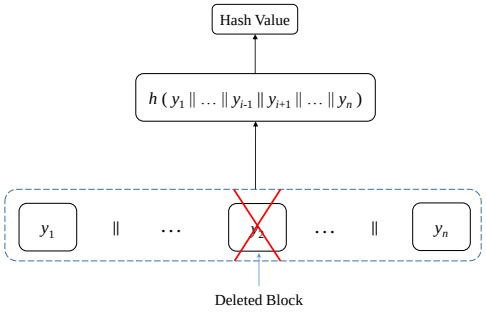

Fig. 17. An update hash operation: Block deletion. Here the block M_i is deleted.

4 Definition of *i*SHAKE

Recently, NIST proposed the *DRAFT SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions* [16], containing definitions for two Extendable-Output Functions named SHAKE128 and SHAKE256. We just briefly mention their definitions:

$$\text{SHAKE128}(M, d) = \text{RawSHAKE128}(M||11, d), \text{ where}$$
$$\text{RawSHAKE128}(M, d) = \text{KECCAK}[256](M||11, d),$$

and

$$\text{SHAKE256}(M, d) = \text{RawSHAKE256}(M||11, d), \text{ where}$$
$$\text{RawSHAKE256}(M, d) = \text{KECCAK}[512](M||11, d).$$

iSHAKE128 is the instantiation of the incremental hash function from Definition 1 where for the hash function h we use SHAKE128 with the output size of 2688 up to 4160 bits. Similarly for iSHAKE256 the output size is in the range of 6528 and 16512 bits.

Using appropriate values for the time complexity of Wagner's generalized birthday attack (Proposition 1), we have the following:

Proposition 2. *Let for iSHAKE128 parameter $k = 2688$ (for iSHAKE256, $k = 6528$) and let the maximal allowed number of blocks be $N = 2^{25}$ ($N = 2^{28}$ for iSHAKE256). Then*

$$\min_{K \leqslant N} O(K \cdot 2^{\frac{k}{1+\lg[K]}}) = 2^{128.385} \quad (2^{253.103}). \tag{3}$$

By a simple multiplication $b \times N$ we have the following:

Proposition 3. *The lower bound of 2^{128} on the complexity of Wagner's generalized birthday attack on* iSHAKE128 *for block sizes of 1 KB, 2 KB and 4 KB for the data blocks M_i, can be achieved by hashing files long 32 GB, 64 GB and 128 GB correspondingly. Also for the 2^{256} security bound for* iSHAKE256 *for block sizes of 1 KB, 2 KB and 4 KB for the data blocks M_i, the hashing files should be long 256 GB, 512 GB and 1 TB correspondingly.*

It is normal to expect that iSHAKE128 would be used for hashing files of size less than 32 GB. In this case there is a tradeoff between the security of finding second-preimage and the size of the hashed files which is expresses by the equation (3). For example, for small size files such as 160 KB the complexity of finding second-preimage is 2^{254} and for files of 1.25 TB, the complexity drops down to 2^{112}. Figure 18 shows that tradeoff for different file sizes.

A similar reasoning applies to iSHAKE256 for hashing files of size less than 256 GB. For example for file sizes of 1 MB the complexity of finding second-preimage is 2^{479} and for files of as much as 8 TB the complexity of finding collisions drops down to 2^{212}. Figure 19 shows that tradeoff for different file sizes.

If the length of the messages is not restricted, then the low bound security of 2^{128} or 2^{256} in equation (3) is achieved for messages with parameter values $k = 4160$ bits for iSHAKE128 and $k = 16512$ bits for iSHAKE256.

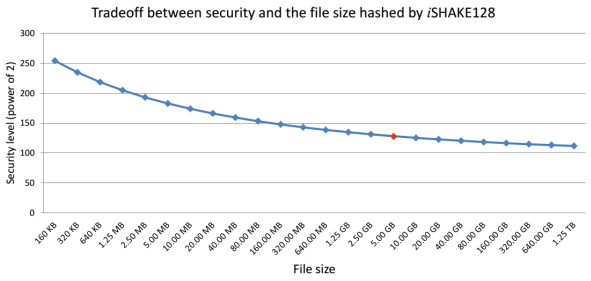

Fig. 18. A tradeoff between finding collisions with the Wagner's generalized birthday attack and the size of the hashed file with iSHAKE128

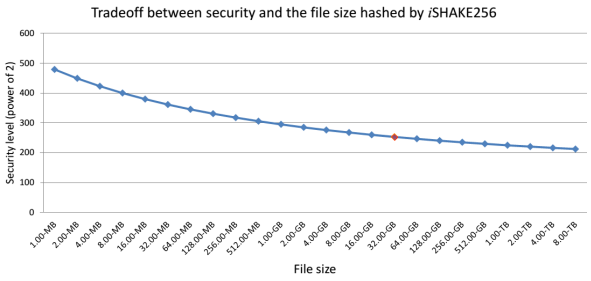

Fig. 19. A tradeoff between finding collisions with the Wagner's generalized birthday attack and the size of the hashed file with iSHAKE256

Incremental Mode	**Incremental Operation**	**Update cost in operations**	**Data overhead**	**Collision between parallel and sequential hashes**
Incremental hashing in fixed size setting	Block Substitution	2 data block hash op.	k-bits of hash output ($2600 \leqslant k \leqslant 16000$)	No
Incremental hashing in variable size setting (without migration)	Block Substitution	2 data block hash op.	k-bits of hash output ($2600 \leqslant k \leqslant 16000$)	No
	Block Insertion	3 data block hash op.		
	Block Deletion	3 data block hash op.		
Incremental hashing in variable size setting (with migration)	Block Substitution	2 data block hash op.	k-bits of hash output ($2600 \leqslant k \leqslant 16000$) + $n \times 64$ bits for the data structure	No
	Block Insertion	3 data block hash op.		
	Block Deletion	3 data block hash op.		
Incremental tree hashing with a tree depth of 1	Block Substitution	1 data block hash operation + 1 hash operation on the intermediate leaves hashes	$n \times k$ bits of intermediate hash values + k bits of final hash output = $(n+1) \times k$ bits ($160 \leqslant k \leqslant 512$)	Yes [13]
	Block Insertion	1 data block hash operation + 1 hash operation on the intermediate leaves hashes		
	Block Deletion	1 hash operation on the intermediate leaves hashes		

Table 1. Comparison analysis between our incremental hash function approach and tree based hashing.

5 Comparison Analysis

In order to show the advantages of our new incremental schemes, we thoroughly compared the different performance aspects of our schemes to suitably chosen tree based hashing schemes. We note that a comparison of our approach to a sequential hashing mode does not make sense both because it is not parallel and because it is not incremental. The only fair comparison would be to schemes with these properties, and currently, tree hashing is the best know method for achieving incrementality.

We compared the update effort for different operations and data overhead that introduces additional storage cost. The results in terms of the needed number of operations, are given in Table 1.

We also compared the performance in terms of speed of iSHAKE and one level tree hashing. Table 2 and Table 3 show an evident speed advantage of iSHAKE over the corresponding incremental tree hashing of as much as 5 to 6 orders of magnitude. The results in the two tables can be interpreted as follows: For a fixed data overhead for both approaches, what amount of data should be digested in one incremental operation. If we assume an equal digest time per data byte, this can be directly translated into a speed comparison between the two. As an example, consider an input file of size 1MB. If we use iSHAKE128 with blocks of 1KB, then the amount of bits that we need to store is just the output of the hash function, or 2688 bits. If we bound the overhead to the same (or approximate) amount of bits for tree hashing, then we can split the message to a maximum

Fixed data overhead of 2688 bits (*i*SHAKE128) and 2816 bits (SHA-3 One Level Hash Tree)												
	1MB			10MB			100MB			1GB		
Block size in KB	1	4	8	1	4	8	1	4	8	1	4	8
Speed advantage (times)	102.4	25.6	12.8	1024	256	128	10240	2560	1280	104857.6	26214.4	13107.2

Table 2. Speed advantage of *i*SHAKE128 in comparison with SHA-3 One Level Hash Tree when one block is updated

Fixed data overhead of 6528 bits (*i*SHAKE256) and 6656 bits (SHA-3 One Level Hash Tree)												
	1MB			10MB			100MB			1GB		
Block size in KB	1	4	8	1	4	8	1	4	8	1	4	8
Speed advantage (times)	85.3	21.3	10.7	853.3	213.3	106.7	8533.3	2133.3	1066.7	87381.3	21845.3	10922.7

Table 3. Speed advantage of *i*SHAKE256 in comparison with SHA-3 One Level Hash Tree when one block is updated

of 10 blocks. In this case, each block will be of size 102.4KB. Thus, in case we have a change of few (up to several hundreds of bytes) that fall in one block of 1KB, *i*SHAKE will rehash only that small block of 1KB, while the tree version of SHA-3 will have to digest significantly bigger block of 102.4KB. This translates to speed advantage of *i*SHAKE of 102.4 times.

6 Conclusion

The need for incremental hashing in the upcoming Zettabyte era is imminent. In this paper we defined two incremental hash functions *i*SHAKE128 and *i*SHAKE256 with security level against collision attacks of 128 and 256 bits respectively. Both are based on the recent NIST proposal for SHA-3 Extendable-Output Functions SHAKE128 and SHAKE256. We presented constructions for two practical settings: fixed size data and variable size data. In the first one, our proposed scheme has obvious advantage in the small overhead that it caries out, compared with any other tree based hash scheme. Moreover the speed-up is present even in the case where the same data overhead is used. In the second practical setting, our proposed scheme behaves approximately the same as tree based hashing when the dynamic data structure representing the unique identifier of the blocks should be stored. In the case where the unique identifiers of the data blocks are tightly coupled with the media where they are stored, the situation is the same as in the fixed size setting. That is, again, our schemes offer show much better performance than tree hashing.

References

1. Amazon web services. An Amazon Company, 2015. `http://aws.amazon.com/ec2/instance-types/`.
2. Virtual machine and cloud service sizes for azure. Microsoft, 2015. `https://msdn.microsoft.com/en-us/library/azure/dn197896.aspx`.
3. Historical cost of computer memory and storage. *hblok.net • Freedom, Electronics and Tech*, February 2013. `http://hblok.net/blog/storage/`.
4. Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In Yvo Desmedt, editor, *CRYPTO*, LNCS, pages 216–233. Springer, 1994.
5. Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In Walter Fumy, editor, *EUROCRYPT*, LNCS, pages 163–192. Springer, 1997.

6. Guido Bertoni, Joan Daemen, Michal Peeters, and Gilles Van Assche. Sakura: A flexible coding for tree hashing. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *Applied Cryptography and Network Security*, LNCS, pages 217–234. Springer International Publishing, 2014.
7. Guido Bertoni, Joan Daemen, Michal Peeters, and Gilles Van Assche. Sufficient conditions for sound tree and sequential hashing modes. *International Journal of Information Security*, (4):335–353, 2014.
8. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - vol. 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.
9. Cisco. Cisco visual networking index: Forecast and methodology, 2012-2017. *White Paper*, May 2013. `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.pdf`.
10. EMC. The emc digital universe study with research and analysis by idc. *Open Report*, April 2014. `http://www.emc.com/leadership/digital-universe/index.htm?pid=home-dig-uni-090414`.
11. Jane K. Hart and Kirk Martinez. Environmental sensor networks: A revolution in the earth system science? *Earth-Science Reviews*, 78(34):177 – 191, 2006.
12. Mark Hornby. Review of the best cloud storage services. TheTop10BestOnlineBackup.com, 2015. `http://www.thetop10bestonlinebackup.com/cloud-storage`.
13. John Kelsey. What Should Be In A Parallel Hashing Standard? NIST, 2014 SHA3 Workshop. Available at `http://csrc.nist.gov/groups/ST/hash/sha-3/Aug2014/documents/kelsey_sha3_2014_panel.pdf`.
14. Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology*, CRYPTO '87, pages 369–378, London, UK, 1988. Springer-Verlag.
15. S Mike. How are digital movies distributed and screened? every question answered! Wirefresh, July 2010. `http://www.wirefresh.com/how-are-digital-movies-distributed-and-screened-every-question-answered/`.
16. NIST. Draft sha-3 standard: Permutation-based hash and extendable-output functions. *FIPS 202*, April 2014. `http://csrc.nist.gov/publications/PubsDrafts.html#FIPS-202`.
17. National Centers for Environmental Information NOAA. Climate Forecast System Version 2 (CFSv2). Available at `https://www.ncdc.noaa.gov/data-access/model-data/model-datasets/climate-forecast-system-version2-cfsv2`.
18. David Wagner. A generalized birthday problem. In Moti Yung, editor, *CRYPTO*, LNCS, pages 288–303. Springer, 2002.