AN EFFICIENT PARALLEL ALGORITHM FOR SKEIN HASH FUNCTIONS

Kévin Atighehchi, Adriana Enache, Traian Muntean, Gabriel Risterucci
ERISCS Research Group
Université de la Méditerranée
Parc Scientifique de Luminy-Marseille, France
email : muntean@univmed.fr

ABSTRACT
Recently, cryptanalysts have found collisions on the MD4, MD5, and SHA-0 algorithms ; moreover, a method for finding SHA–1 collisions with less than the expected amount of work complexity has been published. The National Institute of Standards and Technology (NIST : `http://www.nist.gov/index.html`) has decided that it is prudent to develop a new hash algorithm that shall be referred to as "SHA–3", and will be developed through a public competition (`http://www.nist.gov/itl/csd/ct/hash_competition.cfm`). From the set of proposal accepted for the second round of the competition, the solution we have chosen to explore in this paper for providing an efficient parallel algorithm, is the Skein hash function family. Its design combines speed, security, simplicity, and a great deal of flexibility in a modular package that is easy to analyze. The main reason for parallelizing such an algorithm is to obtain optimal performances when dealing with critical applications which require implementation on multi-core target processors. For parallelizing Skein we have used the tree hash mode which virtually creates one thread for each node of the tree. We claim that this is one of the first parallel implementation with associated performances evaluation of this SHA-3 candidate algorithm.

KEY WORDS
Skein, SHA-3, parallel cryptography, secure communicating systems.

1 Introduction

Skein [10] is a new family of cryptographic hash functions, candidate in the SHA-3 competition [13] (see `http://www.nist.gov/itl/csd/ct/hash_competition.cfm`). Its design combines speed, security, simplicity, and a great deal of flexibility in a modular package.

In 2005, security flaws were identified in SHA-1 [14][1], indicating that a stronger hash function is quite desirable. The NIST (: `http://www.nist.gov/index.html`) then published four additional hash functions in the SHA family, named after their digest length (in bits) : SHA-224, SHA-256, SHA-384, SHA-512, also known as the SHA-2 family which does not share the weakness of SHA-1.

The SHA-3 project was announced in Nov. 2, 2007 and was motivated by the collision attacks on commonly used hash algorithms (MD5 and SHA-1). The new hash function shall not be linked to its predecessor, so that an attack on SHA-2 is unlikely applicable to SHA-3.

Since the new proposals are intended to be a drop-in replacement, some properties of the SHA-2 family must be preserved. However, some new properties and feature must be provided by SHA-3 : it may be parallelizable, more suitable for certain applications, more efficient to implement on actual platforms, or may avoid some of the incidental "generic" properties (such as length extension) of the Merkle-Damgard construct ([8] [7]) that may often result in insecure applications.

The work described in this paper refers to sequential and parallel algorithms for Skein cryptographic hash functions, analysis, tests and optimizations. Our new approach for parallelizing Skein is by using the tree hash mode which creates one thread task for each node of the tree.

The paper is structured as following : Section 2 and 3 give a brief description of Skein and a detailed description of its Tree Mode, section 4 presents the potential approaches for parallelism. Then we present the work done for parallelizing the hash algorithm : speedup, implementation description, testing and elements of performance evaluation. Finally, the conclusion and some recommendations for future work are given in the last chapter.

2 Brief description of Skein

The structure of the Skein algorithm (Unique Block Iteration, UBI chaining) has its origin in the sponge hash functions [5][6].

Definition [6] : Let A be an *alphabet* group which represent both input and output characters and C be a finite set whose elements represents the inner part of the state of a sponge. A sponge function takes as input a variable-length string p of characters of A that does not end with 0 and produces an infinite output string z of characters of A . It is determined by a transformation f of $A \times C$.

Skein works like a Sponge function : it takes a variable length string of characters as its input and produces an infinite output string (it can generate long output by using the threefish block cipher in counter mode). The capacity of the sponge function is replaced in Skein by a tweak which is unique for each block.

A sponge function works by absorbing and squeezing its input; these steps in Skein correspond to the processing stage and the output function. The main difference between sponge functions and UBI chaining is that the input chaining value for the UBIs of the output function is the same, while in the sponge function it depends on the previous state.

Skein also has a configuration block that is processed before any other blocks.

The Skein family of hash functions use three different internal state size : 256, 512 and 1024 bits :

- Skein-512 : the primary proposal; it should remain secure for the foreseeable future.
- Skein-1024 : the ultra-conservative variant. If some future attack managed to break Skein-512, it should remain secure. Also, with dedicated hardware it can run twice as fast as Skein-512.
- Skein-256 : the low memory variant. It can be implemented using about 100 bytes of memory.

Skein uses Threefish as a tweakable block cipher, with the Unique Block Iteration (UBI) chaining mode to build a compression function that maps an arbitrary input size to a fixed output size. Figure 6 shows a UBI computation for skein-512 on a 166-byte (three blocks) input, which use three calls to Threefish-512.

The core of Threefish is a non-linear mixing function called MIX that operates on two 64-bit words. This cipher repeats operations over a block a certain number of rounds (72 for Threefish-256 and Threefish-512, 80 for Threefish-1024), each of these rounds being composed of a certain number of MIX functions (2 for Threefish-256, 4 for Threefish-512 and 8 for Threefish-1024) followed by a permutation. A subkey is injected every four rounds. From a parallelization point of view, each mix operation could be assigned to one thread since, for a given round, they operate on different 128-bits blocks. In theory, we could achieve a maximum speedup of 2 with Threefish-256, 4 with Threefish-512 and 8 with Threefish-1024 provided that at each round waiting times (for instance for scheduling) between threads are quite negligeable, a permutation being performed at the end of each round.

Skein is built with three elements : the block cipher (Threefish), the UBI, and an argument (containing a configuration block and optional arguments). The configuration block is mainly used for tree hash, while optional arguments make it possible to create different hash functions for different purposes, all based on Skein.

Skein can work in two modes of operation, which are built on a chaining of UBI operations :

- Simple hash : it take a variable sized input and return the corresponding hash. It's a simple and reduced version of the full Skein mode. For instance with a hash processus where the desired output size is equal to the internal state size it consists of three chained UBI functions, the first process the configuration string, the second the message and the latter is used to supply the output.
- Full skein : The general form of Skein admits key processing, tree hashing and optional arguments (personalization string, public key, key identifier, nonce...). The tree mode replaces the single UBI call which processes the message by a tree of UBI calls.

The result of the last UBI call (the root UBI call in case of tree processing) is an input to the Output function which generates a hash of desired size.

The followings section presents the two modes of Skein intended to be widely used.

3 Simple Hash Mode

3.1 Specification

This section recalls the specification of the Simple Hash mode (refer to the Skein paper for more information).

A simple Skein hash computation has the following inputs :

N_b The internal state size, in bytes. Must be 32, 64, or 128.

N_o The output size, in bits.

M The message to be hashed, a string of up to $2^{99}-8$ bits ($2^{96}-1$ bytes).

Let C be the configuration string for which $Y_l = Y_f = Y_m = 0$. We define :

$$\begin{aligned} K' &:= 0^{N_b} \quad \text{a string of } N_b \text{ zero bytes} && (1)\\ G_0 &:= UBI(K', C, T_{cfg}2^{120}) && (2)\\ G_1 &:= UBI(G_0, M, T_{msg}2^{120}) && (3)\\ H &:= Output(G_1, N_o) && (4) \end{aligned}$$

where H is the result of the hash.

If the three parameters Y_l, Y_f and Y_m are not all zero, then the straight UBI operation of the equation (3) is replaced by a tree of UBI operations as defined in the section 4.1.

3.2 Remarks

UBI is a chaining mode for the Threefish cipher, so there is no underlying parallelism other than that which can be obtained into the Threefish block encryption as explained above. The Output operation of the equation (4) is in fact a sequence of UBI operations iterated according to a counter mode, so this last one can be done in parallel by assigning one UBI operation to one thread.

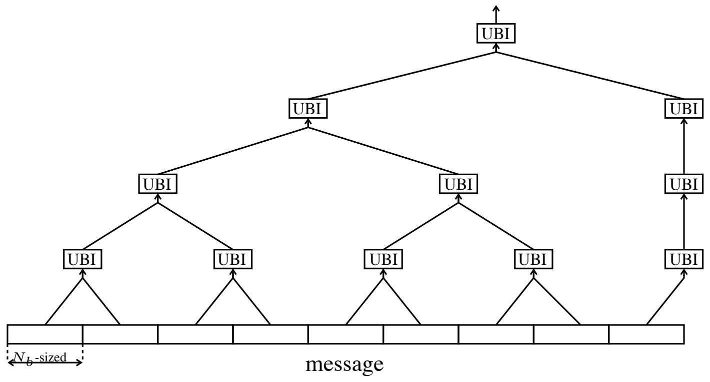

FIG. 1: Tree hashing with $Y_l = Y_f = 1$

4 Hash Tree Mode

4.1 Specification

This section recalls the specification of the Hash Tree mode (refer to the Skein paper for more information), a mode specifically designed for parallel implementations.

Tree processing vary according to the following input parameters :

- Y_l The leaf size encoding. The size of each leaf of the tree is $N_l = N_b 2^{Y_l}$ bytes with $Y_l \geq 1$ (where N_b is the size of the internal state of Skein).
- Y_f The fan-out encoding. The fan-out of a tree node is 2^{Y_f} with $Y_f \geq 1$. The size of each node is $N_n = N_b 2^{Y_f}$.
- Y_m The maximum tree height ; $Y_m \geq 2$. If the hieght of the tree is not limited this parameter is set to 255.
- G_0 The input chaining value and the output of the previous UBI function.
- M The message data.

We define the leaf size $N_l = N_b 2^{Y_l}$ and the node size $N_n = N_b 2^{Y_f}$.

We first split the message M into one or more message blocks $M_{0,0}, M_{0,1}, ..., M_{0,k-1}$, each of size N_l bytes and the latter possibly smaller. We now define the first level of tree hashing by :

$$M_1 = \mathop{\Big\Vert}_{i=0}^{k-1} \mathrm{UBI}(G, M_{0,i}, iN_l + 1 \cdot 2^{112} + T_{msg} \cdot 2^{120})$$

The rest of the tree is defined iteratively. For any level $l = 1, 2, ...$ we use the following rules :

1. If M_l has length N_b then the result G_0 is defined by $G_1 = M_l$.
2. If M_l is longer than N_b bytes and $l = Y_m - 1$ the we have almost reached the maximum tree height. The result is then defined by :

$$G_1 = \mathrm{UBI}(G, M_l, Y_m \cdot 2^{112} + T_{msg} \cdot 2^{120})$$

3. If neither of these conditions holds, we create the next tree level. We split M_l into blocks $M_{l,0}, M_{l,1}, ..., M_{l,k-1}$, where all blocks are of size N_n, except the latter possibly smaller. We then define :

$$M_{l+1} = \mathop{\Big\Vert}_{i=0}^{k-1} \mathrm{UBI}(G, M_{l,i}, iN_n + (l+1) \cdot 2^{112} + T_{msg} \cdot 2^{120})$$

and apply the above rules to M_{l+1} again.

The result G_1 is then the chaining input to the output transformation.

4.2 Sequential implementation

The straightforward method consists to implement this algorithm as it is presented in its specifications. This implementation would constitute a scheduling method for the nodes processing that we might call *Lower level and leftmost node first* (that we will call for short *Lower level node first*). Such an implementation has the disadvantage of consuming lots of memory. For instance if we take $Y_l = 1$, we should be ready to store in memory up to half of the message we want to hash, which may be impossible for long messages.

There is an effective algorithm (refer to [11]) which computes a value of height h node, while storing only up to $h + 1$ hash values. The idea is to compute a new parent hash value as soon as possible before continuing to compute the lower level node hash values, so that we can call this method *heigher level node first*. The interest of this method, which maintains a stack in which are stored the intermediate values, is to rapidly discard those that are no longer needed. This stack, empty at the beginning, is used as follows : we use (push) leaf values one by one from left to right and we verify at each step if the last two values on the stack are of the same height or not. If such is the case, these last two values are popped and the parent hash value is computed and pushed is the stack, otherwise we continue to push a leaf value and so on. Note that we could use a 2 hash-sized buffer at each level (1 to h) instead of a unique stack, even though it is useless in such a sequential implementation. This algorithm can be applyed to Skein trees on the condition that we include a special termination round seeing that they are not necessarily full trees (as we can see in figure 1).

We assume existence of followings elements :

- **oracles :**
  - $S(n)$ which returns the node value.
  - $LEAFCALC(l)$ which returns a pair of elements $(S(n_l), t)$ where $S(n_l)$ is the leaf value (a N_b-sized block of the message) and t a binary variable indicating whether it is the last leaf (1) or not (0).
  - $TOPNUMBER(s)$ which returns the number of top nodes on the stack of equal height.

  - $SIZE(s)$ which returns the number of staked nodes.
- **variables :**
  - l : a counter which start from 0, the leftmost leaf.
  - np : the number of nodes processed.
  - cl : the current level.
  - t : the terminate variable
  - D_l : the internal node degree at level 1.
  - D_n : the internal node degree at level > 1
  - s : the stack.
- **other notations :** n_l, n_p, n_r, n_i denotes respectively a leaf node, a parent node, a root node and the i-th child of a parent node.

Then an algorithm for Skein tree hashing is the following :

Algorithm 1 Skein tree hashing using a stack (without Output transformation)

```
1:  Set $l = 0$, $np = 0$, $cl = 0$, $t = 0$, $D_l = 2^{Y_l}$, $D_n = 2^{Y_f}$ and $s = [\,]$.
2:  if $cl < Y_m - 1$ and ($t \neq 0$ or ($TOPNUMBER(s) = D_l$ and $cl = 0$) or ($TOPNUMBER(s) = D_n$ and $cl > 0$)) then
3:      Increment $cl$
4:      Compute $N = TOPNUMBER(s)$
5:      for $i = N - 1$ to 0 do
6:          Pop $S(n_i)$ from $s$
7:      end for
8:      Concatenate $S = ||_{i=0}^{N-1} S(n_i)$
9:      Compute $S(n_p) = \text{UBI}(G, S, \max(np - N, 0) \cdot N_b + cl \cdot 2^{112} + T_{msg} \cdot 2^{120})$
10:     Push $S(n_p)$ onto $s$
11:     Compute $np = \lfloor \frac{np}{N} \rfloor$
12: else
13:     Compute $(S(n_l), t) = LEAFCALC(l)$
14:     Push $S(n_l)$ onto $s$
15:     Set $np = l$
16:     Increment $l$
17: end if
18: Compute $R = TOPNUMBER(s)$
19: if $t \neq 0$ and $R = SIZE(s)$ and $cl > 0$ then
20:     Increment cl
21:     for $i = R - 1$ to 0 do
22:         Pop $S(n_i)$ from $s$
23:     end for
24:     Concatenate $S = ||_{i=0}^{R-1} S(n_i)$
25:     Compute $S(n_r) = \text{UBI}(G, S, cl \cdot 2^{112} + T_{msg} \cdot 2^{120})$
26:     Return $S(n_r)$
27: else
28:     Loop to line 2
29: end if
```

5 Approaches for parallelism

In the following sections, we denote N_t the number of threads. These threads are then indexed $0, 1, ..., N_t - 1$.

We assume that $k_1 = k$ is the number of N_b-sized blocks of level 1. We need to define a recursive sequence starting at an initial value k_2 by :

$$k_2 = \lceil \frac{k}{2^{Y_f}} \rceil \text{ and } k_i = \lceil \frac{k_{i-1}}{2^{Y_f}} \rceil$$

There exists an index v for wich $k_v = 1$. The tree height is then $p = \min(v, Y_m)$. The bytes string produced at a level i of the tree (excepted the base level $i = 0$) can be splitted into k_i blocks $M_{i,0}, M_{i,1}, M_{i,2}, ..., M_{i,k_i}$ of size N_b.

5.1 Lower level node priority

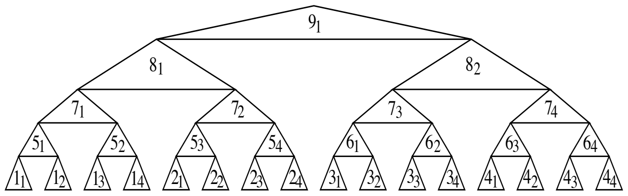

FIG. 2: Lower level node first
$(Y_f = Y_l = 1,\ Y_m = 255,\ N_t = 4)$

This method consists in treating the tree levels completely and successively. It should, theoretically, offer the best performances in practice because of the near absence of synchronization between threads, out of synchronization due to inevitable dependencies between worker threads and main thread which provides the input data. Let's observe the figure 2 in which a job is indexed as i_j where i denotes the step and j the index of the assigned thread. In this example if we count the jobs on each level from left to right, then we can assign a job j to a thread indexed $j \mod N_t$. This method, although intended to get the best performances, has the drawback of requiring huge amount of memory as explained above.

5.2 Higher level node priority

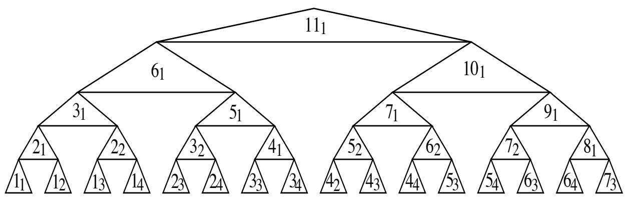

FIG. 3: Higher level node first
$(Y_f = Y_l = 1,\ Y_m = 255,\ N_t = 4)$

This method consists, for a thread, to assign [1] the higher level node among all those that may be assigned

[1]An assignment of a node to a thread means that this last one is responsible for producing the hash value of this node thanks to the hash values of its children.

to it. There are two possible ways to apply this method, in a deterministic way, in which case a thread indexed j must take into account at the current step the predictable behavior of the threads indexed $0, 1, 2, ..., j-1$, and a non deterministic way, in which case the higher level node is assigned to the first ready thread. The advantage of this approach is to conserve at best the memory usage during the hash process. Indeed, if we denote N_t the number of threads, p the height of the produced tree, we define a recursive sequence by :

$$n_1 = N_t,\ n_{p-1} = k_{p-1} \text{ and}$$

$$n_i = \max(\lfloor \frac{n_{i-1}}{2^{Y_f}} \rfloor, 2^{Y_f}) \text{ for } i \in [\![1, p-1[\![$$

Thus, we can use buffers at each level of the tree, of respectively $n_1 \cdot N_b$ bytes for the first one, $n_2 \cdot N_b$ bytes for the second, and so on. If the Y_m parameter does not constrain this tree, then a memory space of only $N_b \cdot \sum_{i=1}^{p-1} n_i$ bytes seems sufficient. Note that this estimate does not represent the maximal memory space really used at every moment because buffers may be not entirely filled. In fact it is much lower, the worst case occurring when all the buffers are not empty and not necessarily filled. Furthermore, we are not sure that first level buffers at the bottom of the tree have the length a multiple of 2^{Y_f} (take an example with 5 or 6 threads), so that we have to consider them as cyclic buffers. Note also, this is only the recommanded memory for the produced/consumed digests at the nodes, we must add the data input buffer cost and some other data such as mutexes, semaphores or eventual conditional variables needed for synchronization. Also, if we look at the figure 3, we must be careful that thread 3_2 does not produce a digest before the thread 3_1 has finished to consum the digest produced by 2_1, forcing these last ones to perform a data recopy in order not to lose too much parallelism.

This scheduling method is not easy to implement and it is not clear it offers good performances in practice because of the large number of synchronization mechanisms required. Therefore, a deterministic case implementation should be avoided since the threads might wait for themselves uselessly. Finally, we can notice (observe the right sides of the trees on the figure 3) that the total number of steps increases compared to the first scheduling method because of the number of purely sequential steps, which can approach the height of the tree. This number of additional steps depends on the configuration of the tree and the number of threads generated. This inherent unbalanced loading between threads, due to this scheduling approach, can therefore reduce the speedup.

5.3 Priority to a fixed number of higher level and same level nodes

A third method takes again the idea of using a stack (see section 4.2), but apply it to an arbitrary number of threads. For N_t threads, we use at each level buffers which can receive $N_t \cdot 2^{Y_f}$ blocks of size N_b, excepted the base

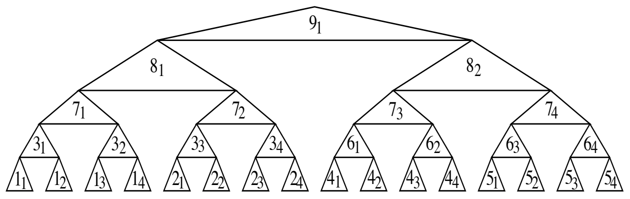

FIG. 4: Fixed number and same level nodes first
($Y_f = Y_l = 1,\ Y_m = 255,\ N_t = 4$)

level where the leaves are the input data buffer blocks. These threads have to compute for the same level $N_t \cdot 2^{Y_f}$ nodes values in order to move up and compute N_t nodes values at the next level. Once these N_t nodes values are computed, the $N_t \cdot 2^{Y_f}$ children values are removed. If the current level occupied by threads is greater than 1 and the lack of ressources on the level below prevents them finishing the compution of the $N_t \cdot 2^{Y_f}$ blocks, then they return down to level 1, otherwise they continue, and so on (see the figure 5). Obviously, a termination phase for the end of the message has to be foreseen, in which case buffers' contents less than $N_t \cdot 2^{Y_f}$ blocks have to be processed. Furthermore, top levels may need narrower buffers (see figure 4) and when the $Y_m = p$ parameter constraints the tree, the penultimate level buffer must always have a capacity of k_{p-1} blocks. Such an algorithm requires about N_t times more memory for internal nodes values storage than the sequential one using levels buffers instead of a stack.

In theory this *fixed number and same level nodes first* scheduling is as efficient as the *lower level node first* version described above section 5.1 and conserve also the first method's speedup. Just like the *higher level node first* scheduling, any recurrent waiting between threads should reduce performances, but, it seems simpler to implement.

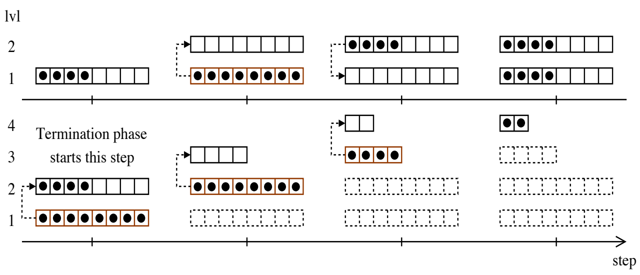

FIG. 5: levels buffers filling for a message of $32N_b$ bytes
($Y_f = Y_l = 1,\ Y_m = 255,\ N_t = 4$)

5.4 Assigning subtrees

If we consider the case where a thread is processing a subtree, the user could control an additional parameter, the height h_s of the subtree. Threads should be able to process full subtrees, not necessarily full subtrees at the right side of the original tree and finally a last top subtree of height lower or equal to h_s.

Although the use of sub-trees may slightly unbalance loading between threads, it would have the advantage of reducing the total number of dependencies during the execution and improve performances. Note that the effect of this parameter could be similar to the Y_f and Y_l effect but the user might have an interest in treating a tree of a particular configuration, for instance (s)he might want to check a hash issued from a tree of a particular configuration.

The scheduling policies outlined above can always be applied.

6 From Simple hash to Tree hash

The Tree mode will allow to calculate a hash in an incremental way ; that is to say it will allow to update the hash whenever a new data field is concatenated before or after the actual data. It also offers the possibility of authentication and update of the hash when the data to be authenticated is never truncated or concatenated with additional fields (e.g : memory authentication [9], statical dictionary).

The object of this section is to confront the Simple Hash Mode to the Tree Hash Mode. Then, we estimate the followings speedup :

– parallel tree processing compared to the sequential UBI operation of the equation (3).

– parallel tree processing compared to the sequential tree processing.

In each case, we give the potential speedup for which the number of hardware processing units is large enough to not be a limiting factor.

6.1 Elementary operations and time complexity

The time complexity of a function UBI for the evaluation UBI(G, M, T_s) can be described by :

$$T(l) = a \cdot (\lceil \frac{l}{8N_b} \rceil \cdot \mathbb{1}_{l>0} + \mathbb{1}_{l=0}) + b$$

where l is the message length in bits, the constant a is the time complexity for a block ciphering operation and the constant b corresponds to the time complexity for the initialization operations such as padding operation and argument evaluation.

We can safely say that b is much lower than a, then we parametrize b by αa with $\alpha \in [0, 1]$ and define this time complexity by :

$$\overline{T}(n) = a \cdot (n + \alpha) \qquad (5)$$

where $n = \lceil \frac{l}{8N_b} \rceil$ is not zero.

The block ciphering operation by Threefish is then considered as an elementary operation, it constitutes one iteration of the UBI chaining mode, framed in red in the figure 6 which gives an example of a three-block message hashing using UBI.

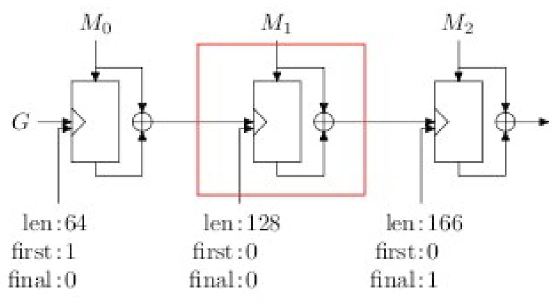

FIG. 6: Processing a message of 166 bytes

6.2 Comparing the two algorithms

The number of basic operations of a single UBI application is $n + \alpha$. Let's consider the Tree hash mode in the best case where there is a $k \in \mathbb{N}$ such that $n = 2^{Y_l} k$ and also a $h \in \mathbb{N}$ such that $k = 2^{Y_f h}$. The associated tree is a complete tree of 2^{Y_f} nodes and of deep h. If $h \leq Y_m - 1$, the number of basic operations without the contributions in α, is :

$$N_s^{nc}(n) = n \cdot \left(1 + \frac{2^{Y_f - Y_l}}{2^{Y_f} - 1}\right) - \frac{2^{Y_f}}{2^{Y_f} - 1}$$

For $h > Y_m - 1$, this number is :

$$N_s^c(n) = n \cdot \left(1 + \frac{2^{Y_f(Y_m-1)} - 1}{2^{Y_l + Y_f(Y_m-2)}(2^{Y_f} - 1)}\right)$$

The associated contributions in α are :

$$\begin{aligned} M_s^{nc}(n, \alpha) &= n \cdot \frac{2^{Y_f - Y_l} - 1}{2^{Y_f} - 1} \cdot \alpha \\ M_s^c(n, \alpha) &= (n \cdot \frac{2^{Y_f(Y_m-1)} - 1}{2^{Y_l + Y_f(Y_m-2)}(2^{Y_f} - 1)} + 1) \cdot \alpha \end{aligned}$$

Following the model of the equation (5), the time complexity $\overline{T_t}(n)$ of a calls tree UBI for n of the form $2^{Y_l + Y_f h}$ is :

$$\overline{T_t}(n) = \begin{cases} a \cdot (N_s^{nc}(n) + M_s^{nc}(n, \alpha)) & \text{if } h \leq Y_m - 1 \\ a \cdot (N_s^c(n) + M_s^c(n, \alpha)) & \text{otherwise} \end{cases}$$

From $N_s^{nc}(n)$ and $N_s^c(n)$, for a fixed message size, one can observe that :

– if $Y_l = Y_f = 1$ and $h \leq Y_m - 1$ then the maximum number of basic operation is reached. Such parameters can be of interest if we can use $\frac{n}{2}$ parallel processing units.

– if $Y_m = 2$ and $h > Y_m - 1$, Y_f is not used, the number of operations are function only of Y_l and are optimized for $Y_l = 1$.

– Augmenting Y_l will minimize the overhead of the number of operations but requires the use of larger buffers, mainly in a multi-threaded implementation.

– The choice of the two parameters Y_l and Y_f influences the deep of the tree and therefore the memory usage overhead. Augmenting these two parameters will decrease the overhead. A constraint Y_m on the tree deep is not affecting the amount of memory required for the hash (but, in case of the use of mechanisms for memory authentication one needs to store the intermediary levels of the tree).

The choice of parameters Y_f, Y_l and Y_m depends on the degree of parallelism for a particular implementation, the synchronization primitives of a specific implementation, and, the constraints associated with the memory requirements.

Now we consider the optimal configuration with k processing units. If $h \le Y_m - 1$, the number of operations by processing unit, without the contribution in α, shall be :

$$N_p^{nc}(n) \quad = \quad 2^{Y_l} + h \cdot 2^{Y_f}$$

For $h > Y_m - 1$, this number is :

$$N_p^{c}(n) \quad = \quad 2^{Y_l} + (Y_m - 2) \cdot 2^{Y_f} + 2^{Y_f(h - Y_m + 2)}$$

The associated contributions in α are :

$$\begin{aligned} M_p^{nc}(n, \alpha) &= (h+1) \cdot \alpha \\ M_p^{c}(n, \alpha) &= Y_m \cdot \alpha \end{aligned}$$

Following always the model of the equation (5), the time complexity $\overline{T_t^p}(n)$ of a calls tree UBI performed by a system with at least k processing units and for n of the form $2^{Y_l + Y_f h}$ is :

$$\overline{T_t^p}(n) = \begin{cases} a \cdot (N_p^{nc}(n) + M_p^{nc}(n, \alpha)) & \text{if } h \le Y_m - 1 \\ a \cdot (N_p^{c}(n) + M_p^{c}(n, \alpha)) & \text{otherwise} \end{cases}$$

We define $PS_{pt/u}(n)$ the potential speedup of the Tree mode in an optimal configuration system, for n of the form $2^{Y_l + Y_f h}$ and when compared to the Simple hach (a straight UBI operation), by :

$$PS_{pt/u}(n) = \frac{\overline{T}(n)}{\overline{T_t^p}(n)}$$

If we fix a value for α, we can deduce the following result :

Lemma 1. *In the general case, for all n but a not constrained tree in height, $PS(n) \in \Omega(\frac{n}{\log n})$, so the potential speedup increases with the size of the message.*

Note that if we take $\alpha = 0$, for n of the form $2^{Y_l + Y_f h}$ and $h \le Y_m - 1$, we have

$$\begin{aligned} PS_{pt/u}(n) &= \frac{n}{N_p^{nc}(n)} \\ &= \frac{n \cdot \ln(2^{Y_f})}{(\ln(n) - \ln(2^{Y_l})) \cdot 2^{Y_f} + \ln(2^{Y_l}) \cdot 2^{Y_l}} \end{aligned}$$

Similarly, in this case and if $h > Y_m - 1$, the speed-up is then $PS_{pt/u}(n) = \frac{n}{N_p^c(n)}$.

6.3 Speed-up of the Tree hash

We define $PS_{pt/st}(n)$ the potential speedup of the Tree mode in an optimal configuration system, for n of the form $2^{Y_l + Y_f h}$ and when compared to a one processor implementation, by :

$$PS_{pt/st}(n) = \frac{\overline{T_t}(n)}{\overline{T_t^p}(n)}$$

Note that if we take $\alpha = 0$, for n of the form $n = 2^{Y_l + Y_f h}$ and $h \le Y_m - 1$, then the potential speed-up of Hash Tree when compared to a single processor implementation is $PS_{pt/st}(n) = \frac{N_s^{nc}(n)}{N_p^{nc}(n)}$ and in case of a constrained tree with $h > Y_m - 1$ this last one is $PS_{pt/st}(n) = \frac{N_s^{c}(n)}{N_p^{c}(n)}$.

7 Java implementation

Thread scheduling in Java. There are two kinds of schedulers : green and native. A green scheduler is provided by the Java Virtual Machine (JVM), and a native scheduler is provided by the underlying OS. In this work, tests were performed on a Linux operating system with a JVM using the native thread scheduler. This provides a standard round-robin strategy.

Threads can have different states : initial state (when not started), runnable state (when the thread can be exaecuted), blocked state and terminating state. The significant point is when a thread is in the blocked state, i.e waiting for some event (for example, a specific I/O operation or waiting for a signal notification). In this case, the thread is not consuming CPU resources at all, meaning that having a large number of blocked threads doesn't impact much on the efficiency of the system.

7.1 Class organization

Skein's implementation is composed of several classes, splitting the core functionality and special code of the algorithm :

- Main algorithm : Skein core is implemented as three classes, Skein256, Skein512 and Skein1024. They all provide the same interface, and support Simple Hash as well as Full Skein. Tree and thread management is done in other support classes.
- Tree and thread support : different class were implemented, each representing the different kind of nodes we can have in the hash tree. All of those classes have a similar interface :
  - TreeNode : used when hashing a file with a hash tree
  - NodeThread : used in a hash tree with one thread per node

- NodeJob : used in a hash tree with one job per node, and process those jobs with a thread pool
- ThreadPool : manage the pool of thread used with NodeJob instance.
- Other classes are needed for the pipeline implementation of Skein : SimplePipeFile and TreePipeFile are used, the first one for Simple Hash and the second one for Full Skein with tree.

In addition to these classes, two "main" classes where written. The Speed class is used to test the speed-up of the algorithm, and the Test class implements direct calls to the different hash methods on different inputs, as well as running tests provided in the Skein reference paper.

7.2 Sequential Skein

To do a Simple Hash, one simply call the update() and digest() methods on a Skein class. There are also methods available to perform the Tree hash computation sequentially.

7.3 Parallel Skein

1. *One thread per node*
   We create one thread per node, and let the scheduler handle how they are executed (figure 7).

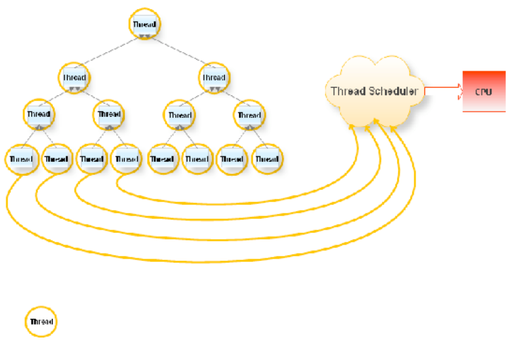

FIG. 7: Parallel Skein using one thread per node

2. *One thread per node with a thread pool*
   To optimize the first implementation we created a thread pool that has a fixed number of threads. Those threads can accept jobs in a FIFO manner and executes them (figure 8).
3. *Pipe input file*
   Because most of the time people hash many files at a time, we have decided to implement a "pipe" that applies the hash function in parallel for each input file. This implementation uses the thread pool with a thread count equal to the number of files to be hashed. It is implemented using both the Simple hash and the Tree hash methods.

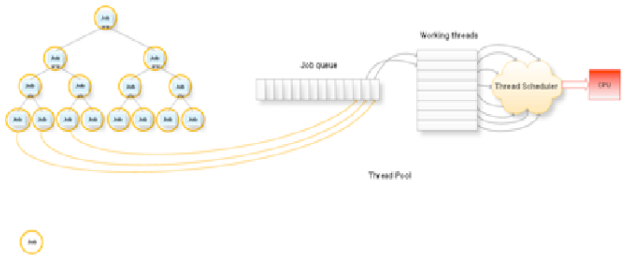

FIG. 8: Parallel Skein using one thread per node and a thread pool

8 Testing and performance

For testing the performance evaluations of the various implementations we wrote the Speed class, used in conjunction with the YourKit profiling tool [2], which allow monitoring the CPU and memory usage. The test were done using a Dell Latitude D830, Intel(R) Core(TM)2 Duo CPU T7500 @ 2.20Ghz, 2GB RAM, L2 cache size 4MB with Ubuntu 9.10. In order to determine the speed of the implementation, it was tested using a file of 700MB. The performance results are illustrated in the chart below :

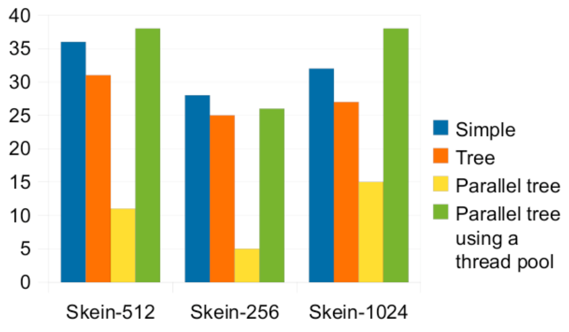

FIG. 9: Processing speed (in MB/s) comparison between the Skein versions

Although the fastest version should theoretically be Skein-1024, from this chart we can deduce that the version using a block size of 512 bits is faster. That is because the computer used for these tests has a 64 bit processor. Furthermore, the slowest is Skein-256 but this one would be the fastest on a 32 bit CPU.

The tests using the YourKit profiler showed the parallel versions use more heap memory, but the *CPU load* stay close to 100%, meaning both processors available on the platform are used at their full capacity.

In termes of execution time the *One Thread per Node* implementation is the slowest. This is mainly due to the overhead of the thread scheduler and poor memory management. Creating a lot of threads for single use is very costly : it triples the heap memory usage in comparison to the sequential version, but is not very effective. It's also slower than the sequential version, due to excessive synchronization required between threads.

To optimize this parallel implementation, we created a thread pool class. With a limited number of threads, we use less memory, although it's still more when compared to the sequential version. On the other hand, the execution time for this implementation is almost half of the sequential version.

The last parallel implementation uses the thread pool class with as many threads as input files. This is an efficient implementation as the execution time is half of the sequential version, and the difference for the used heap memory is quite small – for the simple mode hash it is almost 0.2MB. Also an important fact is that when running the tests on a computer with two CPUs, having two input files mean that both threads stay in the runnable state. We're then using the maximum capacity of the platform this way.

Comparing the three versions of Skein implementations, we noticed that for the simple sequential implementation the amount of heap memory used is almost the same. A small difference was registered for Skein-1024, that uses 0.1MB more heap memory and 0.1MB more non-heap memory. That is because this last version uses blocks of 1024 bits. In terms of execution time the results reflected the ones in the chart above.

For the tree implementations, we used the same parameters for all three versions of Skein. The results showed that for the sequential version Skein-256 uses less memory, and for the parallel implementations Skein-1024 uses less heap memory. The reason is that the number of nodes is smaller for Skein versions with bigger block sizes, and the size of each node doesn't vary much between the three versions.

9 Comparing with other implementations

Our implementations were also tested by comparing them to other Skein implementations, one in Java, from sphlib-2.0, and the second in C from the NIST submission of Skein.

In Java, using our Speed class to test both our implementation and sphlib-2.0 implementation, we get the following results :

Implementation	Processings speed
Skein-512 – our implementation	36MB/s
Skein-512 – sphlib-2.0	34MB/s
SHA-512 – sphlib-2.0	27MB/s

TAB. 1: Speed results - sphlib-2.0

As we can see the Skein implementation from the sphlib-2.0 is slower. Also it is important to notice that this Skein implementation is much faster when compared to the SHA-512 one (Skein is therefore a good candidate for replacing the current SHA-2).

For the second comparison we used a 700MB file and hashed it using both our implementation in Java and the C reference implementation of Skein :

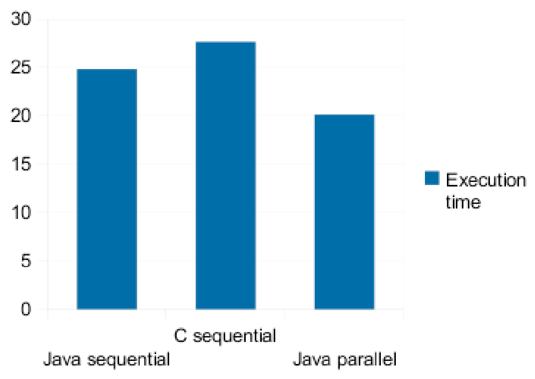

FIG. 10: Comparison with a C implementation

The Java implementation of the tree mode was of course slower than the C version, but not significantly ; therefore some Java applications can use Java implementation of Skein. On the other hand, the parallel implementation using the thread pool is faster than the C implementation.

10 Conclusion and suggestions for further work

Hash functions are the most commonly used cryptographic primitives. These functions can be found in almost any application and they secure the very fundamental levels of our information infrastructures. Currently the SHA family of functions is the most popular, but because the SHA-1 version was broken a new SHA family is needed.

Skein is one of the candidates to the second round of the SHA-3 competition and, judging by the results obtained, it is one of the quite promising candidates.

Skein is appropriate for hardware implementation for both, devices with little memory and high speed needs. Furthermore, software implementations of this family of hash functions in C or Java can be used immediately, increasing its accessibility. The C version is the fastest, but the availability of a pure Java implementation makes it interesting for Java applications with performances quite acceptable.

Further work is in progress for testing the parallel implementation on a highly multi-core/multi-processor system. Moreover, further research should be done to implement the decryption phase of Skein and possibly a specific thread scheduling policy that would speed up the implementation (i.e by reducing the scheduling overhead).

Références

[1] Schneier on security (web site). `http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html`.

[2] Yourkit profiler (web site). `http://www.yourkit.com`.

[3] G. S. Almasi and A. Gottlieb. *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.

[4] M. Bellare, T. Kohno, S. Lucks, N. Ferguson, B. Schneier, D. Whiting, J. Callas, and J. Walker. Provable security support for the skein hash family, 2009. `http://www.skein-hash.info/sites/default/files/skein-proofs.pdf`.

[5] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In *EUROCRYPT*, pages 181–197, 2008.

[6] Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche, Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions, 2007.

[7] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle–damgård revisited : How to construct a hash function. pages 430–448. Springer-Verlag, 2005.

[8] Ivan Damgård. A design principle for hash functions. In *CRYPTO '89 : Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*, pages 416–427, London, UK, 1990. Springer-Verlag.

[9] Reouven Elbaz, David Champagne, Catherine H. Gebotys, Ruby B. Lee, Nachiketh R. Potlapally, and Lionel Torres. Hardware mechanisms for memory authentication : A survey of existing techniques and engines. volume 4, pages 1–22, 2009.

[10] Niels Ferguson, Stefan Lucks Bauhaus, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The skein hash function family (version 1.2), 2009.

[11] Markus Jakobsson, Tom Leighton, Silvio Micali, and Michael Szydlo. Fractal merkle tree representation and traversal, 2003.

[12] Alfred J. Menezes, Paul C. Van Oorschot, Scott A. Vanstone, and R. L. Rivest. Handbook of applied cryptography, 1997.

[13] Andrew Regenscheid, Ray Perlner, Shu jen Chang, John Kelsey, Mridul Nandi, Souradyuti Paul Nistir, Andrew Regenscheid, Ray Perlner, Shu jen Chang, John Kelsey, Mridul Nandi, and Souradyuti Paul. The sha-3 cryptographic hash algorithm competition, 2009.

[14] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full sha-1. In *In Proceedings of Crypto*, pages 17–36. Springer, 2005.