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 SHA1 collisions with less than the expected calculus complexity has been published. The NIST [1] has thus decided to develop a new hash algorithm, so called SHA-3, which will be developed through a public competition [3]. From the set of accepted proposals for the further steps of the competition, we have decided to explore the design of an efficient parallel algorithm for the Skein [12] hash function family. The main reason for designing such an algorithm is to obtain optimal performances when dealing with critical applications which require efficiently tuned implementations on multi-core target processors. This preliminary work presents one of the first parallel implementation and associated performance evaluation of Skein available in the literature. To parallelize Skein we have used the tree hash mode in which we create one virtual thread for each node of the tree.

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

1 Introduction

Skein [12] is a new family of cryptographic hash functions which is one of the candidates in the SHA-3 competition [15]. 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 [16][2], indicating that a stronger hash function is highly desirable. The NIST [1] 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. These are known as the SHA-2 family which does not share the weakness of SHA-1.

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

Since the new proposals are intended to be a drop-in replacement for the SHA-2 family of algorithms, some properties of the SHA-2 family must be preserved. However, some new properties and features shall be provided by SHA-3: it may be parallelizable, more suitable for certain classes of 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 ([10] [9]) that often results in insecure applications.

This paper describes sequential and parallel algorithms for Skein cryptographic hash functions, and the analysis, testing and optimization thereof. Our approach for parallelizing Skein uses the tree hash mode which creates one virtual thread for each node of the tree, thus providing a generic method for fine-grain maximal parallelism.

The paper is structured as following: Section 2 and 3 give a brief description of Skein and a detailed description of the associated Tree Mode, the Section 4 presents the potential approaches for parallelism. Then we present the work done for parallelizing the hash algorithm: speedup, implementation description, testing and first elements of performance evaluation for basic platforms parallel implementations. Finally, 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 [7][8].

Definition [8]: Let A be an *alphabet* group which represents both input and output characters and let C be a finite set whose elements represent 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 acts 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 acts 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 hash functions in the Skein family use three different sizes for internal state : 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 UBI chaining mode to build a compression function that maps an arbitrary input size to a fixed output size. For instance, Figure 6 shows a UBI computation for Skein-512 on a 166-byte (three blocks) input, which makes 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 on 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. For a parallel implementation, each mix operation could be assigned to one thread since, for a given round, they operate on different 128-bit 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 negligible, a permutation being performed at the end of each round.

Skein is built with three basic 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 chaining the UBI operations:

- Simple hash: Takes a variable sized input and returns the corresponding hash. It is a simple and reduced version of the full Skein mode. For instance, with a hash process where the desired output size is equal to the internal state size it consists of three chained UBI functions, the first processes the configuration string, the second the message and the last is used to supply the output.
- Full Skein: The general form of Skein admits key processing, tree hashing and optional arguments (for example, personalization string, public key, key identifier, nonce, and so on). 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 the case of tree processing) is an input to the Output function which generates a hash of desired size.

The followings sections recall the two modes of Skein intended to be widely used, the Simple Hash mode and the Hash Tree mode, a mode specifically designed for parallel implementations (see [12] for more information).

3 Simple Hash Mode

3.1 Specification

A simple Skein hash computation has the following inputs:

N_b The internal state size, in bytes (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:

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

where H is the result of the hash.

If the three parameters Y_l, Y_f and Y_m are not all 0, 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 with 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, thus the output operation can be done in parallel by assigning the UBI operations to each thread according to a round robin arrangement.

4 Hash Tree Mode

4.1 Specification

Tree processing varies 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.

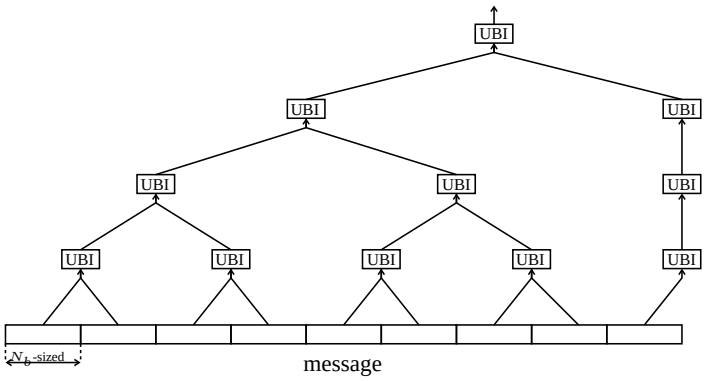

Figure 1: Tree hashing with $Y_l = Y_f = 1$

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 except the last, which may be smaller. We now define the first level of tree hashing by:

$$M_1 = \bigparallel_{i=0}^{k-1} \mathrm{UBI}(G_0, M_{0,i}, iN_l + 1 \cdot 2^{112} + T_{msg} \cdot 2^{120})$$

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 $l = Y_m - 1$, then we have almost reached the maximum tree height. The result is then defined by:

$$G_1 = \mathrm{UBI}(G_0, M_l, Y_m \cdot 2^{112} + T_{msg} \cdot 2^{120})$$

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 last which may be smaller. We then define:

$$M_{l+1} = \bigparallel_{i=0}^{k-1} \mathrm{UBI}(G_0, M_{l,i}, iN_n + (l+1) \cdot 2^{112} + T_{msg} \cdot 2^{120})$$

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 would consist of implementing this algorithm as it is described in its specifications. This implementation constitutes a scheduling method for the node processing that we call *Lower level and leftmost node first* (or *Lower level node first* for short). Such an implementation has the disadvantage of consuming a lot of memory. For instance if we take $Y_l = 1$, we need an amount of avalaible memory space of up to half of the message size, which may be impossible for long messages. There is an effective algorithm (see [13]) which computes a value of a node of height h, 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; we call this method *heigher level node first*. The interest of this method, which maintains a stack in which the intermediate values are stored, is to rapidly discard those that are no longer needed. This stack, which is initially empty, is used as follows: we use (push) leaf values one by one from left to right and we check at each step whether or not the last two values on the stack are of the same height. If such is the case, these last two values are popped and the parent hash value is computed and pushed onto the stack, otherwise we continue to push a leaf value and so on. Note that we could use a two hash-sized buffer at each level (from 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, in which case the memory consumption does not exceed $(h-1)(2^{Y_f}-1) + 2^{Y_l}$ blocks of size N_b for the computation of a node of height h, on the condition that we include a special termination round since they are not necessarily full trees (as we can see in Figure 1).

We assume the existence of the following 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 starts from 0, the leftmost leaf.
  - np: the number of nodes processed.
  - cl: the current level.
  - t: the termination variable
  - D_l: the internal node degree at level 1.
  - D_n: the internal node degree at level > 1.
  - s: the stack.

- **other notation:** n_l, n_p, n_r, n_i denote respectively a leaf node, a parent node, a root node and the i-th child of a parent node.

Then the Algorithm 1 describes the message processing stage with Skein tree hashing.

Algorithm 1 Skein tree hashing using a stack

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

5 Approaches for parallelism

In the following sections, we denote n the number of blocks[1] of the message and N_t the number of threads. These threads are then indexed $0, 1, ..., N_t - 1$.

We assume that $k_1 = k = \left\lceil \frac{n}{2^{Y_l}} \right\rceil$ is the number of N_b-sized blocks of level 1. We define a recursive sequence starting at an initial value k_2 by

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

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

5.1 Addressing parallelism

Two possible ways to address parallelism can be applied: (i) a deterministic way, in which a thread with index j must take into account, at the current step, the predictable behavior of the threads $0, 1, 2, ..., j - 1$; and (ii) a non-deterministic way, in which the *first* node whose child values are available is assigned to the first ready thread. The meaning of the term *first* depends on the strategy adopted to parallelize this algorithm as described bellow. The following sections illustrate several methods.

5.2 Lower level node priority

This method consists in processing the tree levels successively. It should, in theory, offer the best performances due to the (almost) absence of synchronization between threads, apart from synchronization due to dependencies between worker threads and main thread which provides the input data. An example is shown in Figure 2, in which a job is indexed as i_j where i denotes the iteration step and j the index of the assigned thread. If one counts 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.

[1] When it is not specified, the blocks are of size N_b bytes and we include the last block which can be of size less or equal than N_b.

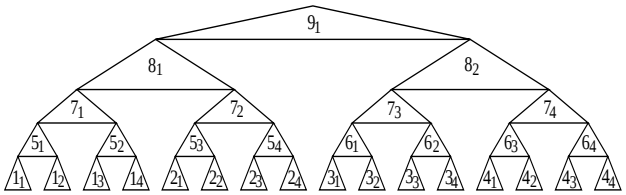

Figure 2: Lower level node first
($Y_f = Y_l = 1,\ Y_m = 255,\ N_t = 4$)

For an implementation, worker threads could wait for themselves when the end of the tree level is reached. This does not minimize the number of steps. Indeed, at the same iteration it is possible to assign the last nodes of level i to the first threads and the first nodes of level $i+1$ to the last threads[2], when the number of nodes of level i is not a multiple of N_t. Note that the other methods described hereafter do not seem to offer the opportunity to gain a few steps in order to optimize speed.

5.3 Higher level node priority

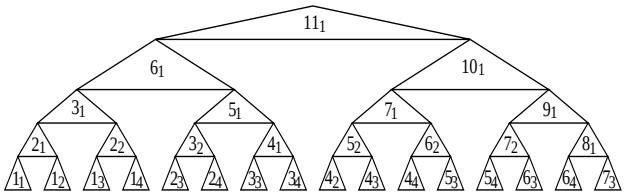

Figure 3: Higher level node first
($Y_f = Y_l = 1,\ Y_m = 255,\ N_t = 4$)

This method consists in assigning[3] to a thread the higher level node among all those that may be assigned to it. Apply this method in a deterministic way need that each thread maintains a vector describing, at each step, the number of nodes that can be processed on each level taking into account the tasks performed by all threads. Thus, knowing the state of the tree, a thead indexed 0 will choose the higher level node, a thread indexed 1 will choose the second higher level node, and so on (an example is shown in Figure 3). Apply this method in a non-deterministic way means that the higher level node whose child values are avalaible 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 by N_t the number of threads, and by p the height of the produced tree, we define a recursive sequence by

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

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

Thus, at each level of the tree, we can use buffers of size $n_1 N_b$ bytes for the first one, $n_2 N_b$ bytes for the second, and so on. If the Y_m parameter does not constrain this tree, then a memory space of only $N_b \sum_{i=1}^{p-1} n_i$ bytes seems sufficient for a deterministic implementation (the memory consumption for the non-deterministic case should approach the deterministic case with high probability). Note that this estimate does not represent the maximum memory used at every moment because not all buffers will be entirely filled. In fact the real memory usage is much lower, the worst case occurring when all the buffers are not empty and not necessarily filled. Furthermore, we cannot be sure that the lengths of the first level buffers at the bottom of the tree are multiples of 2^{Y_f} (take an example with 5 or 6 threads), so we have to consider them as cyclic buffers. Note also that this is only the recommended 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 Figure 3, we must be careful that thread 3_2 does not produce a digest before thread 3_1 has finished consuming digest produced by 2_1, forcing these threads to perform a data recopy in order not to lose too much parallelism.

This scheduling method, which must be further studied, seems not easy to implement and it is not clear if it offers good performance 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, note 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 (see, for example, the right side of the tree in Figure 3). This number of additional steps depends on the configuration of the tree and the number of threads generated. Thus the inherent unbalanced loading between threads of this scheduling approach can induce a performance penalty, *a priori* negligible.

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

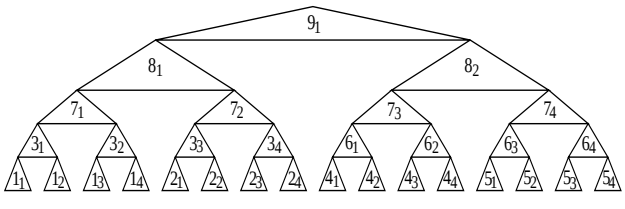

Figure 4: Fixed number and same level nodes first
($Y_f = Y_l = 1,\ Y_m = 255,\ N_t = 4$)

A third method takes again the idea of using a stack (see Section 4.2), but applies it to an arbitrary number of

[2] A step in between two successive levels is possible if the child values of the first nodes of level $i+1$ do not belong to the same iteration step.

[3] An assignment of a node to a thread means that the thread is responsible for producing the hash value of this node using the hash values of its children.

threads. For N_t threads, at each level we use buffers which can receive $N_t 2^{Y_f}$ blocks of size N_b, except the base level where the leaves are the input data buffer blocks. For the same level these threads have to compute $N_t 2^{Y_f}$ node values in order to move up and compute N_t node values at the next level. Once these N_t nodes values are computed, the $N_t 2^{Y_f}$ child values are removed. If the current level occupied by threads is greater than 1 and the lack of resources on the level below prevents them from finishing the compution of the $N_t 2^{Y_f}$ blocks, then they return down to level 1, otherwise they continue, and so on (see Figure 5). In the termination phase for the end of the message, buffers' contents of less than $N_t 2^{Y_f}$ blocks have to be processed. Furthermore, top levels may need narrower buffers (see Figure 4) and when the $Y_m = p$ parameter constrains the tree, the penultimate level buffer must always have a capacity of k_{p-1} blocks. When a level l is reached, buffers are not all filled, except one, and the effective consumption does not exceed $(l(2^{Y_f} - 1) + 1)N_t$ blocks of size N_b. Such an algorithm requires about N_t times more memory for storing internal node values than the sequential algorithm using level buffers instead of a stack.

We may think that this *fixed number and same level nodes first* scheduling is as efficient as the *lower level node first* version described above in Section 5.2. Just like the *higher level node first* scheduling, any recurrent waiting between threads should reduce performance, though it has the advantage of being simpler to implement.

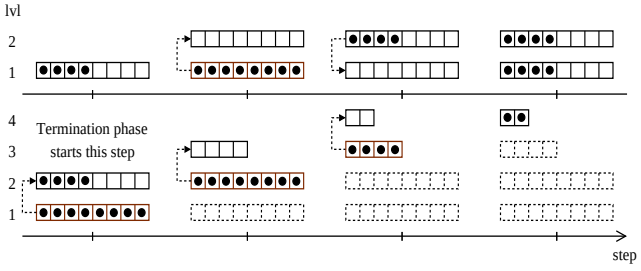

Figure 5: level buffer utilization for a message of $32N_b$ bytes ($Y_f = Y_l = 1,\ Y_m = 255,\ N_t = 4$)

5.5 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 necessarilly full subtrees at the right side of the original tree and finally a last top subtree of height less than 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 thus improve performance. 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 example 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 allows the calculation of a hash in an incremental way; that is to say, it allows updating the hash whenever a new data field is concatenated after the actual data. It also offers the possibility of authentication and updating the hash when the data to be authenticated is never truncated or concatenated with additional fields (*e.g*: memory authentication [11], or a static dictionary).

Note that computing a hash with the Hash Tree mode requires more basic operations than Simple Hash mode (besides, there are more sophisticated mechanisms for these types of application, such as incremental hash and memory authentication). So if these features are absent, because of the overhead that represents the tree structure, it would not be worth providing this functionality in a non-multithreaded implementation.

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

- 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 $\mathrm{UBI}(G, M, T_s)$ can be described by

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

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 assume that b is much lower than a, so we parametrize b by αa with $\alpha \in [0, 1]$ and define this time complexity by

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

where $n = \left\lceil \frac{l}{8N_b} \right\rceil$ 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 Figure 6, which gives an example of three-block message hashing using UBI.

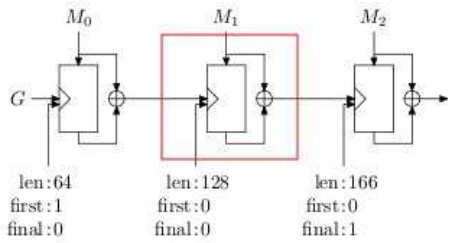

Figure 6: Processing a message of 166 bytes

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 an $h \in \mathbb{N}$ such that $k = 2^{Y_f h}$. The associated tree is a complete tree of 2^{Y_f} nodes and of depth h. If $h \leq Y_m - 1$, the number of basic operations without the contributions in α is given by

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

For $h > Y_m - 1$, this becomes

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

The associated contributions in α are

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

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

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

From $N_s^{nc}(n)$ and $N_s^{c}(n)$, for a fixed message size and not taking into account the contributions in α, one can observe the following:

- 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 $h > Y_m - 1$, then Y_f is not used, the number of operations is function only of Y_l and is maximized for $Y_l = 1$.
- Increasing Y_l will minimize the overhead of the number of operations but requires the use of larger buffers, mainly in a multi-threaded implementation.
- Increasing Y_f tends to decrease the number of operations as well, but it is much less significant compared to Y_l.
- The choice of the two parameters Y_l and Y_f influences the deep of the tree and therefore the memory usage overhead. Increasing these two parameters will decrease the overhead. A constraint Y_m on the tree depth does not affect the amount of memory required for a hash computation (though when using 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 $h \leq Y_m - 1$, the number of operations by processing unit, without the contribution in α, shall be

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

For $h > Y_m - 1$, this becomes

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

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 equation (5), the time complexity $\overline{T_t^p}(n, \alpha)$ 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 given by

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

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

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

For any n, there exists two full trees, one with a lower value $h_1 = \left\lfloor \frac{\ln(\lfloor \frac{n}{2^{Y_l}} \rfloor)}{\ln(2^{Y_f})} \right\rfloor$ and the other with an upper value $h_2 = \left\lceil \frac{\ln(\lceil \frac{n}{2^{Y_l}} \rceil)}{\ln(2^{Y_f})} \right\rceil$, which allow to bound the not

constrained tree generated over the n-block-sized message ($h_2 - h_1 \leq 1$). Then, for all n and a not constrained tree,

$$PS_{pt/u}(n) \geq \frac{\overline{T}(n,0)}{\overline{T_t^p}(2^{Y_l+Y_f h_2},1)}$$

and we can deduce the following result.

Lemma 1. *In the general case, for all n but a not constrained tree in height, $PS_{pt/u}(n) \in \Omega(\frac{n}{\log n})$.*

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

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

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

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

In the general case, for all n and a not constrained tree, we can give a following lower bound

$$PS_{pt/st}(n) \geq \frac{\overline{T_t}(2^{Y_l+Y_f h_1},0)}{\overline{T_t^p}(2^{Y_l+Y_f h_2},1)}.$$

Then, adequate lower bound and upper bound for $\overline{T_t}(2^{Y_l+Y_f h_1},0)$ and $\overline{T_t^p}(2^{Y_l+Y_f h_2},1)$ respectively allow to express exclusively in terms of n a lower bound for $PS_{pt/st}(n)$.

Note that if we take $\alpha = 0$, for n of the form $n = 2^{Y_l+Y_f h}$ and $h \leq Y_m - 1$, 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 $h > Y_m - 1$ we have $PS_{pt/st}(n) = \frac{N_s^c(n)}{N_p^c(n)}$.

6.4 Numerical estimates

In order to compare implementation speedup to a not pessimistic speedup reference, a user must estimate time complexity $T_{st}^{N.E.}(n)$ of the sequential version of the tree for all n by

$$T_{st}^{N.E.}(n,\alpha) = n + \sum_{i=1}^{p-1} k_i + \sum_{i=1}^{p} k_i \alpha.$$

In a same way, an upper bound for the time complexity $T_{N_t}^{N.E.}(n,\alpha)$ of the parallel version executed by N_t processing units is given by

$$T_{N_t}^{N.E.}(n,\alpha) = \left\lceil \frac{k_1}{N_t} \right\rceil (2^{Y_l} + \alpha) + \sum_{i=2}^{p} \left\lceil \frac{k_i}{N_t} \right\rceil (2^{Y_f} + \alpha).$$

Then a good lower bound for the performance improvement $S_{N_t/st}^{N.E.}$, when an implementation dedicated to N_t processing units is used, is given by

$$S_{N_t/st}^{N.E.} = \frac{T_{st}^{N.E.}(n,0)}{T_{N_t}^{N.E.}(n,1)}.$$

7 Java implementation

This section provides some details on a Java implementation of Skein based on an approach like the first one in Section 5, which offers maximum parallelism in theory and is independent of the algorithm parameters (in particular the parameters influencing the node sizes and tree structure in Skein Hash Tree mode). Details of the performance given here are for illustrative purposes, only. We have not attempted to optimize the method for practical use; we are aiming solely to demonstrate the performance improvements that can be obtained on a lambda system configuration.

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 executed), blocked state and terminating state. The main issue 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 which case the thread is not consuming CPU resources at all, meaning that having a large number of blocked threads does not impact much on the efficiency of the system.

7.1 Class organization

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

- Main algorithm: The 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 these 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 processes 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 were 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 implementation

To do a Simple Hash, one simply calls 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 implementation

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

2. *One thread per node with a thread pool*
   To optimize the first implementation we create a thread pool that has a fixed number of threads. These threads accept jobs in a FIFO manner and then executes them (Figure 7).

3. *Pipe input file*
   Because most of the time people hash many files at the same 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.

8 Testing and performances

The tests were done using a basic platform for illustrative purpose only: a Dell Latitude D830, Intel(R) Core(TM)2 Duo CPU T7500 @ 2.20Ghz, 2GB RAM, L2 cache size 4MB with a Ubuntu 9.10 operating system. For evaluating

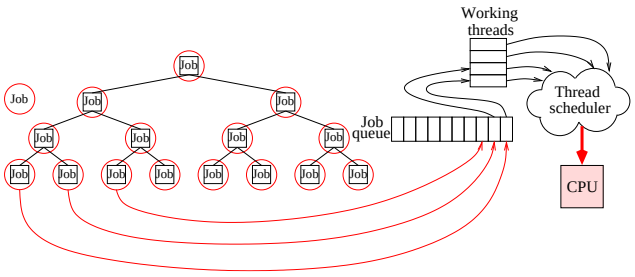

Figure 7: Parallel Skein using one thread per node and a thread pool

the performance of the various implementations we wrote the Speed class, used in conjunction with the YourKit profiling tool [4], which allows monitoring the CPU and memory usage. In order to determine the efficiency of the implementation, we have performed tests using a fixed file of 700MB. The performance results are illustrated in the chart below:

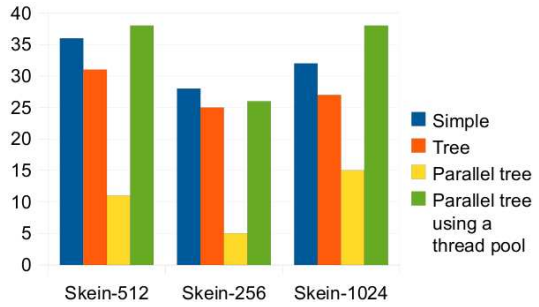

Figure 8: Processing speed (in MB/s) comparison between the Skein versions

Although the fastest version should theoretically be Skein-1024, from this chart we can see 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 for our test is Skein-256, but this one would be the fastest on a 32 bit CPU.

The tests using the YourKit profiler showed that 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 terms 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, although highly scalable, for single use is quite costly: it triples the heap memory usage in comparison to the sequential version, but is not very effective. It is also slower when compared to 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 is still high when compared to the sequential version. On the other hand, the execution

time for this implementation is less than 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 less than 0.2MB. Also an important factor is that when running the tests on a computer with two CPUs, having two input files means that both threads stay in the runnable state, which allows us to maximize CPU utilization.

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 noted for Skein-1024, which uses 0.1MB more heap memory and 0.1MB more non-heap memory. This 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 does not vary much between the three versions.

9 Comparing with other implementations

Our implementations were also tested and compared 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 obtained the results in Table 1. 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).

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

Table 1: Speed results - sphlib-2.0

For the second comparison in terms of execution time we used a 700MB file and hashed it using both our implementation in Java and the C reference implementation of Skein. The execution times are the followings: 27 seconds with the Java sequential version, 20 seconds with the Java parallel version and 24 seconds with the C reference (sequential) version. 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 with no very significant loss of performances. On the other hand, the parallel implementation using the thread pool is faster than the C implementation.

10 Conclusion and further work

Hash functions are the most commonly used cryptographic primitives. These functions can be found in almost any application and they secure the 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 promising candidates.

Skein is appropriate for hardware implementation, both for 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 with acceptable performance is interesting for a large class of Java applications.

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 a more specific thread scheduling policy that would increase performances by minimizing the scheduling overhead.

References

[1] Nist. `http://www.nist.gov/index.html`.

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

[3] Sha-3 competition. `http://www.nist.gov/itl/csd/ct/hash_competition.cfm`.

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

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

[6] 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`.

[7] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the indifferentiability of the

sponge construction. In *EUROCRYPT*, pages 181–197, 2008.

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

[9] Jean-Sbastien Coron, Yevgeniy Dodis, Ccile Malinaud, and Prashant Puniya. Merkledamgrd revisited: How to construct a hash function. pages 430–448. Springer-Verlag, 2005.

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.