Journal of Physics: Conference Series **513** (2014) 032042 doi:10.1088/1742-6596/513/3/032042

Efficient computation of hashes

Raul H C Lopes[1], Virginia N L Franqueira[2] and Peter R Hobson[3]

[1,3]Particle Physics Group, School of Engineering & Design, Brunel University, Uxbridge, UB8 3PH, UK
[2]School of Computing, Engineering & Physical Sciences, University of Central Lancashire, Preston, PR1 2HE, UK

E-mail:
[1]raul.lopes@brunel.ac.uk,[2]vnlfranqueira@uclan.ac.uk,[3]peter.hobson@brunel.ac.uk

Abstract. The sequential computation of hashes at the core of many distributed storage systems and found, for example, in grid services can hinder efficiency in service quality and even pose security challenges that can only be addressed by the use of parallel hash tree modes.

The main contributions of this paper are, first, the identification of several efficiency and security challenges posed by the use of sequential hash computation based on the Merkle-Damgård engine. In addition, alternatives for the parallel computation of hash trees are discussed, and a prototype for a new parallel implementation of the Keccak function, the SHA-3 winner, is introduced.

1. Motivation

The computation of hashes at the core of distributed storage systems as found in the Worldwide LHC Computing Grid (WLCG) services is essentially sequential and non-incremental. It can hinder efficiency in service quality and even pose security challenges that could be addressed by the use of parallel hash tree modes.

Consider the case of a file transfer in a computing grid that is performed in two steps: transfer of the file, and computation of a Message Authentication Code (MAC or checksum) that is used to authenticate the transfer. If the checksum computation phase takes too long, a time out exception is raised, leading to unnecessary repeated transfers of the same file. This paper presents this and other challenges in hash computation.

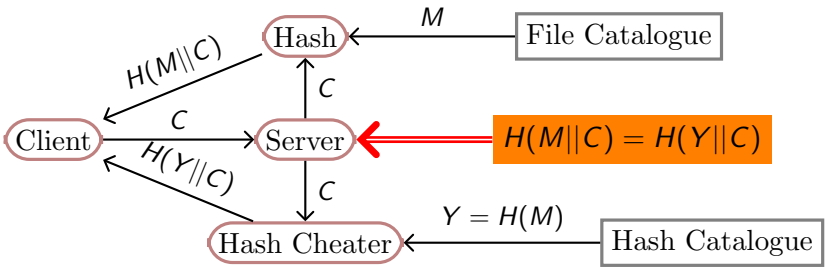
In the next section, both efficiency and security challenges are discussed in the context of hash computation based on the Merkle-Damgård engine [1]. Section 3 introduces computation modes that enable both incremental and parallel computation of hashes and the Keccak hash, the SHA-3 winner. Section 4 outlines a parallel implementation of the *Sakura* tree hashing computation framework, that exposes a functional multi-core algorithm for a parallel deployment of a tree hashing mode for Keccak.

2. Challenges both in efficiency and security

Timeout errors during the computation of a MAC at the conclusion of a file transfer have been the subject of several failures reported in the GGUS ticket system[1] used by the WLCG

[1] `https://ggus.eu/ws/ticket_info.php?ticket=8843` shows an example of ticket issued against a site due to timeout errors in checksum computation after a file transfer.

Figure 1. A security challenge in auditing distributed storage systems.



collaboration. In many cases the impact of the checksum calculation would be reduced if the hash could be evaluated during the file transfer itself. That option, however, may not be available when the transfer is performed using parallel streams (documented for example in ticket 8843 of the GGUS system) and the checksum computation is essentially sequential, as in the case of MD5 and SHA-2, both based on the Merkle-Damgård hash engine [1], which restricts the usage of incremental computation and parallelism [2].

Hall and Jutla [2] first proposed incremental hashing to take on an abstract problem introduced by Blum et al [3]. It represents an abstract challenge for checksum systems that not only generalises the problem above, but also has a potential impact on the use of virtual nodes in grid services. They discuss a problem where a secure processor uses an insecure storage device that can be subject to an attack from an adversary who can possibly modify it. Maintaining a copy of the insecure storage in the secure processor may be impossible due to size constraints. Computing a MAC for the whole storage for every change is too costly, therefore the incremental update of the MAC maintained in the secure processor is a possible solution.

Figure 1 shows a concrete security challenge that can be found in the context of distributed storage commonly used in grid contexts. This potential vulnerability was first identified by Ristenpart et al [4] and it poses a risk to hash-based storage auditing. In this context a *Client* tries to check that a *Server* has a file that it advertises in its *File Catalogue*. In a faithful operation, the *Server*, on receiving a new file M, would save it in the *File Catalogue*, and it would save its hash $H(M)$ in the *Hash Catalogue*. A *Client*, trying to audit the *Server*, would send it a challenge C driving the faithful *Server* to reply with $H(M||C)$, a hash of the file concatenated with the challenge. A not so faithful *Server* might, however, replicate with $H(Y||C)$, where $Y = H(M)$, kept in the *Hash Catalogue*. Given that $H(M||C) = H(Y||C)$, the *Server* would have succeeded to the *Client*'s challenge even if the original file had already been deleted.

Incremental and parallel computing could also alleviate the burden of auditing for dark data in grid storage, and in the WLCG grid infrastructure in particular. Incremental and parallel computation of hashes could open the possibility of maintaining a real time verification of the file catalogue declared by a WLCG grid site, for example.

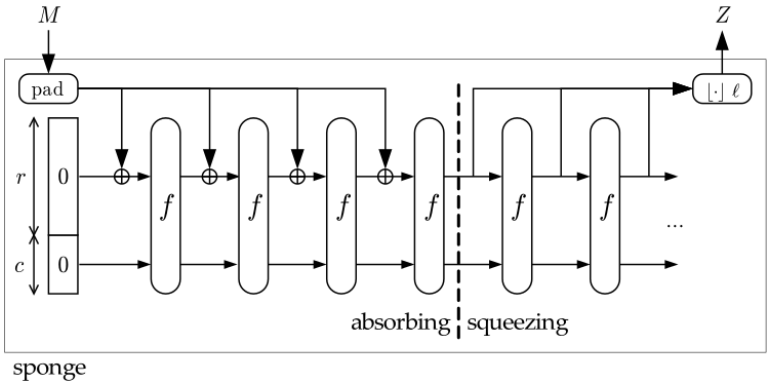
3. From fixed-length to variable-length functions

Most practical hash functions are built from a given input inner function f (typically a fixed-length compression, permutation or a block-cypher transformation), the hash H being comprised of an outer function that sequentially iterates f over the blocks of a given message. Assuming that a checksum is to be computed for a message $M = [M_0, \ldots, M_l]$, the hash is just the result of the application of a reduction or fold pattern (present in languages like Haskell or C++) of computation:

$$H(M) = fold\ f\ \epsilon\ M \tag{1}$$

Figure 2. Keccak's sponge function, from `http://sponge.noekeon.org`.

.



where f maps a partially computed hash value and a block of the message to a newly computed hash value, and ϵ represents an empty string of bits. Presenting the hash template as a *fold* pattern is important because fold patterns can be compiled into parallel programs.

The restriction of using sequential iteration of a fixed length input function severely limits the usage of multi-processors or even the application of SIMD parallelism at the level of the inner transformation function [1]. It is worthwhile noting that Damgård [5] introduced the parallel computation of a hash over a message of length n. The resulting algorithm, however, showed the prohibitive cost of demanding $O(n)$ processors, which could represent a serious limitation in the presence of long message as it would happen in the computation of checksums of datasets in a grid-like environment.

Keccak, the winner of the SHA-3 competition, seems to offer a set of desirable properties for an environment where either parallel or incremental hashing (or both) might be in demand. It is a variable-length input and output function based on the concept of sponge function (first introduced in [6]), that enables the application of both SIMD parallelism at the inner function and multi-processing at the block level computation, also enabling incremental hashing in a natural way.

The Keccak sponge function [7] is a combination of a fixed-length input and output transformation function (actually a permutation in Keccak) and a padding rule to produce a variable length function. Given a parameter n, it takes arbitrary length sequences of bits, strings in $(Z_2)^*$, and maps them to strings in $(Z_2)^n$. Its computation, depicted in figure 2, can be decomposed in two phases, *absorbing* and *squeezing* phases, both operating on a buffer S that comes to be the state of the algorithm.

The *absorbing* phase uses the template algorithm of equation 1, splitting the input message in blocks of size r bits. Each block is padded with c zero valued bits before being submitted to a *xor* operation with the state S. The result is then passed through the transformation f. The output of this phase then feeds onto the *squeezing* phase which repeatedly alternates the extraction of the first r bits of the state S, concatenating them as final output with an application of the transformation function to the state buffer.

Both the absorbing and squeezing phases of Keccak offer clear opportunities for applying SIMD parallelism at both the level of the *xor* and the permutation function [8]. An extensive performance comparison is shown for one core implementation of Keccak in [9]. It shows numbers of cycles consumed to process one byte of message, cycles per byte being the preferred unit of comparison in this context given that times for processing short messages can be so small that they might not be good indicators. The first table in [9] indicates that Keccak and SHA-512 are

both faster than SHA-256. Also, the benchmark in [10] indicates that SHA-512 is faster than SHA-256. As such, in that next section, we compare a parallel implementation of Keccak with an implementation of SHA-512.

4. A parallel realisation of tree hashing

A prototype of a tool for experimenting with Keccak has been implemented, based on the *Sakura* [11] mode of tree hash computation. *Sakura* offers the realisation of tree hashing independent of the availability of parallel resources. Its target is to define a framework for the computation of tree hashes that can be evaluated using either parallel or sequential modes of computation and that can support incremental computing. The *Sakura* modes see a message as being split in blocks that are padded with frame and mode marking bits. It presents two possible computation modes: *final node growing*, where the number of leaves (and degree of internal nodes of the tree) increases with message length; and *leaf interleaving*, showing fixed tree height and number sizes of leaves, with message blocks interleaved onto trees.

The hash computation is abstractly defined in a tree of hops. Hops can be either message hops, that contain the padded blocks of the initial message, or chain hops, denoting application of the inner Keccak hashing function to the results of their children hops. A parallel implementation must map hops to computational tree nodes and decide on the scheduling of the nodes.

The prototype implements the leaf interleaving mode of *Sakura*. It was first realised in Haskell, which enables the development and understanding of the *fold* operator. Its most efficient version is implemented in C with OpenMP offering the macro parallelism. SIMD parallelism is exploited in the computation of *xor* and permutation functions. The macro parallelism follows the pattern of equation 1 which matches the tree defined by the leaf interleaving of *Sakura*. It is exploited by using the OpenMP's multi-threaded facilities to implement a parallel reduction as presented in [12]. Results for a test on computing the checksum of a $2GB$ buffer is shown in Table 1. The table displays results for running on 1, 6 and 12 cores of the same machine: a dual Intel Xeon 5645 with a $2.40GHz$ clock and $12MB$ of cache, running a Gnu/Linux operating system with kernel 3.8.0. The code was compiled with gcc 4.4.7. Results show millions of bytes per second of input buffer processed, which can be useful to compare the implementation's performance with that given by SHA-512 in [10]. Time in seconds is also shown as it makes it straightforward to apply the usual formulas of speed-up (S_p), i.e., rate of time on 1 core divided by time on p cores, and efficiency (E_p), i.e., the division of speed-up by the number of cores used. A computation of SHA-512 for the same buffer is expected to deliver 187 million of bytes per second on an Intel Xeon 5530, according to [9]. The same computation using the GNU/Linux program *sha512sum* demands 18.5 seconds, but includes the time to read the file, which takes around 5 seconds.

Table 1. Time for the parallel implementation of Keccak, running on an Intel Xeon 5645, to compute the checksum of a 2GB buffer.

Architecture	MB/s	Time (s)	Speed-up (S_p)	Efficiency (E_p)
Intel Xeon 5645 (1 core)	208	9.8	1.0	1.0
Intel Xeon 5645 (6 cores)	839	2.4	4.03	0.67
Intel Xeon 5645 (12 cores)	1077	1.9	5.15	0.43

5. Related work

The foundations for the parallel computation of hashes were already discussed in [5], even if using an impractical framework due to the demand on the number of processors. Hall and Jutla [2] presented one of the first works addressing the usage of incremental hashing to defend against the RAM security risk presented in section 2. Their work draws on previous work by Blum et

al [3] on ascertaining correctness of shared insecure memory. This is particularly relevant in the context of cloud and grid computing. Recent work by Ristenpart et al [4] identifies the limitations on the usage of indifferentiability in face of composition and discusses variations of the security challenge of figure 1.

Sarkar and Schellenberg [13] seem to be the first to discuss the security of tree hashing and its realisation using parallel algorithms supported by a limited number of computational resources. Bertoni et al [14] introduce conditions for sound tree hashing and parallelism at both micro (SIMD level) and macro-level (multi-processor) [8, 7]. Also discussed in [8] is the usage of both hardware (FPGA, for example) and GPU parallelism to implement Keccak in parallel.

6. Final remarks

At the time of writing, the final SHA-3 standard defining sizes of capacity and padding had not yet been released by NIST. Indeed NIST had been the target of some criticism due to a recent announcement that would reduce the capacity of Keccak's state buffer, a reduction that would enable faster computation, but would weaken the strength of the function. In that sense other functions that also offer opportunities in parallel and incremental computation may be considered. However, SHA-3 is now a NIST standard with available reference implementations [8] and is clearly a tool for tackling the challenges presented in section 2. Keccak offers the alternatives of parallel and incremental computation that can be used to achieve a real time verification of large datasets being generated and stored, e.g., in the WLCG file systems. The tool described in section 4 offers a fast multi-core implementation of Keccak, and it would be released as public software should the approval of the SHA-3 patent holders be granted.

7. Acknowledgments

Lopes and Hobson are members of the GridPP collaboration and wish to acknowledge funding from the Science and Technology Facilities Council, UK.

References

[1] Coron J S, Dodis Y, Malinaud C and Puniya P 2005 *CRYPTO'05* (Springer-Verlag) pp 430–448
[2] Hall E and Jutla C S 2002 *In Cryptology ePrint Archive*
[3] Blum M, Evans W, Gemmell P, Kannan S and Naor M 1995 *Algorithmica* pp 90–99
[4] Ristenpart T, Shacham H and Shrimpton T 2011 *30th Annual international conference on Theory and applications of cryptographic techniques: advances in cryptology* EUROCRYPT'11 (Berlin, Heidelberg: Springer-Verlag) pp 487–506
[5] Damgård I B 1989 *Advances in Cryptology* CRYPTO'89 (Springer-Verlag) pp 416–427
[6] Bertoni G, Daemen J, Peeters M and van Assche G 2007 Sponge functions Ecrypt Hash Workshop 2007
[7] Bertoni G, Daemen J, Peeters M and van Assche G 2010 KECCAK sponge functions family main document Ecrypt Hash Workshop 2007 nIST SHA-3 submission (update), `http://keccak.noekeon.org`
[8] Bertoni G, Daemen J, Peeters M, van Assche G and Keer R V 2011 Keccak implementation overview `http://keccak.noekeon.org`
[9] Bertoni G, Daemen J, Peeters M and van Assche G 2011 The KECCAK sponge function family — software performance figures Internet Archive `http://keccak.noekeon.org/sw_performance.html` retrieved 2014-01-30
[10] Crypto++ 5.6.0 benchmarks `http://www.cryptopp.com/benchmarks-amd64.html` retrieved 2014-01-30
[11] Bertoni G, Daemen J, Peeters M and van Assche G 2013 Sakura: a flexible coding for tree hashing Cryptology ePrint Archive report 2013/231, `http://eprint.iacr.org`
[12] Blelloch G E 1990 Preffix sums and their applications Tech. Rep. CMU-CS-90-190 School of Computer Science – Carnegie Mellon University
[13] Sarkar P and Schellenberg P J 2002 A parallelizable design principle for cryptographic hash functions Cryptology wPrint Archive report 2002/31, `http://eprint.iacr.org`
[14] Bertoni G, Daemen J, Peeters M and van Assche G 2013 Sufficient conditions for sound tree and sequential hashing modes Cryptology wPrint Archive report 2009/210 (revised April 2013), `http://eprint.iacr.org`