*International Journal For Technological Research In Engineering*
*Volume 4, Issue 6, February-2017* *ISSN (Online): 2347 - 4718*

# IMPLEMENTATION OF HASH FUNCTION FOR CRYPTOGRAPHY (RSA SECURITY)

Syed Fateh Reza[1], Mr. Prasun Das[2]
[1]M.Tech. (ECE), [2]Assistant Professor (ECE), Bitm,Bolpur

***ABSTRACT:*** ***In this thesis, a new method for implementing cryptographic hash functions is proposed. This method seeks to improve the speed of the hash function particularly when a large set of messages with similar blocks such as documents with common Headers are to be hashed. The method utilizes the peculiar run-time configurability Feature of FPGA. Essentially, when a block of message that is commonly hashed is identified, the hash value is stored in memory so that in subsequent occurrences of The message block, the hash value does not need to be recomputed; rather it is Simply retrieved from memory, thus giving a significant increase in speed. The System is self-learning and able to dynamically build on its knowledge of frequently Occurring message blocks without intervention from the user. The specific hash Function to which this technique was applied is blake, one of the SHA-3 finalists.***
***Keyword: Hash Function, FPGA, blake, SHA-3***

## I. INTRODUCTION

### 1.1 Cryptographic hash functions

There is no doubt about the fact that electronic communication has revolutionized our world. The world has progressed from communication with mainly letters written on paper and sent through the post office to instant communication via email, chat and social networking websites like Facebook and Google+. Many communication activities that were traditionally done via post are now done through electronic means. These activities include transferring documents, images, audio and video. A cryptographic hash function is one which converts an input data of arbitrary length into a fixed-length output. Cryptographic hash functions are somewhat different from ordinary hash functions used in computer programs; however, for simplicity cryptographic hash functions will simply be referred to as hash functions throughout the rest of this thesis. The output of a hash function must have certain properties; these are: pre-image resistance, second pre-image resistance and collision resistance. These properties ensure that the hash function is secure. The properties stem from the ways in which hash functions have been attacked. Pre-image resistance implies that the hash function is a one-way function. That is, it should be infeasible for an attacker to determine the original data (or message) from a given hash code or digest (the digest is another name for the hash code or hash value). Second pre-image resistance guarantees that even the slightest change in a message will change the digest. That is, if an attacker is given a message, it should be infeasible for the attacker to manipulate the message and still obtain the same digest as the original message digest. Collision resistance gives the general analogy of fingerprint with respect to the message digests. That is, every message is expected to have a unique hash code and it should be generally difficult for an attacker to find two messages with the same hash code.

Mathematically, a hash function (H) is defined as follows:

$$H: \{0, 1\}^* \rightarrow \{0, 1\}n$$

In this notation, {0, 1}* refers to the set of binary elements of any length including the empty string while {0, 1}n refers to the set of binary elements of length n. Thus, the hash function maps a set of binary elements of arbitrary length to a set of binary elements of fixed length. Similarly, the properties of a hash function are defined as follows:

$$x \{0, 1\}^*; y \{0,1\}n$$

Pre-image resistance: given y= H(x), it should be difficult to find x.

Second pre-image resistance: given x, it should be difficult to find x' such that H(x) = H(x') (where x x').

Collision resistance: it should be hard to find any pair of x and x' (with x x') such that H(x) = H(x')

The properties of second pre-image resistance and collision resistance may seem similar but the difference is that in the case of second pre-image resistance, the attacker is given a message (x) to start with, but for collision resistance no message is given; it is simply up to the attacker to find any two messages that yield the same hash value. The word "difficult" or the phrase "hard to find" in this context implies that it will take a long time (many years) and a huge amount of memory for a computer to perform the computation. That is, for example, it will take many years and a lot of memory for a computer with today's technology standards to compute a message from its digest value; thus, the computation is regarded as infeasible. It is interesting to note that as processing power of computers have increased over the decades, some hash functions that were previously considered secure (possessing all the properties of pre-image, second pre-image and collision resistance) are now considered "broken". Also, if an attacker is able to prove that the time it will take to 'break' a hash function, though not small has been significantly reduced, that hash function will be considered weak. As computational power increased and cryptanalysis of hash functions were performed, certain hash function standards have also been revised because they were found to be weak. It is desirable to have a hash function that is secure and computationally efficient.

### *1.2 Applications of hash functions*

As mentioned earlier, hash functions are used in certain information security schemes. These include: digital signatures, Message Authentication Codes (MACs) and digital image watermarking. There are also simple applications of hash functions such as password storage. In

  

password storage application, the password entered by a user at the first log-in is not stored in the computer system; rather the hash of the password is stored. To log into the system at subsequent times, the user needs to enter the password; the system hashes it and compares it with the stored hash. If there is a match, the user is granted access to the system otherwise the user is denied access. The advantage of this scheme lies in the fact that if an attacker manages to gain access to the system's storage devices, only the hash of the password can be retrieved and this cannot be used to recover the original password since the hash function is a one-way function.

*1.3 Problem statement*

Hash functions, as previously established, are very useful in information security schemes. Apart from the above mentioned applications (digital signatures, digital image watermarking and so on), hash functions are also utilized in generating pseudo random numbers which are in turn utilized in many cryptographic schemes. In most of these applications, particularly digitalsignatures, digital image watermarking and Message Authentication Codes, it is desirable to have the hash function operate as fast as possible especially when a huge traffic or load of messages are expected to be operated on. Consequently, a lot of research effort has been expended in the area of high speed implementation of standardized or widely used hash functions. The US National Institute of Standards and Technology (NIST) has organized a competition to select a new hash function standard that is expected to be atleast as secure as and significantly faster than the current hash function standard (SHA-2). This is in line with the objective of making the hash function run faster and increase overall performance when it is utilized along with other primitives in information security schemes. The goal of this thesis is to explore the high speed implementation of hash functions using Field Programmable Gate Arrays (FPGAs) and the Blake hash function (one of the final round candidates in the competition organized by NIST).

## II. LITERATURE SURVEY

As previously mentioned, a significant amount of research effort has been expended in the area of high speed implementation of hash functions. The Blake hash function like many other hash functions was designed with the intent of making it capable of running at high speed. It has a relatively simple algorithm; its compression function is a modified "double round" version of Bernstein's stream cipher "chacha" which has been intensively analyzed and found to be of excellent performance and parallelizable [4]. Blake has been examined by researchers seeking ways of providing high speed operation. One of the techniques for speed optimization of Blake that is found in literature is parallelism [5]. Other speed optimization techniques that have been applied to Blake are pipelining (in an area of the algorithm where pipelining is feasible) [6] and the use of carry-save adders [6] in the compression function. These techniques focus on the main 'core' of the hash function. Nowhere, to the best of our knowledge has any attempt been made to improve the speed of the hash function by looking at the iterative/ repetitive process of hashing. The hash functions in use today evolved from weaknesses found in previous hash functions. The first publicly known hash function was developed by Ronald Rivest in 1989 and it was known as Message-Digest Algorithm (MD2). In 1990, Rivest developed another hash algorithm named MD4. MD4 was based on the Merkle-Damgard construction [7]. In 1991, Rivest again developed another hash algorithm to replace MD4; this new algorithm was named MD5. Meanwhile the National Institute of Standards and Technology (NIST) was also working on a hash function standard. In 1993, NIST developed the Secure Hash Standard (SHA). This standard was published by NIST as a US Federal Information Processing Standard (FIPS). However, shortly after the publication, the algorithm was withdrawn due to an undisclosed "significant flaw". It was replaced by a revised version named SHA-1. SHA-1 has been widely used in information security schemes such as Transport Layer Security (TLS), Secure Sockets Layer (SSL), Internet Protocol Security (IPsec), Secure Shell (SSH) and Pretty Good Privacy (PGP). SHA-2, a set of hash functions (SHA-224, SHA-256, SHA-384, and SHA-512) was designed by the National Security Agency (NSA) and published by NIST in 2001. These hash functions in SHA-2 are named according to the number of bits of their digest; SHA-256 for instance has 256 bits in its digest. SHA-2 was created as an update to the former standard (SHA-1).

## III. PREVIOUS WORKS (HIGH-SPEED IMPLEMENTATION OF BLAKE)

Certain techniques have been applied to hardware implementations of Blake in an attempt to optimize the speed of the hash function. These techniques are: parallelism, pipelining and the use of fast adders. In the following sections we shall examine each of these techniques.

*3.1 Parallelism*

Parallelism is one of the methods that have been applied for the speed optimization of Blake. The main task that consumes time in the hash function's algorithm is the state update. The initialization is a process that simply depends on a few XOR gates and combinational logic; this doesn't consume time. Similarly, the finalization is a process that depends on XOR gates and utilizes only combinational logic; it consumes a relatively small amount of time. However, for the state update; first of all it utilizes the g-functions which have addition, rotation operations; these can consume some time. Secondly, the full state update takes 14 rounds of similar g-function operations. Thus, if the speed of the hash function is to be increased, one of the main areas to consider would be the state update. The update of the state columns and diagonals can be done sequentially; that is, one column (or diagonal) updated at a time or it could be done with all 4 columns updated simultaneously. However, all the columns must be updated before the diagonals are updated because the diagonal update makes use of the new state variable values obtained from the column update. Parallelism is applied to Blake by updating all the columns of the state simultaneously and then similarly updating all the diagonals

  

of the state simultaneously.

*3.2 Pipelining*

In the g-function, some of the operations performed could take a relatively long period of time. The g-function is a modified 'double' round of the stream cipher chacha. The fact that it is a double round implies that the outputs of some operations in the g-function are inputs to some other operations in the same g-function. In particular, computations involving the XOR of message and constant words, one of which is given below:

$$vx1 = vx1 + vx2 + (m\sigma r(2i) \wedge C\sigma r(2i+1))$$

consume a longer time because there are three major operations involved. Thus, these operations constitute the critical path of the g-function (the path with the longest delay).The critical path influences the speed (throughput) of the overall computation. A long critical path delay requires a long the clock period and hence the speed of the hash function is reduced. However, a pipeline stage may be used to improve the speed. Since there are 14 rounds of repeated g-function computations, if a pipeline register is inserted into the critical path of the g-function; thereby creating a two stage pipeline, then the first stage of the pipeline for the next round can be executed while the second stage of the pipeline for the current round is executing. The net effect is an increase in the clock rate and consequently an increase in speed (throughput) of the hash function. This pipeline technique was applied in [6]. Figure 5 illustrates the method.

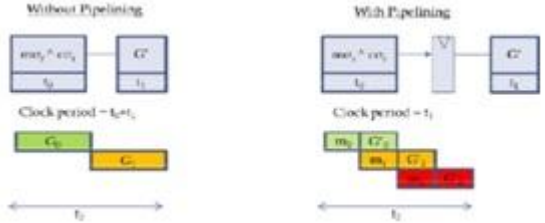


Figure 3.1 Pipelining applied to Blake

As seen in figure 5, when pipelining is applied, 3 g-function computations were accomplished within a time period of t2; whereas without pipelining only 2 g-functions were accomplished within the same time period.

*3.3 Fast adders*

The third technique that has been applied for speed optimization in Blake is the use of carry-save and carry-look ahead adders in the g-function. These are fast adders. The technique was applied in [6]. Carries are a major source of delay in additions when ripple adders are used because a carry needs to propagate to the last full adder before the sum can be considered valid. The additions in the g-function of Blake are 32-bit additions; thus if ripple adders are used, then the time it takes for a carry to propagate from the full adder (FA) at the least significant bit (LSB) position to the full adder at the most significant bit position (MSB) can be significant. To overcome this source of delay, 2 carry-save adders (CSA) are used when three numbers are to be added, with a carry -lookahead adder (CLA) performing the final stage of the addition. This is shown in figure 6 for a 2 bit number. The CSA is a FA connected in such a way that it adds corresponding bits of the 3 numbers directly similar to the way we add numbers on paper. This saves a significant amount of time since the few carries generated are added with a CLA. The arrangement can be easily extended to 32 bits.

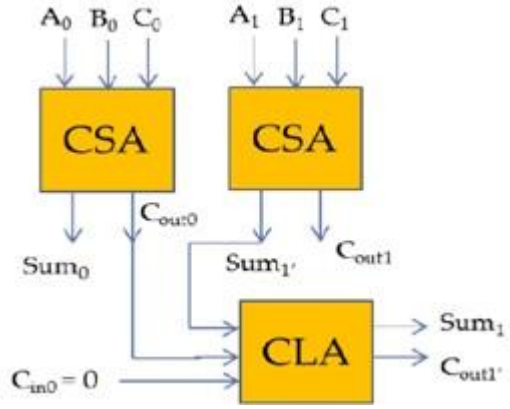


Figure 3.2: Fast adders

## IV. PROPOSED DESIGN

The speed optimizations techniques discussed in the previous chapter essentially focus on the process of hashing one message block. These techniques are effective and aim at reducing the time spent in hashing each message block, thereby reducing the overall time spent in hashing a message which may contain many blocks. For example, if the time spent in hashing a message block has been reduced through the use of fast adders from 50ns to 45ns and there are 1000 messages to be hashed, each containing 10 message blocks; the minimum time that would be spent in hashing these messages would be reduced from 0.5ms to 0.45ms. Thus, the speed has been improved. A similar analogy holds for the techniques of parallelism and pipelining. However, there is a particular situation in which the speed of the hash function can be potentially increased further but these techniques cannot bring about the improvement. This situation occurs when many messages which have some identical message blocks are to be hashed. For instance, if message blocks 1 and 2 out of the 10 message blocks in the messages of our previous example are identical but message blocks 3 to 10 are different, these messages will still give distinct hash codes. However, the chain values (intermediate hash values) obtained for message blocks 1 and 2 will be the same for all the messages. The implication of this, is that in computing the digests of the 1000 messages, the hash function will perform the same computation 2,000 times (the hash code of message block 1 will be computed 1000 times, the same goes for message block 2, so in total there will be 2,000 identical or repeated computations). If there was a way to bypass these repeated computations, this would certainly lead to a significant increase in speed; the time taken to compute the hash codes would be reduced by an additional 0.1ms.

Our design takes the situation in which messages with common blocks are to be hashed into consideration and provides a way of bypassing the redundant computations that would otherwise have to be made; thus providing high speed

  

operation. The design allows the previously discussed techniques of parallelism, pipelining and fast adders to be applied to the Blake hash function but in addition it provides a method of avoiding redundant computations, thereby leading to a further increase in the speed of the hash function. The design is self-learning; that is, it builds up its knowledge of common message blocks without intervention from the user. The design incorporates three major components to facilitate these:
Message preprocessor: This component independently identifies common message blocks in the messages that are being hashed, determines their initial values, counter values and computes their hash codes.

Memory: A memory device is used to store the hash code of any common message block that has been identified by the message preprocessor.

Decoder: This component is used to determine if an inputted message block is a common message block. If the inputted message block is a common message block, the decoder outputs the address of the memory location containing the hash code of the common message block. In addition, it also outputs a signal which indicates to the hash function unit that the hash code of the inputted message blocks is already available in memory and consequently, there is no need to compute it.

In the following sections we shall discuss each of these components and how they interconnect to achieve the desired operation.

**4.1 Objective of Proposed Work**

Hash functions, as previously established, are very useful in information security schemes. Apart from the above mentioned applications (digital signatures, digital image watermarking and so on), hash functions are also utilized in generating pseudo random numbers which are in turn utilized in many cryptographic schemes. In most of these applications, particularly digital signatures, digital image watermarking and Message Authentication Codes, it is desirable to have the hash function operate as fast as possible especially when a huge traffic or load of messages are expected to be operated on. Consequently, a lot of research effort has been expended in the area of high speed implementation of standardized or widely used hash functions. This is in line with the objective of making the hash function run faster and increase overall performance when it is utilized along with other primitives in information security schemes. The goal of this thesis is to explore the Implementation of Cryptographic Hash Function through Integrated Simulation work.

REFERENCE

[1] M. Bellare, R. Canetti and H. Krawczyk, "Keying hash functions for message authentication," in Advances in Cryptology — CRYPTO ’96, N. Koblitz, Ed: Springer Berlin / Heidelberg, 1996, pp. 1-15.

[2] J. Black, S. Halevi, H. Krawczyk, T. Krovetz and P. Rogaway, "UMAC: fast and secure message authentication," in Advances in Cryptology — CRYPTO’ 99, M. Wiener, Ed: Springer Berlin / Heidelberg, 1999, pp. 79-79.

[3] P.W. Wong and N. Memon, "Secret and public key image watermarking schemes for image authentication and ownership verification," Image Processing, IEEE Transactions on, vol. 10, no. 10, pp. 1593-1601 2001.

[4] J.P. Aumasson, L. Henzen, M. Willi and C.W.R. Phan, SHA-3 Proposal BLAKE, January 11, 2011. Available: http://www.131002.net/blake/ [Accessed: August 2011].

[5] S. Tillich, M. Feldhofer, M. Kirschbaum, T. Plos, J.M. Schmidt and A. Szekely, “High-speed hardware implementations of BLAKE, Blue Midnight Wish, CubeHash, ECHO, Fugue, Groestl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein”, Cryptology ePrint Archive, report 2009/510 2009.

[6] L. Jianzhou and R. Karri, “Compact hardware architectures for BLAKE and LAKE hash functions,” in Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on, 2010, pp. 2107-2110.

[7] I. Damgard, "A design principle for hash functions," in Advances in Cryptology — CRYPTO’ 89 Proceedings, G. Brassard, Ed: Springer Berlin / Heidelberg, 1990, pp. 416-427.

[8] H. Dobbertin, “Cryptanalysis of MD4,” in Fast Software Encryption 1996 Proceedings, volume 1039 of Lecture Notes in Computer Science, D. Gollmann, Ed: Springer, 1996, pp. 53– 69.

[9] X. Wang, H. Yu and Y. L. Yin, “Efficient collision search attacks on SHA-0,” in Advances in Cryptology – CRYPTO 2005 Proceedings, volume 3621 of Lecture Notes in Computer Science, V. Shoup, Ed: Springer, 2005, pp. 1–16.

[10] X. Wang, Y. L. Yin and H. Yu, “Finding collisions in the full SHA-1,” in Advances in Cryptology – CRYPTO 2005 Proceedings, volume 3621 of Lecture Notes in Computer Science, V. Shoup, Ed: Springer, 2005, pp. 17–36.

  