International Journal of Electronics and Communication Engineering (IJECE)
ISSN 2278-9901
Vol. 2, Issue 1, Feb 2013, 77-86

International Academy of Science, Engineering and Technology
Connecting Researchers; Nurturing Innovations

HARDWARE IMPLEMENTATION OF ADVANCED CRYPTOGRAPHIC HASH FUNCTION ON FPGAs

ATULIIKA SHUKLA[1], SUMIT SHARMA[2] & RAVI MOHAN[3]

[1]EC Department, Shri Ram Institute of Technology, Jabalpur, Madhya Pradesh, India

[2]HOD, EC Department, Shri Ram Institute of Technology, Jabalpur, Madhya Pradesh, India

[3]HOD, ME/M.Tech EC Department, Shri Ram Institute of Technology, Jabalpur, Madhya Pradesh, India

ABSTRACT

A cryptographic hash function is a deterministic procedure whose input is an arbitrary block of data and output is a fixed-size bit string, which is known as the (Cryptographic) hash value. Cryptographic hash functions are the workhorses of cryptography, and can be found everywhere. Originally created to make digital signatures more efficient, they are now used to secure the very fundamentals of our information infrastructure, message authentication codes (MACs), [1] secure web connections, encryption key management.

Here is an algorithm which is implemented on FPGA. An essential part of this work is hardware performance evaluation of the hash function algorithms. In this work we present efficient hardware implementations and hardware performance evaluations of the algorithm. We implemented and investigated the performance of efficient hardware architectures on latest Xilinx FPGAs. we conclude the results in the form of chip area consumption, throughput and throughput per area on most recently released devices from Xilinx on which implementations have not been reported yet. We have achieved substantial improvements in implementation results from all of the previously reported work. This work serves as performance investigation of the given algorithm on most up-to-date FPGAs.

KEYWORDS: Hash Function, SHA-3, Skein, Threefish, FPGA

INTRODUCTION

Over the last three decades, the use of information technology in our everyday lives has increased dramatically. Due to this, the growth rate for e-commerce has been double-digit over the last decade, with an estimated $301 billion expected online retail sales in 2012 [1]. This extreme increase in online trading has lead to a rise in online attacks to obtain money through deception or other illegal means. Due to this, companies and consumers using e-commerce have become more aware of security risks exchanging information over such an open medium. This increased knowledge has lead to several third parties setting up secure areas for credit card and bank account details to be shared with minimal risk of the numbers being obtained and used fraudulently When shopping on The Internet, a connection is set up between the computer being used and the company server. This is done using a “Challenge and Response” through the Transport Layer Security (TLS), [10].

Challenge and Response uses a mixture of symmetric block ciphers and Message Authentication Codes (MAC). The MAC is constructed using a Hash Function.A cryptographic hash function is a deterministic procedure whose input is an arbitrary block of data and output is a fixed-size bit string, which is known as the (Cryptographic) hash value. Cryptographic hash functions are the workhorses of cryptography, and can be found everywhere.[2] Originally created to make digital signatures more efficient, they are now used to secure the very fundamentals of our information infrastructure, message authentication codes (MACs), secure web connections, encryption key management, virus- and malware-

scanning, and almost every cryptographic protocol in current use.[3] Without hash functions, the Internet would simply not work.

OVERVIEW

We have designed a family of cryptographic hash functions. The proposed design has three different internal state sizes: 256, 512, and 1024 bits. Each of these state sizes can support any output size. The proposed design is built from three components, Threefish tweakable block cipher, Unique Block Iteration (UBI) and Optional argument system. The tweakable block cipher makes every instance of compression unique by hashing configuration data along with input message. The compression function of the proposed design consists of a layer of non-linear MIX operations and permutation. MIX operation consists of addition modulo 2^{64}, rotation and XOR operation on a pair of 64-bit words. The Threefish compression function is used in UBI chaining mode to compress arbitrary length of input data to fixed size hash digest.

RELATED WORK

There are two main streams of hardware implementations of algorithms on FPGA and ASIC platforms: *high speed implementations* and *compact implementations*.[4] Various groups around the world are working on hardware performance evaluation of cryptographic hash functions using these two types of implementations. Most of the reported work is focused on high speed architectures as it provides a direct snapshot of the basic operations' cost for a given algorithm. The relevant category for our work is high speed implementations on FPGAs.

People discussed and reported their results for various architectures using pipelining, folding and loop unrolling approaches. For performance comparison, we quote here the results of architecture based on basic iterative approach. We have been calculated specifications based on the reported clock frequencies and number of clock cycles consumed for the proposed design.[5]

ENVIRONMENT

It effects the implementations in terms of the level of expertise, language, coding techniques, design methodology, and development tools. We implemented the design using VHDL as the language and using Xilinx's ISE 13.1/Altera's Quartus-II as the development tool [6].

IMPLEMENTATION METHODOLOGY

We have implemented the 512-bit variants of the proposed design. Our design is fully autonomous with complete I/O interfaces.[7] We targeted for efficient implementations but keeping in mind the fair hardware performance comparison the proposed design. We assure this approach by catering for the following constraints:

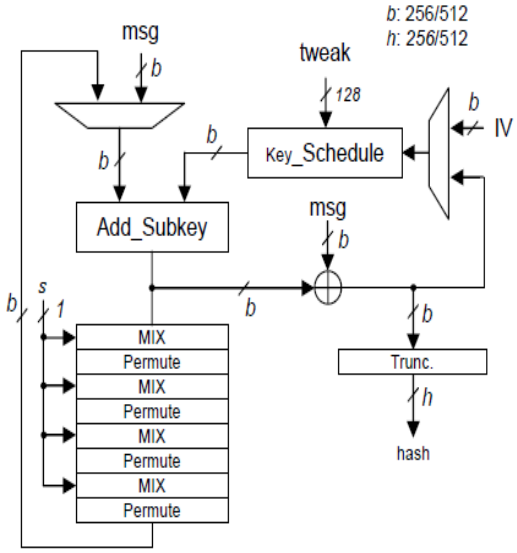
The proposed Design is built from these three components:

- Threefish. Threefish is the tweakable block cipher at the core of design, defined with a 512-bit block size.
- Unique Block Iteration (UBI). UBI is a chaining mode that uses Threefish to build a compression function that maps an arbitrary input size to a fixed output size.
- Optional Argument System. This allows design to support a variety of optional features without imposing any overhead on implementations and applications that do not use the features.

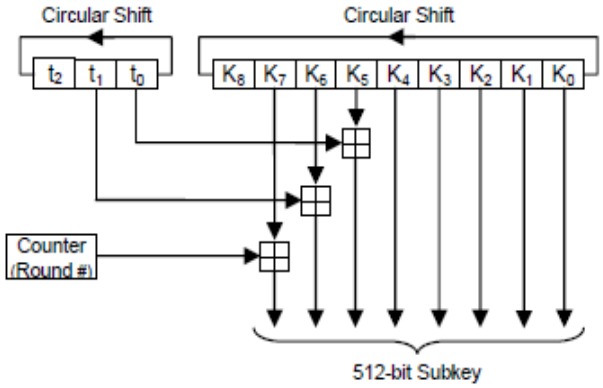
Dividing up our design in this way makes it easier to understand, analyze, and prove properties about. The underlying Threefish algorithm draws upon years of knowledge of block cipher design and analysis.[11] UBI is provably secure and can be used with any tweakable cipher. The optional argument system allows design to be tailored for different purposes. These three components are independent, and are usable on their own, but it's their combination that provides real advantages.

Datapath Architectures for Propose Design

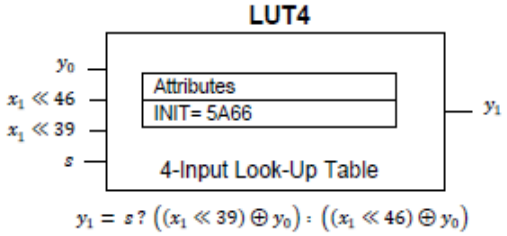
The datapath implemented for the proposed design shown in Fig.(a). Add_Subkey module consists of 8, 64-bit adders, implemented using fast carry chain logic available in Xilinx FPGAs. The Threefish compression function of



(a) Data Path of the Design



(b) Key_Schedule Module



(c) Selection between Two Rotation Constants in MIX Operation

proposed design is partially implemented using 4 unrolled rounds. These 4 rounds are then iteratively used to complete 72 rounds of compression function. The novel idea in implementation of these 4 unrolled rounds is that, we do not need separate MIX modules and multiplexers to select between different rotation constants in second step of MIX operation. We have efficiently implemented second step in MIX module using a LUT4 primitive depicted in Fig.(c).

The select bit s, selects between two rotated instances of x_1 according to round number to XOR with y_0. For first four rounds s is zero and upper half rows of rotation constants' table are used for respective MIX modules. For next four rounds s will be 1 and lower half rows of rotation constants' table are used for respective MIX modules. For example x1<< 46 will be selected and XORed with s0 in first round and x1<< 39 will be selected and XORed with y0 in fifth round. Hardware architecture of key schedule module is shown in Fig.(b). The extended key K8 is obtained by XORing the input 64-bit key words (K0…..K7) and constant C240. The extended teak t2 is obtained by XORing the two input 64-bit tweak word (t0 and t1). The extended key and tweak words are then loaded into the circular shift registers K (576 bit) and t (192 bit). These two registers are clocked and rotated once for each subkey. Key Schedule module generates subkeys on every falling edge of clock pulse. Add_Subkey module gives output on the rising edge of each clock pulse. Next subkey is available on falling edge of the same clock pulse. In this way one clock cycle is required to complete four rounds, subkey addition and subkey generation. Therefore to complete 72 rounds and 19 subkey addition of design, 19 clock cycles will be required. The next chaining hash value will be available after 19 clock cycles.

A Full Specification of Proposed Design Type Values

The Design has many possible parameters. Each parameter, whether optional or mandatory, has its own unique type identifier and value. Type values are in the range 0..63. Design processes the parameters in numerically increasing order of type value, as listed in Table 1.

Table 1: Values for the Type Field

Symbol	Value	Description
Tkey)	0	Key (for MAC and KDF)
Tcfg	4	Con_guration block
Tprs	8	Personalization string
TPK	12	Public key (for digital signature hashing)
Tkdf	16	Key identi_er (for KDF)
Tnon	20	Nonce (for stream cipher or randomized hashing)
Tmsg	48	Message
Tout	63	Output

The Configuration String

The configuration string contains the following data:

- A schema identifier. This is a literal constant. If some other standardization body wants to define an entirely different function based on UBI and Threefish, it can chose a different schema identifier and ensure that its function is different from Skein.
- A version number, to support future extensions.
- No: the output length of the computation, in bits. This ensures that two Skein computations

  that di_er only in the number of output bits give unrelated results.

- Yl: Tree leaf size encoding. Set to 0 if tree hashing is not used.
- Yf : Tree fan-out encoding. Set to 0 if tree hashing is not used.
- Ym: Max tree height. Set to 0 if tree hashing is not used.

The Output Function

The function Output(G;No) takes the following parameters:

G the chaining value.

No the number of output bits required. and produces No bits of output.

The result consists of the leading [No/8] bytes of

O = UBI(G,ToBytes(0, 8), Tout2^{120})jj

UBI(G;ToBytes(1; 8); Tout2^{120})jj

UBI(G;ToBytes(2; 8); Tout2^{120})jj_ _ _

If No mod 8 = 0 the output is an integral number of bytes. If No mod 8 $\neq$ 0 the last byte is only partially used.

Using Function as Simple Hashing

A simple hash computation has the following inputs:[8]

Nb The internal state size, in bytes. Must be 32, 64,

No 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 con_guration string defined as with

Yl = Yf = Ym = 0

We define:

K'= 0^{N}_{b} a string of Nb zero bytes

G0 := UBI(K, C, $T_{cfg}2^{120}$)

G1 := UBI(G0, M, $T_{msg}2^{120}$)

H := Output(G1, No)

where H is the result of the hash.

In its full general form, a design computation has the following inputs:

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

No The output size, in bits.

K A key of Nk bytes. Set to the empty string (Nk = 0) if no key is desired.

Yl Tree hash leaf size encoding.

Yf Tree hash fan-out encoding.

Ym Maximum tree height.

L List of t tuples (Ti;Mi) where Ti is a type value and Mi is a string of bits encoded in a string of bytes.

We have:

L := (T0;M0),…….. (Tt-1,Mt-1)

We require that T_{cfg} < T0, Ti < Ti+1 for all i, and

Tt-1 < Tout. An empty list L is allowed. Each

Mi can be at most 2^{99}- 8 bits (= 2^{96} - 1 bytes) long.

The first step is to process the key. If Nk = 0, the starting value consists of all zeroes.

$$K' = 0^{Nb}$$

If Nk 6= 0 we compress the key using UBI to get our starting value

$$K' = UBI(0^{Nb}, K; Tkey2^{120})$$

Let C be the con_guration string de_ned in Section 3.5.2. We compute:

$$G0 := UBI(K',C, Tcfg2^{120})$$

The parameters are then processed in order:

$$Gi+1 := UBI(Gi,Mi, Ti2^{120}) \text{ for } i = 0\ldots\ldots\ldots\ldots t-1$$

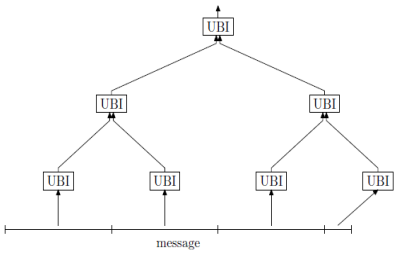
with one exception: if the tree parameters Yl, Yf , and Ym are not all zero, then an input tuple with Ti = Tmsg is processed ,rather than with straight UBI [9].

And the final result is given by:

$$H := Output(Gt;No)$$

Tree Processing

The message input (type Tmsg) is special and can be processed as a tree. Figure 10 gives an example of how tree hashing works. Tree processing is controlled by the three tree parameters Yl, Yf , and Ym in the con_g block. Normally (for non-tree hashing), these are all zero. If they are not all zero, the normal UBI function that processes the Tmsg field is replaced by a tree hashing construction, this is a drop-in replacement of that one UBI function; all other parts of Skein are unchanged.The tree hashing uses the following input parameters:.



An Overview of Tree Hashing

Yl The leaf size encoding. The size of each leaf of the tree is $Nb2^{Yt}$ bytes with Yl ≥1.

Yf The fan-out encoding. The fan-out of a tree node is 2^{Yf} with Yf ≥1.

Ym The maximum tree height; Ym ≥2. (If the height of the tree is not limited, this parameter is set to 255.)

G The input chaining value. This is the G input of the UBI call that the tree hashingreplaces, and the output of the previous UBI function in the Skein computation.

M The message data.

We de_ne the leaf size Nl := Nb2^{Yt} and the node size Nn := Nb2^{Yt}

The message data M is a string of bits encoded in a string of bytes. We _rst split M into one or more message blocks M0,0,M0,1;M0,2; :::;M0,k-1. If M is the empty string, the split results in a single message block M0;0 that is itself the empty bit string. If M is not the empty string, then blocks M0;0; : : : ;M0;k-2 all contain 8Nl bits and block M0;k-1 contains between 1 and 8Nl bits. We now define the first level of tree hashing

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

Note that in the tweak, the tree level field is set to one and the Position field is given an offset equal to the starting offset (in bytes) of the message block.

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

If Ml has length Nb then the result Go is defined by Go := Ml.

If Ml is longer than Nb bytes and l = Ym -1 then we have almost reached the maximum tree

height. The result is defined by:

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

If neither of these conditions holds, we create the next tree level. We split Ml into blocks Ml;0, Ml;1……Ml;k-1 where all blocks but the last one are Nn bytes long and the last block is between Nb and Nn bytes long. We then define

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

and apply the above rules to Ml+1 again.

The result Go is the output of the tree hashing. It becomes the chaining input to the next UBI function in design. (Currently there are no types defined between Tmsg and Tout, so Go becomes the chaining input to the output transformation.) As Yf ≥1 each node of the tree has a fan-out of at least 2, so the height of the tree grows logarithmically in the size of the message input.

Security Claims

The design has been developed to be secure for a wide range of applications, including but not limited to digital signatures, key derivation, pseudorandom number generation, and stream cipher usage. Design supports personalized and randomized hashing. Under a secret key, Design can be used for message authentication and as a pseudorandom function.[12].

Below, we write n for the state size, and m for the minimum of state and output size. We claim the following

levels of security against standard attacks

- First pre-image resistance up to 2^{m}.

- Second pre-image resistance up to 2^m.
- Collision resistance up to $2^{m/2}$.
- Resistance against r-collisions up to roughly min ($2^{n/2}$, $2^{(n-1)m/r}$.) (An r collision consists of r different messages M1, . . . , Mr with H(M1) =…… = H(Mr).)

Summary

In this paper, an architecture with multi-mode operation and the VLSI implementation of the hash function is proposed. The system can support efficiently the security needs, with higher offered security level compared with the previous existing standard hash functions. Furthermore, this proposed system could substitute the implementations of the existing hash standard, in all types of applications, such as digital signatures, message authentication codes and random number generators, with better achieved performance and higher supported security level. The introduced system performs efficiently for the three SHA-2 standard functions (256, 384 and 512). The proposed system covers less area resources compared with previous published implementations [13], and achieves higher operation frequency compared with other related works [13]. In some cases, it also achieves higher performance at about 277 and 417% than other hardware integrations [14].

As we have shown, it is feasible in fact, quite easy to create pseudo-near-collisions and pseudo near-second-preimages for up to eight rounds of any variant of the design. Here, \near" means Hamming-distance 2. Using techniques, one can push this from eight to twelve rounds, at the cost of some significant but feasible amount of work. Assuming close to 2n units of work, it may even be possible to find pseudo-near-second-preimages for up to sixteen rounds of the design-n compression function, for either n = 256, n = 512, or n = 1024.

REFERENCES

1. 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.
2. S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk. Cryptographic hash functions: A survey. Technical Report
3. 95-09, Department of Computer Science, University of Wollongong, July 1995.
4. O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt. Dynamic scheduling of tasks on partially reconfigurable FPGAs. *IEE Proc., Comput. Digit. Tech.*, 147(3):181–188, 2000.
5. N. Shirazi,W. Luk, and P. Y. K. Cheung. Framework and tools for run-time reconfigurable designs. *IEE Proc., Comput. Digit. Tech.*, 147(3):147–152, 2000.
6. R. Stoica, D. Zebulum, R. Keymeulen, T. Tawel, A. Daud, and A. Thakoor. Reconfigurable VLSI architectures for evolvable hardware: From experimental field programmable transistor arrays to evolution-oriented chips, *IEEE Transactions on VLSI*, 9(1):227–232, 2001.
7. 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," manuscript in preparation, 2008
8. J. Goodman and A. P. Chandrakasan. An energy-efficient reconfigurable public-key cryptography processor. *IEEE Journal of Solid-State Circuits*, 36(11):1808–1820, 2001.
9. Xilinx, San Jose, California, USA, Virtex, 2.5 V Field Programmable Gate Arrays, www.xilinx.com, 2002. N. Ferrguson, S. Luck s, B. Schneier, D. Whiting, M . Bellare, T. Koohno, J. Callas, J. Walker, Thee Skein Hash Function Familyy Version 1.3, ppp. 1-100, (2010),,

10. E. Biham and O. Dunkelman, “A Framework for iterative hash functions — HAIFA,” presented at the 2nd NIST hash function workshop, Santa Barbara, 2006.
11. B. Baldwin, A. Byrne, M. Hamilton, N. Hanley, R.P. McEvoy, W. Pan and W.P. Marnane, "FPGA Implementations of SHA-3 Candidates: CubeHash, Groestl, LANE, Shabal and Spectral Hash," in *Digital System Design, Architectures, Methods and Tools, 2009*
12. A.H. Namin, G. Li, J. Wu, J. Xu, Y. Huang, O. Nam, R. Elbaz and M.A. Hasan, "FPGA implementation of CubeHash, Grostel, JH, and SHAvite-3 hash functions," in *NEWCAS Conference (NEWCAS), 2010 8th IEEE International*, 2010, pp. 121-124.
13. M. McLoone, and J. V. McCanny. Efficient single-chip implementation of SHA-384 & SHA-512. In *IEEE Proc., International Conference on Field-Programmable Technology (FTP)*, pp. 311–314, 2002
14. S. Dominikus. A hardware implementation of MD4-family hash algorithms. In *IEEE Proc., International Conference on Electronics Circuits and Systems (ICECS)*, vol. III, pp. 1143–1146, 2002