A VHDL Implementation of the Lightweight Cryptographic Algorithm HIGHT

Fernando Melo Nascimento
Departamento de Computação
Universidade Federal de Sergipe
São Cristóvão – SE, Brazil
Email: nascimentofm@ufs.br

Fernando Messias dos Santos
Departamento de Computação
Universidade Federal de Sergipe
São Cristóvão – SE, Brazil
Email: fernando@ufs.br

Edward David Moreno
Departamento de Computação
Universidade Federal de Sergipe
São Cristóvão – SE, Brazil
Email: edwdavid@gmail.com

***Abstract*—In the modern world, with intensive use of the Internet via mobile devices, is increasingly required new and appropriate security alternatives. Due to size and capabilities limitations of many existing embedded devices, there has been a possibility, using lightweight cryptography. This paper presents a new hardware implementation (VHDL and FPGAs) of the lightweight cryptographic algorithm HIGHT, as well as an overview of cryptography. We show the resources used and the time to encrypt and decrypt a message, and the results are promising.**

Keywords*—*HIGHT; lightweight cryptography; VHDL; FPGA.

I. Introduction

Cryptography can be described as a set of techniques to encrypt (encode) information. The main purpose of encryption is to protect the information contained in a document, as it usually becomes unreadable after this process. To retrieve the information contained in the document, a reverse process called decrypt is required. Encryption techniques may be described by algorithms, which makes it feasible implementation by computers [1].

Over time, several different methods have been developed in order to make it difficult to decode encrypted files. Currently there are sophisticated methods that require a password to retrieve information, such as asymmetric key algorithms, for example. Other methods, such as hash functions, do not allow the decryption of once encrypted content, serving as digital signature of files [1].

The growing evolution of cryptography can be explained by the development of computing. Strongly widespread in modern world, computing has become ubiquitous and is present in the daily lives of most people. A simple example of the importance of encryption in the world today is the growing use of smartphones for banking transactions. If there were no safe ways to protect the information entered on banking applications, invasions to accounts of thousands of people would be easily seen [2], [3].

However, classical cryptographic algorithms such as RSA, ECC and AES are not suitable for all currently devices. Most of them operate based on battery, which requires a low consumption by the algorithm. Others have a reduced capacity and may not support keys over 128 bits, as is common in most asymmetric algorithms. For these devices it is necessary a different approach: the use of lightweight cryptographic algorithms [2], [4], [5].

The basic feature needed for a lightweight cryptographic algorithm is to meet the limitations of the devices for which it was designed, without losing the security focus. Even with limitations in relation to classic algorithms, the lightweight cryptography must match the classical in terms of functionality and safety [2], [6].

Aiming hardware implementations, usually faster than those in software, this paper presents an implementation of the HIGHT algorithm [7] in hardware description language VHDL. This algorithm was chosen based on its results in previous works, discussed in Section III.

The rest of the paper is organized as follows: Section II presents concepts of cryptography. Section III list some related work and Section IV, the adopted methodology. The Subsections IV-A and IV-B contain the description of the algorithm HIGHT and details of its implementation in VHDL, respectively. Section V presents the results obtained. Finally, we present our conclusions and suggestions for further work in Section VI.

II. Technical Background

Two major categories of cryptographic algorithms are the block cipher and stream cipher. The former groups the information to be encrypted into blocks from 8 to 16 bytes before the encryption process and then encrypts the whole block. It is possible to use chaining encryption to hinder attacks. The latter encrypts text one bit at time, using the logical operation xor between the bit and the key [1].

A key is a unique string that is used to encrypt and/or decrypt a file. There are two classes of algorithms with respect to the key: symmetric and asymmetric. Symmetric algorithms use only one key to encrypt and decrypt the message, which is a private key that must be distributed among all parties involved in the communication. In the asymmetric cryptography, there are two keys, one public and another private. They function as a plug: while one is used to encrypt, the other is able to restore the information and vice-versa [1].

A lightweight encryption application is its possible use in Radio Frequency Identification (RFID) technology, an automatic identification method using radio signals. RFID tags generally have high limitations of computational resources, such as those listed by Saarinen and Engels in [4]:

- The total integrated circuit area available for implementing the entire device's logic, including the security, is reduced.
- The power used can affect the maximum distance of communication between a reader and the RFID tag – when there are power peaks, this range is decreased.
- The power depends on the device's clock frequency. Thus, security implementations need to minimize the use of clock cycles.

The lightweight cryptography is, in fact, the best security option for such devices. As noted by Katagi and Moriai [2] lightweight cryptographic algorithms meet precisely the restrictions lifted by Saarinen and Engels [4]. Especially algorithms designed to hardware fit this case: low power consumption and small footprint, which leads to interest in implementing such algorithms in VHDL.

III. Related Work

Eisenbarth et al. [5] compared some light cryptographic algorithms implemented in software. The difficulty in reducing the size of HIGHT code did not impact on performance: both encryption and decryption led 2964 cycles per block, reaching a transfer rate of 80.3 Mbps at 4 MHz – the fastest of hardware-oriented algorithms.

Cazorla et al. [8] made the implementation of 17 Cryptographic Algorithms in C programming language, analyzing aspects such as software performance and memory usage. HIGHT used 18 and 3130 bytes of RAM and ROM, respectively, one of the few hardware-oriented algorithms with good results.

Alizadeh et al. [9] implemented the algorithms HIGHT, KATAN, KLEIN and TEA in Atmel AVR ATtiny45 microcontroller using assembly. The KLEIN performed better in terms of energy consumption ($25.98\,\mu$J against $80.87\,\mu$J of HIGHT), but its memory usage was the highest among all others. KATAN obtained the best results in this regard, but consumed more energy. HIGHT and TEA had similar performances, being a good alternative in general cases.

Koo et al. [10] implemented the algorithms RC5, Skipjack and HIGHT on Mica2 Mote sensor based on 8 bit Atmel AVR processor. The usage of the RAM and ROM of HIGHT was 568 and 3906 bytes (greater usage of RAM and second largest ROM use), respectively. The operation time 7.413 s was 200 ms higher than the other two algorithms. In power consumption, measured in CPU cycles, HIGHT reached second place with 64355 cycles. Although Skipjack provided better results than HIGHT, it does not provide enough security [11].

The VHDL implementation of HIGHT in FPGAs by Yalla and Kaps [12] reached good results: only 91 slices consumed and encryption speed of 65.5 Mbps. A Xilinx Spartan-3 board was used for compilation. This implementation exceeded all lightweight implementations of AES published so far.

IV. Methodology

A. Algorithm Description

HIGHT algorithm *(HIGH security and light weighT)* was proposed in 2006 by Hong et al. [7]. This is a block cipher lightweight cryptographic algorithm. The size of each block is 64 bits and the keys have 128 bits. This algorithm is characterized as a low-cost and low-power, having an ultra-light implementation [7], [8].

HIGHT operating principle is based on xor operations, additions to applying $mod\ 2^8$ and shifts bitwise to the left. Original paper [7] implementation was performed with 3048 logic elements in a $0.25\,\mu$m technology. Tables I and II contain the meaning of variables and operators used in the algorithm, respectively.

TABLE I. Meaning of used Variables

Variables	**Meaning**
P	Initial block to be encrypted
C	Final block (already encrypted)
MK	128 bits user Master key
WK	128 bits Whitening key
SK	Set of 128 8 bits Sub keys
$X_{[0,32]}$	Auxiliary blocks used during processing
delta	Constant vector generated by LFSR

TABLE II. Meaning of used Operators

Operators	**Meaning**
(xor)	Exclusive OR bitwise
(plus)	Modular addition ($mod\ 2^8$)
(shl)	Bitwise left shift

HIGHT algorithm can be described by the pseudo code in Figure 1, based on [7]. The described algorithm takes into account only the encryption process since, the decryption is simple reverse the functions of processing and rotation, and also the order of application thereof. Each procedure used in the algorithm can be described as follows:

1) *HightEncryption*: Receives the text P and the user's master key MK. This is the main procedure.
2) *KeySchedule*: Receives the master key MK and call the methods of key generation 3) and 5).
3) *WhiteningKeyGeneration*: Use the master key MK to generate the 8 bytes white key WK used in the initial and final transformations.
4) *ConstantGeneration*: Generates 128 7 bit delta constant values to be used in the generation of sub keys SK by a LFSR process. The initial delta is $(1011010)_2$ – 0x5A in hexadecimal.
5) *SubkeyGeneration*: Generate, from the delta constant vector, the 128 8 bits sub keys SK used in text rotation function – 4 bytes of SK by RoundFunction.
6) *InitialTransfomation*: Transforms the input text P into X_0 through (xor) and (plus) operations with the first 4 bytes of WK key.
7) *RoundFunction*: Uses the auxiliary functions F0 and F1 to rotate and generate new intermediaries texts X_i.
8) *FinalTransfomation*: Transforms the last intermediate text X_{32} into output text C through (xor) and (plus) operations with the last 4 bytes of key WK.

```
HightEncryption(P, MK)
  KeySchedule(MK, WK, SK)
  InitialTransfomation(P, X, WK)
  for i := 0 to 31
    RoundFunction(X, SK, i)
  FinalTransfomation(X, C, WK)

KeySchedule(MK, WK, SK)
  WhiteningKeyGeneration(MK, WK)
  SubkeyGeneration(MK, SK)

WhiteningKeyGeneration(MK, WK)
  for i := 0 to 7
    if (i in [0,3]) then WK_(i) := MK_(i+12)
    else WK_(i) := MK_(i-4)

ConstantGeneration
  s0 := 0; s1 := 1; s2 := 0; s3 := 1; s4 := 1; s5 := 0; s6 := 1;
  delta_0 := s6||s5||s4||s3||s2||s1||s0
  for i = 1 to 127
    s_(i+6)    := s_(i+2) (xor) s_(i-1)
    delta_(i) := s6||s5||s4||s3||s2||s1||s0

SubkeyGeneration(MK, SK)
  Run ConstantGeneration
  for i := 0 to 7
    for j := 0 to 7
      SK_(16i+j)    := MK_(j-i mod 8) (plus) delta_(16i+j)
      SK_(16i+j+8)  := MK_(j-i mod 8 + 8) (plus) delta_(16i+j+8)

InitialTransfomation(P, X, WK)
  X_(0,0) := P_(0) (plus) WK_(0)
  X_(0,1) := P_(1)
  X_(0,2) := P_(2) (xor)  WK_(1)
  X_(0,3) := P_(3)
  X_(0,4) := P_(4) (plus) WK_(2)
  X_(0,5) := P_(5)
  X_(0,6) := P_(6) (xor)  WK_(3)
  X_(0,7) := P_(7)

F0(X) := (X(shl)1) (xor) (X(shl)2) (xor) (X(shl)7)
F1(X) := (X(shl)3) (xor) (X(shl)4) (xor) (X(shl)6)

RoundFunction(X, SK, i)
  X_(i+1,1) := X_(i,0)
  X_(i+1,3) := X_(i,2)
  X_(i+1,5) := X_(i,4)
  X_(i+1,7) := X_(i,6)
  X_(i+1,0) := X_(i,7) (xor)  (F0(X_(i,6)) (plus) SK_(4i+3))
  X_(i+1,2) := X_(i,1) (plus) (F1(X_(i,0)) (xor)  SK_(4i+2))
  X_(i+1,4) := X_(i,3) (xor)  (F0(X_(i,2)) (plus) SK_(4i+1))
  X_(i+1,6) := X_(i,5) (plus) (F1(X_(i,4)) (xor)  SK_(4i))

FinalTransfomation(X, C, WK)
  C_(0) := X_(32,1) (plus) WK_(4)
  C_(1) := X_(32,2)
  C_(2) := X_(32,3) (xor)  WK_(5)
  C_(3) := X_(32,4)
  C_(4) := X_(32,5) (plus) WK_(6)
  C_(5) := X_(32,6)
  C_(6) := X_(32,7) (xor)  WK_(7)
  C_(7) := X_(32,0)
```

Fig. 1. HIGHT's Pseudo-code

B. VHDL Implementation

Our VHDL implementation of HIGHT was based on a software implementation in C programming language by Cazorla et al. [8]. We used a finite state machine (FSM) abstraction in our implementation shown in Figure 2.

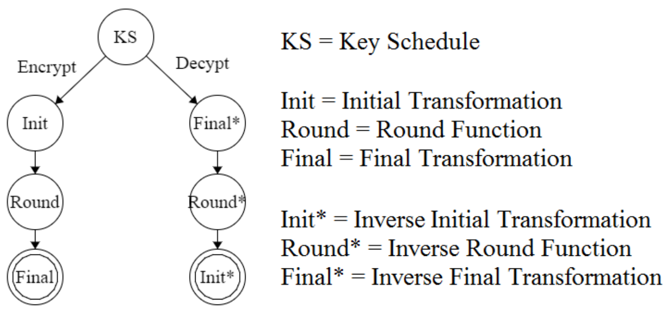
KS
Encrypt
Decypt
Init
Final*
Round
Round*
Final
Init*
KS = Key Schedule
Init = Initial Transformation
Round = Round Function
Final = Final Transformation
Init* = Inverse Initial Transformation
Round* = Inverse Round Function
Final* = Inverse Final Transformation

Fig. 2. HIGHT's Finite State Machine

The algorithm input is a text P and a master key MK. There is also a control signal, designating which operation should be performed: encrypt or decrypt the text. Moreover, as the implementation was based on FSM, it was necessary to use clock and reset, since, to transition to a next state, it must be ensured that all operations of current state finished.

For a better code organization, it was divided into two files: one containing the logic of the FSM and the other with the implementations of the functions and procedures. The main code IV-B contains the entity and architecture, where inputs, outputs, operations and FSM transitions are defined. Auxiliary code consists of a package included by main code.

```
---------------------------------- Libraries: ----------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use work.hight_package.all;
----------------------------------- Entity: ------------------------------------
entity HIGHT is
  port (
    txt_in: in std_logic_vector (0 to 63);
    mk: in std_logic_vector (0 to 127);
    clk: in std_logic;
    reset: in std_logic;
    encrypt: in std_logic;
    txt_out: out std_logic_vector (0 to 63) );
end HIGHT;
-------------------------------- Architecture: ---------------------------------
architecture HIGHT of HIGHT is
------------------------------- Type Definition --------------------------------
type states is (KEY_SCHEDULE, INITIAL, ROUND, FINAL,
INVERSE_FINAL, INVERSE_ROUND, INVERSE_INITIAL, OUTPUT);
----------------------------- Signals Declaration ------------------------------
signal STATE: states;
signal wk: std_logic_vector (0 to 63);
signal sk: std_logic_vector (0 to 1023);
signal txt: std_logic_vector (0 to 63);
-------------------------------- Logic of FSM  ---------------------------------
begin
  process (clk, reset)
  begin
    if reset = '1' then
      STATE <= KEY_SCHEDULE;
    elsif clk'event and clk='1' then
      case STATE is
        when KEY_SCHEDULE =>
          WhiteningKeyGeneration (mk, wk);
          SubkeyGeneration (mk, sk);
          if encrypt = '1' then
            STATE <= INITIAL;
          else
            STATE <= INVERSE_FINAL;
          end if;
          txt <= txt_in;

        when INITIAL =>
          InitialTransfomation (wk, txt);
          STATE <= ROUND;

        when ROUND =>
          RoundFunction (sk, txt);
          STATE <= FINAL;

        when FINAL =>
          FinalTransfomation (wk, txt);
          STATE <= OUTPUT;

        when INVERSE_FINAL =>
          inverseFinalTransfomation (wk, txt);
          STATE <= INVERSE_ROUND;

        when INVERSE_ROUND =>
          inverseRoundFunction (sk, txt);
          STATE <= INVERSE_INITIAL;

        when INVERSE_INITIAL =>
          inverseInitialTransfomation (wk, txt);
          STATE <= OUTPUT;

        when OUTPUT =>
          txt_out <= txt;
      end case;
    end if;
  end process;
end architecture;
```

Fig. 3. VHDL Main Code

V. Results and Discussion

We used the Altera tools Quartus II – versions 11.0 and 13.0.0 (web edition) – for the VHDL code compilation. The target board belongs to Cyclone II family, model EP2C3F672C6 with a total of 33216 logic elements. Table III contains a summary of the build process by the two versions of the tool.

TABLE III. Time Simulation Results

Attributes	Version 11.0	Version 13.0.0
Logical Elements	6447	6608
Combinational Functions	6321	6321
Total Registers	1014	1014
Total Pins	259	259

We observed that the total amount of logic elements in version 11.0 was 6447, whereas in version 13.0.0, the number rose to 6608. Both results have more than twice as many logical elements suggested by Hong et al. [7] (3048). We justify this fact by the presence of both encryption and decryption processes.

In functional simulations, generated by an Altera simulation tool, we could verify the correctness of our implementation. We performed an encryption with random P and MK. The result text C was then set as input in two decryption processes: the first with the same MK and the second with an one-bit different MK. As expected, the former was successful in retrieving the original text P, while the latter was not.

Time simulations were also performed for both versions of Quartus II. Table IV summarizes the results obtained in those simulations. Column **# Clocks** represents the needed number of clocks to complete the encryption/decryption process. **Delay** is relative to the amount of time elapsed after the rise of the last clock to stabilize output, a period in which there is generation of noise at the output.

TABLE IV. Time Simulation Results

Version	Frequency	# Clocks	Clock size	Delay	Transfer Rate
11.0	9.25 MHz	5	108.0 ns	$\approx$ 13.7 ns	128 Mbps
13.0.0	11.36 MHz	5	88.0 ns	$\approx$ 9.8 ns	157 Mbps

Table V synthesizes ours results of frequency and transfer rate of HIGHT implementation and those obtained by other authors.

TABLE V. Synthesized Results

Implementation	Frequency	Transfer Rate	Implementation
Version 11.0	9.25 MHz	128 Mbps	Hardware (VHDL)
Version 13.0.0	11.36 MHz	157 Mbps	Hardware (VHDL)
Eisenbarth et al. [5]	4 MHz	80.3 Mbps	Software
Yalla and Kaps [12]	not available	65.5 Mbps	Hardware (VHDL)

Eisenbarth et al. [5] implementation has a better transfer rate than Yalla and Kaps [12]. However, the former used a software implementation, while the latter, a hardware one. Our results outperformed both implementations. It is important to notice that Yalla and Kaps [12] used a optimized FSM approach, with a counter and additional logic for generate control signals, while we implemented a simpler FSM, using clock and reset in VHDL code.

VI. Conclusion

Cryptography is a constantly advancing area, being driven by ubiquitous computing. Due to the large amount of important devices with limited space and power consumption, the lightweight cryptography is a viable solution for security on such devices. In particular, algorithms implemented in hardware are usually more efficient than those designed for software.

This work presented a VHDL implementation of HIGHT algorithm. Some tests were conducted to demonstrate the functionality and correctness of it. With simulations in versions 11.0 and 13.0.0 of the Altera Quartus II tool, we reached a transfer rate of, respectively, 128 Mbps and 157 Mbps at 9.25 MHz and 11.36 MHz. These results were higher than those found by Eisenbarth et al. [5] and Yalla and Kaps [12].

The main suggestion of future work is to optimize our VHDL code to reduce the amount of logic elements used. To do so, one way is to optimize the FSM used. Other possible path to follow is the implementation of other lightweight cryptographic hardware-oriented algorithms in VHDL language to compare the results obtained previously by other authors.

References

[1] E. D. Moreno, F. D. Pereira, and R. B. Chiaramonte, *Criptografia em Software e Hardware*. Novatec Editora, 2005.

[2] M. Katagi and S. Moriai, *Lightweight Cryptography for the Internet of Things*, Sony Corporation, October 2008.

[3] A. Poschmann, *Lightweight Cryptography: Cryptographic Engineering for a Pervasive World*, ser. IT-Security. Europ. Univ.-Verlag, 2009.

[4] M.-J. O. Saarinen and D. W. Engels, "A do-it-all-cipher for rfid: Design requirements (extended abstract)." *IACR Cryptology ePrint Archive*, vol. 2012, p. 317, 2012, informal publication.

[5] T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel, "A survey of lightweight-cryptography implementations," *IEEE Des. Test*, vol. 24, no. 6, pp. 522–533, 2007.

[6] E. D. Moreno, "Platforms and applications in hardware security: Trends and challenges," *International Journal of Information and Computer Security*, vol. 7, no. 5, pp. 289–304, October 2013.

[7] D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, H. Kim, J. Kim, and S. Chee, "Hight: A new block cipher suitable for low-resource device," in *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop*, ser. Lecture Notes in Computer Science, vol. 4249. Springer, 2006, pp. 46–59.

[8] M. Cazorla, K. Marquet, and M. Minier, "Survey and benchmark of lightweight block ciphers for wireless sensor networks," *IACR Cryptology ePrint Archive*, vol. 2013, p. 295, 2013.

[9] M. Alizadeh, M. Salleh, M. Zamani, J. Shayan, and S. Karamizadeh, "Security and performance evaluation of lightweight cryptographic algorithms in RFID," *WSEAS Conference on Recent Researches in Communications and Computers*, pp. 45–50, July 2012.

[10] W. K. Koo, H. Lee, Y. H. Kim, and D. H. Lee, "Implementation and analysis of new lightweight cryptographic algorithm suitable for wireless sensor networks," in *2008 International Conference on Information Security and Assurance (isa 2008)*. IEEE, April 2008.

[11] E. Biham, A. Biryukov, and A. Shamir, "Cryptanalysis of skipjack reduced to 31 rounds using impossible differentials," in *Advances in Cryptology*, ser. Lecture Notes in Computer Science, J. Stern, Ed. Springer Berlin Heidelberg, 1999, vol. 1592, pp. 12–23.

[12] P. Yalla and J.-P. Kaps, "Lightweight cryptography for FPGAs," in *International Conference on ReConFigurable Computing and FPGAs – ReConFig'09*. IEEE, Dec. 2009, pp. 225–230.