Electronics and Computer Science
Faculty of Physical and Applied Sciences
University of Southampton

Author: Lewis Smith

April 28, 2018

Low Power Hardware Accelerated Internet of Things Cryptography

Project Supervisor: Mark Zwolinski
Second Examiner:

A project report submitted for the award of MEng Electronic Systems with Computer Systems

Abstract

Contents

Acknowledgements

Chapter 1

Introduction

As the speed and global reach of the internet has expanded over the years the number of devices connected to it has rapidly increased. These devices are no longer just the servers and the PC's connected to them, they now include consumer devices like smartphones, tablets, and games consoles. However, even more recently the idea of connecting the internet to various 'dumb' appliances like: simple light switches; kettles; fridges and many more; to make them 'smart' devices that can be controlled through small embedded processors has emerged. The idea of connecting such devices to internet has been dubbed 'Internet of Things' or 'IoT' and has the aim to make our lives simpler. The 'IoT' concept is also being explored for more industrial applications such as: automated factories; city electrical grids; and even a network of self-driving cars[1] but this is far more advanced than the basic 'smart' home.

Due to the wide range of products that a connected IoT device can be applied to improve the efficiency and/or usefulness, it has been predicted that billions of devices will be in use by 2020[2]. This also means that the complexity of the devices varies greatly. The one thing that all of these devices have in common though is that they need to be secure as they communicate sensitive and private data through an open channel on the internet between the user and the device. To keep the potential adversaries from accessing the data and possibly controlling numerous connected devices, maliciously or not, an encryption algorithm can be used.

POSSIBLY CHANGE/IMPROVE BELOW

There are many encryption algorithms that perform this function and most can be implemented in both software and dedicated hardware such as an Application Specific Integrated Circuit (ASIC) or a Field Programmable Gate Array (FPGA). As a majority of IoT devices are implemented on small embedded processors which have limited resources, the hardware option might possibly be a better solution for IoT devices. However, due to the fact that most IoT devices are always on, and likely battery powered, power consumption is a very important factor when considering options for adding hardware accelerated encryption and for battery powered devices it is often more critical than the actual encryption.

The goals of this project are to explore various encryption algorithms and compare their performance based on data throughput, accuracy, security and power consumption when implemented in software and hardware. To evaluate these parameters the

same algorithms can be coded in C or C++ for the software versions and a Hardware Development Language (HDL) such as System Verilog can be first simulated in ModelSim, before programming a FPGA for the hardware version. These comparisons can then be used to match the algorithms to the appropriate IoT device as they all have different requirements for relative security level and power consumption, as for example a light switch does not necessarily need to be protected from the same level of attack as a set of digital locks or private data storage. In order for the hardware to work with IoT devices it will also need a communication protocol like I^2C or SPI to work with embedded processors, and possibly Ethernet or WiFi to act as the gateway to internet for the device. Some of these protocols are available on FPGA development boards but can be implemented in System Verilog code.

Chapter 2

Background Research & Literature

2.1 Internet of Things

CHANGE/IMPROVE BELOW [100]

As mentioned in chapter 1 there are many IoT devices that require varying levels of security and have to be protected against different of attacks, like side channel attacks. The purpose of this subsection is to discover what these devices are and what properties are required in the encryption algorithm in terms of throughput, power or energy consumption, software or hardware restrictions and security. Due to IoT devices having limited resources a provisional limit of 2000 Gate Equivalents in hardware is the maximum size for most embedded platforms but even that might be to big for devices like RFID tags[**?**]. Power consumption should also be kept to as little as possible but a limit of tens of micro Watts (μW) for RFID tags is suggested[**?**].

2.2 Cryptography

POSSIBLY CHANGE/IMPROVE BELOW

After evaluating the conditions required to provide the appropriate level of security in section 2.1 this subsection explores the cryptographic principles and algorithms available that satisfy those conditions. The primary objective of cryptography is to convert, or encrypt, a readable message known as plaintext into an unreadable form, ciphertext, so that adversaries cannot read the contents, but over the years the scope of cryptography has widened. Throughout history encryption has been has been used allow people and groups to exchange secret messages, especially in times of war. Since the early transposition and substitution ciphers, where each character in a message are rearranged and replaced by others a certain number further down the alphabet respectively, encryption has evolved to include techniques for identity authentication, integrity checks and much more[insert reference]. Cryptography is therefore the study of encryption and other techniques, including identity authentication and integrity

checks. Its counterpart: the study of breaking the encryption to find the original message, is known as cryptanalysis[?]. Eventually, for most encryption techniques a weakness is found, and subsequently exploited, so more complex techniques are conceived and with the invention of computers the complexity of the algorithms has increased greatly. However, the computer power is also available for cryptanalysis so the cycle of continuous improvement of the algorithms hasn't stopped.

In cryptography there are two main concepts that the algorithms are based on: symmetric and asymmetric keys[insert reference]. In asymmetric key cryptography a unique key is used to encrypt data and different, but a related key, is used to decrypt it. The relationship between the two keys is often defined by maths problem that is very difficult to solve which is the basis of the encryption[insert reference]. On the other hand, symmetric key cryptography uses the same key for both encryption and decryption, hence symmetric, with the security usually provided by a combination of simple logic operations[insert reference].

2.2.1 Asymmetric Key

POSSIBLY CHANGE/IMPROVE BELOW

Asymmetric key cryptography can be referred to as public key due to the fact that one of the related keys can be publicly available without compromising the security of the encrypted data. This is because the keys are usually generated based on mathematical problems that have no solution or the solution is impossible for a computer to solve efficiently, such that solving it takes longer than an exhaustive key search[?]. There are many problems that fit this criteria but the most popular in use today are the integer factorization and elliptic curve problems used by the RSA[insert reference] and the family of Elliptic-curve cryptography (ECC) techniques[insert reference] respectively. Public key cryptography can be used in two different modes as if data is encrypted with the intended recipients public key only they can decrypt it with their private key, thus encryption. However, if a private key is used for encryption then using the public key to decrypt it ensures the senders identity, authentication[?].

2.2.2 Symmetric Key

POSSIBLY CHANGE/IMPROVE BELOW

Similar to the symmetric/private comparison symmetric key cryptography is also known as private key, as in order to keep the encrypted data secure the key used must be kept secret. There are two main types of private key algorithms that operate on the plaintext differently: block ciphers which uses a fixed number of bits, block; or stream ciphers which encrypts data bit by bit[?].

Modern block ciphers are based on Claude Shannons work on product ciphers[?], in which he suggested that iterating a cipher for multiple rounds, with subkeys, improves the security. Hence, the cipher to be iterated didn't need to be complex operations and simple logic operations such as XOR, substitution or permutation of the plaintext could be used[insert reference]. The base cipher that is iterated is

known as the round function and it takes as an input a block of plaintext and a subkey, which is generated from the main key by a separate key expansion function, and outputs a block of ciphertext. The output is usually the result of the round function XORed with the subkey. The round functions are mostly designed using either a Feistel network[?] (F network) or a Substitution Permutation network (SP network)[?].

The Fesitel network was named after physicist Horst Feistel who was a integral part of the team at IBM that developed the early block cipher Lucifer, which of course used a Feistel network[insert reference]. The F network works by splitting the input plaintext into two equal words, known as the left (MSB) and right (LSB) words. The round function is then applied to the right word before the result is XORed with the left word and then the words are swapped over and iterated as in Equation 2.1 and 2.2, with the ciphertext being equal to (R_{n+1}, L_{n+1}) where n is the number of rounds iterated. The advantage of using a F network is that decryption is just applying the same algorithm but with the sub keys in reverse as in Equation 2.3 and 2.4, with the ciphertext (R_{n+1}, L_{n+1}) as the input and the plaintext (L_0, R_0) returned.

$$L_{i+1} = R_i \tag{2.1}$$

$$R_{i+1} = L_i \oplus F(R_i, K_i) \tag{2.2}$$

$$R_i = R_{i+1} \tag{2.3}$$

$$L_i = R_{i+1} \oplus F(L_{i+1}, K_i) \tag{2.4}$$

On the other hand, Substitution Permutation networks operate on the whole plaintext block using S-boxes for substitution and P-boxes for permutation. Individually, these operations aren't particularly strong as a S-box and a P-box can be thought of as simple substitution and transposition ciphers respectively. However, when combined in a SP network over multiple rounds the security can be very strong due to Shannon's confusion, provided by the S-boxes, and diffusion, P-boxes, properties being satisfied[?]. The S-boxes usually take in a certain number of bits and outputs the same number of bits but of a different value. P-boxes are then used to spread the bits around such that the output of the S-boxes are used by as many S-boxes in the next round. After the S-boxes and P-boxes and before the next round occurs the block is XORed with the round key so the round equation is Equation 2.5. Decryption, Equation 2.6 is achieved using inverted S-boxes and P-boxes and the round keys in reverse order which means that different hardware or operations are needed.

$$B_{i+1} = F(B_i) \oplus K_i \tag{2.5}$$

$$L_i = R_{i+1} \oplus F(L_{i+1}, K_i) B_i = F'(B_{i+1}) \oplus K_i \tag{2.6}$$

CHANGE/IMPROVE BELOW

Stream ciphers were initially designed to approximate the One Time Pad (OTP) cipher that was proved to be completely unbreakable by Claude Shannon[insert reference]. The OTP works by combining each digit of the plaintext with a completely random keystream. The stream ciphers work by generating a pseudo-random keystream to combine with the plaintext[?]. Because the keystream is pseudo-random

and not completely random a stream cipher is breakable. The keystream is created by a pseudo-random number generated with a cryptographic key used as a seed.

There are also some modes of operation for block ciphers, in [?], that provide better security by using feedback of the ciphertext to the next block. Some of these modes of operation also allow block ciphers to behave similar to stream ciphers as they encrypt an initialization vector with the key and the resulting ciphertext can be combined with the plaintext.

2.2.3 Decisions

POSSIBLY CHANGE/IMPROVE BELOW

Due to the fact that asymmetric key algorithms are hard to solve they require complex hardware or software to implement which is undesirable for this project. Also, with the exception of ECC the key sizes needed for the security can be very large so with the limited IO pins available on FPGAs they could prove difficult to program. On the other hand, many private key algorithms are designed to be efficient in hardware especially Feistel networks as an inverted round function isn't required. While a stream cipher can be useful to encrypt serial data that will most likely be the source, the modes of operation available for block ciphers provide more flexible functionality including stream cipher modes. Therefore, the algorithm chosen for this project will most likely be a block cipher with a Feistel network.

2.3 Conventional Algorithms

POSSIBLY CHANGE/IMPROVE BELOW

There are many block ciphers that are considered very secure and therefore popular, they include: DES[?], AES[?], Blowfish[?]. DES operates on a block of 64 bits for 16 rounds using a key length of 64 bits but it has an effective key length of 56 bits as 8 bits were used for parity. AES, an upgrade to DES, is far more secure as uses a 128 bit block and has the flexibility of using three different key lengths: 128, 192 and 256. The number of rounds that AES iterates depends on the key length with 10 rounds used for a 128 bit key, 12 for 192, and 14 for the largest key. Blowfish, like DES, operates on a 64 bit block and iterates for 16 rounds, but it can use a variable key length in the range 32 to 448 bits.

2.3.1 Standardization

DES, which stands for Data Encryption Standard, is one the earliest block ciphers used in the computer age and it was developed by IBM in the 1970s based on their earlier cipher Lucifer[insert reference]. As with Lucifer it was designed around a Feistel network but the round function used also has a SP network structure to it[insert reference],but the S-boxes aren't a one-to-one function but rather output 4 bits from a 6 bit input. It has the name Data Encryption Standard as it was

accepted as the standard encryption algorithm by the US National Bureau of Standards (NBS), now the National Institute of Standards and Technology (NIST), in 1977 after it was altered by the National Security Agency (NSA), which caused some controversy[?].

DES was used for about two decades but in the 1990s several successful attacks proved its weakness[?] so in 1997 NIST started a selection process to find its replacement. Due the controversy of the NSAs involvement and comments from the cryptography community the selection process was as transparent as possible[insert reference]. Many algorithms were submitted as candidates for the standard but the finalists were: MARS, RC6, Rijndael, Serpent, and Twofish[insert reference]. It took three years to decide on the algorithm to be set as the standard which was announced as Rijndael in 2000 and the standard was et in 2001, with the 128, 192, 256 bit keys being used in the standard[?]. Unlike DES, AES uses a SP network as it is efficient, in time, in both hardware and software. The round function treats the block as a $4x4$ byte matrix and performs multiple steps on the data: sub bytes; shift rows; mix columns and add round key[insert reference].

Since its standardization in 2001 AES has been used almost exclusively because its security is trusted. Because of this there are many different software and hardware implementations produced with some concentrating on side-channel attack resistance[?] or efficient S-box implementations[?]. However, even area optimized designs like [?] use 2400 gate equivalents which is too much for lightweight applications.

2.3.2 Other Algorithms

The US standardized algorithms quickly became very popular and can be considered the unofficial global standard. Although, there are many other algorithms that are considered secure and are commonly used. One of these algorithms, TripleDES, is actually based around DES which increases the security by encrypting the plaintext three times with separate keys making the effective key length of 168 bits. These algorithms might be used because there is still some distrust of the NSAs involvement in the algorithms and they are more open source. This is the case of the Blowfish algorithm as it is unpatented and can therefore be used in any product without legal consequence.

Blowfish was designed by Bruce Schneier in the early 1990s as he, and many others, noticed the insecurity of DES particularly with the 56 bit key length making a brute force attack more plausible[?]. The design of the algorithm is based around a Feistel network with, similar to DES, the round function using S-boxes. The sub keys and S-box lookup tables are generated using the hexadecimal digits of pi which are provided by the designers[insert reference]. As with AES there are many FPGA and ASIC implementations of the Blowfish algorithm, [?] and [?], but they require too many FPGA resources to be considered for the lightweight nature of this project.

2.3.3 Decisions

Due to the conventional algorithms not being explicitly designed for hardware and definitely not for lightweight applications they are not appropriate for this project. Although, they are useful for comparison with the lightweight algorithms in section 2.4 in terms of security and throughput.

2.4 Lightweight Algorithms

After deciding that the conventional algorithms might not be suitable for the low power devices targeted by this project, some more lightweight algorithms were found including: PRESENT[**?**], PRINCE[**?**] and the SIMON and SPECK algorithms[**?**]. However, as IoT is an emerging technology and is the main reason for lightweight cryptography there isn't a standard set by NIST, but the process has begun in [**?**]. These algorithms are considered lightweight because they make sacrifices in and security or throughput, or both, to achieve small area and low power designs.

2.4.1 SIMON & SPECK

The SIMON and SPECK family of algorithms are the lightweight techniques proposed by the NSA that were designed to perform well in both software and hardware while still being secure; and to be flexible in terms of block and key size, listed in Table 2.1. The algorithms are similar but SIMON was optimised for hardware implementations and SPECK for software. As with AES the number of rounds iterated depends on the key size but as the block size varies as well it also has an effect as shown in Table 2.1. The structure of both algorithms is a Feistel network and thus it works on words of N bits, where $2N$ is the block size, and with a key of $M * N$ bits. This lends it self to the naming format of SIMON or SPECK $2N/N * M$, which means, for example, SIMON48/72 has a word size of 24 and uses 3 words for the key[insert reference].

Block Size	Key Size	N	M	SIMON Rounds	SPECK Rounds
32	64	16	4	32	22
48	72	24	3	36	22
48	96	24	4	36	23
64	96	32	3	42	26
64	128	32	4	44	27
96	96	48	2	52	28
96	144	48	3	54	29
128	128	64	2	68	32
128	192	64	3	69	33
128	256	64	4	72	34

Table 2.1: A table of the modes of operation for the SIMON & SPECK Algorithms. Adapted from [**?**].

Even though they are relatively new algorithms there are still a few FPGA implementations available for review, including [**?**] and [**?**] as well as those provided in [**?**]. These all show that very small designs are possible with even the 128/256 versions fitting below the 2000 GE limit.

2.4.2 Other Algorithms

The PRESENT cipher is another option for lightweight cryptography as it achieves a 1570 GE FPGA design with a 80 bit working on a 64 bit block. there is also an version that uses 128 bit key. The design of the cipher is based around a SP network, **??**, with 64 bit subkeys.

PRINCE is a lightweight cipher that can encrypt data in one clock cycle with an unrolled SP network, as SP networks require less rounds than a F network. The steps include S-boxes and a matrix layer as well as the XORing of the round key and a round constant. Due to the unrolled nature of the algorithm the FPGA implementations are mainly combinational so the register count is lower than other algorithms.

2.4.3 Decisions

Based on the research explored in chapter 2 I chose the SIMON and SPECK algorithms from the NSA, mainly because of their flexibility in security levels, with different key lengths, that could be applied to the different devices explored in section 2.1. This means that multiple algorithms don't need to be developed and I could concentrate on making my code as efficient as possible. When compared in terms of power consumption the SIMON64/96 version in [**?**] also shows lower power consumption than the other algorithms explored. Also, while both PRESENT and PRINCE meet the lightweight specification they are also SP networks and in subsection 2.2.3 it was decided that a Feistel network is preferable.

After deciding on the SIMON and SPECK family I explored how each version works in order to make a more informed decision on which to work with in this project. As SIMON was designed primarily for hardware it only makes use of XOR ($\oplus$), AND ($\&$) and circular rotate operations ($R^j[x]$) on the n bit wide words. For the rotate operation the word x is rotated by j bits to the left or right if j is negative. The encryption and decryption functions take the Feistel network form described in **??** with the round function Equation 2.7.

$$F(x) = (R^1[x] \& R^8[x]) \oplus R^2[x] \tag{2.7}$$

SPECK on the other hand, being optimised for software implementations uses XOR ($\oplus$), modulo 2^n addition ($+$) and circular rotate operations ($R^j[x]$), with the circular rotate being equivalent to what is used in SIMON. The encryption and decryption functions take a slightly different form to the basic Feistel network, **??**, and are shown in Equation 2.8 and 2.9 where $\alpha = 7$ and $\beta = 2$ if $n = 16$, but $\alpha = 8$ and $\beta = 3$ otherwise.

$$L_{i+1} = (R^{-\alpha}[L_i] + R_i) \oplus K_i \tag{2.8}$$

$$R_{i+1} = R^{\beta}[R_i] \oplus (R^{-\alpha}[L_i] + R_i) \oplus K_i = R^{\beta}[R_i] \oplus L_{i+1} \tag{2.9}$$

As the aim of this project is to compare how an algorithm performs in hardware and not software SIMON was chosen even though SPECK shows almost as good performance in hardware and much better in software.

Chapter 3

Previous Work & Initial Design Approaches

Unfortunately, unlike some of the algorithms explored in chapter 2 there isn't a standard SIMON/SPECK software library available for use and benchmarking with my design. For that a reason a software version needed to be developed that offers similar functionality to the hardware version, so both software and hardware were developed in parallel. Also, by working on the software version, in C or C++, it allowed me to increase my familiarity with the algorithm and provided some insight in to how the hardware version, in System Verilog, would be designed, and vice versa.

For all versions of the SIMON algorithm developed for this project the most important factor is efficiency in terms of power consumption and resource use, with time efficiency not being an initial priority. All versions of this algorithm were developed not only with the description but also the pseudocode[insert reference]. In that document there is also a set of test vectors that define the ciphertext the algorithm should produce with given key and plaintext. Due to Feistel network structure of SIMON the decryption is just the same as encryption but with the key schedule reversed. This means that the encryption can be done in parallel to the key expansion but decryption requires the key to be pre-expanded. As some of the block cipher modes of operation[insert reference] only require encryption, two variants might be useful: one that can just encrypt data; and one with full functionality.

3.1 First Approach

The first approach for this project was to develop the software and hardware to be flexible to the various data block and key sizes defined for SIMON, shown in Table 2.1. As each system would only be computing with one mode at a time the decision of the variable sizes could be made at compile time and not during in runtime which be introduce some unnecessary inefficiency to the system. This could be achieved with C preprocessor macros for the software and, even though something similar is possible in System Verilog, parameters were passed between the modules. The macros and parameters were used to define the n and m values for each mode so

that the input and output variables of each function in software and each module in hardware would be correct. They were also used to define the number of rounds that the round function should iterated and the value of j used in the key expansion.

3.1.1 Hosted C

Initially the software was only developed to be used in a hosted environment but future approaches could made be for an embedded processor or something similar. With most languages used in software development the variable sizes and constrained to the data types available, with C offering only 8, 16, 32 and 64 bit variable types. This produced some difficulty with the 24 and 48 bit sizes required for this algorithm, but with the use of typedefs and bitfields in a structure it was solved. Using the values of N and M a *word* data type was defined based on unsigned integers, uint$N$_t, and the block, key and key schedule data types were setup using arrays of the base *word* type. All of the code used to setup the data sizes and type can be seen in Listing 3.1.

```
#if defined S32_64
    #define    N    16
    #define    M    4
    #define    T    32
    #define    j    0
#elif defined S48_72
    ...
#elif defined S48_96
...

typedef uint32_t  uint24_t;
typedef uint64_t  uint48_t;
#define TYPE_(x)  uint ## x ## _t
#define TYPE(x)   TYPE_(x)
#define UINT(x, n)  typedef struct n  { TYPE(x) v : x;  } n;
UINT(N, word);

typedef word    block[2];
typedef word    key[M];
typedef word    keys[T];

```

Listing 3.1: Macro definition of the word, block, key and key schedule types.

After the data types were defined the rotate functions, which are used frequently in algorithm, were made to rotate the bits in a *word* around a given number of bits. These rotate functions were used in the Feistel and round functions which is iterated for both encryption and decryption. The code used for these functions can be seen in Listing 3.2 but a full listing is available in the Appendix.

CHANGE/IMPROVE BELOW

```
TYPE(N) F    (word x          )
{
  return (ROTL(x,1) & ROTL(x,8)) ^ ROTL(x, 2);
}

void  ROUND (block b, word k     )
{
  word tmp = b[0];                      //  SAVED TO SWAP
  b[0].v = b[1].v ^ F(b[0]) ^ k.v;      //  GENERATE NEW WORD
  b[1] = tmp;                           //  SWAP FROM SAVED
}
```

Listing 3.2: Round and Feistel functions

```
void  KEXP_PRE  (keys ks, key k    )
{
  word tmp;
  int8_t i;
  //  LOAD KEY
  for(i=M; i>0; i--)  ks[i-1].v = k[M-i].v;

  //  GENERATE NEW KEYs
  for(i=M; i<T; i++)
  {
    tmp.v = ROTR(ks[i-1], 3);

#if (M == 4)  tmp.v ^= ks[i-3].v; #endif

    tmp.v ^= ROTR(tmp, 1);
    ks[i].v = ~ks[i-M].v ^ tmp.v ^ (z[j][(i-M) % 62]) ^ 3;
  }
}

void  KEXP_INL  (key k, TYPE(8) i    )
{
  //  GENERATE NEW KEY
  word tmp;
  tmp.v = ROTR(k[M-1], 3);

#if (M==4)  tmp.v ^= k[M-3].v;  #endif

  tmp.v ^= ROTR(tmp, 1);
  tmp.v ^= ~k[0].v ^ (z[j][i % 62]) ^ 3;

  //  SHIFT KEYS
  k[0] = k[1];

#if (M>2) k[1] = k[2];     #endif

#if (M>3) k[2] = k[3];     #endif

  k[M-1] = tmp;
}
```

Listing 3.3: Key Expansion Functions

The key expansion functions, Listing 3.3, also use the rotate functions but they are more complicated than the round function as they differ slightly depending the number of words in the key, M, but more compiler directives can handle that problem. There are two separate key expansion function: one that can expand the key at the same time as the encryption in the iterative loop (INL); and one that computes the schedule before encryption or decryption occurs in its own iterative loop (PRE).

For simplicity in the early stages the the software didn't have the capability of reading a file and encrypting or decrypting it so it could only use data stored in the code. This still allowed for the test vectors to be used and therefore correct functionality could still be confirmed. By simply changing the desired mode at compile time and setting up the test vectors for each mode they could be tested with the same code. The results of testing the code, in mode 0 (32/64), can be seen in [reference figure]. For the in loop key expansion the test plaintext and key was inputted and encrypted with the result compared with the expected ciphertext. With the pre loop key exapansion the encrypted ciphertext could also be decrypted so when that produced the original plaintext then the whole system was confirmed to function correctly.

INSERT TESTING RESULTS FIGURE

3.1.2 System Verilog

As mentioned in section 3.1 the parameter system was used to pass the values for word and key size and also the number of rounds to be iterated. Similar to the functions used in subsection 3.1.1 modules were used for the Feistel function, round function, key expansion function before being combined in the top level control module. Although, unlike the software version the rotate and logic operations used could be done with just combinational logic so the result is available almost instantaneously not after a few CPU instruction cycles.

The main functionality of the system was provided by the control module which provides the sequential operation with the clock (clk), and active low reset signal (nR). As with the software version two control modules were created: one that expands they key while encrypting (INL) and one that waits for the key to be fully expanded before the data processing begins (PRE). They both achieve this by operation as a state machine with with different states: waiting for data; loading data; processing data; and writing the output data. The state is controlled by the combinational block which is dependant on the current state; the inputs: $newData$, $readData$ and enc_dec*; and the internal $count$ variable and the output $doneKey$*.

INSERT STATE MACHINE FIGURE

ADD SYSTEM VERILOG LISTINGS BELOW [100]

To test the individual modules various input data was selected and the expected outputs were calculated either manually for the simple operations or using the software. As with the software when this functionality was correct they were combined into the top level control module to be further tested. Again, this was only for the most basic 32/64 mode of SIMON. To do this a testbench was created that with

a clock that had a $100ns$ period, or $10MHz$ [CHECK], and with the initial inputs setup during the reset of the module. After this the reset signal could be disabled and then the $newData$ signal could be set $HIGH$ to indicate to the state machine to move into the loading and processing states. When the module finished processing the data the ciphertext outputted could read by the testbench and then compared with the expected result before being decrypted in the module capable of that.

INSERT TESTING RESULTS FIGURE

ADD SYSTEM VERILOG SYNTHESIS BELOW [100]

3.2 Second Approach

With the basic functionality working in section 3.1 improvements could be made to the power consumption and resource use with methods like: code minimisation; logic minimisation; clock gating. Also more functionality could be added to increase the usability of the algorithm. As the aim of this project is to design an efficient version of this algorithm in hardware not software, most of the changes in this approach were made in the System Verilog code.

3.2.1 Hosted C

There were only some minor changes to the software version for this approach which were mainly what was learnt from developing the System Verilog version, subsection 3.1.2. One of these changes was ensuring that the key schedule was stored, before or during the main loop, and only expanded once. This was done with a $doneKey$ flag variable which is set to true at the end of the key expansion.

One other improvement was to add basic file reading capability to the software which allowed any '$.txt$' file to be read and stored in data blocks to be processed. Although, the same test data was used for the key and first data block to ensure that the cipher was still performing correctly.

ADD C LISTINGS BELOW [100]

ADD C TESTING BELOW [200]

3.2.2 System Verilog

To add the clock gating to the control module wasn't simple as the clock should not be halted while data is being processed or written to the output, so it could only be updated during the wait state. This was attempted with the clock gating signal only updating in this state on the same rising edge as the state machine. However, this caused glitches because the state machine moved onto the next state before the clock was stopped and was thus no longer in the correct state the clock gating when required. This was because the same edge that is needed to update the control signal moves the machine onto the next state. One option to solve this is to add more signals and a latch to have the gated clock to only stop when the clock is LOW. A

similar method to this was eventually used as the clock gating signal sensitivity was changed to the falling edge of the clock, hence updating half a clock cycle before the system changes state.

Gating the clock also enabled a few more enhancements to the code. One of these was because the halted clock provided waiting for new data functionality, the waiting state was no longer necessary. The clock gating was moved onto the loading state and the waiting state was just used for initialisation to improve the reliability of the system. Another change was the option to clock both the encryption/decryption as well as the key expansion and have them both in the same module. This also removed the need for two variants needed for in loop and pre key expansion as the clock gating could also be dependant on the *enc_dec* signal. It also helped with the logic minimisation as the key and the expanded key didn't need to be passed between multiple modules.

Another addition to the system was the ability to load a new key after the initial loading was done. This was done with another input signal $newKey$ which operated in the same way as the $newData$ signal did for data. Also a few more signals were outputted that indicate when the data and keys were loaded into the system and thus the inputs could change to be ready for the next set of data.

ADD SYSTEM VERILOG LISTINGS BELOW [100]

ADD SYSTEM VERILOG SIMULATIONS BELOW [200]

ADD SYSTEM VERILOG SYNTHESIS BELOW [100]

Chapter 4

Final Design Approach

For the final approach the idea was to ensure that the control module remains in the *execute* state as much as possible to ensure that the system is always processing data and thus increase the throughput. Having the hardware active all the time does increase the overall power consumption and might negate the effects of the clock gating added in section 3.2 but as the throughput is increased the power consumed per bit or byte of data processed is lower. This is done by sending packets of data instead of just individual blocks of data and having extra functions or modules to handle the input and output of these packets. This adds another system to be tested and compared in both hardware and software but even if it performs worst it does represent a more usable and complete system.

In a full system the packets would be sent using a serial protocol like I^2C but the they would also include information about what and how much data is being sent in a packet. As there are ten modes of the SIMON algorithm, labelled as $0-9$, four bits of the information are used to identify what mode of SIMON the data is intended for. This leaves another 4 bits to be used in a byte that were used individually for different parameters of the packet. One bit was used to indicate whether the data is to be processed or a new key to be loaded. Another tells the hardware to encrypt or decrypt the data, obviously irrelevant if a new key is being sent. As for most modes the number of words in the key are more than the two words in a block it was decided that a constant four words, or two blocks, would be sent in each packet but an option to just send one block would be controlled by one bit. The last bit in this byte tells the hardware that the packet is either meant to be inputted to it or outputted from it. To help the hardware keep track of the packets another byte was used to as a packet ID, which increments with each new packet.

ADD PACKET DIAGRAM FIGURE

Having this information attached to each block of data being processed could introduce a weakness to the cipher that might be exploited by an attacker. Although, ensuring and increasing the security of the algorithm is not the main purpose of this project.

4.1 Hosted C++

As the packets need to be setup by the a processor and includes a few data types and with some individual bits needing to be controlled it made sense to switch to an object orientated language instead of C so naturally C++ was chosen. While it would be possible to develop the functionality required in C it would take time and effort to get functionality already provided by C++. A Class was setup for each of the data types used in the previous approaches with appropriate constructor, mutator and accessor functions. Classes were also used for the packets, to store the data read from a file and for the cipher object.

The packet class stored the packet information, packet count and the four words of data. It also has an array of one byte unsigned integers that compacts the data into the smallest possible size for transmission. This is required because the information and count can always be stored in two bytes but depending on the mode the words vary in size and don't always fit exactly into the data types available. So before being sent a function is used to copy the data from the words byte by byte into the byte array which results in no redundant data being sent by accident. The setting up of these packets can be done separately or as part of the encryption, or decryption, process.

A class was used to reading files because the data were read into a temporary byte array before being stored in a the four words in a packet. From there with the packet setup it could be added to a vector of input vectors or sent straight to the cipher used, either software or hardware. If the vector method is used then the packets could be stored in another file to be sent to the cipher at a later point. The class is also used for reading the packets returned to it by the cipher and then written to a file with the same as the original name but with an encrypted or decrypted indicator. The class could also keep track of where in the file it has read to as it will not be read as a whole.

The reason for using a class for the cipher object was so that the key schedule and the state of the encrypted/decrypted block could be stored in one variable it similar to how it is stored in the hardware version. There is also a function added for handling the packets to ensure that data is used correctly.

ADD C++ LISTINGS BELOW [100]

ADD C++ TESTING BELOW [200]

4.2 System Verilog

With the packets being setup in software, as described in section 4.1, the System Verilog module only needed to be handle the information and hence route the packeted data to the correct part of the system at the right time. The whole system also needed to ability to pass the packet information and count through the modules as each one processed the data to ensure that the correct packet is outputted. To do this two modules were added: one to handle the input packet; and one to handle the output packets.

To handle the input packets the module takes an input of bytes equal to the size of the packets as well as an input control signal $newIN$. The module operates in a similar way to how the control module did in the first approach, subsection 3.1.2, with states for waiting for packets; loading packets; processing packets; and writing the relevant words to the relevant output, the key or the input block of the control module. After the packet is loaded the processing state checks the information byte is in the correct form: as an input, for the correct mode and the packet count; and then decodes what data is in the packet which is used by the writing state. When a packet contains two blocks of data, identified in the information, the first writes two of the words to the the input block and then waits for the control module to be ready for new data before writing the other two words.

The control module was also changed slightly to enable the packet information and count to propagate through the module with the connected data. This removed the need for the *enc_dec* signal as that is contained in the information. The input or output flag is also flipped by the control module for the output handling module.

The output handling module receives the packet information and data from the control module and then writes the data to the relevant part of the output packet. It then checks and decodes the packet information similar to the input dandling module but then instead of loading, processing and then writing it processes the packet information, reads the data and then writes the packet to the output. If the packet information suggests that there should be two blocks of data in the packet then the module waits for a second block of data to be processed by the control module before writing the output packet.

ADD SYSTEM VERILOG LISTINGS BELOW [100]

ADD SYSTEM VERILOG SIMULATIONS BELOW [200]

As the control module didn't change much from the previous approach its testing wasn't important but the other modules needed to be functionality correctly with absolutely no glitches for them to all interact as a whole system. To this the, as with all of the previous testbenches, the inputs were all setup with the reset signal active and then changed after the reset was deactivated.

For the input handling module the input packet had the basic packet information for a key, initial packet count of 0 and then the key from the test vectors.

ADD SYSTEM VERILOG SYNTHESIS BELOW [100]

Chapter 5

Experiments & Results

The experiments used in this project mainly focussed on the parameters that were outlined in chapter 1: throughput, power consumption, and resource use; with the accuracy confirmed throughout the development stage and the security level assumed to be good enough. Unfortunately, the hardware was never fully implemented on an FPGA as it was felt that the work required to setup the modules to work with a communication protocol was too much and would not yield any more information that couldn't be discovered with the various simulation tools. As the software and hardware versions do differ in how they compute the results, some of the desired comparisons couldn't be made directly as, for example, the power consumption is not available in software and the resources used are not the same. Even with the throughput the clock speed used in each version is vastly different, but even with that disadvantage the hardware should perform better.

5.1 Throughput

5.1.1 Method

To test the throughput of the algorithm is as simple as putting some data through it and timing how the implementation takes to process that data. From there, using the number of bytes and the time taken, the throughput can be calculated in bytes per second (Bs^{-1}) with [insert equation]. For the both software and hardware the data needed to be setup in software by reading an example file containing approximately $10kB$ of lorum ipsum[insert reference]. As the reading of the file takes some time, and is not part of the hardware testing, it is not included for the throughput testing in software by reading the file and setting up the packets and blocks before recording the start time. To input the data into the ModelSim testbenches it was simply written to a file in a SystemVerilog format and then copied and pasted into the relevant testbench.

5.1.2 Results

ADD THROUGHPUT TESTING RESULTS BELOW [300]

As each mode of SIMON processes a different number of bytes depending on the block the throughput of each block size (32, 48, 64, 96 and 128) is shown as it doesn't vary much key size.

ADD THROUGHPUT FIGURE

5.2 Power Comsumption

5.2.1 Method

ADD POWER TESTING METHOD BELOW [100]

5.2.2 Results

ADD POWER TESTING RESULTS BELOW [300]

5.3 Resource Use

5.3.1 Method

ADD RESOURCE TESTING METHOD BELOW [100]

5.3.2 Results

ADD RESOURCE TESTING RESULTS BELOW [300]

Chapter 6

Conclusion & Future Work

Bibliography

[1] Z. Hegde, "IoT now : how to run an IoT enabled business." [Online]. Available: https://www.iot-now.com/2017/03/09/59388-iot-applications-autonomous-vehicles-smarter-cars-cellular-iot-vehicle-telematics/

[2] D. Evans, "The Internet of Things How the Next Evolution of the Internet Is Changing Everything," 2011. [Online]. Available: https://www.cisco.com/c/dam/en{_}us/about/ac79/docs/innov/IoT{_}IBSG{_}0411FINAL.pdf