Exploring an AES Crypto-processor Design Using a Secure Asynchronous Toolset

Seyedeh Pardis Miri, Behnam Ghavami, Mehrdad Najibi, Hossein Pedram
Department of Computer Engineering, Amirkabir University of Technology, Tehran, Iran
E-mail: {miri, ghavami, najibi, pedram}@ce.aut.ac.ir

Abstract

Asynchronous circuit design techniques, especially balanced, finely-grained pipelines are able to play a key role, particularly in making hardware designs inherently resistant to side-channel attacks. In this paper, the asynchronous design of the AES crypto-processor is elaborated. Having the asynchronous processor being synthesized by an automatic tool for synthesizing asynchronous circuits, full functional test at various levels of design was performed. The results have shown that the asynchronous AES crypto-processor has an area overhead which was not unexpected for the asynchronous design, but the benefit of being resistant against side-channel attacks compensates this area overhead. Although, the area overhead, gained through this synthesis method was much higher as compared with those gained through other secure implementations methods, it provides security against the multiple side-channel attacks.

1. Introduction

Although good cryptography algorithms are designed to withstand against rigorous cryptanalysis attacks, they are vulnerable to which, the analysis of physical characteristics of hardware implementation is needed for. Many attacks have been developed exploiting physical properties of implementations and information leakage through side channel attacks [1], [2], [3].

Synchronous circuit design has been in use for a long time and various synthesis tools are available for such designs, but synchronous circuits have many problems, including power consumption, clock skew, and etc. One of the main problems in cryptography spot is vulnerability to side channel attacks [1]. The name stems from the fact that data dependent variations in timing, power consumption, electromagnetic emanations and behaviour in the presence of faults can be exploited by an adversary.

Several solutions have been proposed to countermeasure against side-channel attacks [5], [4]. Recently it was proved that, Quasi Delay Insensitive (QDI) asynchronous circuits, if implemented in a balanced way, have several features that are convenient for a secure hardware design [6], [7]. It demonstrates that dual-rail cryptography asynchronous circuits improve security by eliminating data depending on the amount of power consumption. To assure the independence of power consumption from input values in data- processing circuits and to exploit the propagating alarm signals defending against fault injections, design aspects are available to increase the resistance against side channel attacks. The asynchronous approach to increase the security is based on balancing the operation through special QDI logic cells. Properties of a dual-rail encoding and four-phase handshaking protocol significantly, improve the DPA resistance.

Despite of having so many advantages, there are some drawbacks regarding asynchronous circuits such as complex design procedure, and larger number of transistors. As a result, having an automatic asynchronous design tool is extremely helpful in popularizing asynchronous design methods. In this regard, Persia design toolset was developed and customized for secure implementations of crypto-systems which are resistant against side-channel attacks [7].

The Motivations for using asynchronous circuits in secure hardware design is explained in section 2 and in section 3, Persia synthesis tool is introduced. Section 4 describes the architecture of AES crypto-processor along with its asynchronous design. Section 5 and Section 6 present the synthesis results and the paper conclusion, respectively.
2. Motivations for using asynchronous circuits in secure hardware design

Asynchronous circuits could become a trustworthy platform for implementation of secure crypto-systems against side channel attacks. Some of the benefits include[8][6]:

- Absence of clock signal — having no clock means that clock glitch attacks are removed.
- Environment tolerance — Asynchronous circuits can adapt to their environment, which means, that they can tolerate many forms of fault injections such as power glitches, thermal gradients and etc. Having such a feature, fault sensing could be much easier since only major fault are need to be detected and reacted to. As minor fluctuations are inevitable, this feature is very desirable.
- Electromagnetic reduction — Electromagnetic signature is strongly reduced by replacing the synchronous processor with an asynchronous one (no clock harmonics). Removing clock, results in significantly flatter noise and electro magnetic interference (EMI) spectrum across the frequency domain.
- Balanced power consumption — Circuits comprising dual-rail (or multi-rail) codes can be balanced to reduce data-dependent emissions. No matter having a logical-0 or a logical-1, the encoding of the bit ensures that the data is transmitted and computations are performed with constant Hamming weight. This is important since side-channel analysis is based on the leakage of the Hamming weight of the sensitive data. Whilst dual-rail coding is used in a clocked environment, it had to be ensured that combinational circuits were balanced and glitch free. Return to-zero (RTZ) signaling is also required to ensure data independent power emissions.

QDI circuit implementations appear to be the most appropriate class of asynchronous circuits that can be synthesized automatically from large high-level behavior specifications. Many different QDI asynchronous circuit design methods are proposed [9]. The most efficient QDI implementations are based on per-charge logic. However, one of the most important problems in popularity of QDI asynchronous circuits is having a complex synthesis method and lack of automatic tools. Most importantly, few of QDI-asynchronous EDA tools address fin-grained asynchronous dynamic logic pipelining which is of major importance for security. In next section a synthesis tool for QDI asynchronous circuits is presented.

3. Persia Synthesis Toolset

Persia is an asynchronous synthesis toolset developed for automatic synthesis of QDI asynchronous circuits. The structure of Persia is based upon the design flow shown in figure 1 which can be considered as the following three individual portions: QDI synthesis, layout synthesis, and simulation at various levels. The simulation flow is intended to verify the correctness of the synthesized circuit in all levels of abstraction.

Persia uses Verilog-CSP[10]-an extension of the standard Verilog which supports asynchronous communication as the hardware description language for all levels of abstractions except the net list which uses standard Verilog. In this way, the Verilog is equipped by some READ and WRITE PLI* macros to emulate CSP language communication actions on the channels. The input of Persia is a Verilog description of a circuit that includes READ and WRITE macros for sending and receiving data via communication channels. This description will be converted to a netlist of standard-cell elements through several steps of QDI synthesis flow. To have simpler synthesis process, the arithmetic operations are extracted from the code and then, the major steps of synthesis process operate on the codes which contain no arithmetic operation. This is done by the AFE which also replaces the arithmetic functions by standard library modules. The two major steps in Persia synthesis process are Decomposition and Template synthesizer (SYN). For more information refer to [10].

![Persia design flow](image)

* Programming Language Interface
4. AES Asynchronous Crypto-processor Design Flow

As mentioned before, the asynchronous circuits are clock-less and the interaction of different modules is done by handshaking signals. The first step for designing an asynchronous processor is to design the modules which the circuit is consists of. The top module of the design is named as “Asynch_AES_module”. This module, as it shows in Figure 2, has two inputs which are the 128-bit “Key” and 128-bit “Input”, plaintext, and one 128-bit “Output” referred to as ciphertext.

![Asynch_AES_module](image)

Figure 2. AES Top Module

“Asynch_AES_module” is consists of three sub modules. These sub modules are as followed: “key_expansion_module”, “cipher_module” and “subbyte_module”. As it shows in Figure 3, key_expansion_module reads the 128-bit “Key” and performs a key expansion routine to generate the linear array of byte words, denoted as W[0] to W[43]. The cipher_module will read the “Input” via 1 input port and the W[0] to W[43] through 44 input ports and then perform the ten rounds of four stages of AES algorithm to generate the “Output”. “subbyte_module” is a module both of the “key_expansion_module” and “cipher_module” refer to. This module applies the Subbytes() function to any 32 bits block it reads. In section 4.1 and 4.2, the sub-module of each of these modules will be explained in detail.

![Diagram of sub-modules](image)

Figure 3. AES Sub-Modules

As we know, no matter what the Key is, the plaintext and the ciphertext can be converted to one another. In other words, the “Key” and the “Input” are totally independent of each other. Thus, we can process each separately and to implement more security it is always better to receive the “Key” and the “Input” in to two different modules. As it can be seen in the figure 3, first the Key is processed and the resulting key schedule, W[i], will be gained. Then, having W[0] to W[43] and the “Input” as the inputs of the cipher_module module, we can gain the output block. This way of processing is called sequential processing.

![Diagram of key expansion and cipher module](image)

As we discussed both parallel and sequential ways of design implementation, the parallel processing was not chosen for this design. The sequential processes may not gain the Output as fast as the previous way, but it doesn’t have that much overhead and the problem of accessing the shared module will never happen. Since, the module of key_expansion_module is done reading and writing in and from the “Subbyte_module” and producing key schedule, when the module of “cipher_module” would access the “subbyte_module” module for the first time.

In section 4.1 and 4.2, the sub modules of these three modules, mentioned above, and their functionality would be explained in details. In section 4.3 the top module with all its sub-modules would be explained.
4.1. “key_expansion_module” and “subbyte_module” sub-modules

The “key_expansion_module” is consists of two submodules: 1. AddRoundKey_module; 2. ShiftRow_module.

**Figure 4. key_expansion_module**

1. AddRoundKey_module: This module reads the 128-bit key as an input, splits it into four 32-bit blocks which are called W[0] to W[3]. After that, “AddRoundKey_module” module, in each round, writes W[i] to W[i+3] (i=0,4,8,12,...,39) and the appropriate 32-bit round constant word array value, “Rcon[i]” (i=0 to 9), containing the values given by \[x^{i-1}, 00, 00, 00\] with \(x\) being powers of \(x\) in the field GF(28), in to the input ports of the other module, and reads four values written by the “ShiftRow_module” module on its four 32-bit input ports referred as W[i+4] to W[i+7]. The values of W[0] to W[39] would be stored in this module, till the last four Words (W[40] to W[43]) are read from the other module. After that, “AddRoundKey_module” writes W[0] to W[43] in to its 44 output ports, simultaneously.

2. ShiftRow_module: This module first, reads W[i] to W[i+3] (i=0,4,8,12,...,40) and the Rcon[i] (i=0 to 9) from the output ports of “AddRoundKey_module” module and applies the ShiftRow() function to the W[i] (i=0,4,8,12,...,36). After that, it writes the W[i] as an input of the other module named “subbyte_module” which applies the Subbyte() operation. Having read the new value of W[i] (i=0,4,8,12,...,36), W[i], from the output ports of “subbyte_module” module, the W[i+4] to W[i+7] would be gained through the pseudo code shown in figure 5. It shows that, after ten rounds, all the Ws could be read by the “AddRoundKey_module” module.

```
// i=0,4,8,12,16,20,24,28,30,32,36;
W[i+5] = W[i+1] ^ W[i+4];
W[i+6] = W[i+2] ^ W[i+5];
W[i+7] = W[i+3] ^ W[i+6];
```

**Figure 5. Part of Verilog-CSP code of shiftRow_module**

Subbytes_module: This module is a module that both “cipher_module” and “key_expansion_module” modules access to it in a sequential way. This module consists of two sub modules. One of them is the module which reads a 32-bit block, W[i] (i=0,4,8,12,...,36), from “ShiftRow_module” module and splits it in to four bytes. Thus, each time it writes 8 bit in the output port A, and reads 8 bit from the input port B. This module is named “four_one_byte_module”. The other module is “lookup_table_module” which contains the look up table values needed for the function of Subbytes(). As we can see in figure 4, these sub-modules of “subbyte_module” module interact with each other through two 8-bit input and output ports. Since the look up table had been set to replace 1byte with its new value which is 1 byte too, four 8-bit reads and writes has to be occurred till the whole 32bit could be substituted by the new value in the S-Box(Lookup table).

4.2. “cipher_module” sub-modules

The “cipher_module” is consists of three submodules:
1. cipher_AddRoundKey_module; 2. cipher_shiftRow_mixcolum_module; 3. cipher_shiftRow_module;

**Figure 6. Cipher_module**
1. Cipher_AddRoundKey_module: This module reads the Input block and W[0] to W[43], and then combines W[i] to W[i+3](i=0,4,8,...,40) to build a 128bit array. This module produces array0 to array10 which each are 128bit and consists of four sequential blocks of 32bit W(s). Ten 128bit arrays would be gained through this way. After that, as it shows in the pseudo code in figure 6, this module applies the bitwise XOR operation to 128bit array0 and 128bit Input block. The result, “temp”, would be written in to the 128bit input ports of “cipher_shiftRow_mixcolum” module and the new value of the array0, “newtemp”, would be read from the 128bit output port of “cipher_shiftRow_mixcolum”.

After that, this module applies the bitwise XOR operation to newtemp and array1. This sequence of writing, reading and applying the XOR operation would be repeated until the final output is gained which is the output of the top module “Asynch_AES_module”.

```verilog
always Begin
  temp=input^array0;
  `WRITE(OUT,temp)
  `READ(IN,newtemp)
  temp=newtemp^array1;
  `WRITE(OUT,temp)
  `READ(IN,newtemp)
  temp=newtemp^array2;
  ...
  temp=newtemp^array10;
  `WRITE(FINALOUT,temp)
end
```

Figure 7. Part of Verilog-CSP Code of the Cipher_AddRoundKey_module

2. cipher_shiftRow_mixcolum_module: This module is just the same as cipher_shiftRow_mixcolum_module module except having the MixColun() function. This module is only referred to once, in the last iteration which the MixColun() function is not needed. Thus, no problem of using the “subbyte_module” by two modules of “cipher_shiftRow_mixcolum_module” and “cipher_shiftRow_module” may ever happen.

4.3. “Asynch_AES_module” sub-modules

The “Asynch_AES_module” is the top level module of the AES crypto-processor and is consisted of six sub-modules. The function of each module was explained in details in section 4.1 and 4.2. As we can see in figure 8, the “Key” and the “Input” are read as the inputs of this top module and the Output is the encrypted block of “Input”.

5. Results

Having the AES crypto-processor asynchronous design synthesized by the Persia asynchronous tool which used the security library cell which as introduced in [7]. The results from the synthesis operation over all the modules are shown in Table1.

<table>
<thead>
<tr>
<th>Module</th>
<th>Number of modules</th>
<th>Number of transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>shiftRow_module</td>
<td>1457</td>
<td>10618</td>
</tr>
<tr>
<td>AddRoundKey_module</td>
<td>8265</td>
<td>58833</td>
</tr>
<tr>
<td>key_expansion_module</td>
<td>9723</td>
<td>69451</td>
</tr>
<tr>
<td>cipher_module</td>
<td>28469</td>
<td>175199</td>
</tr>
<tr>
<td>cipher_AddRoundKey_module</td>
<td>6579</td>
<td>45327</td>
</tr>
<tr>
<td>cipher_shiftRow_mixcolum_module</td>
<td>15182</td>
<td>79984</td>
</tr>
<tr>
<td>subbyte_module</td>
<td>1173</td>
<td>7849</td>
</tr>
<tr>
<td>four_one_byte_module</td>
<td>514</td>
<td>3458</td>
</tr>
<tr>
<td>lookup_table_module</td>
<td>659</td>
<td>4391</td>
</tr>
<tr>
<td>asych_AES_module</td>
<td>48721</td>
<td>316780</td>
</tr>
</tbody>
</table>

Table 1 shows the overhead resulted form our design (secure-asynchronous). However, the overhead of our methodology is much better compared with synchronous one that was resist against Differential Power Attack.
We performed a full analysis of the side-channel information leakage from an asynchronous AES crypto-processor implementation. Differential Fault and Power Attacks were applied to the attack performed on our implementation and desirable simulations would be resulted. In other words, the occurrence of such attacks is not possible.

6. Conclusion

This paper presented an asynchronous approach to the implementation of the AES Crypto-processor with the key of 128-bit. Two different ways of AES processing was presented (sequential and parallel processing). Although a better performance could be gain through parallel processing but the problem of overlapping of using a shared module and area overhead was irresistible. Thus, the sequential way had been chosen for the design implementation.

Having our asynchronous design being synthesized by the secure library cell in Persia asynchronous tool, the results show that our implementation was secure against multiple side channel attacks.

7. References