2013 III Brazilian Symposium on Computing Systems Engineering

An Implementation of the AES cipher using HLS

Rodrigo Schmitt Meurer Tiago Rogério Mück Antônio Augusto Fröhlich
Software/Hardware Integration Lab
Federal University of Santa Catarina
Florianópolis, Brazil
Email: {rmeurer,tiago,guto}@lisha.ufsc.br

***Abstract*—The Advanced Encryption Standard (AES) is the main algorithm used to ensure security and privacy in several different applications ranging from massive data servers to small low-power embedded systems. Such embedded systems often rely on dedicated hardware implementations of AES in order to meet tight power budgets. In this scenario, C/C++ High-Level Synthesis (HLS) solutions are gaining acceptance as traditional hardware design methodologies can no longer match the strict time-to-market requirements of current applications. In this paper, we describe a C++ implementation of the AES algorithms and explore different hardware micro-architectures by using HLS solutions. We focus on describing the process of obtaining an efficient synthesizable C++ description form plain software code.**

I. Introduction

Advances in technology are increasingly changing our day-to-day. In the upcoming world of *Internet of Things*, technologies such as wireless sensor nodes, contactless smart cards, and *radio frequency identification* (RFID) are of uttermost importance and need strong cryptographic protocols to ensure privacy[1]. In this scenario, the *Advanced Encryption Standard* (AES) is a standardized algorithm approved by the *National Institute of Standards and Technology* (NIST)[2]. It has become the default choice in numerous applications, from high-end computers to low power portable devices, including standard wireless technologies such as IEEE 802.11i [3] and IEEE 802.15.4[4]. Its widespread adoption on small embedded devices has brought new design challenges, however. The AES algorithm is computationally intensive and a software implementation on general purpose processors may not meet the requirements of such small and low power devices. Current embedded systems may depend on dedicated hardware accelerators to perform encryption and decryption of information.

Approaches for deploying AES hardware accelerators are often based on *register transfer level* (RTL) implementations synthesized to *application-specific integrated circuits* (ASICS) or reconfigurable hardware devices such as *field programmable gate arrays*(FPGA). However, the strict time-to-market requirements of most applications demand a better productivity than what is possible with current RTL design methodologies, thus leading to a growing demand for *high-level synthesis* (HLS) solutions. HLS is a (semi-)automatic process that creates cycle-accurate RTL specifications from untimed or partially timed behavioural specifications. The main advantage of HLS is the possibility of automatically generating different hardware micro-architectures from a single high-level implementations, thus enabling quick design space explorations and dramatically reducing time-to-market.

Current HLS tools support hardware synthesis from high-level C/C++ specifications. This motivates the reuse of efficient, well validated and time-tested software algorithms through automatic hardware synthesis. However, plain C/C++ software code usually cannot be used "as it is". Some constraints must be met to achieve a good hardware implementation using HLS. These constraints include identifying concurrency opportunities, creating an architecture top level describing the system interface, and choosing the mapping between data arrays and hardware resources, such as registers and RAM blocks.

In this paper we describe the design and implementation of a AES block cipher for both hardware and software. During the design process, we identify which constraints must be met in order to achieve a C++ code that can be efficiently used for architectural exploration using hardware HLS tools. We first provide an efficient baseline software implementation and then explore different hardware implementations considering two main scenarios: one that relies on registers for storing data, thus allowing a high throughput implementation; and one that relies on RAM/ROM blocks, thus reducing the overall hardware by sacrificing performance. For each scenario, we explain which changes made to the original C++ code to allow the desired exploration.

The remaining of this paper is organized as follows: Section II provides a quick background on HLS and the AES algorithm; Section III reviews other works that describes AES implementations; Sections IV and V describe our design, hardware microarchitecture exploration, and our results; Section VI closes the paper with our conclusions.

II. Background

This section provides a quick overview on the AES standard and HLS concepts.

A. Advanced Encryption Standard

AES is a round-based symmetric block cipher which uses the same key for both encryption and decryption. Figure 1 shows the operation flow for encryption and decryption. AES is defined for 128-bit data blocks size and uses key lengths of 128, 196, and 256 bits. According to the key length, AES variants are called AES-128, AES-196, and AES-256.

The cipher keeps the to-be-en/decrypted data in an internal 4x4 matrix of bytes, called *state*, in which the operations are performed. Similar to other symmetric ciphers, AES operations are organized into *rounds* that are applied iteratively according to the key length (10, 12, and 14 rounds for AES-128, AES-196, and AES-256, respectively). Each round consists of four

2324-7894/14 $31.00 © 2014 IEEE
DOI 10.1109/SBESC.2013.36

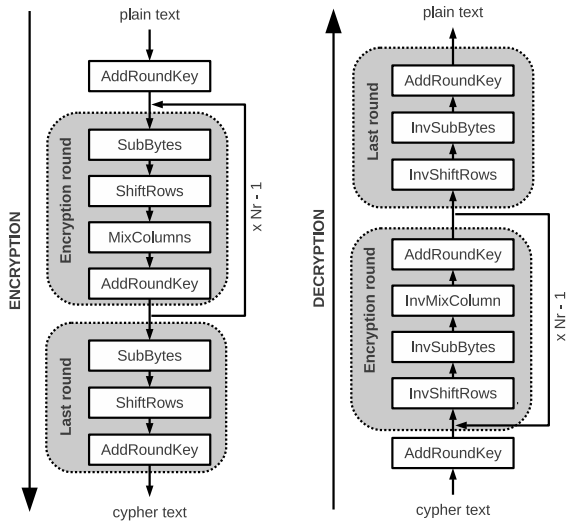

Figure 1. AES Encryption/Decryption Flow

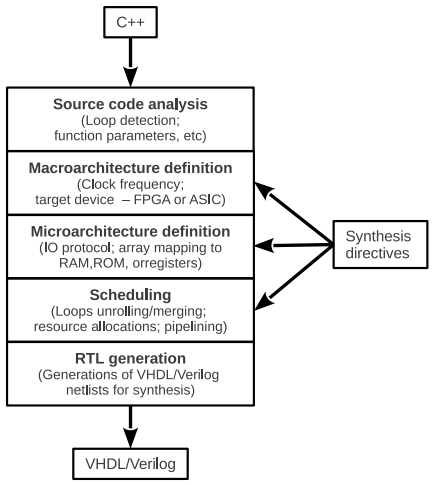

Figure 2. High-level synthesis steps

transformations:
addRoundKey consists in mixing the state array with a round key derived from the cipher key by XORing the bytes in respective positions of the array.
subBytes is an invertible non-linear transformation. It uses 16 identical 256-byte substitution tables (*S-box*) for mapping each byte of state into another byte. The *S-box* values are generated by computing the multiplicative inverses over in *Galois-Field* GF(2^8) and applying affine transformations.
shiftRows operation consists basically of a left shift of the second, third and fourth rows of the *state* matrix by one, two and three bytes respectively
mixCollumns performs a polynomial multiplication in GF(2^8) on each column. More information can be obtained from [2].

B. High-level Synthesis

High-level synthesis — also known as *electronic system-level* (ESL) synthesis, is an semi-automatic process that takes a untimed algorithmic description and generates RTL hardware implementations. Figure 2 gives an overview of the HLS flow.

The state-of-art of HLS tools (e.g. CatapultC [5], Vivado [6], and others[7], [8]) already supports hardware synthesis from algorithms coded in languages such as C and C++. Due to some characteristics of high-level synthesis, extra care must be taken in order to produce C++ code that can be synthesized to hardware and run efficiently in software. HLS tools limit the use of C++ features that rely on dynamic structures in software (e.g. heaps, stacks, virtual method tables), such as recursion and dynamic polymorphism. Furthermore, the same high-level algorithm can span several different hardware implementations. For instance, C++ loops can have each iteration executed in a clock cycle, or can be fully unrolled in order to increase throughput at the cost of additional silicon area. This kind of synthesis decision is usually taken based on user defined *synthesis directives* which are provided separately from the algorithm descriptions.

Based on these inputs, algorithm operations are scheduled in different clock cycles and functional units. The output of the tool is usually a RTL descriptions in hardware description languages such as VHDL or Verilog, which is used as input for the lower levels of the implementation flow.

III. Related Works

Since the standardization of AES, a lot of people developed hardware architectures to implement the algorithm. However, there are few works that uses HLS to implement the cipher.

In [9], the authors focus on developing a highly regular and scalable hardware architecture. They achieve their objective by using similarities of encryption and decryption. In their work they also achieve a very good performance and relatively small area by balancing the combinational paths of the design.

The AES encryption core hardware presented in [10] is suited for small embedded applications or devices that require low power consumption. The authors propose a core constituted of a novel 8-bit architecture that supports AES-128. They achieve a throughput of 121 Mbps at a maximum frequency of 153 MHz. The authors achieved a significantly higher throughput(comparing to previous 8-bit implementations) with corresponding area and lower energy consumption per processed block.

The authors of [11], provide a comprehensive hardware architecture comparison between the AES cipher and new lightweight cryptographic algorithms standardized by the *International Organization for Standardization* (ISO), such as *Clefia* [12] and *Present* [13]. The comparison is performed on 128-bit version of all the ciphers, with the same design goals, and targeting the same FPGA platform and synthesis tools. Their area/speed results for the serial versions of AES and CLEFIA were similar, however, the extra storage register required by CLEFIA for the mixed key means that lower area AES designs with a similar throughput are possible. They also note that the PRESENT serial and iterative architectures differ little in area while increasing a lot in throughput.

In [14], the authors expose how they effectively leveraged on a C-based AES implementation to generate a hardware

core using HLS. They provide an explanation of the C2R [15] methodology and compiler for co-processor synthesis. Exploration of different architectures to meet area and speed constraints are also shown in their work. Their baseline implementation using a 128-bit datapath resulted in an encryption core that required 12 clock cycles to encrypt one block.

IV. AES BASELINE SOFTWARE IMPLEMENTATION

Our baseline software implementation is directly based on the operations flow described in Section II and supports the 128-bit version of the AES algorithm. It is implemented using the C++ languages using *object-oriented programming* (OOP) techniques. Figure 3 shows shows an overview of our implementation.

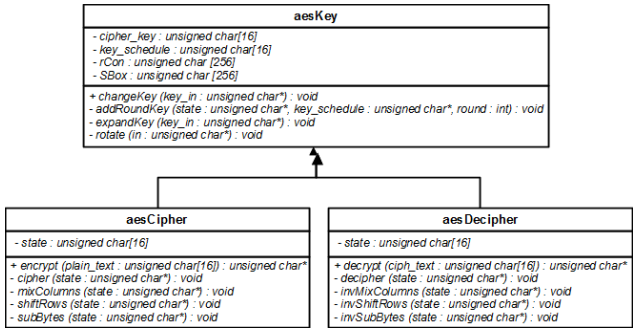

Figure 3. UML class diagram showing the structure of our software implementation

The *aesKey* class encapsulates all key operations that are shared by both encryption and decryption flows. This class consists of the key expansion operations the *addRoundKey* method. The *aesKey* class also contains the *rCon* and *sBox* tables. The *rCon* table is used to generate all the round keys, and the *sBox* is used both to generate the round keys and to encrypt. The *aesKey* class is extended by the *aesCipher* and *aesDecipher* classes using OOP inheritance. The classes implements the encryption and decryption flow, respectively. This is necessary since their internal implementation differs dramatically between encryption and decryption.

A. Encryption flow

In order to show how our baseline software works, we describe next how we have implemented the sequence of operation in the encryption flow. The first method to be called is the *changeKey* method (defined in the *aesKey* class). It receives the new key to use in encryption and stores it's value in the *cipher_key* array. After the value is stored, the *expandKey* method is called. The key expansion operates with words of 4 bytes and starts by copying the key into the 16 first positions of the *key_schedule* array, then expands the key into another 10 round keys as shown in Figure 4.

With the key expanded and stored, the encrypt method can be called passing the to-be-encrypted text as argument. The round operations are shown in Figure 5. Usually, the round operations are performed in the *state* matrix. In our case, we implemented the state matrix in form of an array, as shows Figure 6. The main reason to implement the matrix as an array

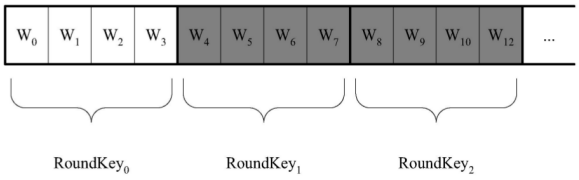

Figure 4. Mapping of the key words to round keys

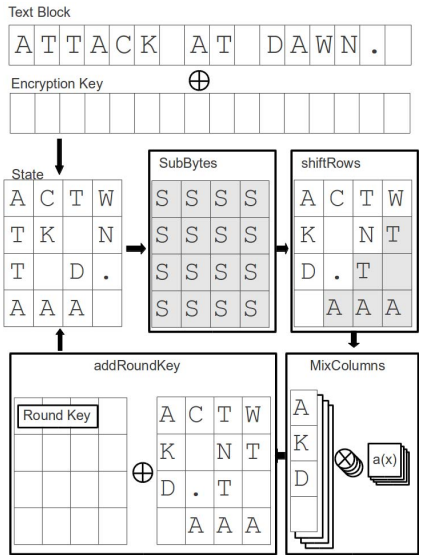

Figure 5. AES round operations.

is avoid nested loops, which facilitates the profiling of loop iterations. The method *encrypt* triggers the encryption flow and starts by copying the plain_text to the state array and then calling the *cipher* method. The *cipher* method executes the methods in the same sequence shown in Figure 1:
addRoundKey performs a 16-iteration loop which is executes a byte-by-byte XOR with the state array and the respective round key.
subBytes, like *addRoundKey*, performs a 16-iteration loop in which each iteration replaces a byte by it's correspondent value from the *sBox* table.
shiftRows cyclically shifts the bytes in each row by a certain offset. The first row is left unchanged. Each byte of the second row is shifted one to the left. Similarly, the third and fourth rows are shifted by offsets of two and three respectively.
Finally, the *mixColumns* method does operations on the GF for each column of the state matrix. In our case the state columns is each sequence of 4 bytes of the state array.

B. Results

We compared our baseline implementation of the AES with a widely used C library available from Texas Instruments [16]. We started by validating the implementation against a series of test-vectors and comparing the results with Texas'. The results were obtained by running both implementations in a Intel Core 2 Quad CPU Q9550 @ 2.83GHz x 4 using 32-bit Ubuntu 12.04 LTS. Texas Implementation encrypted one block in $8\mu s$

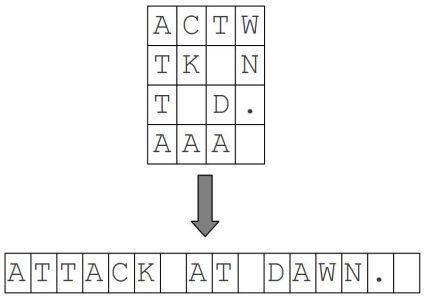

Figure 6. Example of the state matrix implemented as array

Table I. TABLE OF MAIN LOOPS, UNROLLING CAPABILITY AND ITERATION NUMBER

Loop:	Unroll	Iterations
addRoundKey	yes	16
subBytes	yes	16
mixColumns	yes	4
round	no	9
expandKey	no	40

while our encrypts one block in $9\mu s$. This shows that our C++ OOP-based implementation yields performance comparable to a industry-optimized C-based implementation.

V. MODIFICATIONS FOR HARDWARE SYNTHESIS

In this section we describe the process of obtaining an efficient synthesizable C++ description from plain software code. The first step is identifying target loops for parallelization. Table I shows the main loops of the cipher, if they can be unrolled and how many iterations. As presented in Section IV the *addRoundKey, subBytes* and *mixCollumns* are implemented in loops. Since the next iteration of these loops does not depend on the previous, we can unroll the loop and perform all the operations in parallel. The *round* loop and the *expandKey* can not be unrolled due the fact that each new iteration depends on the values obtained from the previous one.

Figure 7 summarizes all performed modifications and generated hardware architectures. Each step is explained in more details below.

A. Baseline implementation synthesis

We made some changes to adapt the software C++ code for a HLS tools[1]. This changes mostly consisted in creating an architecture top level. For designs that have multiple functions within the C++ input file, it is necessary to specify one function to be the top-level function. The architecture top-level is necessary to describe the system interface and behaviour. We defined that the input is a sequence of 17 bytes of which the first bytes defines the operation(*encrypt*, *decrypt* or *changeKey*) and the other 16 bytes are either the to-be-en/decrypted text or the new key. The output is a sequence of 16 bytes with the encrypted/decrypted text. Before synthesizing the software implementation obtained from Section IV, we defined some constraints aiming at obtaining results considering a real scenario. We synthesized only the encryption and key expansion cores. The design target frequency was established as 13.56 MHz (ISO standard for contact-less smart cards) on a Xilinx Virtex-6 6VCX130TFF484 FPGA platform.

Considering an application that encrypts more than one block with the same key, we defined that the round keys were pre-calculated and stored, this way this approach can be more energy-efficient. The *sBox* and *rCon* are also pre-calculated values. With these constraints defined above, we synthesized the code from Section IV with just the addition of the top-level function. The synthesis tool mapped the constant arrays *rCon* and *sBox* to ROM and the key schedule array to RAM. Without changing any of the tool constraints, we synthesized the hardware to verify what would be generated. The result of the synthesis was not satisfactory. It required an enormous amount of cycles to encrypt one block. We then considered two optimization scenarios: one for throughput and another one for area. Results are shown in Table II. This table shows the number of cycles required for each method and the total number of cycles need for encryption of a whole block.

B. High-throughput implementation

In our first scenario, we considered an application in which we don't need to worry about the area occupied but high throughput is required. In this case would be very interesting to have all the arrays mapped into registers because it would allow all indexes to be accessed within a single clock cycle, therefore increasing parallelism opportunities. Since the target frequency of the design is also low (13.56 MHz), the a full encryption round operations can also fit in just one cycle. Using this configurations, we were allowed to fully unroll the loops (loops that could be unrolled shown in Table I) and parallelize most of the operations. This resulted in an encryption core where each 16 byte block takes 12 cycles to encrypt.

The single modification we made to the software implementation at this point was to rewrite the *shiftRows* method. The method was re-written to do the respective changes of byte's order in the array without mapping it to a matrix. This way the operation used less resources than it would if mapped to the matrix. By just changing the HLS tool constraints we were able to generate this. The tool was set to store the constant tables *rCon* and *sBox* to registers. The *key_schedule* was also set to be stored in registers. These changes made possible the reduction of the number of cycles, because *addRoundKey*, *subBytes*, *ShiftRows* and *mixColumns* operations could be entirely unrolled.

C. Low area implementation

Since not all applications have area to spare, we also considered an application that is limited in area, so we would need a low area implementation. In this scenario would be very interesting to have the constant arrays in ROM and the *key_schedule* array in RAM. But this is what we had in the first time we tried to synthesize. To optimize the hardware synthesis we would need to make changes to the software in a way that the tool would generate a different implementation. Using one RAM to store the array we can only access one address per cycle making it impossible to obtain any improvement. But we

[1]due to legal constraints regarding licensing, we cannot mention the HLS used in our work

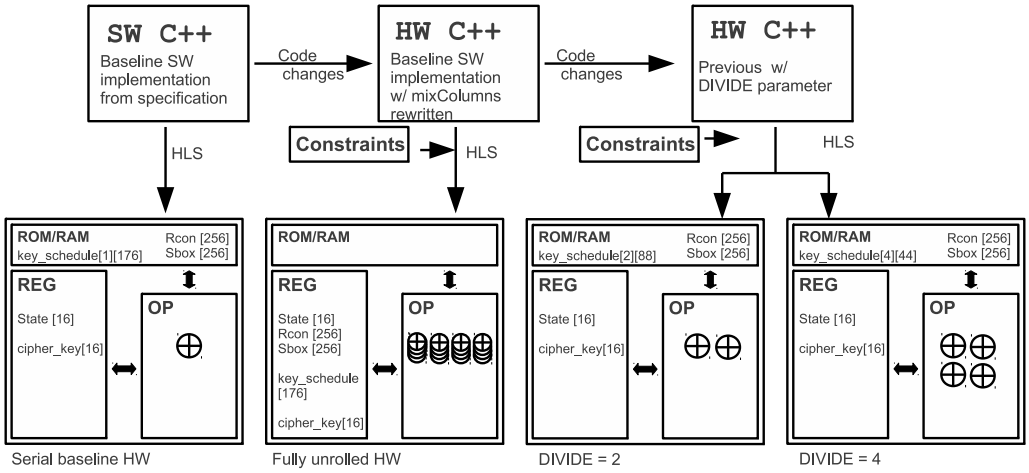

Figure 7. Overview of the performed modifications and generated hardware micro-architectures

Table II. RESULTS OF HARDWARE SYNTHESIS. THE NUMBER OF CYCLES FOR THE ENCRYPTION OF ONE BLOCK INCLUDES THE INTERFACE DELAY

. The *Area Rating* is a generic are metric reported by the synthesis tool.

Implementation		Number of Cycles	Area Rating	LUTs	Flip-Flops	36 Kbits BRAM
Baseline Implementation:	*mixColumns*:	4	4447.78	2674	1723	1
	addRoundKey:	32				
	subBytes:	32				
	shiftRows:	4				
	Cycles per Round :	**70**				
	Encryption of one block:	**748**				
Tables and key_schedule in registers:	*mixColumns*:	1	16936.01	15464	4810	0
	addRoundKey:	1				
	subBytes:	1				
	shiftRows:	1				
	Cycles per Round :	**1**				
	Encryption of one block:	**12**				
key_schedule w/ DIVIDE = 2	*mixColumns*:	4	4528.42	2740	1718	1
	addRoundKey:	16				
	subBytes:	32				
	shiftRows:	1				
	Cycles per Round :	**51**				
	Encryption of one block:	**542**				
key_schedule w/ DIVIDE = 4	*mixColumns*:	4	4596.57	2634	1804	1
	addRoundKey:	8				
	subBytes:	32				
	shiftRows:	1				
	Cycles per Round :	**43**				
	Encryption of one block:	**454**				
[14]	**Encryption of one block:**	**12**	-	64282	4890	0

can divide the array into different RAMs, thus we can access a different number of addresses per cycle, that number being the number of RAMs we divide the array on.

We then needed a new software implementation where the tool would map the *key_schedule* array into a different number of RAMs. To do this we re-implemented the *keyExpansion* and the *addRoundKey* methods using a parameter called *DIVIDE*. *DIVIDE* can assume three different values:

- if *DIVIDE = 1* we have the original code where the entire array is mapped into the same RAM block, with dimensions [1][176], meaning we can only access one word per cycle.
- if *DIVIDE = 2* we store the array into 2 different RAMs, resulting in 2 arrays of length [88]. In this case, each array is read eight times, but we can read 2 arrays per cycle.
- if *DIVIDE = 4* we store the array into 4 different RAMs, resulting on 4 arrays of length [44]. In this case, it reads four arrays per cycle four times to obtain a round key.

By using the *DIVIDE* parameter we were able to optimize the number of cycles used in most of the operations in which the *key_schedule* is used. Using RAM, 32 cycles were required when we use *DIVIDE = 1* to perform the *addRoundKey* operation, where one cycle was used to read the memory and another to perform the XOR for each of the 16 bytes. When we use *DIVIDE = 2* the number of cycles required to perform an entire *addRoundKey* decreases to 16. It also takes one cycle to read the memory and another to perform the XOR, however, it performs two operations per cycle. In one cycle it reads 2 addresses of memory and in the other it performs two XOR

operations.

Using *DIVIDE = 4* we were able to reduce the number of cycles required to perform an entire *addRoundKey* to 8 cycles. In this case four memory addresses are read in one cycle and the four XORs occur in the second cycle. This way we were able to reduce a considerable amount of clock cycles to perform the entire encryption operation with just a little gain in the area rating. We could also improve the *subBytes* method, but instead of dividing the *subBytes* table we would need to store the same table in different memory addresses so we could read more than one per cycle.

Table II also compares our result with a previous work [14]. Our Baseline implementation using 128-bit data path (fully unroled with key_schedule and constant tables stored in registers) matched the number of clock cycles required to encrypt a block obtained in [14][2], but with a significantly low area. This shows that we were able to achieve a very efficient implementation.

VI. Conclusions and Future Work

In this paper we have explored different hardware implementations of the AES using HLS. We have developed a high performance software implementation that required only small changes to be efficiently synthesized to hardware. By exploring different HLS directives and memory partitioning optimizations, we were able to reduce the area occupied by the design while also decreasing the number of cycles necessary to encrypt a block.

Despite enabling the fast design space exploration of hardware micro-architectures, the HLS process required modifications on the source code to generate efficient hardware. This yields at least two different base AES implementations: one aiming at software, and one aiming at hardware. Considering a scenario in which the AES could be implemented as both hardware or software, maintaining two different implementations for the same component can be error prone and lead to functional mismatches. Therefore, as future work, our goal is to solve this issue by redesigning the AES cipher using *aspect-oriented programming* techniques and the *unified design* approach [17].Our focus in this sense is on isolating aspects that are specific to hardware and software into *aspect programs* that can be automatically applied to the base C++ AES code. This would enable the generations of hardware- and software-specific C++ without the need of error-prone manual modifications.

References

[1] A. A. Frohlich, A. M. Okazaki, R. V. Steiner, P. Oliveira, and J. E. Martina, "A Cross-layer Approach to Trustfulness in the Internet of Things," in *9th Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS)*, Paderborn, Germany, 2013, pp. 884–889. [Online]. Available: http://www.lisha.ufsc.br/pub/Frohlich_SEUS_2013.pdf

[2] National Institute of Standards and Technology, "Advanced encryption standard (aes)," FIPS-197, 2001.

[3] "Ieee standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications," *IEEE Std 802.11-2012 (Revision of IEEE Std 802.11-2007)*, pp. 1–2793, 2012.

[4] IEEE Computer Society, "IEEE Standard 802 Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)," The Institue of Electrical and Electronics Engineers, Technical report, 2006.

[5] Calypto Design Systems, "CatapultC Synthesis," 2011, http://www.calypto.com/.

[6] Xilinx, "Vivado Design Suite," 2013, http://www.xilinx.com/products/design-tools/.

[7] Synopsys, "Synphony C Compiler," 2011, http://www.synopsys.com.

[8] Cadence Design Systems, "C-to-Silicon Compiler," 2011, http://www.cadence.com.

[9] S. Mangard, M. Aigner, and S. Dominikus, "A highly regular and scalable aes hardware architecture," *Computers, IEEE Transactions on*, vol. 52, no. 4, pp. 483–491, April.

[10] P. Hamalainen, T. Alho, M. Hannikainen, and T. Hamalainen, "Design and implementation of low-area and low-power aes encryption hardware core," in *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, 2006, pp. 577–583.

[11] N. Hanley and M. O'Neill, "Hardware comparison of the iso/iec 29192-2 block ciphers," in *VLSI (ISVLSI), 2012 IEEE Computer Society Annual Symposium on*, 2012, pp. 57–62.

[12] Sony Corporation, "Clefia algorithm specification," http://www.sony.net/Products/cryptography/clefia/download/index.html, 2007.

[13] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "Present: An ultra-lightweight block cipher," in *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4727. Springer, 2007, pp. 450–466.

[14] S. Ahuja, S. Gurumani, C. Spackman, and S. Shukla, "Hardware coprocessor synthesis from an ansi c specification," *Design Test of Computers, IEEE*, vol. 26, no. 4, pp. 58–67, 2009.

[15] CebaTech, "C2r compiler," "http://www.cebatech.com/c2r_compiler".

[16] Texas Instrments, "Aes-128," http://www.ti.com/tool/aes-128.

[17] T. R. Mück and A. A. Fröhlich, "Towards Unified Design of Hardware and Software Components Using C++," *IEEE Transactions on Computers*, 2013, to appear. [Online]. Available: http://www.lisha.ufsc.br/pub/Muck_TC_2013.pdf

[2] the number of LUTs required by [14] was adjusted in Table II due to differences in the devices. We have deployed our AES implementation on an Xilinx Virtex6 FPGA with 6-input LUTs, while [14] uses a Virtex5 with 4-input LUTs.