Formal Verification of Hardware Support For Advanced Encryption Standard

Anna Slobodová
Intel Corporation

***Abstract*—The Advanced Encryption Standard (AES), approved by National Institute of Standards and Technology, specifies a cryptographic algorithm that can be used to protect electronic data. The next generation of Intel micro-processor introduces a set of instructions known as AES-NI, that promises multi-folded acceleration of the AES encryption and decryption process. In this paper, we report about the formal verification of hardware support for these new instructions. The verification is based on use of Symbolic Trajectory Evaluation that lies at the base of formal verification methodology used by Intel Corporation. To our knowledge, this is the first formal verification of AES hardware support.**

I. Introduction

In the competitive world of computer business, each generation of micro-processors is expected to come up with something new – more performance, lower power, higher degrees of parallelism and, last but not least, higher degrees of security. The new features of the next generation of Intel's product include hardware support for encryption and decryption algorithms. The Federal Information Processing Standards Publication 197 [7] specifies the Advanced Encryption Standard (AES) that is based on the Rijndale algorithm [5], [6] – a block cipher encryption and decryption algorithm that uses Galois Field arithmetic. The latest Intel architecture introduces six Intel$^{(R)}$ SSE instructions known as AES-NI that promise threefold-ed speed-up of the AES algorithms. Their description can be found on Intel's official web site [2], [3]. Four of the instructions – AESENC, AESENCLAST, AESDEC, and AESDECLAST correspond to one round of encryption/decryption algorithm. The other two facilitate the key expansion procedure: AESKEYGENASSIST is used for generating the round keys for encryption; AESIMC is used for converting the encryption round keys to a form usable for decryption.

Each of these instructions is implemented using AES micro-operations – primitives understandable by hardware. Our focus is on the verification of these micro-operations whose complexity matches the complexity of the AES-NI instructions.

For the verification, we use an internal formal verification system Forte [1]. Our specification of micro-operations is written in Reflect [8] – a ML-like functional language that is a successor of FL. Reflect includes Binary Decision Diagrams as first-class objects and Symbolic Trajectory Evaluation (STE) [16] as built-in functions. For details on Intel's verification methodology using Forte, we refer the reader to the published papers [10], [11], [12], [17].

For each micro-operation, a proof constitutes a run of Symbolic Trajectory Evaluation on a formal model derived from the Register Transfer Level model of Execution Cluster (EXEC) – a part of design responsible for the execution of micro-operations. All external assumptions – the assumptions made about hardware outside EXEC – are checked by traditional simulation on the full-chip model.

To our knowledge, this is the first verification of HW support for AES. There is a previous work describing verification of Galois Field Algorithms. Morioka *et.al.* proposed a logic system for the verification of algorithms over Galois fields $GF(2^m)$ and carried out a proof for One Shot Reed-Solomon Decoding Algorithm [14]. Mukhopadhyay *et. al.* suggested [15] an improvement to the theorem-based approach in [14] by using an observation that the verification of a property in $GF(2^m)$, where $m = np$, can be reduced to the verification of the property in a composite field $GF(2^n)^p$. This observation allowed the authors to take advantage of a hierarchical approach to the verification problem. For the transformation of elements between the fields, they used Reduced Ordered Functional Decision Diagrams (ROFDDs) [9]. They demonstrated their approach on the same Reed-Solomon decryption algorithm. Recently, a group in Galois Connections developed a domain-specific language Cryptol and a tool set for formally specifying, implementing and verifying cryptographic algorithms. It has been applied to the verification of some aspects of the AES algorithm. Their verification methodology uses And-inverter graphs and combinational equivalence checking and requires both specification and implementation be written in Cryptol.

Our work differs from the papers mentioned above in several aspects. Our focus is not on the AES algorithms but on the verification of hardware that implements a basic step of the algorithm, which is equivalent to one round of the AES encryption/decryption. An important aspect of our work is that the hardware has been designed with high performance rather than verification in mind. The main message of this paper is that the verification of the AES implementation can be performed using general purpose BDD-based symbolic simulation engine instead of a special-purpose diagrams ROFDDs tailored for XOR-AND-based functions.

In the next section, we explain all terms needed to understand the specification of AES instructions. Section III describes verification methods.

[1] A publicly available version of the tool that can be used for non-commercial purposes can be downloaded from *http://www.intel.com/software/products/opensource/*

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

II. AES And Its HW Support in Intel(R) New Generation Microprocessors

Our specifications of micro-operations were written based on the Micro-Architecture Specification for the Intel(R) new generation microprocessor – Westmere; while the specification of each particular transformation is defined as in FIPS 197 document [7] to avoid replication of possible errors made by designers. It is comprised of about five hundred lines of Reflect code. Although the MAS is confidential, the description of AES instructions can be found on Intel public web-pages [2], [3].

The FIPS 197 document [7] specifies a symmetric block cipher that can process data blocks of 128 bits, using cipher keys with lengths of 128, 192, and 256 bits. The AES instructions operate on one or on two 128 bits sources: State and Round Key. From the architectural point of view, the state is input in an xmm register (xmm1) and the Round key is input either in an xmm register (xmm2) or in memory (m128).

Because of lack of space we will describe just one representative AES instruction – AESENC that performs one round of AES encryption. This will give the reader a flavor of the complexity of the design and the specification. We will start with the definition of the instruction, followed by basic terms and operations that the instruction is composed of. Note, that because of the symmetric character of the AES encryption and decryption algorithm, for each transformation, there is an inverse transformation that is used for decryption. Here we introduce only transformations that are needed for the definition of the chosen instruction.

Instruction AESENC xmm1, xmm2/m128, performs one round of AES encryption applied to the content of a xmm register xmm1 using a 128-bit key from the register xmm2 or memory m128 (we will show the version for xmm2).

```
AESENC xmm1, xmm2
Tmp:= xmm1;
RoundKey:= xmm2;
Tmp:= ShiftRows (Tmp);
Tmp:= SubState (Tmp);
Tmp:= MixColumns (Tmp);
xmm1:= Tmp xor_bitwise RoundKey
```

Any sequence (of bits, bytes or words) is represented as a list. **Byte** is a sequence of 8 bits with lsb indexed by 0. **Word** is a sequence of 4 bytes $[w_0, w_1, w_2, w_3]$. Note that this differs from the usual notion of a computer word as a sequence of two bytes. **State** is a sequence of words in Big Endian notation. State might be viewed as a matrix, so it makes sense to talk about **columns** (that match words) and **rows**. If not stated otherwise, in $s = [s_0, s_1, s_2, s_3]$, si denote columns.

Example: A state in hexadecimal notation and viewed as a matrix:
$[[h00, h01, h02, h03], [h04, h05, h06, h07],$
$[h08, h09, h0a, h0b], [h0c, h0d, h0e, h0f]]$

word3	word2	word1	word0
h00	h04	h08	h0c
h01	h05	h09	h0d
h02	h06	h0a	h0e
h03	h07	h0b	h0f

Elements in $GF(2^8)$ can be represented as 7-degree polynomials with binary coefficients. **Addition** and **subtraction** over $GF(2^8)$ are equivalent operations and are defined as bitwise XOR, denoted by $\oplus$.

Multiplication in $GF(2^8)$ is multiplication of polynomials modulo an irreducible polynomial $m(x) = x^8+x^4+x^3+x+1$ and it results in an element in $GF(2^8)$. $m(x)$ is represented by a binary vector $\text{AES}_M = [1,0,0,0,1,1,0,1,1]$. In our specification, multiplication over $GF(2^8)$ is implemented as suggested in the FIPS 197 document using an efficient iterative algorithm, where each iteration step consists of a multiplication by polynomial x. **Inverse element** of Galois field is computed using **Extended Euclidean Algorithm** with irreducible polynomial $m(x)$.

Another structure we use is a four-term **polynomial with coefficients from** $GF(2^8)$. $a(x) = a_3x^3 + a_2x^2 + a_1x + a_0$ can be represented by 4 bytes: $[a_0, a_1, a_2, a_3]$ i.e., one word. These polynomials behave differently from the polynomials we mentioned above:

Addition of such polynomials is defined as bit-wise XOR applied to the corresponding bytes:

$$(a_3x^3 + a_2x^2 + a_1x + a_0) + (b_3x^3 + b_2x^2 + b_1x + b_0) = (a_3 \oplus b_3)x^3 + (a_2 \oplus b_2)x^2 + (a_1 \oplus b_1)x + (a_0 \oplus b_0)$$

Or

$$[a_0, a_1, a_2, a_3]+[b_0, b_1, b_2, b_3] = [a_0 \oplus b_0, a_1 \oplus b_1, a_2 \oplus b_2, a_3 \oplus b_3]$$

Multiplication of two polynomials (of degree less than 4) is computed modulo $x^4 + 1$. To distinguish this multiplication from the $*$-multiplication defined above, we denote it by $\otimes$ and call a **modular multiplication**.
$a \otimes b$ can be expressed as a left multiplication of b by matrix

$$M(a) = \begin{pmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix}$$

$a \otimes b = M(a) * b^t$, where * is usual matrix multiplication with * applied to the elements of the field, and b^t is a vector transposition of b.

ShiftRows is a transformation that rotates i-th row by i bytes to the left (rows are labeled in top-down manner 0 to 3).

$$\begin{pmatrix} S_0 & S_4 & S_8 & S_c \\ S_1 & S_5 & S_9 & S_d \\ S_2 & S_6 & S_a & S_e \\ S_3 & S_7 & S_b & S_f \end{pmatrix} \longmapsto \begin{pmatrix} S_0 & S_4 & S_8 & S_c \\ S_5 & S_9 & S_d & S_1 \\ S_a & S_e & S_2 & S_6 \\ S_f & S_3 & S_7 & S_b \end{pmatrix}$$

SubByte transformation applied to a byte b consists of finding its multiplicative inverse b^{-1} and applying affine transformation defined by matrix S and vector c. S is generated by left-rotation of vector $[1,1,1,1,0,0,0,1]$:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

and $c = [0,1,1,0,0,0,1,1]$ is a binary representation of h63. SubByte $b = S * b^{-1} + c$, where b^{-1} is inverse element of b. Note that the bits in the rows of the S matrix published in the FIPS 197 document are in reversed order. This is because our bytes have bits ordered $[b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0]$.
SubWord is a byte-wise application of SubByte to the word.
SubState is a word-wise application of SubWord to the state.
MixColumn_word $w = q \otimes w = M(q) * w^t$,
where $q = [h02, h01, h01, h03]$.
MixColumns is defined as a word-wise application of MixColumn_word to a state.

Another polynomial used in the algorithm is x^3 represented as $x3 = [h01, h00, h00, h00]$. The corresponding matrix $M(x3)$ is used to implement the cyclic shift of the bytes of the word to the left:
RotWord $w = x3 \otimes w = M(x3) * w^t$
Example:
RotWord $[h03, h02, h01, h00] = [h02, h01, h00, h03]$

III. Formal Verification of AES Support in Intel(R) New Generation Processor Design

Verification of AES was a part of an extensive verification effort undertaken by our team on Execution Cluster of the next generation micro-processor. After several years of experience in verification of arithmetic units described in numerous publications, Intel has in place a solid methodology that is based on Forte system [10], [12], [11], [17]. Even though the methodology advanced substantially in the past couple of years, its base remains the same. Our proofs consist of data-path proofs (one for each particular micro-operation) and control proofs that assure that there is no write-back conflicts between the unit that implements AES and the other units. All of them were completed using Symbolic Trajectory Evaluation [16].

Since we haven't had any previous experience with arithmetic over Galois fields, we were not sure about the complexity of the proof. Because the representation of elements in Galois Field use module-2 sum-of-products, it seems to be natural to use ROFDDs for their representation. However, our STE engine is based on Reduced Ordered Binary Decision Diagrams (BDDs) [4]. The involvement of multiplication in some operations made us uncertain whether the capacity of the tool will be sufficient to carry out the verification. As it turned out, with the right variable ordering, we had no BDD-size blow-up issues and no decomposition of the proof/design was needed, which eliminated the need for using theorem proving.

The data-path proof for each micro-operation was performed in one STE run that starts with sources (after bypass) and control signals, and ends at the write-back signals at the boundary of EXEC. Since all AES instructions have fixed latency, STE is a suitable verification engine. We assumed that sources carry legal values, and control signals have values that correspond to a valid AES micro-operation in the pipe, i.e. micro-operation code and additional control signals (power-up, reset, valid signal, etc.) have specific values. Assumptions that concern the design outside execution cluster are called *external*. Examples of external assumptions are:
– Reset signal is not asserted.
– When a micro-operation is executed, the resepective unit is powered up.
– Scheduling of Micro-operations are non-conflicting.
External assumptions used in our proofs were identical to assumptions used in the verification of other micro-operations inthe EXEC. They were checked dynamically by means of simulation based full-chip validation. The bypass logic has been formally verified, but the verification is out of the scope of this paper.

There were some of the usual challenges in writing the specification from incomplete and imprecise documents. Some confusion also came from the use of different byte and bit ordering formats in the design and in the specification. Since we did not want to duplicate potential errors made by designers, we wrote our definition of GF operations based on the FIPS 197 document [7]. However, the FIPS 197 document is Big Endian oriented, while Intel hardware is Little Endian oriented. That means that the behavior of the hardware requires inputs and outputs to be byte-reflected, in order to match the FIPS specification.
Therefore, in the specification, we translate source values from Little Endian to Big Endian, transformations are applied to these translated values; then results are translated back to Little Endian to be compared to values taken from write-back signals.

Variable Ordering

To nobody's surprise, part of the success of any BDD-based verification lies on a right choice of variable ordering. Let us look closer at the transformation included in our instruction:

An important observation is that all of the transformations just shuffle bits and bytes around without "re-using" them in multiple places. The intutition behind this vague statement is that we will never come into a conflict that one variable should be in two differnt positions in the order. The other observation is that the most operations are defined on bytes or words.

Let us consider the ShiftRows transformation first. It is important to note that it does not change relative position of bytes within words. The first byte of each new word was the first byte of some other word; the second byte of each new word was the second byte of some other word; and so on. If we

consider a variable ordering where words are interleaved, the variable order of bits in original and transformed work remains consistent with this order. When we look at our example, word [S0, S1, S2, S3] is transformed to [S0, S5, Sa, Sf]. Bits of S0 are followed by bits of S1 and S5 interleaved, those are followed by bits of S2 and Sa interleaved, and those are followed by S3 and Sf interleaved. Obviously, such order is a good variable ordering for checking the ShiftRows transformation.

Let us consider SubState transformation. This state transformation consists of byte-wise substitution. The FIPS 197 document published a pre-computed SubByte transformation in a form of a table labeled by possible values of half-bytes. In our specification, we actually choose to generate the table using the operations over GF as described in the specification, which has been compared to the published table. That provided an additional assurance that our specification of the operations is correct. For any variable order with variables bound to inputs on the top, a BDD for S-table consists of a complete tree of depth 8, with 8-node BDDs at each tip. A BDD for one bit of the result is even smaller.

MixColumns is defined as a word-wise application of Word-Column_word. The words neither overlap, nor are combined. What we deal here with are four 8x8 bit multiplications (mod $m(x)$), followed by bit-wise XOR. Since for each multiplication, multiplicand is a constant, there is no BDD growth to expect. The XOR-combination of bytes would call for interleaving bytes in the variable ordering, if we would want to write our check as AND of XOR-ed bits, but we don't need to do it. Each bit of the result can be checked separately. Because of that, XOR of bytes works fine even if the variables are in order of sequential concatonation of the bytes. As a result, the variable order where words are interleaved is a good variable ordering for this transformation as well.

To summarize, we choose variable ordering with variables for control signals (such as reset signal, power-up signal, micro-operation code, etc.) on the top, followed by 128-bit data (xmm1) chopped by words and interleaved, and 128-bit key (xmm2) closes the list. In fact, the order of variables bound to the key does not matter. However, for the instructions that involve transformations applied to the key, we use the order where the key is chopped into words and interleaved.

IV. Conclusion and final remarks

Our formal proofs cover correct execution (as defined in [7] and Westmere Micro-Architecture Specification) of all valid micro-operations issued to the AES unit for all legal inputs. Following assume/guarantee principle, we also proved conjecture of other EXEC units which assume that AES does not interfere with their write-backs. Our conjecture that other units do not interfere with AES write-back has been proven within the respective units as well. With our choice of variable ordering we did not encounter BDD blow-ups in the course of the verification. Time and memory consumption of the proofs were very moderate. The data-path and control proofs for all six micro-operations took about 12 minutes and the peak memory usage was less than 2GB on a 64-bit Intel(R) Xeon (TM) machine). As a collateral of doing FV, the Micro-Architecture Specification has been improved and extended with examples. The proof scripts were added to our rich library of proofs that we routinely apply to IA32/64 micro-processor designs and are expected to be reused on the successor projects. To our knowledge, there have been no published reports on the verification of AES hardware support.

Acknowledgment

I would like to thank Naren Narasimhan for challenging me with the problem and Brent Boswell, designer of the AES unit, for answering all of my annoying questions about the design and its intended functionality.

References

[1] AES page available via http://www.nist.gov/CryptoToolkit
[2] Intel(R) Advanced Vector Extensions Programming Reference. March 2008. 319433-002. http://softwarecommunity.intel.com/isn/downloads/intelavx/Intel-AVX-Programming-Reference-31943302.pdf
[3] S. Gueron: Advanced Encryption Standard (AES) Instruction Set, http://softwarecommunity.intel.com/articles/eng/3788.htm
[4] Bryant, R.E.: Graph-based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Comps.*, C-35, pp. 677-691, 1986.
[5] J. Daemen and V. Rijmen, AES Proposal: Rijndael, AES Algorithm Submission, September 3, 1999, available at [1].
[6] J. Daemen and V. Rijmen, The block cipher Rijndael, Smart Card research and Applications, LNCS 1820 (Springer) pp. 288-296.
[7] Federal Information Processing Standards Publication 197, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
[8] Jim Grundy, Thomas F. Melham, John W. O'Leary: A reflective functional language for hardware design and theorem proving. J. Funct. Program. 16(2): 157-196 (2006)
[9] T. Kropf: Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems, Springer, 1999.
[10] R. Kaivola, M, Aagard: Divider Circuit Verification with Model Checking and Theorem Proving. *TPHOL 2000, (Springer) LNCS 1869, pp.338-355.*
[11] R. Kaivola, K. Kohatsu: Proof Engineering in the Large: Formal Verification of Pentium(R) 4 Floating-Point Divider. *Software Tools for Technology Transfer, Springer, 2003, Vol.4, Issue 3, pp 323-335.*
[12] R. Kaivola, N. Narasimhan: Formal Verification of the Pentium(R) 4 Floating-Point Multiplier *Proc. of the conference on Design, automation and test in Europe (DATE), 2002.*
[13] J. Lewis: Cryptol: specification, implementation and verification of high-grade cryptographic applications. In *FMSE'07: Proceedings of the 2007 ACM workshop on Formal methods in security engineering, pp. 41.*
[14] S. Morioka, Y. Katayama, T. Yamane: Towards Efficient Verification of Arithmetic Algorithms over Galois Fields $GF(2^m)$, in Proc. CAV 2001, LNCS 2102, pp. 465-477.
[15] D. Mukhopadhyay, G. Sengar, D.r. Chowdhury: Hierarchical Verification of Galois Field Circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26, No. 10, October 2007.
[16] C.-J.H. Seger, R.E. Bryant: Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Formal Methods in System Design, 6(2):147-189,1995.*
[17] A. Slobodová: Challenges for Formal Verification in Industrial Setting. In Proc. of FMICS/PDMC 2006, pp.1-22.
or *Formal Methods: Applications and Technology, 11th International Workshop*, FMICS 2006 and 5th International Workshop PDMC 2006, Bonn, Germany, August 26-27, and August 31, 2006, Revised Selected Papers. Springer (2007), LNCS 4346.