Formal design and verification of cryptographic circuits: Application to symmetric block ciphers

Abir. BITAT[1], Salah MERNIZ[1]

[1]MISC Laboratory, IFA departement, NTIC Faculty, Abdelhamid MEHRI-Constantine2 University

Abstract

Cryptographic circuits have become indispensable in most systems where security is the main criteria. That is why it is important to verify the correctness of their design. In this paper we propose a formal methodology for design and verification of the cryptographic circuits; it combines two techniques, the *SAT Solver* technique, which is automatic and has been used to verify the correctness of combinational logic parts; and, the *induction* technique, which has been used for sequential logic parts.

The proposed approach consists of using the functional Hardware Description Language (HDL) Lava to describe both behavioural and structural aspects of a circuit; using Finite State Machines (FSMs). And then to verify the specification and check the equivalence of the implementation against it, in order to prove the circuit's correctness.

To show the features of the proposed approach, it was applied to verify implementations of both sequential and pipelined architectures of the symmetric block cipher AES (Advanced Encryption Standard).

Keywords

Formal design, Formal verification, Cryptographic circuits, Functional approach, FSMD, Finite State Machine with Data-path, Symmetric cryptography, AES, Functional language, Functional HDL

1. Introduction

Nowadays networks are trusted with extremely sensitive information; hence the importance of cryptography which makes its security possible. As a consequence, cryptographic co-processors have become indispensable in many modern applications. A co-processor is a hardware module that adds functions to a standard CPU; a cryptographic one adds cryptographic IP cores specialized for encryption and decryption operations. Due to the sensitivity of the security matter; it is important to design and implement these circuits correctly.

Circuits design process goes through several steps; the first one consists in converting the informal description of the design to an algorithmic specification. From this latter, a micro-architectural implementation is derived through refinement; followed by a series of design steps reducing the abstraction levels until a realizable description is obtained. After each of these design steps; a verification process is performed.

Design verification of hardware can either concern non functional aspects like time, power and layout, or functional verification; this latter, consists in checking that the structural implementation provides the same behavior mentioned in the behavioral specification; which can

RIF 2022: The 11th Seminary of Computer Science research at Feminin, March 10, 2022, Constantine, Algeria

abir.bitat@univ-constantine2.dz (Abir. BITAT); salah.merniz@univ-constantine2.dz (S. MERNIZ)

0000-0003-1748-4286 (Abir. BITAT); 0000-0001-6544-7063 (S. MERNIZ)

be done in quite a few ways, such as simulation, formal proof, and semi-formal verification in which the former techniques are combined.

In this work, we use formal methods in order to design and verify the cryptographic circuits through functional verification. In section 2 we discuss related literature. In section 3, we explain our design and verification methodology. In section 4, we demonstrate how we applied our approach in order to verify sequential and pipelined architectures of the symmetric cryptographic circuit AES. And finally conclusions are drawn in section 5.

2. Related works

Imperative HDLs like VHDL and verilog, are still the most used for describing digital circuits. Therefore, the majority of the existing similar works are based on using them [1], [2], [3], [4], [5]. These languages are intended for description, simulation and synthesis of hardware; but not for formal verification due to their lack of formal semantics; in addition, they do not allow descriptions of the highest levels of design. As a consequence, most of the mentioned works use the simulation technique [2], [3], [4], which is not sufficient for critical systems such as the cryptographic circuits because it does not guarantee the absence of errors. It is possible to use formal verification on circuits described with imperatives HDLs, but with a great deal of difficulties; the behavioral specification that hides all implementation details therefore needs more powerful abstraction and structuring mechanisms [5]. For the verification of such complex circuits, deductive methods were used [1]; which is quite difficult, because it requires user guidance through all the verification process. Symbolic computation was used for the verification of cryptographic circuits in some algebraic approaches [6], [7]; similarly to the work presented herein, they use the hierarchy technique in order to reduce the complexity of design and therefore simplify the verification task. Equivalence checking and completion functions were used in a formal approach [8] for the verification of RTL descriptions in imperative HDL VHDL. A language and tool that support automated formal verification of cryptographic assembly code was proposed in [9]. As for the pipelined implementations of cryptographic circuits; the imperative HDL VHDL was used for the description and simulation of some crypto-processors in [10]; and formal verification was applied, but no details about the method that was used were mentioned.

Even though there exists several functional approaches that were applied for the verification of digital circuits; to the best of our knowledge; beside the work presented herein, there is only one approach for the implementation and verification of cryptographic circuits [11]. However; this approach uses the functional language only for the behavioral specification; but it uses an imperative HDL for the structural implementation; which brings back some translation difficulties between the two descriptions.

3. The proposed approach

The design process of a circuit is divided into several steps, where the implementation resulting on a certain abstraction level is used as the specification on the next lower one. A verification phase is mandatory after each step, to avoid the propagation of errors between the abstraction levels. Therefore, in order to verify the correctness of a certain implementation, it is sufficient to compare it to its already verified specification. The most challenging verification step is that of the highest levels of abstraction; this difficulty is caused by the gap between descriptions as the majority of HDLs don't allow ones at those levels.

In this work, we propose a methodology of formal design and verification for the cryptographic circuits, we target the higher levels of abstraction, and we verify that the micro-architectural description of a cryptographic circuit is equivalent to the algorithmic one.

The other attribute of our proposed approach is that we use the functional language Haskell and its embedded HDL Lava to describe the circuits. This choice is motivated by their interesting characteristics such as: having formal descriptions that can be reasoned about; the composition of functions in the same way that complex circuits are composed; having much more concise descriptions, which makes finding errors easier; and last but not least, Lava has some incorporated tools for formal verification of circuits; which in our knowledge have never been used for the verification of the cryptographic circuits.

A FSMD (Finite State Machine with Datapath) model provides a systematic approach for designing sequential circuits; it combines a controller, modeled as a FSM, and a dapapath. We model both descriptions of the circuit (behavioral and structural) as FSMs.

A FSM in Haskell is a recursive function ***fsm***, which represents the *output function*; it takes as input a s_{i-1} with the *next state function* called ***step***, like shown in the following form:

$$\begin{aligned} &fsm\ n\ si = fsm\ (n-1)\ sj \\ &\quad where \\ &\qquad\qquad sj = step\ si \end{aligned}$$

Since the two descriptions are at different levels of abstraction, they deal with different types of data; therefore, a mapping function *abs* is needed between the two them. Thus, in order to verify the implementation; the following theorem must be proved:

$$\forall\ s_i \in states,\ fsmS\ (abs\ s_i)\ =\ abs\ (fsmI\ s_i) \tag{1}$$

The proof of this theorem must be decomposed. Since both descriptions are hierarchical, we need to start by verifying the simplest components at the lowest level of hierarchy. Once they are verified, the verification of the upper components that they compose becomes possible.

The combinational parts of the implementation can be verified automatically using the SAT solver tool of Lava. As for the sequential parts; we use induction and equational reasoning.

When performing equivalence checking using SAT solvers, if the two descriptions are equivalents, the output of the Xor gate should be always *False*. If the output becomes *True* for any input sequence, it means that the SAT solver found a satisfiable assignment, which implies that the descriptions are producing different outputs for the same input.

In order to verify that two descriptions are equivalent, we have to define a safety property that expresses their equivalence. A safety property states that some condition is always true. This property is also defined as a function in the following form:

$$\begin{aligned} &propertyEquiv\ \ in = ok \\ &\quad where \end{aligned}$$

$$
\begin{aligned}
&out1 = description1\ \ in\\
&out2 = description2\ \ in\\
&ok = out1 <==> out2
\end{aligned}
$$

To verify this property we use the Lava function *satzoo*, which is a call to the satisfiability solver: $satzoo\ \ propertyEquiv$.

Since we want to verify that a component implements correctly its specification (which works with a different data type); the equivalence property should be defined in the following way instead:

$$
\begin{aligned}
&propertyEquiv\ \ in = ok\\
&\quad where\\
&\qquad out1 = specification\ \ (abs\ \ in)\\
&\qquad out2 = abs\ \ (implementation\ \ in)\\
&\qquad ok = out1 <==> out2
\end{aligned}
$$

Now, for the verification of the sequential part, which is the FSM it self, meaning the implementation of the whole circuit; we use induction like we mentioned previously. So, we start by verifying the base case (a); then we prove (b) that if the property holds for a step *n*; then it should hold for the following step *n-1*.

$$
\begin{aligned}
&(a)\ \ fsmS\ 0\ (abs\ s_i)\ \ =\ \ abs\ (fsmI\ 0\ s_i)\\
&(b)\ \ fsmS\ n\ (abs\ s_i)\ \ =\ \ abs\ (fsmI\ n\ s_i) \Rightarrow\\
&\qquad\qquad fsmS\ (n-1)\ (abs\ s_i)\ \ =\ \ abs\ (fsmI\ (n-1)\ s_i)
\end{aligned}
$$

The proposed approach for verifying both sequential and pipelined architectures is summarized in Fig.1. We start by checking if the behavioural description of the sequential architecture holds the verification property (1) ; then, we verify the sequential implementation against its specification (2). As for the pipelined architecture, we verify the equivalence of its specification to the sequential one (3); and finally we verify its implementation against the verified specification (4).

We prove the verification property (1) and equivalence property (3) using Lava's SAT solver. We also use this latter to verify the inner sub-components of the circuits, like shown in Fig.2. Then we will be able to prove the equivalence properties (i.e. (2), and (4)) using induction.

4. Application and results

Advanced Encryption Standard (AES) [14] is a symmetric block cipher, constructed based on the Rijandael system; it encrypts blocks of 128 bits using a key of 128 bits size, 129, or 256 bits depending on the system's version. We demonstrate our proposed approach on AES128.

The encryption consists of ten identical rounds of processing; each of which includes four steps: sub bytes, shift rows, mix columns and add key; except for the last one, where the mix

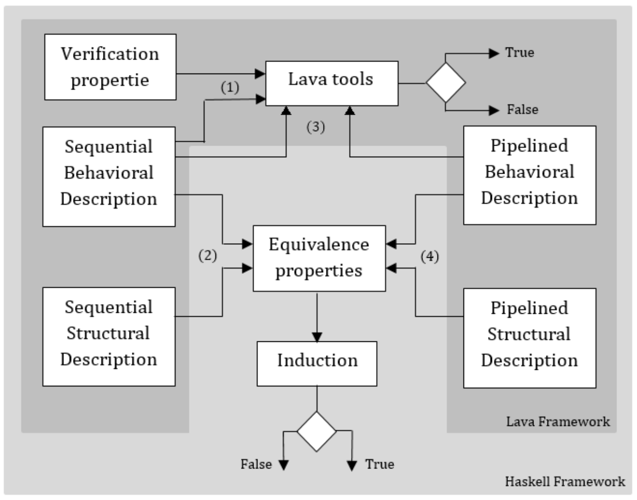
Verification propertie
Lava tools
True
False
(1)
(3)
Sequential Behavioral Description
Pipelined Behavioral Description
Equivalence properties
(2)
(4)
Sequential Structural Description
Pipelined Structural Description
Induction
Lava Framework
False
True
Haskell Framework

Figure 1: Global view of the proposed approach.

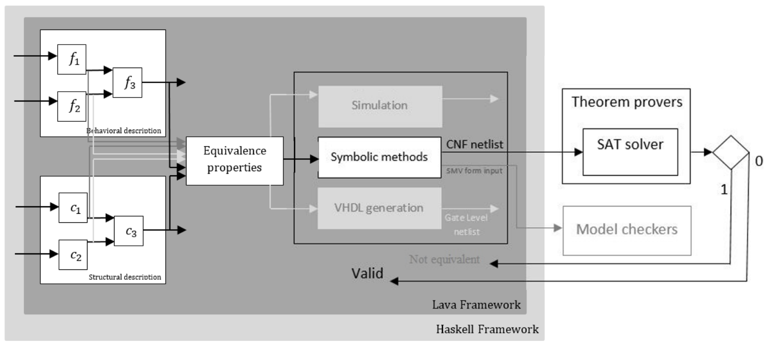

Figure 2: Automatic verification of the combinational parts.

columns function is not performed. The decryption on the other hand, uses the inverse of these functions in a different order.

The key expansion function uses the cipher key to produce sub-keys in the same number of rounds. In this work, we pre-calculate the sub-keys in advance.

4.1. Formal design of the Sequential architecture

The basic hardware architecture used to implement an encryption or decryption unit of the symmetric AES cipher is shown in Fig.3. This architecture characteristics is that only one block of data is encrypted at a time.

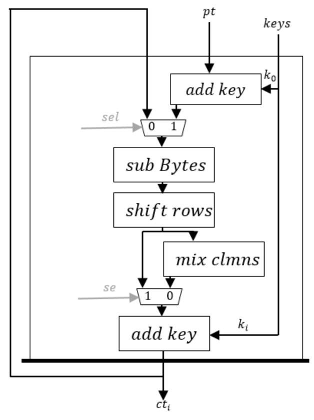

Figure 3: Basic iterative architecture.

4.1.1. Behavioural description

The AES core is composed of three units: encryption unit, decryption unit, and key expansion unit in order to pre-calculate the sub-keys. It is defined by the function *seqSpecAES*, which takes as inputs a text *text*, a key *key* and the operation type *op*.

$$
\begin{aligned}
&seqSpecAES(op, text, key) = textNew\\
&\quad where\\
&\qquad keys = keyExpansionS\ \ 10\ \ key\\
&\qquad textNew = if(op == 1)\ \ then\ (seqSpecEnc(text, keys))\\
&\qquad else\ (seqSpecDec(text, keys))
\end{aligned}
$$

The function *seqSpecEnc* converts the text from *ascii* to *hexadecimal*, and decomposes it into blocks of 128bits, and send them sequentially to the function that encrypts one block of data. The behavioral description of this encryption unit is expressed as a FSM in Lava, represented by the recursive function *seqSpecEncFSM*, which takes as input the *stat* s_i which is composed of: the round number *n*, one block of the plain text *pt*, the current cipher text *ct*, and the pre-calculated sets of sub-keys *keys*; and it calculates the round cipher text using the next state function *stepSeqSpecEnc*.

$$
\begin{aligned}
&seqSpecEncFSM\ (n, pt, ct, keys) = seqSpecEncFSM\ ((n-1), pt, ctNew, keys)\\
&\quad where\\
&\qquad sel = (n == 10)\\
&\qquad se = (n == 1)\\
&\qquad it = muxS\ (sel,\ addKeyS\ (\ (keys!!0)\ , pt),\ ct)\\
&\qquad ctNew = stepSeqSpecEnc(se, it, keys!!(10-(n-1)))
\end{aligned}
$$

$$\begin{aligned}
&stepSeqSpecEnc(se, it, ik) = (ctNew)\\
&\quad where\\
&\qquad ctNew = addKeyS(ik, muxS(se, shiftRowsS(subBytesS\ \ it),\\
&\qquad\qquad mixClmsS(shiftRowsS(subBytesS\ it))))
\end{aligned}$$

The selector *sel* controls whether to take the initial plain text *pt* or the current round cipher text *ct*; and the selector *se* controls whether to perform the mix Columns function or not.

The *seqSpecDecFSM* function is a FSM as well that represents the behavioral description of the decryption unit.

$$\begin{aligned}
&seqSpecDecFSM\ (n, pt, ct, keys) = seqSpecDecFSM\ ((n-1), ptNew, ct, keys)\\
&\quad where\\
&\qquad sel = (n == 10)\\
&\qquad se = (n == 1)\\
&\qquad it = muxS\ (sel,\ addKeyS\ (\ (keys!!10)\ , ct),\ pt)\\
&\qquad ctNew = stepSeqSpecDec(se, it, keys!!(n-1))
\end{aligned}$$

$$\begin{aligned}
&stepSeqSpecDec(se, it, ik) = (ptNew)\\
&\quad where\\
&\qquad ptNew = muxS(se, addKeyS(ik, (invSubBytesS(invShiftRowsS\ it)))\\
&\qquad , invMixClmsS(addKeyS(ik, (invSubBytesS(invShiftRowsS\ it)))))
\end{aligned}$$

4.1.2. Structural description

The hardware implementation of AES is described in the same hierarchical way as the behavioral description; it is composed of three components : the key expansion function *keyExpansionI*, the encryption function *seqImpEnc*, and the decryption function *seqImpDec*.

$$\begin{aligned}
&seqImpAES(op, text, key) = textNew\\
&\quad where\\
&\qquad keys = keyExpansionI\ \ 10\ \ key\\
&\qquad textNew = if(op == 1)\ \ then\ (seqImpEnc(text, keys))else\ (seqImpDec\\
&\qquad (text, keys))
\end{aligned}$$

The encryption function *seqImpEnc* converts the text from *ascii* to *hexadecimal* to *binary* and decompose it into blocks of 128bits and send it sequentially block by block to the function *seqImpEncFSM*, which is the FSM that models the structural description of the encryption unit.

$$
\begin{aligned}
&seqImpEncFSM\ (n, pt, ct, keys) = seqImpEncFSM\ ((n-1), pt, ctNew, keys)\\
&\quad where\\
&\qquad sel = (n == 10)\\
&\qquad se = (n == 1)\\
&\qquad it = muxI\ (sel,\ addKeyI\ (\ (keys!!0), pt),\ ct)\\
&\qquad ctNew = stepSeqImpEnc(se, it, keys!!(n-1))
\end{aligned}
$$

$$
\begin{aligned}
&stepSeqImpEnc(se, it, ik) = (ctNew)\\
&\quad where\\
&\qquad ctNew = addKeyI(ik, muxI(se, shiftRowsI(subBytesI\ it),\\
&\qquad\qquad mixClmsI(shiftRowsI(subBytesI\ it))))
\end{aligned}
$$

The inner components of this circuit (*seqImpEncFSM*), like *muxI*, *addKeyI*... are defined in terms of their structure using gates, unlike the *muxS*, *addKeyS*... which express the behavior of that component.

$$
\begin{aligned}
muxS(sel, a, b)\ &=\ if\ \ sel\ \ than\ \ a\ \ else\ \ b\\
muxI(sel, a, b)\ &=\ orI\ \ (andI\ \ sel\ \ a)\ \ (andI\ \ (notI\ \ sel)\ \ b)
\end{aligned}
$$

The hardware implementations of the main four components of AES are the followings:

- Add Key: is implemented using the bitwise XOR gate; and therefore it is the inverse of itself.
- Shift Rows: is a left cyclic shift of bits. The inverse shift rows is a right cyclic shift.
- Sub Bytes: relies on operations in the Galois Field GF (2^8); it can be implemented using only logic i.e. implementing circuits for the multiplication operation, the inverse operation, and for the affine transformation operation; or using the S-boxes, or Look-Up-Tables (LUTs); which are implemented as Read Only Memories (ROMs). We need 16 8x8 bit ROMs for the encryption and 16 for the Inverse Sub Bytes.
- Mix Columns: relies on operations in GF(2^8) as well, and can be expressed using only logic or LUTs too; which are implemented as 8x8 bit ROMs for each of the multiplication operations of both encryption and decryption process.

seqImpDecFSM function represents the structural description of the decryption unit.

4.2. Formal design of the Pipelined architecture

There are several architectures possible for pipelined block ciphers [15] ; we chose the one with inner-round pipelining depicted in Fig.4. with four pipeline stages. This makes it possible to encrypt four blocks simultaneously.

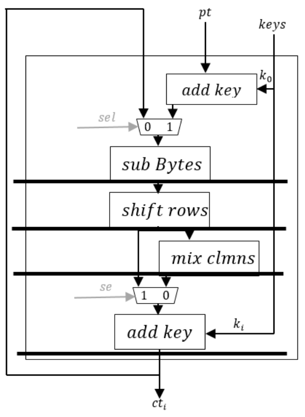

Figure 4: Basic iterative with inner-round pipelining.

4.2.1. Behavioral description

Similarly to the sequential architecture; AES is composed of three units: encryption, decryption, and key expansion. This latter is the same as in the sequential architecture.

$$\begin{aligned}&pipeSpecAES(op, text, key) = textNew\\&\quad where\\&\qquad keys = keyExpansionS\ \ 10\ \ key\\&\qquad textNew = if(op == 1)\ \ then\ (pipeSpecEnc(text,\\&\qquad keys))else\ (pipeSpecDec(text, keys))\end{aligned}$$

The function *pipeSpecEnc* converts the text from *ascii* to *hexadecimal*, decomposes it into blocks of 128bits, and send them to the function *pipeSpecEncFSM* that encrypts four blocks of data. Unlike in the sequential architecture, where the number of states equals the number of rounds; in this architecture, more internal states appear; which represents the pipeline stages registers. Thus, the next-state function *stepPipeSpecEnc* is a recursive function that takes as input the number of internal step *n*, the pre-calculated keys *keys*, and the current state of the four stages registers *(c1,c2,c3,c4)*.

$pipeSpecEncFSM(p4blocks, keys) = c4blocks$
$where$
$e = mixClmsS(shiftRowsS(subBytesS(addKeyS(p4blcks!!0, (keys!!0)))))$
$d = shiftRowsS(subBytesS(addKeyS(p4blcks!!1, (keys!!0))))$
$c = subBytesS(b3 = addKeyS(p4blcks!!2, (keys!!0)))$
$b = addKeyS(p4blcks!!3, (keys!!0))$
$(n, (c1, c2, c3, c4), k) = step(34, (b, c, d, e), keys)$
$ct1 = addKeyS((keys!!10), c3)$
$ct2 = addKeyS((keys!!10), (shiftRowsS\ c2))$

$ct3 = addKeyS((keys!!10), shiftRowsS(subBytesS\ c1))$
$ct4 = addKeyS((keys!!10), (shiftRowsS(subBytesS(addKeyS((keys!!9), c4)))))$
$c4blocks = [ct1, ct2, ct3, ct4]$

$stepPipeSpecEnc(n, (b, c, d, e), keys) = stepPipeSpecEnc((n-1), (bn, cn, dn, en), keys)$
$where$
$k = whatKey\ n\ keys$
$cn = subBytesS\ b$
$dn = shiftRowsS\ c$
$en = mixClmsS\ d$
$bn = addKeyS\ (k, e)$

The decryption function *pipeSpecDecFSM* is also a FSM; it's next-state function is *stepPipeSpecDec*.

4.2.2. Structural description

Like for the sequential architecture, the hardware implementation of pipelined AES is described in the same hierarchical way as its behavioral description; it is composed of three components : the key expansion function *keyExpansionI*, the encryption function *pipeImpEnc*, and the decryption function *pipeImpDec*.

```
$pipeImpAES(op, text, key) = textNew$
    $where$
        $keys = keyExpansionI\ \ 10\ \ key$
        $textNew = if(op == 1)\ \ then\ (pipeImpEnc(text, keys))\ \ else\ (pipeImpDec$
        $(text, keys))$
```

The encryption function *PipeImpEnc* converts the text from *ascii* to *hexadecimal* to *binary* and decompose it into blocks of 128bits and send them to the function *pipeImpEncFSM*, which is the FSM that models the structural description of the encryption unit.

```
$pipeImpEncFSM(p4blocks, keys) = c4blocks$
  $where$
    $e = mixClmsI(shiftRowsI(subBytesI(addKeyI(p4blcks!!0, (keys!!0)))))$
    $d = shiftRowsI(subBytesI(addKeyI(p4blcks!!1, (keys!!0))))$
    $c = subBytesI(addKeyI(p4blcks!!2, (keys!!0)))$
    $b = addKeyI(p4blcks!!3, (keys!!0))$
    $(n, (c1, c2, c3, c4), k) = stepI(34, (b4, c, d, e), keys)$
    $ct1 = addKeyI((keys!!10), c3)$
    $ct2 = addKeyI((keys!!10), (shiftRowsIc2))$
```

$$
\begin{aligned}
&\quad ct3 = addKeyI((keys!!10), (shiftRowsI(subBytesIc1)))\\
&\quad ct4 = addKeyI((keys!!10), (shiftRowsI(subBytesI(addKeyI((keys!!9), c4)))))\\
&\quad c4blocks = [ct1, ct2, ct3, ct4]
\end{aligned}
$$

$$
\begin{aligned}
&stepPipeImpEnc(n, (b, c, d, e), keys) = stepPipeImpEnc((n-1), (bn, cn, dn, en), keys)\\
&\ where\\
&\quad k = whatKey\ n\ keys\\
&\quad cn = subBytesIb\\
&\quad dn = shiftRowsIc\\
&\quad en = mixClmsId\\
&\quad bn = addKeyI(k, e)
\end{aligned}
$$

The *pipeImpDecFSM* function represents the decryption FSM; it's next-state function is *stepPipeImpDec*.

4.3. Formal verification

In order to verify that both sequential and pipelined architectures implement correctly the behavior of their specifications; seven theorems have to be proved.

1. First we need to verify the correctness property of the initial specification, which is expressed by the following theorem:

$$\forall\, m,\ Decryption(Encryption\ m) =\ m \tag{2}$$

2. Then we need to verify the equivalence of the implementation of the encryption unit to its specification:

$$\forall\, n,\ pt,\ ct,\ keys; seqSpecEncFSM(abs(n, pt, ct, keys)) = abs(seqImpEncFSM(n, pt, ct, keys)) \tag{3}$$

3. And the equivalence of the decryption unit implementation against its specification:

$$\forall\, n,\ pt,\ ct,\ keys; seqSpecDecFSM(abs(n, pt, ct, keys)) = abs(seqImpDecFSM(n, pt, ct, keys)) \tag{4}$$

4. Afterwards, we need to verify the equivalence between the specification of the pipelined text encryption unit and the sequential one :

$$\forall\, text,\ keys; seqSpecEnc(text, keys) = pipeSpecEnc(text, keys) \tag{5}$$

5. And the equivalence between the specification of the pipelined text decryption unit and the sequential one:

$$\forall\, text,\ keys; seqSpecDec(text, keys) = pipeSpecDec(text, keys) \tag{6}$$

6. Finally, we need to verify the equivalence of the pipelined implementation of the encryption unit against its specification:

$$\forall\, n,\ pt,\ ct,\ keys; pipeSpecEncFSM(abs(n, pt, ct, keys)) = abs(pipeImpEncFSM(n, pt, ct, keys)) \tag{7}$$

7. And respectively, we need to verify the equivalence of the pipelined implementation of the decryption unit against its specification:

$$\forall\, n,\ pt,\ ct,\ keys; pipeSpecDecFSM(abs(n, pt, ct, keys)) = abs(pipeImpDecFSM(n, pt, ct, keys)) \tag{8}$$

4.3.1. Formal verification of the sequential architecture

Theorem (2) is written in Lava in the following form:

```
$verificationProperty\ (pt,\ keys) = ok$
    $where$
        $pt0 = fromListToState\ \ pt$
        $keys0 = fromListToListOfStates\ \ keys$
        $(n1, pt1, ct1, k1) = seqSpecEncFSM\ (10, pt0,\ [[[\,]]],\ keys0)$
        $(n2, pt2, ct2, k2) = seqSpecDecFSM\ (10, [[[\,]]],\ ct1,\ keys0)$
        $ok = pt0 <==> pt2$
```

Because lists are infinite structures, we need to define a verification property that is explicit about the sizes:

```
$verifPropForSize\ n\ m =$
    $forAll\ (list\ n)\ \$\ \backslash\ pt\ \ ->$
        $forAll\ (list\ m)\ \$\ \backslash\ keys\ \ ->$
            $verificationProperty(pt,\ keys)$
```

To prove this property we call the *satzoo* function:
$satzoo\ \ propVerifForSize\ \ 128\ \ 1408$ (1408 is the total of eleven 128bit size keys); it returns ***Valid***, which means that the specification holds that property.

In order to be able to prove theorem (3) and (4), we need to verify the descriptions internal components; using the same method as for theorem (2); to the multiplexer component, the main four operations (sub bytes, shift rows, mix columns, and add key), and their inverses:

```
propertyEquivSubBytes  s = ok
    where
        x =  fromListToState  s
        out1 = subBytesS  (abs  x)
        out2 = abs  (subBytesI  s)
        ok = out1 <==> out2
```

```
propEquivSubBytesForSize n =
                        forAll (list n) $ \ s  - >
                                propertyEquivSubBytes s
```

```
satzoo  propEquivSubBytesForSize  128
```

All these properties were satisfied and gave back the answer ***Valid***; which means that the implementations of these components are correct.

The sub-circuit *stepSeqImpEnc* that is composed of the previously verified components is also correct according to the theorem proved above.

$$\begin{aligned}
&\forall\, in,\ subBytesS(abs\ \ in) = \ abs(subBytesI\ \ in),\\
&\forall\, in,\ shiftRowsS(abs\ \ in) = \ abs(shiftRowsI\ \ in),\\
&\forall\, in,\ mixClmnsS(abs\ \ in) = \ abs(mixClmnsI\ \ in),\\
&\forall\, in,\ addKeyS(abs\ \ in) = \ abs(addKeyI\ \ in),\\
&\forall\, in,\ muxS(abs\ \ in) = \ abs(muxI\ \ in),\\
&\Rightarrow \forall\, in,\ stepSeqSpecEnc(abs\ \ in) = abs(stepSeqImpEnc\ \ in)
\end{aligned}$$

Now that all the inner-components are verified, we verify the encryption FSM using induction.

Basis:

$$\begin{aligned}
&seqSpecEncFSM(abs(0, pt, ct, keys)) = \ abs(0, pt, ct, keys)\\
&abs(seqImpEncFSM(0, pt, ct, keys)) = \ abs(0, pt, ct, keys)\\
&\Rightarrow\ seqSpecEncFSM(abs(0, pt, ct, keys)) = abs(seqImpEncFSM(0, pt, ct, keys))
\end{aligned}$$

Induction step:

$\forall\ n \in Int;\ pt,\ ct \in [[[Signal\ Bool]]];\ keys \in [[[[Signal\ Bool]]]];$
$seqSpecEncFSM(abs(n, pt, ct, keys)) = abs(seqImpEncFSM(n, pt, ct, keys))$
$\Rightarrow seqSpecEncFSM(abs((n{-}1), pt, ct, keys)) = abs(seqImpEncFSM((n{-}1), pt, ct, keys))$
$abs(seqImpEncFSM(n, pt, ct, keys))$
$= abs(seqImpEncFSM((n-1), pt, ct_i, keys))$
$with\ \ ct_i = stepSeqImpEnc\ \ ct$
$seqSpecEncFSM(abs(n, pt, ct, keys))$
$= seqSpecEncFSM(n, abs\ pt, abs\ ct, abs\ keys)$
$= seqSpecEncFSM((n-1),\ abs\ pt, ct_s, abs\ keys)$
$with\ \ ct_s = stepSeqSpecEnc\ (abs\ ct)$
$stepSeqSpecEnc(abs\ ct) = abs(stepSeqImpEnc\ ct)$
$\Rightarrow ct_s = abs\ ct_i$
$= seqSpecEncFSM((n-1),\ abs\ pt,\ abs\ ct_i,\ abs\ keys)$
$= seqSpecEncFSM(abs((n-1), pt, ct_i, keys))$

$\Rightarrow seqSpecEncFSM(abs((n-1), pt, ct, keys))$
$= abs(seqImpEncFSM((n-1), pt, ct, keys))$

We verified the correctness of the decryption unit's implementation in the same way.

$\Rightarrow \forall\ n \in Int; pt,\ ct \in [[[Signal\ Bool]]];\ keys \in [[[[SignalBool]]]];$
$seqSpecDecFSM(abs(n,\ pt,\ ct,\ keys)) = abs(seqImpDecFSM(n,\ pt,\ ct,\ keys))$

$\Rightarrow \forall\ op \in Int, text, key \in [Char];$
$seqSpecAES(abs(op, text, key)) = abs(seqImpAES(op, text, key))$

4.3.2. Formal verification of the pipelined architecture

For the verification of the pipelined behavioral description against the sequential one, we verify the equivalence of their upper components that encrypts multiple blocks of data; because the next state in the sequential FSM consists of the result after one round; however, the next state in the pipelined FSM is the result of one inner-round pipeline stage.
$equivPropSeqPip(text, keys) = ok$
$where$
$txt = fromTextToBlocks\ \ text$
$ks = fromListToState\ \ keys$
$out1 = seqSpecEnc(txt, ks)$
$out2 = pipeSpecEnc(txt, ks)$
$ok = out1 <\!==\!> out2$

$equivPropSeqPipForSize\ n\ m\ =$
$forAll\ (list\ n)\ \$\ \backslash\ text\ ->$
$forAll\ (list\ m)\ \$\ \backslash\ keys\ ->$
$equivPropSeqPip\ (text, keys)$

We can only verify the property for a fixed text size at the time.
$satzoo\ \ equivPropSeqPipForSize\ (length\ \ text)\ 128$
For the implementation of the pipelined architecture, we check its equivalence to the behavioral description using induction. Since the inner components are the same for the sequential architecture we do not need to verify their equivalence.

$$\forall\, n \in Int; b, c, d, e \in [[[Signal\ Bool]]]]; keys \in [[[[SignalBool]]]];$$
$$stepPipeSpecEnc(abs\ \ (n, (b, c, d, e), keys)) = abs(stepPipeImpEnc\ \ (n, (b, c, d, e), keys))$$

Basis:

$$stepPipeSpecEnc(abs(0, (b, c, d, e), keys)) = abs(0, (b, c, d, e), keys)$$
$$abs(stepPipeImpEnc(0, (b, c, d, e), keys)) = abs(0, (b, c, d, e), keys)$$
$$\Rightarrow\ stepPipeSpecEnc(abs(0, (b, c, d, e), keys)) = abs(stepPipeImpEnc(0, (b, c, d, e), keys))$$

Induction step:
$\forall\, n \in Int;\ b, c, d, e, \in [[[Signal\ Bool]]],\ keys \in [[[[Signal\ Bool]]]];$
$stepPipeSpecEnc(abs(n, (b, c, d, e), keys)) = abs(stepPipeImpEnc(n, (b, c, d, e), keys)) \Rightarrow$

$stepPipeSpecEnc(abs((n{-}1), (b, c, d, e), keys)) = abs(stepPipeImpEnc((n{-}1), (b, c, d, e), keys))$

$stepPipeSpecEnc(abs(n, (b, c, d, e), keys))$
$= stepPipeSpecEnc((n, (abs\ b, abs\ c, abs\ d, abs\ e), abs\ keys))$
$= stepPipeSpecEnc(((n-1), (b_s, c_s, d_s, e_s), keys))$
$with\ b_s = addKS(abs\ e, k)$
$c_s = subBytesS(abs\ b)$
$d_s = shiftRowsS(abs\ c)$
$e_s = mixClmnsS(abs\ d)$

$abs(stepPipeImpEnc(n, (b, c, d, e), keys))$
$= abs(stepPipeImpEnc((n-1), (b_i, c_i, d_i, e_i), keys))$
$with\ b_s = addKI(e, k)$
$c_i = subBytesIb$
$d_i = shiftRowsIc$
$e_i = mixClmnsId$

$addKeyS(abs\ e, k) = abs(addKeyI(e, k)) \Rightarrow b_s = abs\ b_i$

$subBytesS(abs\ b) = abs(subBytesI\ b) \Rightarrow c_s = abs\ c_i$
$shiftRowsS(abs\ c) = abs(shiftRowsI\ c) \Rightarrow d_s = abs\ d_i$
$mixClmnsS(abs\ d) = abs(mixClmnsI\ d) \Rightarrow e_s = abs\ e_i$

$\Rightarrow stepPipeSpecEnc(((n-1),(b_s,c_s,d_s,e_s),keys))$
$= stepPipeSpecEnc(((n-1),(abs\ b_i, abs\ c_i, abs\ d_i, abs\ e_i),keys))$
$= stepPipeSpecEnc(abs((n-1),(b_i,c_i,d_i,e_i),keys)$
$= abs(stepPipeImpEnc((n-1),(b_i,c_i,d_i,e_i),keys))$

$\Rightarrow \forall\ n \in Int;\ b,c,d,e, \in [[[Signal\ Bool]]],\ keys \in [[[[Signal\ Bool]]]];$
$stepPipeSpecEnc(abs(n,(b,c,d,e),keys)) = abs(stepPipeImpEnc(n,(b,c,d,e),keys))$

We verify the decryption unit in the same way as the encryption one.

$\forall\ n \in Int;\ b,c,d,e, \in [[[Signal\ Bool]]],\ keys \in [[[[Signal\ Bool]]]];$
$stepPipeSpecDec(abs(n,(b,c,d,e),keys)) = abs(stepPipeImpDec(n,(b,c,d,e),keys))$

Now that we verified the equivalence between the pipelined *step* function's implementation and its specification; the equivalence of the upper-components is verified.

$\forall\ blocks \in [[[[Signal\ Bool]]]],\ keys \in [[[[Signal\ Bool]]]];$
$pipeSpecEncFSM(abs(blocks,keys)) = abs(pipeImpEncFSM(blocks,keys))$

$\forall\ blocks \in [[[[Signal\ Bool]]]],\ keys \in [[[[Signal\ Bool]]]];$
$pipeSpecDeccFSM(abs(blocks,keys)) = abs(pipeImpDecFSM(blocks,keys))$

$\Rightarrow \forall\ op \in Int, text, key \in [Char];$
$pipeSpecAES(abs(op,text,key)) = abs(pipeImpAES(op,text,key))$

In table I, we show how long the verification process took for the main sub-components and of the complete AES sequential circuit. As we can notice, all the theorems that has been proved automatically using the Lava's SAT solver, took less than 2s. As for theorems (3) and (4), which express the equivalence between the sequential encryption FSM specification (sEFSMS) and its implementation (sEFSMI), and between the sequential decryption FSM specification (sDFSMS) and its implementation (sDFSMI) respectively; they were proved manually using induction in a simple straightforward way. Each of these theorems took from 3 to 5 minutes to be proved. This makes the total amount of time needed for the verification of the whole circuit 490.5 s (**A** = the total of the **A**utomatic verification / **M** = the total of the **M**anual verification).

Table II shows the time that was taken for the verification of the AES pipelined circuit. Theorems (5) and (6), which express the equivalence between the specification of the sequential architecture and that of the pipelined one, where proved automatically using Lava's SAT solver. As for the verification of the pipelined implementation against its verified specification, we proved theorems (7) and (8) using induction. This makes the total of the verification time for the pipelined circuit 421.97 s.

Component	Property .... Theorem number	Method	Verification Time (s)
AES Spec	$\forall x, Decryption(Encryption\ \ x) = x$ .... (1)	SAT	1.05
Mux	$\forall x, muxS(abs\ x) = abs(muxI\ x)$	SAT	1.78
Xor	$\forall x, xorS(abs\ x) = abs(xorI\ x)$	SAT	0.73
AddKey	$\forall x, addKeyS(abs\ x) = abs(addKeyI\ x)$	SAT	1.08
SubBytes	$\forall x, subBytesS(abs\ \ x) = abs(subBytesI\ x)$	SAT	1.18
ShiftRows	$\forall x, shiftRowsS(abs\ \ x) = abs(shiftRowsI\ x)$	SAT	1.18
MixClms	$\forall x, mixClmnsS(abs\ \ x) = abs(mixClmnsI\ x)$	SAT	0.86
KeyExp	$\forall x, keyExpS(abs\ x) = abs(keyExpI\ x)$	SAT	0.72
AESEnc	$\forall x, sEFSMS(abs\ \ x) = abs(sEFSMI\ x)$ .... (2)	Induction	$\simeq 300$
InvSubBytes	$\forall x, invSBS(abs\ x) = abs(invSBI\ x)$	SAT	0.72
InvShiftRows	$\forall x, invSRS(abs\ x) = abs(invSRI\ x)$	SAT	0.6
InvMixClms	$\forall x, invMCS(abs\ x) = abs(invMCI\ x)$	SAT	0.6
AESdec	$\forall x, sDFSMS(abs\ \ x) = abs(sDFSMI\ x)$ .... (3)	Induction	$\simeq 180$

Total (AESseq Circuit)			A= 10.5
			
			M= 480

Table 1
Verification time of the sequential AES circuit and its inner-components.

Finally, in table III, we recapitulate the different works related to the proposed approach. As we can see, the majority of the existing works are based on an imperative approaches. These approaches are well suited for simulation [2],[3], but we notice that it is still possible to use them for formal verification as well [1],[4],[5],[8],[9],[10]. In this case, the description for the design and verification is long and complex. Only one functional approach beside ours exists that targeted the cryptographic circuits [11]; no details about the verification time were mentioned. A couple of the existing works are based on an algebraic approach [6],[7]; similarly to our work, they use the hierarchy technique for proof decomposition; which makes the verification process easier. As we can see, our proposed methodology takes half the time for the verification of the same cryptographic circuit.

5. Conclusions

In this paper, we presented a formal design and verification methodology for symmetric cryptographic circuits. We combine the two techniques (induction and SAT solver) for verifying different parts of the implementation. The proposed approach was demonstrated over both sequential and pipelined architectures of the symmetric cipher AES128. As prospects, we aim to optimize the methodology into a fully automtic one, and to be able to design and verify the

Component	Property .... Theorem number	Method	Verification Time (s)
AESencSpec	$\forall x, seqSpecEnc\ x = pipeSpecEnc\ x$ .... (4)	SAT	1.12
AESdecSpec	$\forall x, seqSpecDec\ x = pipeSpecDec\ x$ .... (5)	SAT	0.85
AESencImp	$\forall x, pEFSMS(abs\ x) = abs(pEFSMI\ x)$ .... (6)	Induction	$\simeq 240$
AESdecImp	$\forall x, pDFSMS(abs\ x) = abs(pDFSMI\ x)$ .... (7)	Induction	$\simeq 180$

Total (AE-Spip Circuit)			A= 1.97
			
			M= 420

Table 2
Verification time of the pipelined AES circuit.

Work	Approach	Verification Method	Cryptographic circuit	Verification Time (s)
[5] 2003	Imperative	Formal methods	DES	59
[10] 2004	Imperative	Formal methods/simulation	AES128	/
[1] 2006	Imperative	Theorem proving	SHA1	/
[8] 2007	Imperative	Equivalence checking	Pipelined KASUMI	180 .. 540
[11] 2007	Functional	Formal methods	AES128	/
[4] 2008	Imperative	Equivalence checking	Pipelined KASUMI	180 .. 540
[6] 2012	Algebraic	Formal methods	AES128 datapath	800
[7] 2014	Algebraic	Formal methods	AES128	844
[3] 2014	Imperative	Simulation	AES256	/
[2] 2015	Imperative	Simulation	TDES	/
[9] 2017	Imperative	Formal methods	Pipelined AES128	2300
This work	Functional	Formal methods	Sequential AES128	490.5
This work	Functional	Formal methods	Pipelined AES128	421.97

Table 3
Comparative table of the related literature's verification methods and time.

super-scalar architectures.

References

[1] Toma, D. "Vérification Formelle des systèmes numériques par démonstration de théorèmes: application aux composants cryptographiques" (Doctoral dissertation) (2006).

[2] Singh, Kirat Pal, and Shivani Parmar. "Design of high performance MIPS cryptography processor based on T-DES algorithm." arXiv preprint arXiv:1503.03166 (2015).

[3] Ali, Imran, Gulistan Raja, and Ahmad Khalil Khan. "A 16-Bit Architecture of Advanced Encryption Standard for Embedded Applications." 2014 12th International Conference on Frontiers of Information Technology. IEEE, (2014).

[4] Lam, Chiu Hong. "Verification of pipelined ciphers". MS thesis. University of Waterloo, (2009).

[5] Clarke, E., Kroening, D. "Hardware verification using ANSI-C programs as a reference". In Proceedings of the ASP-DAC Asia and South Pacific Design Automation Conference. pp. 308-311. IEEE. (2003).

[6] Homma, Naofumi, Kazuya Saito, and Takafumi Aoki. "A Formal Approach to Designing Cryptographic Processors Based on $GF(2^m)$ Arithmetic Circuits." IEEE Transactions on Information Forensics and Security vol. 7, no 1, p. 3-13. (2011).

[7] Homma, Naofumi, Kazuya Saito, and Takafumi Aoki. "Toward formal design of practical cryptographic hardware based on Galois field arithmetic." IEEE Transactions on Computers. vol. 63, no 10, p. 2604-2613 (2013).

[8] Lam, Chiu Hong, and Mark D. Aagaard. "Formal Verification of a Pipelined Cryptographic Circuit Using Equivalence Checking and Completion Functions." 2007 Canadian Conference on Electrical and Computer Engineering. IEEE, p. 1401-1404. (2007).

[9] Bond, Barry, et al. "Vale: Verifying high-performance cryptographic assembly code." 26th USENIX Security Symposium (USENIX Security 17). p. 917-934. (2017).

[10] Kim, Ho Won, and Sunggu Lee. "Design and implementation of a private and public key crypto processor and its application to a security system." IEEE Transactions on Consumer Electronics.vol. 50, no 1, p. 214-224. (2004).

[11] Lewis, Jeff. "Cryptol: specification, implementation and verification of high-grade cryptographic applications." Proceedings of the 2007 ACM workshop on Formal methods in security engineering. p. 41-41.(2007).

[12] Wolfs, Davy, et al. "Design automation for cryptographic hardware using functional languages." Proceedings of the 32nd WIC Symposium on Information Theory in the Benelux. Werkgemeenschap voor Informatie-en Communicatietheorie.; Netherlands, p. 194-201. (2011).

[13] Guo, Xiaolong, et al. "Pre-silicon security verification and validation: A formal perspective." Proceedings of the 52nd Annual Design Automation Conference. p. 1-6. (2015).

[14] Daemen, Joan, and Vincent Rijmen. The design of Rijndael: AES-the advanced encryption standard. Springer Science and Business Media, (2013).

[15] Gaj, K., and Chodowiec, P. (2009). FPGA and ASIC implementations of AES. In Cryptographic engineering (pp. 235-294). Springer, Boston, MA.