Formal Verification of Pipelined Cryptographic Circuits: A Functional Approach

Abir Bitat and Salah Merniz
MISC Laboratory, NTIC faculty, Abdelhamid Mehri University, Constantine, Algeria
E-mail: abir.bitat@univ-constantine2.dz

Keywords: formal design, formal verification, cryptographic circuits, pipelined circuits, functional language, hardware description language

Received: May 28, 2020

Cryptographic circuits are essential in systems where security is the main criteria. Therefore, it is crucial to ensure the correctness of not only the cryptographic algorithms, but also their hardware implementations. Formal methods, unlike the other validation techniques, guarantee the absence of errors.The problem is that designers still use conventional imperative Hardware Description Languages (HDLs), which are poorly suited for formal verification.
This paper presents an automatic verification methodology for the pipelined cryptographic circuits using formal methods. It consists of using the functional HDL Lava to describe and verify the equivalence between the behavioural specification and structural implementation of a circuit. To the best of our knowledge, we are the first to use this functional HDL for that purpose.
To show the features of the proposed approach, it was applied to verify the pipelined implementation of the cryptographic circuit AES (Advanced Encryption Standard).

Povzetek: Za namene preverjanja formalne pravilnosti delovanja vezij je opisan funkcionalni pristop.

1 Introduction

Cryptography plays a major role in modern applications, as the present networks are trusted with highly sensitive information; hence, cryptographic circuits have become indispensable in these systems. To ensure the security of information, not only the cryptographic algorithms have to be verified but also their hardware implementations.

The first step of the design process consists in the conversion of the informal description of the design to a formal *Behavioural* (or *Algorithmic*) specification. From this latter, a *Structural* (or *Micro-architectural*) implementation is derived through refinement, followed by a sequence of design steps that reduce the abstraction levels until a realizable description is obtained. The verification of the circuit is carried out through all these stages; the most critical one is the *Functional verification*, which consists of confirming that the structural implementation provides the same behaviour mentioned in the behavioural specification. It is possible to verify the correctness of the cryptographic circuits, in quite a few ways, such as simulation, formal proof, and semi-formal verification in which formal techniques and simulation are strongly combined.

We focus on our work on functional verification, and we use formal methods to do so. In the following subsection, we present a summary of the related literature and their shortcomings to highlight the problematic; then we present the features of the proposed approach and the contribution of this paper. In section two, we explain the proposed verification methodology. In section three, we demonstrate how we applied our approach for verifying the pipelined cryptographic circuit Advanced Encryption Standard (AES). And finally, conclusions are drawn in section four.

1.1 Related works

The vast majority of the existing literature related to the description and verification of pipelined cryptographic circuits are based on using imperative HDLs such as VHDL and Verilog,[1], [2], [3], [4], [5]. These languages are made for description, simulation, and synthesis of hardware; however, they are poorly suited for formal verification, because of their lack of formal semantics; besides, they do not allow descriptions of the highest design levels [6],[7]. In order to be able to use formal methods for verification, we need formal descriptions that we can reason about; however, imperative HDLs provide descriptions that are hard to express in any formal logic; [8],[6],[9],[10],[11], which requires either a translation of those descriptions to some formal logic or rewriting new "equivalent" descriptions, which eliminates the need for the imperative ones in the first place, because there is no relation between the two [6],[12],[13],[14]. This makes the formal verification of devices expressed in imperative HDLs a quite hard process.

Consequently, simulation is the technique that has been used in most of these approaches [2], [3], [4]. The problem with simulation is that it can not be sufficient as a verification technique for systems as critical as the cryptographic circuits. Formal verification can still be done but with quite a few challenges when using imperative HDLs; therefore, the abstract behavioural specification needs powerful mechanisms of structuring and translating [5]. Mostly, deductive methods are applied for verification of such complex circuits, like the case of [1]; this kind of methods is quite difficult because it usually requires user interference through all the verification process in almost a manual way.

Some algebraic approaches [15], [16] used formal methods for the verification of these circuits; similar to the work presented in this paper, the hierarchy technique was used to reduce the complexity of design and therefore simplify the verification task. However, many details differ in our approach from this work, in particular, the verification time is significantly less than the one presented in [15], which took 13 minutes to formally verify the same 128bit AES circuit. Another approach that was slightly faster than the work mentioned before, with a difference of 5 minutes, is presented in [17]. It consists of a language that supports automated verification of cryptographic assembly code.

Several functional approaches were applied for the formal verification of hardware designs; however, to the best of our knowledge, the only one beside ours that was applied to implement and verify the pipelined cryptographic circuits is the work presented in [18]. This approach uses the functional language only to describe the behavioural specification; but not for the structural implementation; which makes translation difficulties reappear.

Another formal approach was proposed in [19]; it uses the equivalence checking technique. The specification is described at the RTL level using VHDL, and the verification process of some pipelined implementations of the KASUMI cipher took from 3 to 9 minutes, depending on the number of stages. Another work that targeted pipelined implementations is [20]; it uses VHDL to describe private and public key crypto-processor; the verification was done using simulation, formal verification, and static timing analysis.

1.2 Contribution of this work

- The most important feature of our approach, is that it consists of using a functional HDL, which is very suitable for hardware description and for formal verification as well. A comparison study of these HDLs against other types showed that they give the best results [21].

- Secondly, our approach performs the functional verification using formal methods, as this latter consists of proving mathematically the correctness of a design, which is crucial for security systems.

- In addition, our approach uses two techniques: which are hierarchy and modularity, in order to reduce the complexity of the design, which makes the verification much easier.

- Unlike the other conventional HDLs, it is possible to represent the most abstract descriptions with functional HDLs.

- Descriptions of functional languages and HDLs can be executed, which allow the verification through the simulation technique as well as the formal verification.

- The functional HDL used in our approach has some built-in tools that allow automatic formal verification of circuits.

- The proposed approach presents a verification methodology that is easier and faster than the previous related works.

- Lastly, even though our approach was proposed for pipelined cryptographic circuits, it is not exclusive to them.

2 The proposed approach

The proposed approach consists of a formal design and verification methodology for the pipelined cryptographic circuits using a functional HDL. This choice is motivated by the interesting characteristics of these HDLs; such as the composition of functions in the same way that complex circuits are developed, using function composition, renaming, and abstraction; the other major advantage of introducing the functional style to hardware design is having much more concise descriptions, and the ability to provide reusable functions that are abstractions of common patterns. Moreover, functional languages usually have an extremely expressive type system, which allows being more strict on defining the limitations on values. This makes finding errors and violations easier. Several functional HDLs have been created over the decades; their high diversity is due to the complexity of hardware design. A historical survey that discusses these languages can be found in [22]. The language that we chose for our approach is Lava [23], which consists of a simple HDL embedded in the functional programming language Haskell. To the best of our knowledge, no functional HDLs (including Lava) has been used before for both description and formal verification of the cryptographic circuits.

The design flow of hardware devices is depicted in Fig. 1. It starts with an algorithmic description which will be considered as the initial specification of the design; then other descriptions are derived from it. Each implementation resulting in a certain abstraction level will be used as the specification for the next one.

Since the algorithmic level is the most abstract, the architectural details do not appear at it; thus, both sequential and pipelined architectures have the same algorithmic description. Accordingly, we use the same principle of the design flow to verify a pipelined implementation of a cryptographic circuit; we start first by verifying the correctness of a sequential structural implementation against its

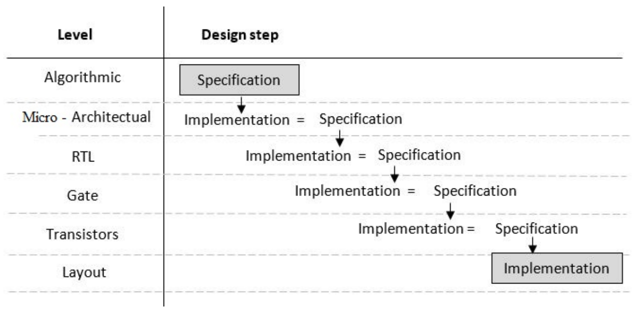
Level
Design step
Algorithmic
Specification
Micro - Architectual
Implementation = Specification
RTL
Implementation = Specification
Gate
Implementation = Specification
Transistors
Implementation = Specification
Layout
Implementation

Figure 1: The design and verification flow of a hardware device.

behavioural specification; once it is verified, we check the equivalence of the pipelined structural implementation to it, as demonstrated in Fig. 2.

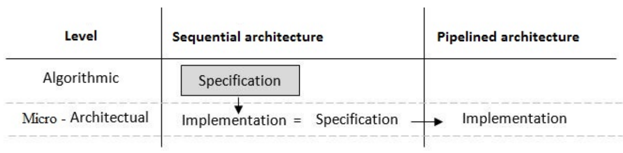

Figure 2: A verification approach of a pipelined implementation of a circuit.

The behavioural specification at the algorithmic level deals with a different type than the structural implementations at the micro-architectural level; thus, we need a mapping function between the two descriptions, as shown in Fig. 3.

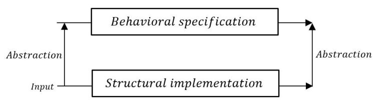

Figure 3: The abstraction function needed between the behaviour specification and structural implementation of a circuit.

The shared behavioural description is referred to by *Spec*; the sequential architecture by *imp1*, and the pipelined one by *imp2*. So, the correctness rule of this latter would be described by the following theorem:

$$\forall\, x,\ spec\,(abs\; x)\ =\ abs\,(imp2\; x) \tag{1}$$

Theorem (1) has to be decomposed to be proven; thus, we must prove the following couple of theorems:

$$\forall\, x,\ spec\,(abs\; x)\ =\ abs\,(imp1\; x) \tag{2}$$

$$\wedge \qquad \forall\, x,\ imp1\; x\ =\ imp2\; x \tag{3}$$

The behavioural specification and both structural implementations are described as functions, the former using the functional language Haskell, and the latter using the functional HDL Lava; which consists of Haskell modules that give the user various facilities to work on circuits.

For formal verification; we use one of Lava tools, which is a SAT solver, that verifies automatically the equivalence between descriptions. Fig. 4 shows an approach of performing equivalence checking using SAT solvers. If both descriptions are equivalents, the output of the *XOR* gate should be always *False*; if it becomes *True* for any input, it means that the two descriptions are producing different outputs for the same input, which negates their equivalence.

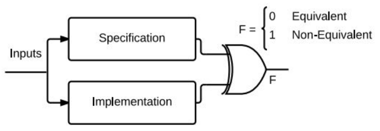

Figure 4: Performing equivalence checking using SAT solvers [24].

We must import three modules to be able to shift from the general-purpose language Haskell to the HDL Lava: *Lava* [25], which defines several operations that we can use to build circuits; *Patterns*, to access wiring circuits and connection patterns; and *Arithmetic*, to access the arithmetical circuits:

$$import\ Lava$$
$$import\ Lava.Patterns$$
$$import\ Lava.Arithmetic$$

Circuits in Lava are described by functions, and their inputs and outputs can either be of type *Signal Bool* which may take one of two values *low* or *high*; or *Signal Int*.

To verify that a circuit's structural implementation meets its behavioral specification, we must define a safety property that expresses their equivalence. Thus, for the verification of the pipelined implementation, we need two safety properties corresponding to the two theorems (2) and (3) mentioned before. These properties are also defined in Lava by functions in the following forms:

```
$propertyEquiv1\ \ in = ok$
    $where$
        $out1 = spec\ \ (abs\ \ in)$
        $out2 = abs\ \ (imp1\ \ in)$
        $ok = out1 <==> out2$

$propertyEquiv2\ \ in = ok$
    $where$
        $out1 = imp1\ \ in$
        $out2 = imp2\ \ in$
        $ok = out1 <==> out2$
```

To verify these properties, we use the Lava function *satzoo*, which is a call to the satisfiability solver:

$$\begin{gathered} satzoo \;\; propertyEquiv1 \\ satzoo \;\; propertyEquiv2 \end{gathered}$$

This operation generates a logical formula that expresses the equivalence property; this formula is then sent to an external theorem prover, which will prove (or disprove) its validity, and the result is taken back into Lava.

The input *in* must be of a finite form; this is not possible in cryptographic circuits, where both data and key are of an important size that can only be represented by lists. But since the inputs of block cipher cryptographic circuits are of a fixed size, we will only verify the properties for that size. Thus, we define new equivalence properties, which are explicit about what size of input we want to verify them.

$$\begin{aligned} & propertyEquiv1ForSize \;\; n = \\ & \quad forAll\ (list\ n)\ \$ \setminus in \ \rightarrow \\ & \quad\quad propertyEquiv1 \;\; in \end{aligned}$$

$$\begin{aligned} & propertyEquiv2ForSize \;\; n = \\ & \quad forAll\ (list\ n)\ \$ \setminus in \ \rightarrow \\ & \quad\quad propertyEquiv2 \;\; in \end{aligned}$$

Then we call the function *satzoo* for both properties with a specific size n:

$$\begin{gathered} satzoo\ (propertyEquiv1ForSize \;\; n) \\ satzoo\ (propertyEquiv2ForSize \;\; n) \end{gathered}$$

This operation will verify that both descriptions give the same output, for all inputs of that size. When it finds that the output of one description always equals the other, it returns *Valid*. The SAT-solver actually checks that the negation of the formula is unsatisfiable, leading to the Valid answer inside Lava [25]. Our proposed approach explained above is depicted in Fig. 5.

3 Application and results

In this section, we demonstrate how we applied the proposed approach to verify the AES sequential circuit, illustrated in Fig. 6; and the pipelined circuit, illustrated in Fig. 7. AES [26] is a symmetric block cipher, constructed based on the Rijandael system. The plain and ciphertexts are taken as blocks of 128 bits. The key, on the other hand, varies depending on the system version, between 128, 192, and 256 bits. We only focus here on the 128-bit key size AES.

The encryption consists of ten identical rounds of processing. Except for the last one, each round includes four steps; the order in which these steps are executed is different in encryption from decryption. The 128-bit input

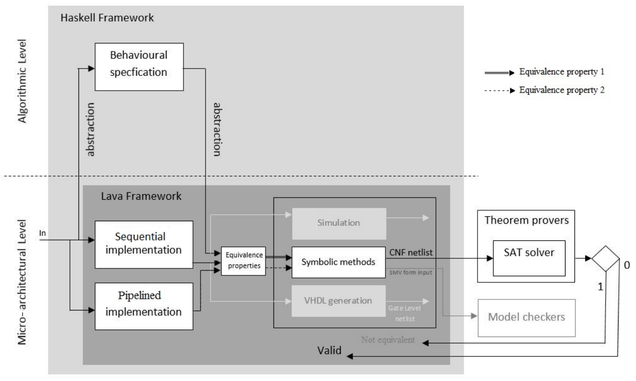

Figure 5: The proposed verification methodology of pipelined cryptographic circuits in a functional framework.

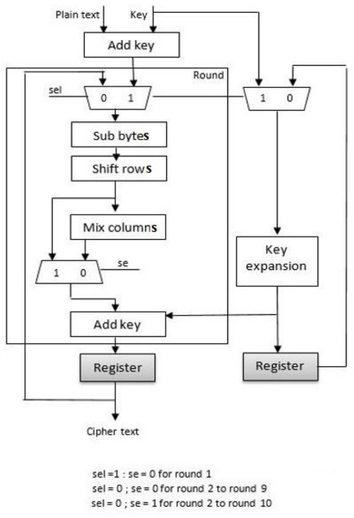

Figure 6: Sequential architecture of AES128 circuit.

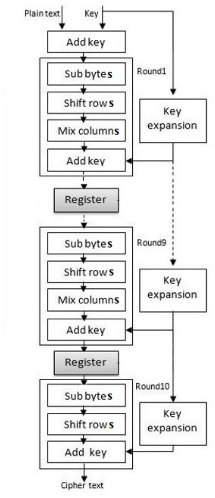

Figure 7: Pipelined architecture of AES128 circuit.

block is organized in a four by four-byte matrix, column by column. The matrix will be added to a sub-key using the *Xor* operation and the result obtained will be transmitted as input for the first round. At each round, the following operations are performed:

- The ***subBytes*** function is performed using S-boxes. These boxes in AES are based on a mathematical model, which is the modular arithmetic using polynomials.
- The ***shiftRows*** function is a simple-circular-left shift of bytes: the first row doesn't change; the second is shifted by one position; the third is shifted by two positions; and the last one, is shifted by three positions.
- The ***mixColumns*** function consists on taking each column of the matrix and multiplying it by a fix matrix, then reducing the answers modulo the polynomial $x^8 + x^4 + x^3 + x^1 + x^0$.
- The ***addKey*** function adds the round sub-key to the block. The sub-key used here is calculated by the *keyExpansion* function.

The application of these four operations takes place at each round except for the last one, where the *mixColumns* function is not applied. The ***KeyExpansion*** is a basic operation in the encryption process; it uses the cipher key to produce sub-keys in the same number of rounds. The first sub-key is just the original cipher key; then, to get the next sub-key, the previous one is passed through a function that involves a rotation P-box, a set of identical S-boxes, and addition modulo 2 to a round-constant.

3.1 Behavioural specification of the AES circuit

The function ***aesSpec*** represents the behavioural description of AES; it takes one input (*m*: plain text, *k*: cipher key); and one output (*c*: cipher text).

```
$aesSpec\,(m,k)\ =\ c$
  $where$
      $subKeys = keyExpSpec\ 1\ \ k$
      $n = addKeySpec\,(m,(subkeys\ !!\ 0))$
      $c = roundsSpec\,1\ (n, subKeys)$
```

The function *roundsSpec* is recursive and defined in the following way:

```
$roundsSpec\,n\ (m1, subKeys) = m2$
  $where$
    $n1 = subBytesSpec\,m1$
    $n2 = shiftRowsSpec\,n1$
    $n3 = mixColumnsSpec\,n2$
    $n4 = addKeySpec\,(n3,(subKeys\ !!\ (n-1)))$
    $m2 = roundsSpec\,(n+1)\ (n4, subKeys)$
```

```
$roundsSpec\ \ 10\,(m9, subKeys)\ = m10$
  $where$
    $n1 = subBytesSpec\,m9$
    $n2 = shiftRowsSpec\,n1$
    $m10 = addKeySpec\,(n2,(subKeys\ !!\ 9))$
```

The *keyExpSpec* function is recursive as well, and is defined as follows:

```
$keyExpSpec\,n\,ki\ = kj : (keyExpSpec\,(n+1)\ kj)$
  $where$
    $w0 = wXorSpec\,(sBox(\ shiftSpec\,(ki!!3)),$
    $ki!!0,\ rconst!!(n-1)$
    $w1 = wXorSpec\,(ki!!1\ \ , w0)$
    $w2 = wXorSpec\,(ki!!2\ \ , w1)$
    $w3 = wXorSpec\,(ki!!3\ \ , w2)$
    $kj = [w0, w1, w2, w3]$
```

```
$keyExpSpec\,10\,k9\ = k10$ : [ ]
    $where$
      $w0 = wXorSpec\,(sBox(\ shiftSpec\,(k9!!3)),$
      $k9!!0,\ rconst!!9$
      $w1 = wXorSpec\,(k9!!1\ \ , w0)$
      $w2 = wXorSpec\,(k9!!2\ \ , w1)$
      $w3 = wXorSpec\,(k9!!3\ \ , w2)$
      $k10 = [w0, w1, w2, w3]$
```

3.2 Structural implementation of the AES circuit

3.2.1 AES sequential architecture

The function ***aesImp1*** represents the structural description of AES sequential architecture. It takes *(m,k)* as input, and outputs c. The definition of this function is quite similar to ***aesSpec***, with the difference in their signature, and their internal functions, because they work with two different types. The *aesSpec* function works with data as hexadecimal, which are taken in Lava as *Signal Int*. However, *aesImp1* works with bits, which are represented by the type *Signal Bool*.

$aesSpec :: ([Signal\ Int], [Signal\ Int]) \rightarrow [Signal\ Int]$

$aesImp1 :: ([Signal\ Bool], [Signal\ Bool]) \rightarrow [Signal\ Bool]$

The function *aesImp1* is defined as it follows:

```
$aesImp1\,(m,k) = c$
    $where$
      $subKeys = keyExpImp1\ \ 1\ \ k$
      $n = addKeyImp1\,(m,(subkeys\ !!\ 0))$
      $c = roundsImp1\ \ 1\ \ (n, subKeys)$
```

The functions that *aesImp1* calls represent the inner components of the AES circuit; they differ from their corresponding functions in *aesSpec*, even though they use the same hierarchical way of description.

3.2.2 AES pipelined architecture

The function ***aesImp2*** represents the structural description of a pipelined architecture of AES. Its definition is different than ***aesImp1***; to allow the hardware parallelism, we need multiple functional components, one for each round; instead of using the same one for all of them; this will allow us to encrypt multiple blocks at the same time. For instance, when the first data block is on the second round of encryption, another block can start its first round; therefore when the first block is at its last round, nine other blocks can be calculated simultaneously. *aesImp2* has the same signature as *aesImp1* and it is defined in the following way:

$aesImp2\ ::\ ([Signal\ Bool], [Signal\ Bool]) \rightarrow [Signal\ Bool]$

```
$aesImp2\,(m, k) = c$
  $where$
      $s0 = addKeyImp2\,(m,\ k)$
      $k1 = getK\ 1\ k$
      $s1 = roundImp2\ (s0,\ k1)$
      $k2 = getK\ 2\ k1$
      $s2 = roundImp2\ (s1,\ k2)$
      $k3 = getK\ 3\ k2$
      $s3 = roundImp2\ (s2,\ k3)$
      $k4 = getK\ 4\ k3$
      $s4 = roundImp2\ (s3,\ k4)$
      $k5 = getK\ 5\ k4$
      $s5 = roundImp2\ (s4,\ k5)$
      $k6 = getK\ 6\ k5$
      $s6 = roundImp2\ (s5,\ k6)$
      $k7 = getK\ 7\ k6$
      $s7 = roundImp2\ (s6,\ k7)$
      $k8 = getK\ 8\ k7$
      $s8 = roundImp2\ (s7,\ k8)$
      $k9 = getK\ 9\ k8$
      $s9 = roundImp2\ (s8,\ k9)$
      $k10 = getK\ 10\ k9$
      $s10 = round10Imp2\ (s9,\ k10)$
      $c = s10$
```

The functions *roundImp2* is recursive as well, and it is defined in the following way:

```
$roundImp2\ (s,\ k) = nextS$
  $where$
    $s1 = subBytesImp2\ s$
    $s2 = shiftRowsImp2\ s1$
    $s3 = mixColumnsImp2\ s2$
    $nextS = addKeyImp2\,(s3, k)$
```

```
$round10Imp2\ (s9,\ k10) = s10$
  $where$
    $s1 = subBytesImp2\ s9$
    $s2 = shiftRowsImp2\ s1$
    $s10 = addKeyImp2\,(s2, k10)$
```

The function *getK* is the corresponding of *keyExpImp1* on the sequential implementation, it calculates only one key, and it is defined in the following way:

```
$getK\ \ n\ \ ki\ = kj$
  $where$
    $w0 = wXorImp2\,(sBox(\ shiftImp2$
    $(ki\ !!\ 3)),\ ki\ !!\ 0,\ rconst\ !!\ (n-1)$
    $w1 = wXorImp2\,(ki\ !!\ 1\ \ , w0)$
    $w2 = wXorImp2\,(ki\ !!\ 2\ \ , w1)$
    $w3 = wXorImp2\,(ki\ !!3\ \ , w2)$
    $kj = [w0, w1, w2, w3]$
```

3.3 Formal verification of the AES circuit

To verify the correctness of the AES pipelined circuit, we need to prove theorem (4) that expresses the equivalence between its behavioural specification and sequential implementation; and then theorem (5) that expresses the equivalence between the sequential implementation and the pipelined one.

$\forall\, m,\ k,\ m \in [SignalBool],\ k \in [SignalBool],$

$$aesSpec\,(abs1\,(m,k))\ =\ abs2\,(\,aesImp1\,(m,k)) \quad (4)$$

$\forall\, m,\ k,\ m \in [SignalBool],\ k \in [SignalBool],$

$$aesImp1\,(m,k)\ =\ aesImp2\,(m,k) \quad (5)$$

The equivalence properties are described by functions with one input *(m,k)* of type *Signal Bool*, which will be passed to both descriptions, in order to verify that they always produce the same output. As a result, an abstraction function called *fromSbToSi* is introduced, it converts from the type *Signal Bool* to *Signal Int*. Both functions *abs1* and *abs2* are defined using *fromSbToSi*.

$$\begin{aligned}
&propertyEquivSeqAES\ \ in = ok\\
&\quad where\\
&\qquad out1 = aesSpec\ \ (abs1\ \ in)\\
&\qquad out2 = abs2\ \ (aesImp1\ \ in)\\
&\qquad ok = out1 <==> out2
\end{aligned}$$

$$\begin{aligned}
&propertyEquivPipAES\ \ in = ok\\
&\quad where\\
&\qquad out1 = aesImp1\ \ in\\
&\qquad out2 = aesImp2\ \ in\\
&\qquad ok = out1 <==> out2
\end{aligned}$$

Since we use the infinite structure *list*, we also need to define equivalence properties that are explicit about the size of inputs:

$$\begin{aligned}
&propEquivSeqAES\_forSize\ \ n =\\
&\quad forAll\ (list\ n)\ \$ \setminus m\ \rightarrow\\
&\qquad forAll\ (list\ n)\ \$ \setminus k\ \rightarrow\\
&\qquad\quad propertyEquivSeqAES\ (m, k)\\
&propEquivPipAES\_forSize\ \ n =\\
&\quad forAll\ (list\ n)\ \$ \setminus m\ \rightarrow\\
&\qquad forAll\ (list\ n)\ \$ \setminus k\ \rightarrow\\
&\qquad\quad propertyEquivPipAES\ (m, k)
\end{aligned}$$

To verify the AES pipelined circuit, we call the *satzoo* function for both implementations:

$$\begin{aligned}
&verificationSeqAES\ =\\
&\quad satzoo\ \ (propEquivSeqAES\_forSize\ \ 128)\\
&verificationPipAES\ =\\
&\quad satzoo\ \ (propEquivPipAES\_forSize\ \ 128)
\end{aligned}$$

The *satzoo* function generates an output of the type *IO proofResult*. The execution of this function outputs the value **Valid** for both properties, which means that the sequential implementation ***aesImp1*** gives the same output as the behavioural specification ***aesSpec***, and the pipelined implementation ***aesImp2*** gives the same output as the sequential one ***aesImp1***, for every possible combination of plain test and key of 128 bits size. Thus, we conclude that the pipelined implementation ***aesImp2*** is equivalent to its behavioural specification ***aesSpec***.

A comparison between the proposed approach and all the similar works mentioned here is summarized in Table .1. As we can see the majority of the previous works are based on using imperative HDLs [1],[2],[3],[4],[5],[17],[19],[20]. Unlike functional HDLs, the imperative ones used in these approaches do not permit abstract descriptions of the high levels of design, which means that their descriptions are more detailed and therefore require longer code-lines; which makes them of higher complexity. Besides, finding errors and correcting them becomes a harder and more tedious process compared to functional HDLs descriptions. Although there is no actual comparison of the different approaches' code-lines (due to the lack of data), we can say with no hesitation that the functional HDLs provide more concise descriptions than the imperative ones for the same circuit. [2],[3],[4] use the simulation technique for the functional verification, which is not sufficient for critical systems such as the cryptographic circuits, even if they provide fast and automatic verification. As we can see, it is still possible to use formal verification with imperative HDLs, but as we established since they lack formal semantics, this makes the verification process very hard, as it requires translation of descriptions into a formal logic to be able to reason about them. [2] used deductive methods, which means that the verification process was not automatic. In the proposed approach we were able to verify the AES sequential data-path automatically in 1.18s, and the AES pipelined one in 1.05s, which makes the total 2.23s, which is faster than all of the other works that we know of their verification time [5], [15],[16],[17],[19].

Work	**Approach**	**Method**	**Circuit**	**Time (s)**
[1]	Imperative	Formal methods	SHA1	/
[2]	Imperative	Simulation	TDES	/
[3]	Imperative	Simulation	AES	/
[4]	Imperative	Simulation	Kasumi	180
[5]	Imperative	Formal methods	DES	59
[15]	Algebraic	Formal methods	AES	800
[16]	Algebraic	Formal methods	AES	844
[17]	Imperative	Formal methods	SHA AES	2100
[18]	Functional	Formal methods	AES	/
[19]	Imperative	Formal methods	Kasumi	180
[20]	Imperative	Formal methods	AES	/
Ours	Functional	Formal methods	AES	2.23

Table 1: Comparative table to the similar works' verification methods and time.

Since the behavioural specification and both structural implementations are specified by functions, they are executable; therefore, we can simulate them and examine the results. This is interesting because it allows us to verify that not only they are equivalents to each other, but that in fact, they give the expected results. All three functions (*aesSpec*,*aesImp1*,*aesImp2*) were simulated and they give the expected output of encryption.

To verify the other versions of AES, with the appropriate changes in the behavioural and structural descriptions, the equivalence properties need to be explicit about two different sizes.

$$\begin{aligned}
&propEquivSeqAES\_forSizes\ \ n\ \ l = \\
&\quad forAll\ (list\, n)\ \$ \setminus m\ \rightarrow \\
&\qquad forAll\ (list\, l)\ \$ \setminus k\ \rightarrow \\
&\qquad\quad propertyEquivSeqAES\ (m, k) \\
&propEquivPipAES\_forSizes\ \ n\ \ l = \\
&\quad forAll\ (list\, n)\ \$ \setminus m\ \rightarrow \\
&\qquad forAll\ (list\, l)\ \$ \setminus k\ \rightarrow \\
&\qquad\quad propertyEquivPipAES\ (m, k)
\end{aligned}$$

For instance, to verify the AES circuit of 192-bit key size, we call the *satzoo* function in the following way:

$$\begin{aligned}
&verificationSeqAES192\ = \\
&\quad satzoo\ (propEquivSeqAES\_forSizes\ \ 128\ \ 192) \\
&verificationPipAES192\ = \\
&\quad satzoo\ (propEquivPipAES\_forSizes\ \ 128\ \ 192)
\end{aligned}$$

When automatic verification is not possible, we can use the bottom-up proof method proposed in [27], where they started by verifying the functions (components) at the lowest level of the hierarchy, once they are verified, they replaced the implementation with its equivalent specification at the upper level, and so on until verifying the whole circuit.

4 Conclusion

In this paper, we presented an automatic formal verification methodology for the pipelined cryptographic circuits. It is the first application of the functional HDLs for the design and verification of such complex circuits. The proposed approach was demonstrated and applied to the 128-bit AES pipelined circuit. As prospects, we aim to verify the super-scalar designs as well, on which the adopted scalable methodology should be able to prove.

References

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

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

[3] Ali, Imran, Gulistan Raja, and Ahmad Khalil Khan. (2014) "A 16-Bit Architecture of Advanced Encryption Standard for Embedded Applications." *12th International Conference on Frontiers of Information Technology.* IEEE, Pakistan, pp 220-225. https://www.doi.org/10.1109/FIT.2014.49

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

[5] Clarke, E., Kroening, D. (2003) "Hardware verification using ANSI-C programs as a reference". *The ASP-DAC Asia and South Pacific Design Automation Conference*, IEEE, Japan, pp. 308-311. https://www.doi.org/10.1109/ASPDAC.2003.1195033

[6] Camilleri, A., Gordon, M., Melham, T. (1986). *Hardware verification using higher-order logic* University of Cambridge, Computer Laboratory.No. UCAM-CL-TR-91.

[7] Damaj, I. W. (2007). *Parallel algorithms development for programmable devices with application from cryptography*. International Journal of Parallel Programming, 35(6), 529-572. https://doi.org/10.1007/s10766-007-0046-1

[8] Salah, M. (2008) *Méthodologie de Vérification Formelle Pour les Microarchitectures RISC: Approche Fonctionnelle* (Doctoral dissertation).

[9] Walker, R. A., Camposano, R. (2012). *A survey of high-level synthesis systems*. Springer Science and Business Media. Vol. 135.

[10] Seger, C. J. (1992). *An introduction to formal hardware verification*. University of British Columbia, Department of Computer Science.

[11] Salem, A. M. E. F. (1992). *Vérification formelle des circuits digitaux décrits en VHDL* (Doctoral dissertation, Université Joseph-Fourier-Grenoble I).

[12] Guo, X., Dutta, R. G., Jin, Y., Farahmandi, F., Mishra, P. (2015). Pre-silicon security verification and validation: A formal perspective. *In Proceedings of the 52nd Annual Design Automation Conference* ACM. United States. (pp. 1-6). https://doi.org/10.1145/2744769.2747939

[13] Araiza-Illan, D., Eder, K. Richards, A. (2014). Formal verification of control systems' properties with theorem proving. *UKACC International Conference on Control (CONTROL)*. United Kingdom. IEEE. (pp. 244-249). https://doi.org/10.1109/CONTROL.2014.6915147

[14] Singh, K. P., Parmar, S. (2015). *Design of high performance MIPS cryptography processor based on T-DES algorithm*. arXiv preprint arXiv:1503.03166.

[15] Homma, Naofumi, Kazuya Saito, and Takafumi Aoki. (2011) "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. https://doi.org/10.1109/TIFS.2011.2157687

[16] Homma, Naofumi, Kazuya Saito, and Takafumi Aoki. (2013) "Toward formal design of practical cryptographic hardware based on Galois field arithmetic." *IEEE Transactions on Computers.* vol. 63, no 10, p. 2604-2613. https://doi.org/10.1109/TC.2013.131

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

[18] Lewis, Jeff. (2007) "Cryptol: specification, implementation and verification of high-grade cryptographic applications." *The 2007 ACM workshop on Formal methods in security engineering.*, ACM, United States, p. 41-41. https://doi.org/10.1145/1314436.1314442

[19] Lam, Chiu Hong, and Mark D. Aagaard. (2007) "Formal Verification of a Pipelined Cryptographic Circuit Using Equivalence Checking and Completion Functions." *2007 Canadian Conference on Electrical and Computer Engineering.* IEEE, Canada, p. 1401-1404. https://doi.org/10.1109/CCECE.2007.352

[20] Kim, Ho Won, and Sunggu Lee. (2004) "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. https://doi.org/10.1109/TCE.2004.1277865

[21] Wolfs, Davy, et al. (2011) "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.

[22] Chen, Gang. (2012) "A short historical survey of functional hardware languages." *ISRN Electronics* vol 2012. https://doi.org/10.5402/2012/271836

[23] Bjesse, Per, et al. (1998) "Lava: hardware design in Haskell." *ACM SIGPLAN Notices* vol. 34, no 1, p. 174-184. https://doi.org/10.1145/291251.289440

[24] Guo, Xiaolong, et al. (2015) "Pre-silicon security verification and validation: A formal perspective." *The 52nd Annual Design Automation Conference.*, Association for Computing Machinery United states, p. 1-6. https://doi.org/10.1145/2744769.2747939

[25] Claessen, Koen, and Mary Sheeran. (2007) *A slightly revised tutorial on lava: A hardware description and verification system.*

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

[27] Abir, Bitat., and merniz. Salah. (2018) "Towards formal verification of cryptographic circuits: A functional approach." *The 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS).* IEEE, Algeria, p. 1-6. https://doi.org/10.1109/PAIS.2018.8598527