Extreme Pipelining Towards the Best Area-performance Trade-off in Hardware

Stjepan Picek[1], Dominik Sisejkovic[2], Domagoj Jakobovic[2], Lejla Batina[3], Bohan Yang[1], Danilo Sijacic[1], and Nele Mentens[1]

[1]KU Leuven ESAT/COSIC and iMinds, Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
[2] Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia
[3]Digital Security Group, ICIS, Radboud University Nijmegen, The Netherlands

Abstract. This paper presents a novel framework for the automatic pipelining of AES S-boxes using composite field representations. The framework is capable of finding positions to insert flip-flops in an almost optimal way, resulting in S-boxes with an almost optimal critical path. Our novel method is using memetic algorithms and is shown to be fast, reliable and successful. We demonstrate our framework for composite field S-boxes using a polynomial and a normal basis, respectively. Our results prove that this method should be consulted when an optimal solution is of interest. Besides experimental results with the new memetic algorithms, we also discuss the ideal model of a circuit, which can be used when assessing the quality of the obtained solutions. We emphasize that this method can be used for any circuit of interest and not only for AES S-boxes.

Keywords: Real-time cryptography, Pipelining, AES S-box, Optimization, Memetic algorithm.

1 Introduction

Implementations of cryptographic primitives present constant challenges in today's security applications. On the one side, embedded security relies on multiple trade-offs in terms of constraints on area, timing, power and energy and at the same time requires implementations to be secure against side-channel adversaries. On the other side, various high-speed implementations in high-bandwidth servers aim at ever faster versions of algorithms without a substantial increase in resources.

Considering block ciphers like AES that are commonly used for bulk encryption applications, a clear preference is often given to the counter mode of operation as it is parallelizable and hence suitable for high throughput, which is required by applications such as VPN setup, IPSec, etc. It may appear that pipelining and parallelism are the terms that do not go well with constrained platforms but it is less certain where one should draw the line defining embedded security devices. For example, ARM has recently announced its next generation ARM Cortex-A72 processor to be used for mobile phones that is based on the 64-bit ARM v8-A architecture. ARM claims that the new chip delivers as much as 50 times the performance compared to processors from just five years ago and that it is at the same time 75% more energy efficient than the previous generation.

The situation is even more unclear with hardware modules. Basically, applications that require hardware implementations such as RFID tags and smart cards are often developed for unique purposes and tailored towards a specific scenario. It may be the case that high speed is of utmost importance even though the application is embedded. It is fair to say that techniques that boost the performance in hardware such as pipelining and parallelism remain important for efficient implementations.

In this work we focus on pipelining and more precisely, we look for the optimal way to put registers (flip-flops) such that we reduce the critical path substantially. Naturally, at the same moment we do not want to pay for it too much with area or power overhead. Our goal is to develop a novel framework that could be useful for hardware designers and in general, implementers. To this end, we use memetic algorithms as a known approach in the Evolutionary Computation (EC) area. We demonstrate our approach on composite field S-boxes, because they result in circuits with a high number of gates and a high number of unbalanced paths. We elaborate on our ideas and contributions in the remainder of this paper.

Motivation and Contributions

The goal of this work is to derive a framework that is applicable to real-world scenarios. The authors of [1] give a proof of concept where they succeed in pipelining an AES S-box with an improved throughput as a result. However, to come up with a generic and at the same time optimal strategy, significant improvements in the choice of algorithm and the optimization function are necessary. Therefore, the main difference with our work is that we use more powerful search algorithm as well as improved evaluation mechanisms. Although maybe at a first glance those differences do not seem important, they are crucial in the transition from a proof of concept work that was not able to produce optimal results, to our framework that produces much better results in a smaller amount of time.

More specifically, our main contributions are:

1. Development of a new optimization algorithm that is able to produce correct solutions with a high certainty. Since we use heuristics, we cannot guarantee that all obtained solutions will be correct. Nevertheless, the experimental results in this paper did not yield any incorrect solution.
2. Improvement of the evaluation process that enables one to obtain results relatively fast. The evaluation process consists of testing whether all paths have the same number of flip-flops.

3. Extensive tests showing the suitability of our approach.
4. Building a whole system that accepts as an input the netlist and outputs a ready to be simulated netlist with inserted flip-flops.
5. Pipelined S-boxes with an optimized critical path compared to related work.

Besides those main contributions, we have a few more things to report on. Firstly, we have conducted all necessary experiments with several optimization techniques to find the best one. Furthermore, we have developed a tool that enables us to test a circuit in order to a priori determine what kind of results are expected. For this purpose we experimented with several different representations of the problem, in order to find the optimal one. Next, we present a framework that is capable to decompose a network (i.e. a circuit) into several subnetworks. Finally, we introduce the notion of the Ideal Circuit Model that helps us to evaluate the quality of our solutions. We give more details on all the aspects of this work below.

The remainder of this paper is organized as follows. In Section 2, we give the necessary information about AES and the methods for implementing S-boxes in hardware. Furthermore, we give the basic circuit terminology that we follow in this work. We continue in Section 3, where we present related work from the cryptographic, the design automation and the evolutionary computation perspective. In Section 4, we give an extensive description of our framework. To justify the model we use, we also present several other options with their advantages and drawbacks. Here, we also present the Ideal Circuit Model, an abstraction that helps us to assess the quality of the obtained solutions. Section 5 gives results of our EC experiments as well as the results of the synthesis process. Furthermore, we give a short discussion on the relevance of those results as well as some guidelines for future work. Finally, in Section 6 we conclude this study.

2 Preliminaries

In this section, we give the necessary background information for following this work. First, we define the network related terminology we use and then we shortly describe the AES cipher.

2.1 Circuit Terminology

Retiming represents a technique that transforms the circuit by moving registers from one location in the circuit to another in such a way that the functional behavior of the circuit as a whole is preserved [2]. Retiming can optimize several objectives [3]:

– minperiod - minimizes the clock frequency of a circuit,
– minarea - minimizes the number of registers in a circuit, and
– constrained minarea - minimizes the number of registers in a circuit subject to a maximum constraint on the clock period.

Pipelining is a system design technique that increases the performance of a system by partitioning a complex combinatorial circuit into a number of circuits. The pipelined circuit has a reduced critical path and could be operated on a higher working frequency [4]. Pipelining can be regarded as a special case of a minperiod objective [2].

A piece of hardware that implements the functionality of a Boolean function is called a logic gate. A circuit (network) is a set of interconnected logic gates. Networks are commonly described using netlists, which contain information about the types of logic gates employed, as well as their interconnections. Therefore, within a netlist logic gates can be perceived as network nodes.

When an output of a logic gate A, contained within a circuit, is connected to an input of a logic gate B we say that the gate A drives (the input of) gate B. Inputs of a circuit are inputs of logic gates within the circuit that are not driven by any of the logic gates of the circuit. Outputs of a circuit are outputs of logic gates that do not drive any of the gates of the circuit.

A path is a unique combination of nodes connecting a single input to a single output. Each node in the path (logic gate) introduces a delay corresponding to the time required for the signal to propagate from node inputs to node outputs. The number of paths denotes the number of different possible paths through the circuit from an input to an output. The delay of a path is equal to the sum of the delays of all its nodes. The path with the largest delay is called the critical path; for it limits the rate at which circuit-inputs may be changed (system clock frequency).

2.2 Standard Cells and Delays

A standard-cell design approach is based on using pre-made logic gates – called cells – that implement a variety of combinatorial and sequential functions. For further information about standard cells and delays, we refer the readers to [5].

In an effort to have results that are possible to compare with those from previous work, we use the same standard cell library, namely the UMC 0.13 μm low-leakage (LL) standard cell library [6]. For versatility we provide results for all operating conditions of this library, as well results for the UMC 0.13 μm high-speed (HS) standard cell library.

2.3 AES Cipher

As already stated, the target for the pipelining in this work is the S-box as used in the AES cipher. Furthermore, we experiment with both polynomial and normal basis. In accordance with that, here we give the necessary details about the AES cipher, and various ways of implementing the S-box. The AES cipher is a symmetric 128-bit block cipher [7]. To obtain a ciphertext, the plaintext needs to pass a number of round transformations. The number of rounds depends on the length of the key and is 10 rounds for a 128-bit key, 12 rounds for a 192-bit key and 14 rounds for a 256-bit key. Each round has a unique key that is calculated from the initial key. The operations in the AES cipher are on a

4×4 byte array, called the state. Those operations are *AddRoundKey*, *SubBytes*, *ShiftRows* and *MixColumns*. All the rounds consist of the same set of operations, except that there is an additional *AddRoundKey* operation before the first round, and the last round does not have the *MixColumns* operation. With regards to the whole AES cipher, there are a variety of objectives for the implementation. As a result, there exist approaches that e.g. maximize the throughput [8], minimize the circuitry [9] or minimize power consumption [10].

The nonlinear layer (*SubBytes* operation) of the AES cipher is the substitution layer realized with S-boxes. This *SubBytes* operation replaces each byte of the input and involves an inversion in the Galois field $GF(2^8)$. This calculation is not easy and therefore there are several approaches to this problem. In the rest of this paper, we only consider AES that has 10 rounds in order to simplify the considerations. Since the S-box is a 8-input 8-output lookup table (LUT), a memory to store the S-box would have a size of 256 bytes. To reduce the circuitry, Rijmen suggested to calculate the inverse of the Galois field by using subfield arithmetic [11]. This idea was further extended by work of Satoh et al. who suggested to use the tower field approach [9]. Works of Canright [12] and Mentens et al. [13] showed that the most compact solutions rely on composite field arithmetic. For details about the polynomial and normal bases, we refer the readers to [12, 13]. For details about tower fields, we refer the readers to [14].

3 Related Work

In the next section, we briefly summarize several important works on hardware implementations and design automation as well as on evolutionary computation techniques for applications in cryptology. First, we list seminal works dealing with the retiming problem. Leiserson and Saxe presented the retiming technique that is able to minimize the area or maximize the clock frequency without changing the functionality of a circuit as a whole [2]. Furthermore, they showed that the problem of determining a retimed circuit with a minimum number of registers is solvable in polynomial time. Maheshwari and Sapatnekar presented an approach for the minarea retiming problem that is able to handle large circuits [15]. Narendra and Rudell discuss implementation issues arising when implementing retiming algorithms and they give a number of experimental results for circuits of various sizes [3]. Münzner and Hemme presented an algorithm that converts combinational circuits into pipelined data paths where the first step is to use timing requirements to find parts of the circuit where the register placement is possible. The second step utilizes a modified maxperiod algorithm to position a minimal number of registers [16]. However, we note that this algorithm does not guarantee to find the global optimum of flip-flops. For more information about retiming, we refer the readers to [17].

Next, we present related works that concentrate on cryptographic hardware implementations where the design choice is similar to ours. The focus is on implementations that use composite field arithmetic to boost compactness or speed.

Satoh et al. were the first to take advantage of the composite field $GF(((2^2)^2)^2)$ for low area implementations, which results in the most compact S-box at the time with a gate complexity of 5.4 kgates [9]. This paper has triggered many related works looking into one or the other tower field approach.

Similarly, Wolkerstorfer et al. use arithmetic in $GF((2^4)^2)$ to achieve an implementation with a gate count comparable to the one presented by Satoh et al. (5.7 kgates) [18]. An additional goal was to make the best out of reusing hardware area for both encryption and decryption. Mentens et al. experiment with the choice of polynomials and representations to optimize the S-box on compactness for polynomial bases [13]. The main result proves that one can make better choices with different irreducible polynomials and representations of elements in this special type of tower field. Canright picked up on this work, applying the ideas to normal bases [12]. Systematically exploring all the possibilities, he deduced the smallest S-box at the time, a result that held up for almost a decade. Only recently Moradi et al. have published the most compact AES implementation of a size of only 2.4 kgates [19]. This result is obtained by focusing on AES encryption and squeezing the area on all the design layers.

Macchetti and Bertoni [20] describe an ASIC implementation for the same composite field $\mathbb{F}((2^4)^2)$ as Wolkerstorfer et al., but looking into a different representation. We mention here just a handful of the most influential papers, but it is obvious that the plethora of implementation options of AES has contributed to a huge amount of results that vary from exploiting one or the other design alternative. Looking into high-speed implementations, Hodjat and Verbauwhede describe an ASIC implementation for the same composite field $GF((2^4)^2)$ as Wolkerstorfer [21]. Their approach was to perform an area-throughput trade-off by fully pipelining the architecture and also optimizing the key-schedule implementation. The same authors also consider a pipelined AES implementation on an FPGA [22]. In [23] and [24], Boyar and Peralta presented a technique to improve the implementation of the AES S-Box. Their result provides different tradeoffs between the implementation area and the logic depth.

From the Evolutionary Computation perspective, we can find a number of papers that explore various applications that could be of interest in cryptology, the most prominent ones being the evolution of Boolean functions and S-boxes [25–27]. However, here we list only a few works that have clear connections with the problem we describe. Yagain and Vijayakrishna present a framework for the retiming problem when considering DSP blocks [28]. They experiment with the multi-objective genetic algorithm and report as a main advantage of their approach a number of viable solutions instead of one.

Batina et al. conduct the first experiments in trying to evolve the AES S-box in the form of a combinatorial circuit with the goal of increased throughput [1]. We point to this paper as a proof of concept, which is also our starting point and we present a complete novel framework that can be used in real-world security systems. However, we note that the results presented in that paper are worse than even those obtained by manually inserting flip-flops in the design phase as given in Section 5.1.

4 The Optimization Framework

In this section, we start by defining an ideal circuit that can be pipelined into circuits of the same size. In an ideal circuit, each part of the circuit in between the pipelining stages has an equal delay. Furthermore, it is always possible to divide the longest path of such a circuit into $n+1$ partitions of exactly the same size where n is the number of flip-flop layers one adds.

As an example, consider a circuit that has a critical path equal to 1 000 ns. After inserting one layer of flip-flops on all necessary positions, the critical path would equal 500 ns (we disregard the delay of flip-flops). Such ideal model can help us when evaluating the quality of obtained solutions and guide us towards the best possible (optimal) solution. Naturally, it is hard to expect that a realistic circuit can be divided so perfectly. Therefore, we aim that the best possible solution should be as close to the ideal solution as possible. Next, we define the maximal number of flip-flops that can be added to a circuit.

Definition 1 *The upper bound of the number of flip-flop layers is equal to the number of cells that can be added to the shortest path connecting the input to the output of the circuit.*

4.1 The Choice of the Optimization Procedure

Similarly to the approach from [1], we regard this as an optimization problem: pipelining of a combinatorial circuit in a way that minimizes the critical path of a circuit while retaining its correctness, can be viewed as an optimization problem.

To be able to run the optimization, we introduce the notion of a correct solution.

Definition 2 *A correct solution is represented by any circuit with flip-flops in which there is the same number of flip-flops on every path connecting any input to any output.*

It is obvious that, to be able to pipeline the signal, there has to be at least one flip-flop on each path; but for the solution to be correct, that number must be the same for each path.

Since we established that we regard pipelining as an optimization problem, next we discuss which algorithm to use. We regard this problem as a black box scenario and therefore we assume no specific knowledge about the circuit. If we start with an initial circuit that has no flip-flops and then randomly add a certain number of flip-flops, it would be highly unrealistic to expect correct solutions. Therefore, we decide to use heuristics. Heuristics are algorithms that find good solutions on a large-size problem instance. Alternatively, heuristics can be defined as parts of an optimization algorithm. There, heuristics use the information currently gathered by the algorithm to help decide which solution candidate should be tested next or how the next solution can be produced [29]. Heuristic algorithms can be divided into specific heuristics and metaheuristics. Specific heuristics are methods that are tailor-made to solve a specific problem and therefore not appropriate here (since we are not aware of any tailor-made heuristic algorithm for this problem). Metaheuristics are general-purpose algorithms that can be applied to solve almost any optimization problem. To classify metaheuristics, one can follow many criteria, but we divide it into single-solution based metaheuristics and population based heuristics [30]. Single-solution based metaheuristics manipulate and transform a single solution during the search as in the case of algorithms like local search or simulated annealing. In contrast, population based metaheuristics work on a population of solutions. On the basis of the aforementioned classification, we decide to use population based metaheuristics, and more precisely evolutionary algorithms (EAs).

We experiment with three different evolutionary algorithms, namely, Genetic Algorithms (GAs) [31], Evolution Strategy (ES) [32] and Genetic Annealing (GAn) [33]. First, in order to conduct the experiments we need to define the representation of the problem as well as the objective function. We use the same objective function as in [1] for an easier comparison of the results. The goal is the **minimization** of the following equation:

$$fitness = max\_delay\_time + (1,000 * number\_invalid\_paths). \qquad (1)$$

In the previous equation, the second term acts as a penalty for solutions that are not correct. In other words, we allow the incorrect (infeasible) solutions while searching the solution space, but guide the search towards correct solutions. Here we presume that the user specifies the target number of flip-flop layers $n >= 1$ to be inserted. Consequently, the number of invalid paths presents the number of paths that contain a different number of flip-flops. After a solution is obtained, we simulate it in the Synopsys tool as described in Section 5.4.

Next, we discuss how to encode the solution of the problem. We use the same representation as in [1] where for a position with no flip-flops, we write 0 and for a position with an inserted flip-flop, we write 1.

We developed a tool that translates a netlist into a bitstring representation that can be used in the optimization algorithm. The same tool returns the solution back into the netlist format after the flip-flops are inserted. The tool itself is written in the Java programming language, but the implementation details are of secondary importance so they are not presented here.

However, the question is what is a possible insertion position? The most general option is to allow an insertion of a flip-flop to every input of every cell in the circuit, which we denote as *input-based* encoding. Thus, a potential solution is represented as a string of bits with a length equal to the product of the number of cells and their inputs. This length may be denoted with S. Since each bit may be independently set to either one or zero, the size of the search space is 2^S. We have shown experimentally that in general only a very small fraction of this space represents correct solutions. Naturally, one can suggest to encode the solution in a way where each cell represents one possible insertion position. Therefore, in this kind of encoding we do not put flip-flops on each input of a cell, but on the output of a cell (*output-based* encoding). In this way we are able to reduce the solution length and size of the search space significantly. However, this also results in the fact that some correct solutions, which can be obtained with the first encoding, cannot be represented using the second one.

4.2 Genetic Algorithms

Genetic algorithms (GAs) are probabilistic algorithms whose search methods model some natural phenomena: genetic inheritance and survival of the fittest [30, 31]. GAs are a subclass of evolutionary algorithms where the elements of the search space are arrays of elementary types like strings of bits, integers, floating-point values and permutations [29, 31]. Usual variation operators in the GA are mutation and crossover [31]. In the context of optimization, exploration (diversification) means finding new points in previously unexplored areas of the search space, which is achieved by mutation in GAs. Exploitation (intensification) represents the process of improving and combining the traits of known solutions which is why crossover is used [29]. For an optimization algorithm to be successful, it needs to have a good balance between those two notions to avoid a too fast convergence to a local optimum from one side, but also a too long operation time from the other side. For further information about GAs, we refer to [31]. After the initial round of experiments, the results have shown that GAs outperform by far the ES and GAn algorithms. Therefore, in the rest of the paper we consider only GAs in our experiments.

4.3 Design of the Optimization Algorithm

As noted, in our experiments we use GAs in order to find suitable locations for the insertion of flip-flops. However, it is easy to notice that a GA on itself is often not enough. Recall our fitness function where we penalize each incorrect path. The smaller the number of incorrect paths, the better the solution. Consider the situation where a GA produces a solution that is not correct, but has only a small number of incorrect paths. Mutation will help to explore new search space areas, but in general will not help to correct a slightly incorrect solution. We noticed that often solutions are incorrect, but we need only a small change to make them correct. To amend this disadvantage of GAs, we add a local search (LS) algorithm that tries to correct almost-correct paths. Since now we combine GAs and local search, we deviate from evolutionary algorithms, and instead go to the evolutionary computation area. Such a combination of algorithms is called a memetic algorithm (MA). Memetic Algorithms (MAs) represent a synergy between evolutionary algorithms (or any other population-based algorithms) and local improvement algorithms [29]. Most MAs can be interpreted as search strategies in which a population of solutions cooperate and compete [34]. Next, we give the pseudocode for our optimization algorithm in Algorithms 1 to 4.

Algorithm 1 represents the main part of our framework and is a somewhat customized version of a genetic algorithm.

Algorithm 1 Greedy Hibrid SSGA.

```
P = createInitPopulation(POP_SIZE)
evaluate(P)
while not termination do
    if LS then
        (I1, I2) = getTwoBestFrom(P)
        for all individual from (I1, I2) do
            I = GreedyLocalSearch(individual)
            if fitness(I) better than fitness(bestOf(I1, I2)) then
                switch I with worst from P
            end if
        end for
    end if
    repeat
        randomly add $k$ individuals to the tournament
        select the worst one in tournament
        (R1, R2) = randomly select two parents from the remaining ones in the tournament
        D = randomCrx(R1, R2)
        evaluate(D)
        replace the worst in P with D
    until POP_SIZE times
end while
```

The LS algorithm presented in Algorithm 2 helps us to locate correct solutions that are close to those obtained by the GA.

Algorithm 2 Greedy Local Search.

```
Require: iteration = 0
  repeat
      N(I(iteration)) = Neighborhood(I);
      I(iteration + 1) = getBestOf(N(I(iteration)))
      LocalOp(I(iteration + 1))
      iteration = iteration + 1
  until MAX_ITER times
```

Next, the Neighborhood algorithm is used to generate a population of solutions that are within Hamming distance of the current solution. Here, by Hamming distance we mean the number of positions (flip-flops) that differ in the two solutions. The Neighborhood algorithm is given in Algorithm 3.

Algorithm 3 Neighborhood.

```
Require: iteration = 0
  while N_SIZE > iteration do
    create_individual at Hamming distance $d$ from individual
  end while
```

Finally, the LocalOp algorithm is used to compare the quality of solutions generated by the local greedy search algorithm and is presented in Algorithm 4.

Algorithm 4 LocalOp.

```
  for all bit postion i in bitsOf(I) do
    oldFitness = fitness(I);
    flip bit on position i in bitsOf(I);
    evaluate(I);
    if fitness(I) worse than oldFitness then
      flip bit on position i in bitsOf(I);
    end if
  end for
```

Common Parameters. To be able to assess the effectiveness of the optimization algorithm, and compare the alternatives, we need to define parameter values for each algorithm variant. Since the observed algorithms are stochastic, their performance must be evaluated on the basis of repeated runs; therefore, the number of independent runs for each setting in our experiments is 100. The other common parameters are selected on the basis of tuning experiments and include the population size, which is set to 50. The tournament size k in the tournament selection is equal to 3. The tournament selection works by randomly choosing 3 individuals and then removing the worst one. From the remaining two individuals, one new solution is created via the crossover operation. The mutation probability is set to 0.01 per individual where we make a choice on the basis of a small set of tuning experiments that showed this was the best result on average. Local search is called every fourth generation, with a maximum of 6 iterations for local search. The neighborhood size is 35 and the Hamming distance is 10. Furthermore, we display all common parameters in Table 1.

4.4 Circuit Decomposition

Here, we briefly discuss the additional functionality that our framework incorporates. It allows to decompose a network on several levels, i.e. subnetworks divided by flip-flops. Each of those subnetworks realize a part of the functionality of the complete network and it is possible to pipeline only a subnetwork. We call this

Table 1: Common parameters.

Parameter	Parameter Value
Number of runs	100
Tournament size	3
Population size	50
Stopping criterion	Stagnation in 10 generations
Mutation rate	0.01
LS rate	4
Max iteration for LS	6
Neighborhood size for LS	35
Hamming distance in LS	10

procedure network decomposition. However, it is important to state that it is not always possible to pipeline a subnetwork (or even a network). Therefore, with regards to Definition 1, we offer the following definition:

Definition 3 *It is possible to add flip-flops only to those subnetworks that do not contain cells with direct inputs to the network.*

5 Experimental Results

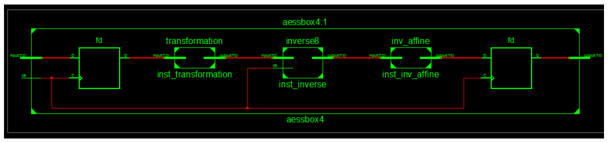
In this section, we first introduce the results obtained with two different methods where the focus is on those results obtained with the optimization algorithm. We perform static timing analysis (STA) on pre-layout netlists synthesized using Synopsys Design Compiler, which is used to report the area of the designs, while we use PrimeTime - a golden timing signoff solution and environment by Synopsys - to perform STA.

In all of our experiments we are using the smallest D-Flip-Flop (DFF) cells from the appropriate libraries (`DFFCLD` and `DFFCHD`) for driving the inputs. Furthermore, we use the load of these cells for all outputs. This models the placement of the combinatorial network between two registers. The same DFF cells are used for the pipeline registers.

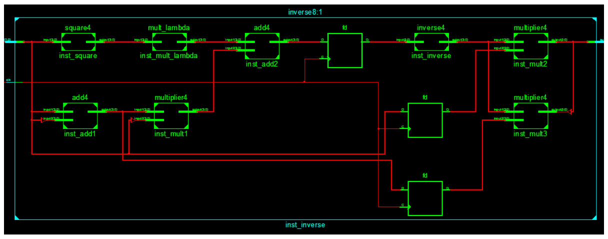
Lastly, in order to depict the impact of the proposed method on purely combinatorial networks, we do not include the setup times of sequential elements in the presented results.

5.1 Introducing Flip-Flops Manually in the Design Phase

We established that randomly setting flip-flops cannot result in a correct network when working with such complex networks as given here. However, what about inserting flip-flops in the design phase? In this way, we avoid working with netlists, but rather with an abstraction of a network that is much easier to comprehend. Furthermore, this approach is the dominant one when considering practical applications [4]. As an example, here we take the AES S-box in polynomial basis and then we insert flip-flops into the inverse8 part. This is represented in Figures 1a and 1b. Flip-flops are depicted as "fd" cells in the latter figure. We note that the tool itself changes the network when adding the cells in the design phase.



(a) Top view of AES S-box in polynomial basis.



(b) Zoom into inverse8 with added FFs.

Fig. 1: Example of inserting FFs in design phase.

5.2 Results for the Optimization Algorithm

In this section, we present the best results we obtained with our memetic algorithm. Alongside, we give basic statistics on the netlists without inserted flip-flops in Table 2. For an example of full statistics, we point the readers to Appendix 6. In order to ease the comparison, we calculated the delays for cells as in [1] where the values are obtained as averages for all possible combinations for each element. To model the delay of a flip-flop, we can use any value from the library as long as it is the same for the whole circuit, and here we work with a D-FF with a single output and no clear signal that has an average delay time of 320.35 ps.

Table 2: Statistics of the preliminary S-box design.

Basis	# of cells	# of inputs	# of paths	Critical path (ps)
Polynomial	165	432	8 023 409	3 884.52
Normal	181	497	139 221 044	4 685.724

In Table 3, we give the best obtained results for our algorithm. If written only Polynomial, it means that the flip-flops are inserted to the input of a cell, when flip-flops are added to the output of a cell we denote it with Polynomial, out.

Table 3: Best solutions.

Basis	Layers	Critical path (ps)	# of added FFs
Polynomial	2	2 065.7435	64
Polynomial, out	2	3 075.6087	11
Normal	2	2 508.8050	73

Finally, in Table 4, we give the percentage value of times that each correct solution reached a certain critical delay time.

Table 4: Obtained number of correct solutions (%).

Basis	Layers	1.5 - 2	2 - 2.5	2.5 - 3	3 - 3.5	3.5 - 4	4 - 4.5	4.5 - 5
Polynomial	2	-	13.04	53.26	32.6	1.08	-	-
Polynomial, out	2	-	-	-	80	20	-	-
Normal	2	-	-	10.52	-	5.26	36.84	47.37

5.3 The Performance of The Memetic Algorithm

After discussing the successfulness of our approach in the previous section, here we discuss its reliability and speed. As already stated, those objectives are what we believe to be the differentiation of a proof of concept from the real-world framework. For all results we use PCs with Intel i5-3470 CPU with 3.2 GHz, 6 Gb of RAM and 64-bit Windows 7 OS. To obtain the following statistics, we run every setup 100 times. We consider the algorithm successful if it generates at least one correct solution. The rationale behind this is supported by the fact that every stochastic optimization algorithm is meant to be run at least several times (in other words, it is meaningless to run a stochastic algorithm on a given problem only once).

When adding one level of flip-flops to the S-box in polynomial representation where flip-flops are positioned on the input and with 100 000 evaluations, we obtain a successfulness of 93%. When running the same setup, but with flip-flops positioned on the output of cells (output-based), the successfulness drops to only 36.8%. When working with S-boxes in normal basis with flip-flops based on the inputs of the cells, the successfulness equals 91.6%. To summarize the previous results, we can conclude that our algorithm is reliable since it has a reasonably high success rate. Next, we discuss the speed of our approach based on the speed of evaluation. Here, an evaluation is the whole process of obtaining a new individual and examining its fitness. Since it is clear that the evaluation process depends on the number of paths, it is easy to see that the evaluation of a solution in polynomial representation will be faster than the one in normal representation since it has a smaller number of paths as given in Table 2. A single evaluation of a polynomial representation solution lasts around 100 ms and of a normal representation solution around 120 ms. However, 10 evaluations last 800 ms, and 100 evaluations last 8 000 ms. We observe that more evaluations are comparably faster since in Java implementation we have a "warm up" phase due to the optimizations and JIT compilation. Finally, on average, our algorithm needs 150 generations to find a solution which amounts to 7 500 evaluations on average. When accounting for the speed of evaluation, we see that our approach needs on average 12.5 minutes to output a correct solution with an improved critical path.

5.4 Static Timing Analysis Results

The critical paths of the synthesized netlists are evaluated using the following design constraints. Firstly, for both libraries that are used we are using the `enG10k` wire load model. Secondly, we perform STA for all available operating conditions in order to take into account the available driving powers of combinatorial networks. Thirdly, we assume the combinatorial networks are driven by the smallest DFFs. Consistently, all outputs are loaded with the same DFFs. The setting used for STA is depicted in Figure 2.

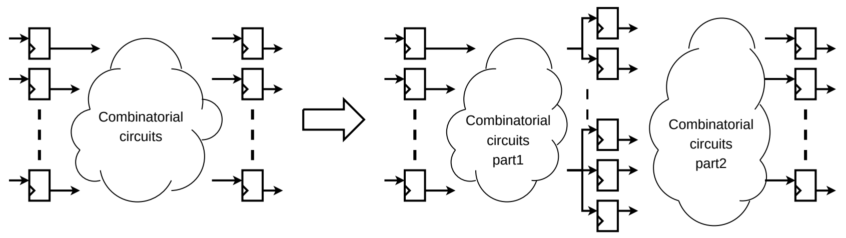


Fig. 2: STA setting.

In order to depict the impact of the proposed method on purely combinatorial networks, we do not include the setup times of the sequential elements in the presented results. Lastly, due to the fact that DFFs typically have a smaller load of input pins—while providing stronger drives—than the combinatorial elements used in the initial network, the sum of delays through both networks is smaller than the delay of the original network.

In Table 5 we give the critical paths of the networks for all observed cases. The last column gives information on the ratio between two critical paths, including the rising edge setup times of the DFFs used for the model.

Table 5: Critical paths of original and pipelined networks.

Basis	Library	Operating Conditions	C. Path (ns)	Pipeline C. Path (ns)	Ratio (%)	With setup (%)
Poly.	UMC0.13LL	BCCOM	5.91	2.87	48.89	49.92
Poly.	UMC0.13LL	TCCOM	9.83	4.78	49.03	49.45
Poly.	UMC0.13LL	WCCOM	17.09	8.36	48.84	49.39
Poly.	UMC0.13HS	BCCOM	2.35	1.19	50.64	53.78
Poly.	UMC0.13HS	TCCOM	3.64	1.85	50.82	52.89
Poly.	UMC0.13HS	WCCOM	6.30	3.20	50.79	52.01
Norm.	UMC0.13LL	BCCOM	6.28	3.07	49.61	50.16
Norm.	UMC0.13LL	TCCOM	10.36	5.08	49.38	49.81
Norm.	UMC0.13LL	WCCOM	18.10	8.84	48.92	49.29
Norm.	UMC0.13HS	BCCOM	2.58	1.28	49.61	52.55
Norm.	UMC0.13HS	TCCOM	4.01	1.98	49.38	51.32
Norm.	UMC0.13HS	WCCOM	6.98	3.45	49.43	50.56

5.5 Discussion and Future Work

The results presented in this work show that our methodology is capable of finding almost optimal positions for adding flip-flops. However, we must also ask the question if it is worth while? Although our framework is capable of generating good results relatively fast, this is still significantly slower than what is the case when adding flip-flops manually in the design phase. Therefore, the answer to the previous question depends on the setting. If we have a setting where we require a critical path that is as small as possible and where we can afford the cost of added flip-flops, this methodology represents a valuable resource. Otherwise, the total cost versus the benefit is much less favorable. Furthermore, as main advantage of our approach compared with the retiming technique is the possibility to divide the circuit in parts of almost the same critical path size and therefore obtaining an optimal solution. The same often cannot be said for the retiming technique due to the optimization towards a minimal number of registers.

We believe our approach can be coupled with the retiming technique to provide even better results (i.e. our critical path, but with a smaller number of registers). We emphasize that although we work here on S-boxes realized in tower fields, there is nothing stopping us to use this method with any other kind of combinatorial circuit. Naturally, the smaller the critical delay, the smaller the benefit of pipelining. In any case, pipelining has a big impact on the efficiency of certain modes of operation. For fully exploiting the power of the AES instructions, one needs a small delay in the mode of operation and that has the unfortunate side effect that the "better modes of operation" such as CBC are much less applied and one tends to do counter mode (fully parallelizable). In our future work we want to extend this research to the whole AES round. The results showed here suggest our technique should be regarded as a viable option when looking for optimal pipelining. However, the final verdict must be done only after a whole cipher round is examined. Besides that, we plan to further improve the local search part of the algorithm since its efficiency has an extreme impact on the efficiency of the whole algorithm. On top of that, one interesting research avenue would be to combine our algorithm with techniques for finding ASAP (as-soon-as-possible) and ALAP (as-late-as-possible) locations [16] for flip-flops which could result in a decrease of the search space size for our optimization algorithm. Finally, it is worth mentioning that the results presented in this paper are pre-layout results. We are aware that the outcome might change when post-layout results are used, as also mentioned in [35] and [36].

6 Conclusion

In this paper we present a framework that is able to pipeline combinatorial circuits. To show its performance, we experimented with the AES S-box realized with tower fields in both polynomial and normal representation. The obtained results show our approach is highly competitive when the goal is to minimize the critical path. Furthermore, our results can be regarded as the best possible since they divide the circuit into two equal parts. The method presented in [1], as well as the method dominantly used today (Section 5.1) give worse results. Naturally, the methodology used in this work can be used in other applications besides cryptography when the goal is to decrease the critical path as much as possible and where each nanosecond makes a difference.

Acknowledgments

This work has been supported in part by the Croatian Science Foundation under the project IP-2014-09-4882. In addition, this work was supported in part by the Research Council KU Leuven (C16/15/058) and IOF project EDA-DSE (HB/13/020). D. Sijacic is supported by the Marie Curie-Sklodowska research fellowship, within the ECRYPT-NET framework.

References

1. Batina, L., Jakobovic, D., Mentens, N., Picek, S., Piedra, A.D.L., Sisejkovic, D.: S-box Pipelining Using Genetic Algorithms for High-Throughput AES Implementations: How Fast Can We Go? In: Progress in Cryptology - INDOCRYPT 2014 - 15th International Conference on Cryptology in India, New Delhi, India, December 14-17, 2014, Proceedings. (2014) 322–337

2. Leiserson, C.E., Saxe, J.B.: Retiming synchronous circuitry. Algorithmica **6**(1) (1991) 5–35
3. Shenoy, N., Rudell, R.: Efficient implementation of retiming. In Kuehlmann, A., ed.: The Best of ICCAD. Springer US (2003) 615–630
4. Lin, M.B.: Introduction to VLSI Systems: A Logic, Circuit, and System Perspective. CRC Press, Boca Raton (2011)
5. Tillich, S., Feldhofer, M., Großschädl, J.: Area, Delay, and Power Characteristics of Standard-Cell Implementations of the AES S-Box. In Vassiliadis, S., Wong, S., Hämäläinen, T., eds.: Embedded Computer Systems: Architectures, Modeling, and Simulation. Volume 4017 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2006) 457–466
6. Corp., F.T.: Faraday Cell Library 0.13 μm Standard Cell (2004)
7. Daemen, J., Rijmen, V.: The Design of Rijndael. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2002)
8. Morioka, S., Satoh, A.: A 10 gbps full-aes crypto design with a twisted-bdd s-box architecture. In: Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on. (2002) 98–103
9. Satoh, A., Morioka, S., Takano, K., Munetoh, S.: A Compact Rijndael Hardware Architecture with S-Box Optimization. In: Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology. ASIACRYPT '01, London, UK, UK, Springer-Verlag (2001) 239–254
10. Morioka, S., Satoh, A.: An optimized s-box circuit architecture for low power aes design. In Kaliski, B., Koç, c., Paar, C., eds.: Cryptographic Hardware and Embedded Systems - CHES 2002. Volume 2523 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2003) 172–186
11. Rijmen, V.: Efficient Implementation of the Rijndael S-box
12. Canright, D.: A very compact s-box for aes. In Rao, J.R., Sunar, B., eds.: CHES. Volume 3659 of Lecture Notes in Computer Science., Springer (2005) 441–455
13. Mentens, N., Batina, L., Preneel, B., Verbauwhede, I.: A systematic evaluation of compact hardware implementations for the Rijndael S-BOX. In: Proceedings of the 2005 international conference on Topics in Cryptology. CT-RSA'05, Berlin, Heidelberg, Springer-Verlag (2005) 323–333
14. Paar, C.: Efficient VLSI architectures for bit parallel computation in Galios [Galois] fields. VDI-Verlag (1994)
15. Maheshwari, N., Sapatnekar, S.: Efficient Retiming of Large Circuits. IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS **6**(1) (March 1998) 74–83
16. Münzer, A., Hemme, G.: Converting combinational circuits into pipelined data paths. In: Computer-Aided Design, 1991. ICCAD-91. Digest of Technical Papers., 1991 IEEE International Conference on. (Nov 1991) 368–371
17. Jiang, J.H., Brayton, R.: Retiming and resynthesis: A complexity perspective. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **25**(12) (Dec 2006) 2674–2686
18. Wolkerstorfer, J., Oswald, E., Lamberger, M.: An ASIC implementation of the AES sboxes. In Preneel, B., ed.: Topics in Cryptology - CT-RSA 2002, The Cryptographer's Track at the RSA Conference, 2002, San Jose, CA, USA, February 18-22, 2002, Proceedings. Volume 2271 of Lecture Notes in Computer Science., Springer (2002) 67–78

19. Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In Paterson, K.G., ed.: EUROCRYPT. Volume 6632 of Lecture Notes in Computer Science., Springer (2011) 69–88
20. Bertoni, G., Breveglieri, L., Fragneto, P., Macchetti, M., Marchesin, S.: Efficient Software Implementation of AES on 32-Bit Platforms. In Jr., B.S.K., Çetin Kaya Koç, Paar, C., eds.: CHES. Volume 2523 of Lecture Notes in Computer Science., Springer (2002) 159–171
21. Hodjat, A., Verbauwhede, I.: Area-throughput trade-offs for fully pipelined 30 to 70 gbits/s AES processors. IEEE Trans. Computers **55**(4) (2006) 366–372
22. Hodjat, A., Verbauwhede, I.: A 21.54 Gbits/s fully pipelined AES processor on FPGA. In: Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on. (April 2004) 308–309
23. Boyar, J., Peralta, R.: A small depth-16 circuit for the aes s-box. In: Information Security and Privacy Research. Springer (2012) 287–298
24. Boyar, J., Peralta, R.: A new combinational logic minimization technique with applications to cryptology. In: Experimental Algorithms. Springer (2010) 178–189
25. Clark, J.A., Jacob, J.L., Stepney, S., Maitra, S., Millan, W.: Evolving Boolean Functions Satisfying Multiple Criteria. In: Progress in Cryptology - INDOCRYPT 2002. (2002) 246–259
26. Burnett, L., Carter, G., Dawson, E., Millan, W.: Efficient Methods for Generating MARS-Like S-Boxes. In: Proceedings of the 7th International Workshop on Fast Software Encryption. FSE '00, London, UK, UK, Springer-Verlag (2001) 300–314
27. Picek, S., Papagiannopoulos, K., Ege, B., Batina, L., Jakobovic, D.: Confused by Confusion: Systematic Evaluation of DPA Resistance of Various S-boxes. In: Progress in Cryptology - INDOCRYPT 2014 - 15th International Conference on Cryptology in India, New Delhi, India, December 14-17, 2014, Proceedings. (2014) 374–390
28. Yagain, D., Vijayakrishna, A.: A novel framework for retiming using evolutionary computation for high level synthesis of digital filters. Swarm and Evolutionary Computation **20** (2015) 37 – 47
29. Weise, T.: Global Optimization Algorithms - Theory and Application. Second edn. Self-Published (2009) Online available at http://www.it-weise.de/.
30. Talbi, E.G.: Metaheuristics: From Design to Implementation. Wiley Publishing (2009)
31. Eiben, A.E., Smith, J.E. In: Introduction to Evolutionary Computing. Springer-Verlag, Berlin Heidelberg New York, USA (2003)
32. Beyer, H.G., Schwefel, H.P.: Evolution Strategies A Comprehensive Introduction. Natural Computing **1**(1) (May 2002) 3–52
33. Yao, X.: Optimization by Genetic Annealing. In: Proc. of 2nd Australian Conf. on Neural Networks. (1991) 94–97
34. Glover, F.W., Kochenberger, G.A., eds.: Handbook of Metaheuristics. 1 edn. Volume 114 of International Series in Operations Research & Management Science. Springer (January 2003)
35. Standaert, F.X., Rouvroy, G., Quisquater, J.J., Legat, J.D.: Efficient implementation of rijndael encryption in reconfigurable hardware: improvements and design tradeoffs. In: Cryptographic Hardware and Embedded Systems-CHES 2003. Springer (2003) 334–350
36. Kerckhof, S., Durvaux, F., Hocquet, C., Bol, D., Standaert, F.X.: Towards green cryptography: a comparison of lightweight ciphers from the energy viewpoint. In:

Cryptographic Hardware and Embedded Systems–CHES 2012. Springer (2012) 390–407

Appendix A

Here, we give an example of the results for our statistical tool for a circuit of interest.

AES S-box Polynomial Basis

```
---  Network report [start]  ---
File: sbox_poli.txt
Num of paths: 8023409
Max path length: 3848.862013890002
Max possible layers: 4 (3 flip-flops)
Max possible num of flip-flops on max path: 31
Solution (BitString) size: 432
Network path delay statistics:
 [0-500>: 2
 [500-1000>: 2164
 [1000-1500>: 149944
 [1500-2000>: 2026442
 [2000-2500>: 3580150
 [2500-3000>: 1899675
 [3000-3500>: 361708
 [3500-4000>: 3324
 [4000-4500>: 0
 [4500-5000>: 0
---  Network report [end]  ---
```

In Fig. 3, we give a graphical representation of the AES S-box in polynomial basis. Blue lines depict internal nodes and red lines direct inputs.

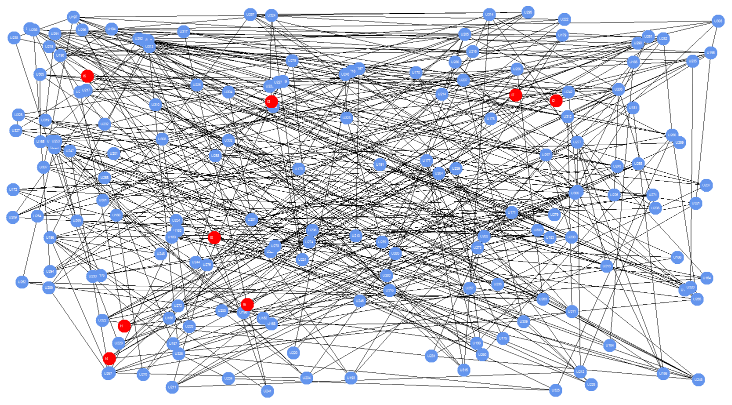


Fig. 3: Graphical representation of the S-box in polynomial basis.