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

No Author Given

No Institute Given

Abstract. Inserting flip-flops into a combinatorial circuit and obtaining the best area-performance trade-off is a difficult problem. One common approach is inserting flip-flops manually in the design phase. However, this method is unlikely to achieve the best possible placement. In this paper, we present a framework capable of finding positions to insert flip-flops in almost optimal way. Our novel method is using memetic algorithms and is shown to be fast, reliable and successful. We demonstrate our framework on the AES S-box, where we separately experiment with the S-box in polynomial and normal basis. Our results prove that this method should be consulted where optimal solution is of interest. Besides experimental results with the new algorithm, we also discuss the ideal model of a circuit, which can be used when assessing the quality of obtained solutions.

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

1 Introduction

Implementations of cryptography 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 on the Internet aim at ever faster 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. As examples, we mention access control, pay TV, medical applications and mobile payments. It is fair to say that techniques that boost the performance in hardware such as pipelining and parallelism remain important for efficient implementations. This leaves the paramount of real-time cryptography within the reach of some mobile devices.

In this work we focus on pipelining and more precisely, we look for the most optimal way to put registers (flip-flops) such that we improve the performance substantially, but without paying for it too much with area (i.e. 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 Evolutionary Computation (EC) applications, which are here used for cryptography, making this study unique. 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 that there are ways to optimize on the flip-flops (FFs) insertion in the AES S-box. 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. We consider the improvements completed with our study.

More specifically, our main contributions are:

1. Development of a new optimization algorithm that is able to produce correct solutions with a high certainty.
2. Improvement of the evaluation process that enables one to obtain results relatively fast.
3. Extensive tests showing the suitability of our approach.

Besides those three 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 is expected. For this purpose we experiment 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 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 present related work from both cryptographic and EC perspective. We continue in Section 3 where we give necessary information about AES and ways how to imple-

ment S-boxes in hardware. Furthermore, we give the basic terminology about circuits we follow in this work. In Section 4, we give 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 here Ideal Circuit Model, an abstraction that helps us to assess the quality of 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 the future work. Finally, in Section 6 we conclude this study.

2 Related Work

In the next section, we briefly summarize several important works on hardware implementations as well as on evolutionary computation techniques for applications in cryptology. First, we list related works that concentrate on hardware implementations where 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 first to take advantage of the composite field $GF(((2^2)^2)^2)$-based implementation for low area, which results in the most compact S-box at the time with a gate complexity of 5.4 kgates [2]. 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) [3]. 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 basis [4]. The main result proves that one can make better choices with different irreducible polynomials and representation of elements in this special type of tower field. Canright picked it up on this work, applying the ideas to normal basis [5]. Systematically exploring all the possibilities he deduced the smallest S-box at the time, the result that help up for almost a decade.

Only recently Moradi et al. have published the most compact AES implementation of the size of only 2.4 kgates [6]. This result is obtained by focusing on AES encryption and squeezing the area on all the design layers.

Following the other line, Macchetti and Bertoni [7] 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 alternative in the design.

Looking into high-speed implementations, Hodjat and Verbauwhede describe an ASIC implementation for the same composite field $GF((2^4)^2)$ as Wolkerstorfer [8]. 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 consider also pipelined AES on FPGA [9].

From the Evolutionary Computation perspective, we can find a number of papers that explore various applications that could be of interest in cryptology. However, here we list only a few works that clearly state cryptographic applications as their goals. Two most relevant topics for using EC in cryptology is evolution of Boolean functions and evolving S-boxes.

Millan et al. use Genetic Algorithms (GAs) to evolve Boolean functions with the goal of high nonlinearity [10]. In conjunction with the GA they use a combination of different optimization algorithms in order to find Boolean functions with even higher nonlinearity.

Clark and Jacob experiment with two-stage optimization to generate Boolean functions [11]. They use a combination of simulated annealing (SA) and hill climbing with a cost function motivated by Parseval theorem in order to find functions with high nonlinearity and low autocorrelation.

Clark et al. use SA to generate Boolean functions with cryptographically relevant properties [12]. In their work, they consider balanced function with high nonlinearity and with the correlation immunity property less or equal to two.

Kavut and Yücel develop an improved cost function for a search that combines SA and hill climbing [13]. With the approach, the authors are able to find some functions of eight and nine inputs that have a combination of nonlinearity and autocorrelation values previously unattained. They also experiment with three-stage optimization method that combines SA and two hill climbing algorithms with different objectives.

On the other hand, Burnett et al. use heuristic method to generate MARS-like S-boxes [14]. With their approach they are able to generate very efficiently a number of S-boxes of appropriate size that satisfy all the requirements placed on MARS S-box. With a combination of several techniques, they are even able to find S-boxes with improved nonlinearity.

Picek et al. use GA to evolve S-boxes of various sizes that have presumably better resistance against Differential Power Analysis (DPA). In their research they experiment with three different DPA-related metrics [15–17].

Batina et al. conduct the first experiments where they try to evolve AES S-box in a 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.

3 Preliminaries

In this section, we give necessary information for following this work. First, we shortly describe the AES cipher, afterwards we discuss S-box implementation options in polynomial and normal basis. Finally, we define network related terminology we use.

3.1 AES Cipher

As already stated, the target for the pipelining in this work is the S-box as used in AES cipher. Furthermore, we experiment with both polynomial and normal basis. In accordance with that, here we give necessary details about the AES cipher, and various ways how to implement S-box. The AES cipher, or Rijndael cipher is a symmetric block algorithm where data is encrypted and decrypted in blocks of 128 bits [**?**]. To obtain a ciphertext, 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 128 bits key, 12 rounds for 192 bits key and 14 rounds for 256 bit key. Each round has a unique key that is calculated from the initial key. Operations in the AES cipher are on a 4×4 column-ordered matrix of bytes, called the state. Those operations are *AddRoundKey*, *SubBytes*, *ShiftRows* and *MixColumns*. All the rounds consist of the same set of operations, except that before the first round there is *AddRoundKey* operation, and the last round does not have *MixColumns* operation. As a note, often it is said that AES is an SPN cipher (Substitution-Permutation Network), but that is not completely true since there is that additional *MixColumns* operation. With regards to the whole AES cipher, there are a variety of objectives when implementing it. In accordance with that, there exist for example approaches that maximizes the throughput [**?**], minimize circuitry [2] or power consumption [18]. In the rest of this paper, we concentrate on maximizing the throughput of as small as possible circuit. Therefore, we combine the fist and third goal.

The only nonlinear part of the AES cipher is the *SubBytes* operation which is the source of confusion in the cipher as defined by Shannon [19]. *SubBytes* operation is realized through the vectorial Boolean function or S-box (Substitution Box). This *SubBytes* operation replaces each byte of the input, where it involves inverse operation in the Galois field $GF(2^8)$. This calculation is not easy and therefore there are several techniques how to approach this problem. The easiest (and fastest) way is to implement S-box as a lookup table (LUT) with all possible values between $0x00$ and $0xFF$. However, this approach is sufficient for software implementations, but not so much for hardware implementations. In the rest of this paper, we only consider AES that has 10 rounds in order to simplify the considerations. Consider that each S-box requires 256 bytes of memory and if wanting to calculate in parallel 16 bytes of state we need 16 copies of the S-box. Furthermore, this is only for one round of AES, the whole cipher would require 160 copies of S-box (not counting the copies necessary for the calculation of round keys). Naturally, depending on the hardware, this can present a significant amount of necessary hardware resources. To reduce the circuitry, Rijmen suggested to calculate the inverse of Galois field by using a subfield arithmetic [**?**]. This idea was further extended by work of Satoh et al. who suggested to use tower field approach [2]. Works of Canright [5] and Mentens et al. [4] showed that the most compact solutions rely on composite field arithmetic.

3.2 S-box Implementation

We describe here two implementation options for the AES S-box that we use in our experiments. Those implementation options are to use polynomial and normal bases. The main advantage of such implementations over LUT implementation is that is relatively simple in regards to the number of gates needed for hardware implementation.

The AES cipher uses Galois field with 8 bits where those bits are coefficients of a polynomial. Furthermore, multiplication is done modulo the irreducible polynomial

$$q(x) = x^8 + x^4 + x^3 + x^2 + 1, \tag{1}$$

where the addition of coefficients is done modulo 2.

To represent some general element E of the Galois field $GF(2^8)$ over $GF(2^4)$ we use linear polynomial in y:

$$E = \gamma_1 y + \gamma_0, \tag{2}$$

where the multiplication is done modulo irreducible polynomial:

$$r(y) = y^2 + \tau y + \nu. \tag{3}$$

With this, we simplified operation since all coefficients are now in the 4-bit subfield $GF(2^4)$. Therefore, element E in polynomial basis $[Y, 1]$ is represented with a pair $[\gamma_1, \gamma_0]$. Here, Y is a root of $r(y)$. In the normal basis we use both roots of $r(y)$ and the basis is $[Y^{16}, Y]$.

We follow the same technique and now present $GF(2^4)$ over $GF(2^2)$ as linear polynomials in z:

$$\gamma = \Gamma_1 z + \Gamma_0, \tag{4}$$

where the multiplication is done irreducible polynomial:

$$s(z) = z^2 + Tz + N. \tag{5}$$

Here, all coefficients are in $GF(2^2)$ and Z is one root of $s(z)$ for polynomial basis $[Z, 1]$. In normal basis we use both roots of $s(z)$ with a basis $[Z^4, Z]$.

Finally, $GF(2^2)$ can be represented over $GF(2)$ with a linear polynomial in w:

$$\Gamma = g_1 w + g_0. \tag{6}$$

Multiplication is done modulo the following irreducible polynomial:

$$t(w) = w^2 + w + 1. \tag{7}$$

Here, $[W, 1]$ represents a polynomial basis $[W, 1$ where W is one root of $t(w)$. Alternatively, in the normal basis, we use both roots of $t(w)$ with a normal basis $[W^2, W]$. For all of the aforementioned fields, the addition is just bitwise XOR for both basis.

Polynomial Basis. To multiply modulo irreducible polynomial as given in Eq. (3), we have:

$$(\gamma_1 y + \gamma_0)(\delta_1 y + \delta_0) = (\gamma_1\delta_0 + \gamma_0\delta_1 + \gamma_1\delta_1\tau)y + (\gamma_0\delta_0 + \gamma_1\delta_1\nu). \tag{8}$$

The inverse is given by:

$$(\gamma_1 y + \gamma_0)^{-1} = \left[\theta^{-1}\gamma_1\right] y + \left[\theta^{-1}(\gamma_0 + \gamma_1\tau)\right], \tag{9}$$

where

$$\tau = \gamma_1^2\nu + \gamma_1\gamma_0\tau + \gamma_0^2. \tag{10}$$

Normal Basis. In the normal basis the multiplication is done as follows:

$$(\gamma_1 Y^{16} + \gamma_0 Y)(\gamma_1 Y^{16} + \gamma_0 Y) = \left[\gamma_1\delta_1\tau + \theta\right] Y^{16} + \left[\gamma_0\delta_0\tau + \theta\right] Y, \tag{11}$$

where

$$\theta = (\gamma_1 + \gamma_0)(\delta_1 + \delta_0)\nu\tau^{-1}. \tag{12}$$

The inverse is as follows:

$$(\gamma_1 Y^{16} + \gamma_0 Y)^{-1} = \left[\theta^{-1}\gamma_0\right] Y^{16} + \left[\theta^{-1}\gamma_1\right] Y, \tag{13}$$

where

$$\theta = \gamma_1\gamma_0\tau^2 + (\gamma_1^2 + \gamma_0^2)\nu. \tag{14}$$

For further information about the polynomial and normal bases, we refer readers to [4, 5]. For details about tower fields, we refer readers to [**?**].

3.3 Circuit Terminology

Circuit (network) is Netlist is Input Output The number of inputs refers to the number of inputs to all standard cells. A path consists of every unique combination of nodes connecting a single input to a single output. The number of paths denotes the number of different possible paths through the circuit from an input to an output. Critical delay of a network is the largest sum of delay times of individual cells on distinct paths.

3.4 Standard Cells and Delays

In an effort to have results that are possible to compare with those from previous work, we use the same standard cell library, namely, UMC 0.13 μm low-leakage standard cell library [20]. To obtain a delay value for each cell, we work with low load capacitance of 1.5 fF and use the average values for all possible combinations (transitions from low to high and from high to low). There is also a possibility to work with the worst case delays, but that would not change the methodology. Furthermore, such scenario is not a very realistic one, since it would assume that each wire has a high load, which does not occur so often. As the delay cell we use a (`QDFFCLD`), D-FF cell that has a single output and no clear, set or enable with an average delay time of 320.35 ps.

Standard cell delay, wire load, fanout, for Nele. For further information about standard cells and delays, we refer readers to [**?**].

4 The Optimization Framework

4.1 Ideal Circuit Model

In this section, we define an ideal circuit which is always possible to pipeline into networks of the same size. Therefore, the critical path after the pipelining equals the beginning critical path divided by the number of added flip-flop layers. As an example, consider a circuit that has a critical path equal to 1000 ns. Ideal circuit when adding one layer of flip-flops would have an even number of cells in the longest path where all cells are the same. As a result, after inserting flip-flops on all necessary position, the critical path would equal 500 ns.

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 expect that the best possible solution should be close to the ideal solution. Furthermore, since we work with the averaged delays, this will also introduce deviation between ideal and obtained results.

Definition 1 *Ideal circuit model consists of only one type of a cell (generic) where it is always possible to add FFs to a circuit. 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 FFs layers one adds.*

Next, we define the maximal number of flip-flops that is possible to add to a network.

Definition 2 *The maximal number of flip-flop layers is bounded above with a maximal number of cells that are possible to add to a shortest path connecting the input to the output of the network.*

4.2 The Choice of the Optimization Procedure

Similarly to the approach from [1], we regard this as an optimization problem. First, we give a formal definition of this problem.

Definition 3 *Pipelining a combinatorial circuit in a way that maximizes throughput 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 4 *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 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, is it possible to obtain a correct solution?

To answer this question, we run AES S-box in polynomial basis and we create 10 000 random solutions (by inserting flip-flops in random positions). Out of those solutions, not one was correct. This should not be surprising, since we do not know where to insert flip-flops or how many FFs in total we need to insert to obtain a correct solution. On the basis of the aforementioned results, 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 [21]. 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 there is yet no heuristic algorithm tailor-made for this problem that we are aware of). Metaheuristics are general-purpose algorithms that can be applied to solve almost any optimization problem [22]. To classify metaheuristics, one can follow many criteria, but we divide it into single-solution based metaheuristics and population based heuristics [22]. 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 aforesaid 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) [23], evolution strategy (ES) [24] and genetic annealing (GAn) [25]. 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] in order for easier comparison of the results. The goal is the **minimization** of the following equation:

$$fitness = max\_delay\_time + (1,000 * number\_invalid\_paths). \tag{15}$$

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 FF 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.

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 optimization algorithm. The same tool after flip-flops are set returns solution back into netlist format. The tool itself is written in Java programming language, but the implementation details are of secondary importance so not presented here.

However, the question is what is a possible insertion position? The most general option is to allow an insertion of an FF 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 of length equal to the product of 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 output of a cell (*output-based* encoding). In this way we are able to reduce the solution length and size of 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. We clarify this with a small example.

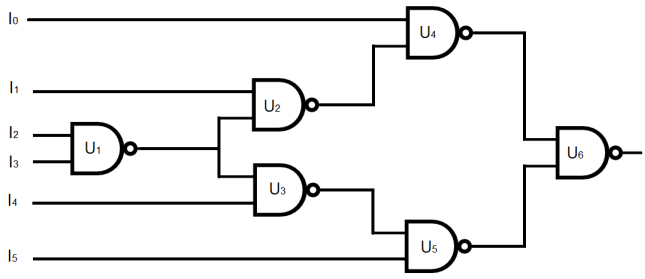


Fig. 1: Example of a circuit.

Consider the path between cells U_1 and U_2. If we set a flip-flop on the output of the cell U_1, then we need to add flip-flips to both I_1 and I_4. However, if we set flip-flops to the input of cell U_1 then it is sufficient to add flip-flop to the I_1. we note that here we do not discuss all necessary positions to obtain a correct solution, but rather we clarify how in one encoding some solutions cannot be represented.

4.3 Genetic Algorithm

Genetic algorithms (GAs) are probabilistic algorithms whose search methods model some natural phenomena: genetic inheritance and survival of the fittest [26]. 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 [21, 27]. Usual variation operators in GA are mutation and crossover [27]. In the context of optimization, exploration (diversification) means finding new points in previously unexplored areas of 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 [21,22]. For an optimization algorithm to be successful, it needs to have a good balance between those two notions to avoid too fast convergence to a local optimum from one side, but also too long operation time from the other side. For further information about GAs, we refer to [27, 28].

In addition to this metaheuristic, we also experimented with evolution strategy and genetic annealing. However, after the initial round of experiments, the results have shown that GA outperforms by far the other two algorithms. Therefore, in the rest of the paper we consider only GA in our experiments.

4.4 Design of the Optimization Algorithm

As noted, in our experiments we use GA in order to find suitable locations for the insertion of flip-flops. However, it is easy to notice that 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 is the solution. Consider the situation where GA produces a solution that is not correct, but it 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 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 GA, we add a local search (LS) algorithm that tries to correct almost-correct paths. Since now we combine GA 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 algorithm) and local improvement algorithms [21]. Most MAs can be interpreted as search strategies in which a population of solutions cooperate and compete [29]. Next, we give a pseudocode for our optimization algorithm as Algorithms 1 until 4.

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

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

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 tour-
    nament
    D = randomCrx(R1, R2)
    evaluate(D)
    replace the worst in P with D
  until POP_SIZE times
end while
```

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
```

Algorithm 3 Neighborhood.

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

Finally, 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 asses 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 include the population size, which is set to 50. The tournament size k in the tournament selection is equal to 3. Mutation probability is set to 0.01 per individual where we choose it on a basis of a small set of tuning experiments where it showed the best results 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, all common parameters we display in Table 1.

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

4.5 Circuit Decomposition

Here, we briefly discuss 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 realizes a part of the functionality of the complete network and it is possible to pipeline only a subnetwork. We call this 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 the Definition 2, we offer the following definition:

Definition 5 *It is possible to add flip-flops only to those subnetworks that do not contain cell with direct inputs to the network.*

5 Experimental Results

In this section, we first introduce results obtained with two different methods where the focus is on those results obtained with the optimization algorithm. The obtained results are synthesized using Synopsys Design Compiler in order to get the pre-layout implementation results for the critical path delay and the area.

5.1 Introducing FFs in the Design Phase

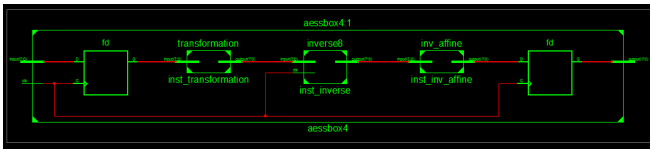
We established that randomly setting flip-flops cannot result in a correct network when working with such complex networks as given here. However, what about inserting flip-flops in the design phase? In this way, we avoid working with netlist, but rather we work with an abstraction of a network that is much easier to comprehend. As an example, here we take AES S-box in polynomial basis and then we insert flip-flops into inverse8 part. This is represented in Figures 2a and 2b. Flip-flops are depicted as "fd" cells in the latter figure. After synthesizing this network, we obtain a critical path of ?? *ns*. We note that the tool itself changes the network when adding the cells in the design phase.

5.2 Results for the Optimization Algorithm

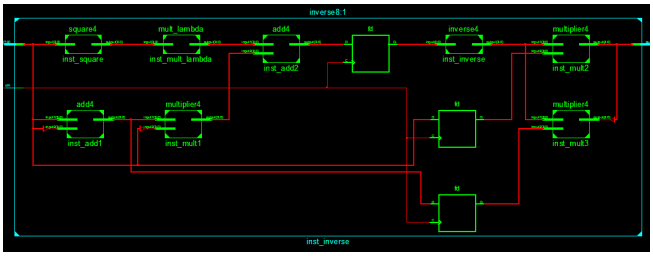
In this section, we present the best results we obtained with our memetic algorithm. Alongside, we give basic statistics about netlists without inserted flip-flops in Table **??**. For extensive statistics, we refer readers to 6.

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

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



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



(b) Zoom into inverse8 with added FFs.

Fig. 2: Example of inserting FFs in design phase.

In Table **??**, 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)	Number of added FFs
Polynomial	2	2 065.7435	64
Polynomial	3	1 988.5056	117
Polynomial, out	2	3 075.6087	11
Normal	2	2 508.8050	73

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

5.3 Speed of the Evolution

After discussing the successfulness of our approach in the previous section, here we discuss its reliability and speed. As already stated, those three objectives are what we believe that differentiates 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 30 times. We consider the algorithm successful if it generates

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	-	-
Polynomial	3	1.58	6.34	79.36	12.69	-	-	-
Normal	2	-	-	10.52	-	5.26	36.84	47.37

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 the 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-box in normal basis with flip-flops based on the inputs of the cells, the successfulness equals 91.6%. When adding two levels of flip-flops, this problem becomes more difficult. For instance, for S-box in polynomial representation and flip-flops on the input of cells, the successfulness is 30%. 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.

We examine separately the speed of evolution and speed of evaluation. When working with S-box in polynomial representation, 100 000 evaluations (here, an evaluation is the whole process of obtaining a new individual and examining its fitness) lasts somewhat less than 10 000 seconds. Therefore, one evaluation lasts around 100 ms. For the S-box in normal basis, the evaluation is much slower since there are significantly more paths; 100 000 evaluations last around 200 000 seconds. When examining only the evaluation of the correctness of a solution, one evaluation lasts 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.

5.4 Synthesizing the Solutions

5.5 Discussion

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 a question is it worth while? Although our framework is capable of generating results relatively fast, this is still significantly slower than what is the case when adding flip-flops in the design phase. Therefore, the answer to the previous question depends on setting. If we have a setting where we require as

high throughput as possible and where we can afford the cost of added flip-flops, this methodology represents a valuable resource. Otherwise, the total cost versus benefit is much less favorable. Although we work here on S-boxes realized in tower field, there is nothing stopping us to use this method with any other kind of combinatorial circuit. In Appendix 6, we also show statistics for the full AES and Keccak rounds that also represents extreme cases when compared with the S-boxes. Keccak can be translated into a solution that has relatively small number of paths, but huge number of positions and therefore huge search space. From the other perspective, full AES does has much larger search space than the S-box only (although still small when compared with Keccak case), but it has extremely large number of paths. Furthermore, we observe that Keccak has relatively short critical path and the real question is whether it is even possible to pipeline it (especially since the delay of FF is more than 20% od the delay of whole critical path). Naturally, the smaller the critical delay, the smaller is the benefit of pipelining.

In any case, pipelining has a big impact on which modes of use are efficient on such an implementation. For fully exploiting the power of the AES instructions, one needs a lot of parallelism in the mode and that has the unfortunate side effects that the "better block encryption modes" such as CBC are much less applied and one tends to do counter mode (fully parallelizable).

5.6 Future Work

In 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. In Appendix 6, we give statistics for full AES round (that has S-box in polynomial basis) and we see that it has almost 400 million of paths. Therefore, this is a much more complex example than those we examined here. When comparing it with the S-box in normal basis, we see it has "only" three times more paths. This suggests that our method should work on it. However, when adding the fact that the evaluation of all path is the most expensive part of our algorithm, we must be aware that finding a solution for full AES round will be much slower than in work done here. Besides that, we plan to further improve our LS part of the algorithm since it efficiency has extreme impact on the efficiency of the whole algorithm.

6 Conclusion

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. 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
3. 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
4. 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
5. 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
6. 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
7. 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
8. 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
9. 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
10. Millan, W., Clark, A., Dawson, E.: Heuristic design of cryptographically strong balanced Boolean functions. In: Advances in Cryptology - EUROCRYPT '98. (1998) 489–499
11. Clark, J., Jacob, J.: Two-Stage Optimisation in the Design of Boolean Functions. In Dawson, E., Clark, A., Boyd, C., eds.: Information Security and Privacy. Volume 1841 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2000) 242–254
12. 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
13. Kavut, S., Yücel, M.: Improved Cost Function in the Design of Boolean Functions Satisfying Multiple Criteria. In Johansson, T., Maitra, S., eds.: Progress in Cryptology - INDOCRYPT 2003. Volume 2904 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2003) 121–134
14. 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
15. Picek, S., Ege, B., Batina, L., Jakobovic, D., Chmielewski, L., Golub, M.: On Using Genetic Algorithms for Intrinsic Side-channel Resistance: The Case of AES S-box. In: Proceedings of the First Workshop on Cryptography and Security in Computing Systems. CS2 '14, New York, NY, USA, ACM (2014) 13–18

16. Picek, S., Ege, B., Papagiannopoulos, K., Batina, L., Jakobovic, D.: Optimality and beyond: The case of 4x4 s-boxes. In: 2014 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2014, Arlington, VA, USA, May 6-7, 2014. (2014) 80–83
17. 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
18. 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
19. Shannon, C.: Communication theory of secrecy systems. Bell System Technical Journal **28**(4) (1949) 656–715
20. Corp., F.T.: Faraday Cell Library 0.13 μm Standard Cell (2004)
21. Weise, T.: Global Optimization Algorithms - Theory and Application. Second edn. Self-Published (2009) Online available at http://www.it-weise.de/.
22. Talbi, E.G.: Metaheuristics: From Design to Implementation. Wiley Publishing (2009)
23. Holland, J.H. In: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. The MIT Press, Cambridge, USA (1992)
24. Beyer, H.G., Schwefel, H.P.: Evolution Strategies A Comprehensive Introduction. Natural Computing **1**(1) (May 2002) 3–52
25. Yao, X.: Optimization by Genetic Annealing. In: Proc. of 2nd Australian Conf. on Neural Networks. (1991) 94–97
26. Michalewicz, Z.: Genetic algorithms + data structures = evolution programs (3rd ed.). Springer-Verlag, London, UK, UK (1996)
27. Eiben, A.E., Smith, J.E. In: Introduction to Evolutionary Computing. Springer-Verlag, Berlin Heidelberg New York, USA (2003)
28. Haupt, R.L., Haupt, S.E.: Practical Genetic Algorithms. John Wiley & Sons, Inc., New York, NY, USA (1998)
29. 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)

Appendix A

In this section we give results of our statistical tool for each of the circuits 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]  ---
```

Next, we give statistics of our best solution with one added level of FFs.

```
---  Solution report [start]  ---
Expected num of layers: 2
Network condition: satisfied
Fitness for 2 layers: 2065.7435
Num of paths (in all layers): 23501
Individual path delay statistics for 2 layers:
 [0-500>: 288
 [500-1000>: 7529
 [1000-1500>: 13428
 [1500-2000>: 2240
 [2000-2500>: 16
 [2500-3000>: 0
 [3000-3500>: 0
Individual flip-flop statistics for 2 layers:
 0: 0
 1: 8023409
---  Solution report [end]  ---
```

Similarly, when added two layers of FFs, the statistics is:

```
---  Solution report [start]  ---
Expected num of layers: 3
Network condition: satisfied
Fitness for 3 layers: 1988.5055833000001
Num of paths (in all layers): 49546
Individual path delay statistics for 3 layers:
 [0-500>: 466
 [500-1000>: 8871
 [1000-1500>: 31166
 [1500-2000>: 9043
 [2000-2500>: 0
 [2500-3000>: 0
Individual flip-flop statistics for 3 layers:
```

```
 0: 0
 1: 0
 2: 8023409
---  Solution report [end]  ---
```

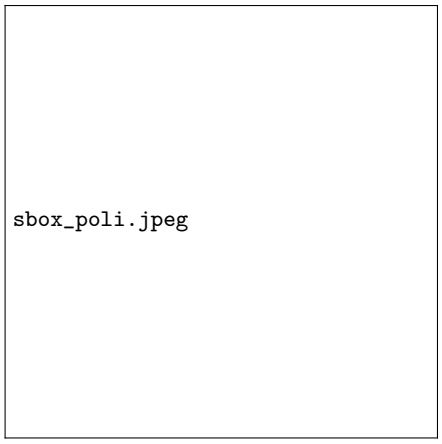
AES S-box Normal Basis

```
---  Network report [start]  ---
File: sbox_normal.txt
Num of paths: 139221044
Max path length: 4685.7242400000005
Max possible layers: 2 (1 flip-flops)
Max possible num of flip-flops on max path: 38
Solution (BitString) size: 497
Network path delay statistics:
 [0-500>: 31
 [500-1000>: 3562
 [1000-1500>: 183550
 [1500-2000>: 2980925
 [2000-2500>: 19120594
 [2500-3000>: 49641279
 [3000-3500>: 49621351
 [3500-4000>: 16471425
 [4000-4500>: 1195795
 [4500-5000>: 2532
 [5000-5500>: 0
 [5500-6000>: 0
---  Network report [end]  ---
```

The network after added one layer of FFs has the following stats:

```
---  Solution report [start]  ---
Expected num of layers: 2
Network condition: satisfied
Fitness for 2 layers: 2508.8050350000003
Num of paths (in all layers): 91352
Individual path delay statistics for 2 layers:
 [0-500>: 396
 [500-1000>: 8593
 [1000-1500>: 38597
 [1500-2000>: 37480
 [2000-2500>: 6280
 [2500-3000>: 6
 [3000-3500>: 0
 [3500-4000>: 0
Individual flip-flop statistics for 2 layers:
 0: 0
 1: 139221044
---  Solution report [end]  ---
```

Next, in Figures **??** and **??**, we give graphical representations of the AES S-box in polynomial and normal basis, respectively. Blue lines depict internal nodes and red lines direct inputs.



(a) Graphical representation of the S-box in polynomial basis.

(b) Graphical representation of the S-box in polynomial basis.

Fig. 3: Graphical representation of the S-box in two bases.

AES One Round

```
---  Network report [start]  ---
File: AESround.txt
Num of paths: 337930631
Max path length: 5132.295189500002
Max possible layers: 2 (1 flip-flops)
Max possible num of flip-flops on max path: 37
Solution (BitString) size: 7638
Network path delay statistics:
 [0-500>: 108
 [500-1000>: 984
 [1000-1500>: 124067
 [1500-2000>: 5428604
 [2000-2500>: 58791831
 [2500-3000>: 142818644
 [3000-3500>: 101842959
 [3500-4000>: 26216596
 [4000-4500>: 2657861
 [4500-5000>: 48927
 [5000-5500>: 50
```

```
 [5500-6000>: 0
---  Network report [end]  ---
```

Keccak One Round

```
---  Network report [start]  ---
File: keccak.txt
Num of paths: 388236
Max path length: 1476.554292
Max possible layers: 3 (2 flip-flops)
Max possible num of flip-flops on max path: 11
Solution (BitString) size: 22543
Network path delay statistics:
 [0-500>: 17974
 [500-1000>: 200851
 [1000-1500>: 169411
 [1500-2000>: 0
 [2000-2500>: 0
---  Network report [end]  ---
```