S-box Pipelining Using Genetic Algorithms for High-Throughput AES Implementations: How Fast Can We Go?

Lejla Batina [1], Domagoj Jakobovic [2], Nele Mentens [3], Stjepan Picek [1,2], Antonio de la Piedra [1](✉), and Dominik Sisejkovic [2]

[1] Digital Security Group, ICIS, Radboud University Nijmegen, Nijmegen, The Netherlands
{lejla,s.picek,a.delapiedra}@cs.ru.nl
[2] Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia
{domagoj.jakobovic,dominik.sisejkovic}@fer.hr
[3] KU Leuven ESAT/COSIC and iMinds, Leuven-Heverlee, Belgium
nele.mentens@kuleuven.be

Abstract. In the last few years, several practitioners have proposed a wide range of approaches for reducing the implementation area of the AES in hardware. However, an area-throughput trade-off that undermines high-speed is not realistic for real-time cryptographic applications. In this manuscript, we explore how Genetic Algorithms (GAs) can be used for pipelining the AES substitution box based on composite field arithmetic. We implemented a framework that parses and analyzes a Verilog netlist, abstracts it as a graph of interconnected cells and generates circuit statistics on its elements and paths. With this information, the GA extracts the appropriate arrangement of Flip-Flops (FFs) that maximizes the throughput of the given netlist. In doing so, we show that it is possible to achieve a 50 % improvement in throughput with only an 18 % increase in area in the UMC 0.13 μm low-leakage standard cell library.

Keywords: Real-time cryptography · Genetic Algorithms (GAs) · S-boxes

1 Introduction

Implementations of cryptography are of constant interest for companies making security products. The challenges vary from very compact, low-power/energy to high-speed implementations of both symmetric and asymmetric cryptographic

This work was supported in part by the Technology Foundation STW (project 12624 - SIDES), The Netherlands Organization for Scientific Research NWO (project ProFIL 628.001.007) and the ICT COST action IC1204 TRUDEVICE.

algorithms. Ever growing applications require security services, introduce more constraints and real-time crypto has become of paramount importance.

This race for the fastest implementations in both hardware and software is especially difficult for algorithms that feature endless implementation options such as Elliptic Curve Cryptography (ECC) and the AES standard. For example AES can be implemented with table look-ups or via multiple composite field representations, each of which has certain advantages for area, performance and security [1–4]. Moreover, other practitioners have relied on resource sharing and folded architectures for reducing the implementation area [5,6]. However, when considering fast hardware implementations, pipelining is an obvious choice to increase the throughput. Nevertheless, as pipelining implies adding FFs, it also increases the area.

An interesting research challenge is to optimize throughput, while keeping the area under control. More precisely, considering compact options for the AES S-box, namely those relying on composite field arithmetic, which are the best pipelining solutions to maximize the throughput? Strategies for solving this problem typically involve hardware tools and rely mostly on good hardware design practice, but are generally far from straightforward. The goal of this research is to investigate this problem of optimizing the performance via pipelining such that the best throughput is found for a given composite field S-box. For this purpose we use genetic algorithms that have already found their place in other cryptographic applications.

Our contribution is in comprehensively evaluating how to boost the performance of AES implementations based on composite field arithmetic. To this end, we consider a hardware implementation where the S-box is implemented using a polynomial basis in $GF(((2^2)^2)^2)$ as used for example by Satoh *et al.* [2] and Mentens *et al.* [3]. We deploy genetic algorithms to find a good solution for the position of the pipelining FFs in order to reduce the critical path as much as possible.

Through our methodology we find a solution that adds one level of pipelining registers in order to increase the throughput of the S-box with 50 % while the extra FFs only increase the area with 18 %. To underline the added value of our approach, we add some statistics on the circuit under investigation, which show that it is far from straightforward to find this solution.

The remainder of this paper is organized as follows. First, in Section 2, we describe how GAs have been coupled with cryptographic applications in the literature as well as different alternatives for exploring the design space of an AES implementation. In Section 3 we illustrate the initial design of the AES S-box based on composite fields that we have selected for this work. Then, in Section 4, we present the framework that we have developed for analyzing Verilog netlists, generating an appropriate input for the GA, evaluating the correctness of the resulting netlist and synthesizing our solution. In Section 5, we describe our results and end in Section 6 with some conclusions.

2 Related Work

Considering previous works on hardware implementations of AES, numerous papers appeared optimizing various implementation properties i.e. throughput, area, power, energy etc. Here we remind the reader to some of those that are using a design choice that is similar to ours. The main focus is on implementations using composite field arithmetic to boost compactness and/or speed.

Satoh *et al.* were the first to introduce a new composite field $GF(((2^2)^2)^2)$-based implementation which resulted in the most compact S-box at the time with a gate complexity of 5.4 kgates. Wolkerstorfer *et al.* [7] used 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). Mentens *et al.* [3] and Canright [4] found the best choice of polynomials and representation to optimize the S-box area for polynomial and normal basis respectively. However, Moradi *et al.* have recently published the most compact AES implementation [8] of the size of only 2.4 kgates. This result is obtained by optimizing AES encryption on all layers and pushing the area to this minimum. Hodjat *et al.* [9] have described an ASIC implementation for the same composite field $GF((2^4)^2)$ as Wolkerstorfer. Their approach was to perform an area-throughput trade-off by fully pipelining the architecture and optimizing the key-schedule implementation. One of the more recent works is from Kumar *et al.* [10] considering an FPGA implementation of AES based on inversion in $GF(2^8)$. The critical path they obtain is 4.6 ns, which is substantially slower than our best result (2.6 ns). In general, hardware design tools consist of algorithms that focus on the optimization of area, speed and/or power/energy consumption, but none of the tools handles automatic pipelining to the extend that we do in this paper.

The literature is rich in examples where researchers have been relying on GAs for solving cryptographic problems. For instance, Jhajharia *et al.* and Sokouti *et al.* proposed the utilization of GAs for generating cryptographic keys [11,12]. Further, Zarza *et al.* and Park *et al.* utilized GAs in the context of cryptographic protocol design [13,14]. Carpi *et al.* studied the selection of security parameters for protecting smart cards against fault attacks via GAs [15]. Moreover, there are many successful applications of evolutionary computation when evolving S-boxes. Clark *et al.* used the principles from the evolutionary design of Boolean functions to evolve S-boxes with desired cryptographic properties [16]. They used the Simulated Annealing (SA) heuristic coupled with the hill-climbing algorithm to evolve bijective S-boxes with high non-linearity. Picek *et al.* used genetic algorithms to evolve S-boxes of various sizes that have better resistance to DPA attacks [17,18].

In this work, we use GAs to explore the design space of a standard cell netlist that has many unbalanced and partially overlapping paths from input to output. In order to achieve high throughput, GAs are used to choose the position of the pipelining FFs. We are not aware of prior work addressing pipelining of digital circuits using evolutionary computation.

3 S-box Implementation

In this section we describe the implementation options for the AES S-box that we experiment with for this study. It was shown before in the works of Canright [4] and Mentens *et al.* [3] that the most compact solutions rely on composite field arithmetic.

Considering various arithmetic options in binary extension fields to optimize the inversion operation in the AES S-box, there are basically two implementation options. We can either perform the subfield operations directly in $GF(2^4)$ or we can perform computations in the tower field $GF(((2^2)^2)^2)$, i.e. working in all subfields and using the fact that inversion is linear in $GF(2^2)$. The latter option is the one we choose.

The field $GF(((2^2)^2)^2)$ is considered as an extension of degree 2 over $GF((2^2)^2)$ constructed using the irreducible polynomial $P(x) = x^2 + p_1x + p_0$, where $p_1, p_0 \in GF((2^2)^2)$. $GF((2^2)^2)$ is a field extension of degree 2 over $GF(2^2)$ using the irreducible polynomial $Q(y) = y^2 + q_1y + q_0$ with $q_1, q_0 \in GF(2^2)$. $GF(2^2)$ is a field extension of degree 2 over $GF(2)$ using the irreducible polynomial $R(z) = z^2 + z + 1$. The constants are given in Appendix A.

In this case, inversion in $GF(2^2)$ requires only one addition:

$$d = t_1z + t_0 \in GF(2^2) : d^{-1} = t_1z + (t_1 + t_0). \tag{1}$$

These operations are implemented as depicted in Fig. 1.

Unlike the inversion in $GF((2^4)^2)$, the building blocks are not implemented as 4-bit look-up tables, but as operations in $GF((2^2)^2)$, which can be computed as follows (with $a_1, a_0, b_1, b_0 \in GF(2^2)$):

- $(a_1y + a_0) \cdot (b_1y + b_0) = (a_1b_1 + a_1b_0 + a_0b_1)y + (a_1b_1\phi + a_0b_0)$;
- $(a_1y + a_0)^2 = a_1y + (a_1\phi + a_0)$;
- $(a_1y + a_0) \cdot \lambda = (a_1y + a_0)\omega y = (a_1 + a_0)\omega y + a_1\omega\phi$.

These equations consist of operations in $GF(2^2)$ that can be computed as follows (with $g_1, g_0, h_1, h_0 \in GF(2)$):

- $(g_1z + g_0) \cdot (h_1z + h_0) = (g_1h_1 + g_0h_1 + g_1h_0)z + (g_1h_1 + g_0h_0)$;
- $(g_1z + g_0)^2 = a_1z + (a_1 + a_0)$;
- $(g_1z + g_0) \cdot \phi = (g_1z + g_0) \cdot z = (g_1 + g_0)z + g_1$;
- $(g_1z + g_0) \cdot \omega = (g_1z + g_0) \cdot (z + 1) = g_0z + (g_1 + g_0)$.

4 Methodology

In this section we provide an explanation of the framework that we have developed for interfacing the GA with different modules for analyzing, simulating and synthesizing evolved netlists (Figure 2). First, we parse a Verilog description of a certain circuit, which is, in our case, the S-box design described in Section 3.

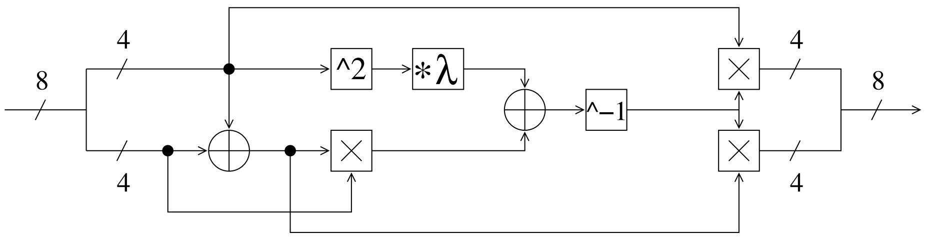

Fig. 1. Schematic of the S-box by Satoh *et al.* [2], where the building blocks are operations in $GF((2^2)^2)$, which are decomposed into operations in $GF(2^2)$

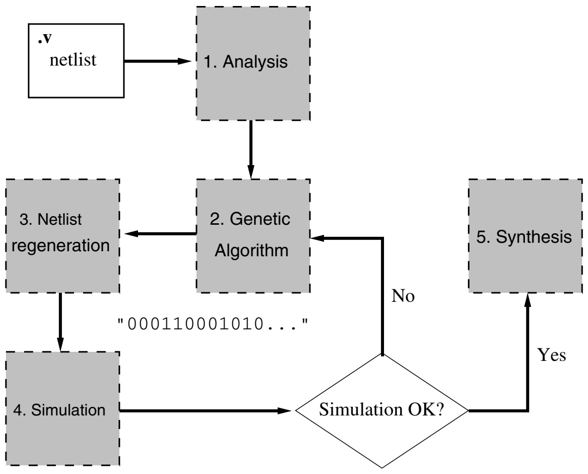

Fig. 2. Workflow of our approach for pipelining the AES S-box

This step provides us with different statistics of the target design such as the number of standard cells (referred to as elements), cell inputs and paths in the circuit. Then, an appropriate input for the GA is generated according to the delay of each element of the netlist. From the GA, we receive a certain arrangement of FFs that maximizes the throughput of the substitution box. The FFs are inserted in a new netlist, simulated and synthesized (Steps 3–5 in Figure 2). In the following sections, we first describe our optimization problem and then we continue on each step and relate our results using the design described in Section 3.

4.1 Pipelining as an Optimization Problem

The task at hand is to insert a combination of FFs in order to increase throughput through a certain number of pipeline stages. A valid solution to this problem requires that all the paths in the circuit contain the same number of FFs, which are placed in a way that minimizes the delay in each pipeline stage. For a given number of stages, the number of FFs at every path is one less than the number of stages (i.e. one FF per path in a two-stage pipeline).

To define this as an optimization problem, we encode each possible solution as a bitstring (a sequence of bits) where each bit represents every input location for all circuit elements. In this encoding, a bit is set to the value "0" if the corresponding input does not have an associated FF and to the value "1" if there is a FF preceding that input (the unmodified circuit is represented with all zeros). The total bitstring length is equal to the sum of all the inputs in the circuit, which in this case amounts to 432. A potential solution is therefore a sequence of bits of length 432, which defines a search space of 2^{432}.

The quality (also called the fitness value) of each potential solution is determined by the delay of the pipeline stage with the greatest delay among all the stages. However, since the optimization algorithm operates with any combination of bits in the search process, a great number of potential solutions are expected to be infeasible, because they will represent a circuit with a different number of FFs in all the paths. To handle that constraint, a penalty factor can be included in the quality estimate to differentiate between feasible and infeasible solutions. The penalty should be great enough such that each feasible solution (a circuit with the same number of FFs in every path), regardless of the delay, is still better than any infeasible solution, to guide the search to valid solutions.

Based on the previous, we define the following fitness function that represents the optimization problem:

$$fitness = max\_delay\_time + (1,000 * number\_invalid\_paths) \tag{2}$$

Here, *max_delay_time* presents the longest delay for every pipeline stage and invalid paths are all those that do not have a correct number of FFs. We experimentally set the weight to be 1,000 in the formula above. Intuitively, the weight needs to be large enough such that even in the case that there is only one invalid path, the total fitness should be worse than for the solution without FFs. For that same reason, every larger weight factor would work the same. The optimization objective is the minimization of the fitness function. Note that to calculate the maximum delay, all possible paths in the network need to be traversed, which poses a fairly large computational demand.

In the next section we describe the first step in Figure 2, focused on the analysis of netlists.

4.2 Analysis of Verilog Netlists

Our framework generates statistical information about a circuit represented as a Verilog netlist. In Table 1 we show the statistical details related to our choice of representation. These parameters are extracted using the framework we developed for pipelining the AES substitution box. The number of elements in the

Table 1. Statistics of the preliminary S-box design

Number of elements	Number of inputs	Number of paths	Shortest path	Critical path (ps)
165	432	8,023,409	4	3,884.52

table denotes the number of standard cells. The number of inputs refers to the number of inputs to all standard cells. Finally, the number of paths denotes the number of different possible paths through the circuit from an input to an output.

As depicted, there are too many possible paths to encode all of them into one solution. Since the total number of elements as well as the total number of inputs is relatively small, we decided to encode the possible solutions as bitstrings where each bit represents every input location for all elements.

Given a netlist of the S-box in Verilog, our framework first parses it according to a predefined grammar and then, each element from the UMC 0.13 μm low leakage standard cell library is identified [19]. This is done using a framework developed in `python 2.7.5-5` in combination with the `pyparsing 2.0.1` library[1].

Relying on that library, we have defined a grammar that deconstructs each entry of a Verilog netlist into a set of cells and their connections. For instance, a NAND gate defined in the standard cell library can appear in a given netlist as "`ND2CLD U181 ( .I1(\input [0]), .I2(\input [1]), .O(n1) );`". Hence, the parser must identify that element as a NAND gate (`ND2CLD`) associated to the `U181` identifier. Moreover, it must detect that is connected to the first two inputs of the S-box and that the `n1` wire routes its output.

This process is performed by creating a grammar that expects a set of entries consisting of:

– The type of the cell.
– The cell identifier.
– A comma-separated list of inputs and outputs (i.e. `I1`, `I2` and `O` in the example) connected to the circuit inputs, outputs or internal Verilog wires (i.e. `\input [0], \input [1]`, and `n1`).

Each element of the netlist is abstracted in a data structure that stores the cell type, the cell identifier, the number of inputs of the cell and all the elements that are connected to their inputs i.e. their adjacent elements. Moreover, the delay associated to each element according to the standard cell library is also stored. This information is later used as an input for the optimization algorithm.

The resulting list contains all the circuit cells together with their number of inputs and their adjacent nodes (that is, the cells that are connected to their inputs) as well as the delay of each element. A small example of the parser output is given below:

[1] http://pyparsing.wikispaces.com/

```
U163     2     U251 U248               146.8
U164     3     U198 U256 U163          86.471
U165     4     U198 U163 U256 U164     98.369
U166     1     U207                    59.39
U167     4     U207 U209 U210 U166     114.406
```

This example describes the number of inputs for the cells `U163-U167` (i.e. 2, 3, 4, 1, 4 respectively) together with the cells that are connected to those inputs and the respective delay of the cell according to the standard cell library.

These values are obtained as average values for all possible combinations (transitions from low to high and from high to low) for each element. For each FF element that will be inserted in order to maximize the throughput, we use a D-FF with a single output and no clear, set or enable (`QDFFCLD`) with an average delay time of 320.35 ps. All delay times are given for a temperature of 25 degrees Celsius, a core voltage of 1.2 V and a load capacitance of 1.5 fF.

Taking the average of low-to-high and high-to-low delays and assuming a low load capacitance of 1.5 fF is an approximation that gives good results. Nevertheless, to improve our methodology, the actual delay information based on the load of each standard cell should be taken into account.

Next, we present the optimization algorithm we used to generate pipelined circuits.

4.3 Genetic Algorithms

In accordance with the given representation, we selected genetic algorithms (GAs) as the optimization method to be used in our experiments.

Prior to going into to the details about genetic algorithms, first we offer a short rationale behind the choice of them. Since there is no previous work that uses any kind of heuristics to evolve the optimal arrangement of FF elements in a combinatorial circuit, we believe we should start with some well-researched algorithm that can be easily adapted.

Genetic algorithms are an evolutionary computation technique that has been successfully applied to various optimization problems. Additionally, bitstring representation is one of several standard representations of GAs [20]. Naturally, there are other heuristic algorithms that also use bitstring representation (e.g. Particle Swarm Optimization [21], Genetic Annealing [22]) that could be used here. In accordance with that, it is not possible to stipulate what algorithm would perform the best. The "No Free Lunch" theorem states that there is no single best algorithm for all the problems, i.e. when averaged over all search problems, all algorithms behave the same [23]. Therefore, only thorough experimental work can give insight into more appropriate algorithms. Further details about genetic algorithms and operators we use are given in Appendix B.

Common Parameters. The parameters used in each run of the algorithm are the following: the number of independent runs for each evolutionary experiment is 30 and the population size is 30. The tournament size k in the tournament selection is equal to 3. Mutation probability is set to 0.45 per individual where we choose it on a basis of a small set of tuning experiments where it showed the best results on average.

Further, our setting has one more important parameter that needs to be set i.e. the number of pipeline stages. With this parameter we control how many levels of FF elements we want in our circuit. From Table 1 we see that the number of elements in the shortest path is 4. Therefore, this path can have only 3 levels of FFs and that is the maximum number of FFs our circuit can have in order to produce a correct output.

Evolutionary Process. After the parameters are set, the GA starts with the generation of the initial population. In this part, the genetic algorithm reads all the elements of the parser output file and for each cell input it reserves one position in the bitstring representation. Notice that our bitstring size is fixed for a given circuit and it can not be dynamically changed during the evolutionary process. This results in the fact that our current setting does not support multiple FF elements one after the other. The initial population is built by creating random bitstrings of the designated length corresponding to randomly setting FFs in the preliminary netlist.

When the initial population is generated, the genetic algorithm starts with the evolution process. In each iteration it randomly chooses k possible solutions (the k tournament size) and eliminates the worst solution among those (this is also to ensure elitism i.e. the best solutions are always propagated to the next generation). The remaining solutions are used as parents which create one offspring via variation operators. The offspring (new solution) then replaces the worst individual in the population and additionally undergoes a mutation with a given probability.

For each offspring, a genetic operator is selected uniformly at random between all operators within an operator class (mutation or crossover). We use simple and mix mutations and uniform [24] and one-point [20,25] crossover operators. These variation operators are selected among those that are commonly used nowadays.

The evolution process repeats until the stopping criterion is met. In our case, the stopping criterion is based on 50 generations without improvement of the best solution.

4.4 Reconstructing Evolved Individuals to the Netlist

Using a list of structures described in Section 4.2 it is possible to compute all the paths of the circuit based on all the possible combinations for the eight inputs and outputs of the AES substitution box. This is done by transforming the list of cell structures described above into a non-directed graph, where the cells are represented by nodes and their connections by edges. Then, it is possible to extract the connections in the circuit and identify all the paths for all the input-output combinations using a graph exploration algorithm such as the

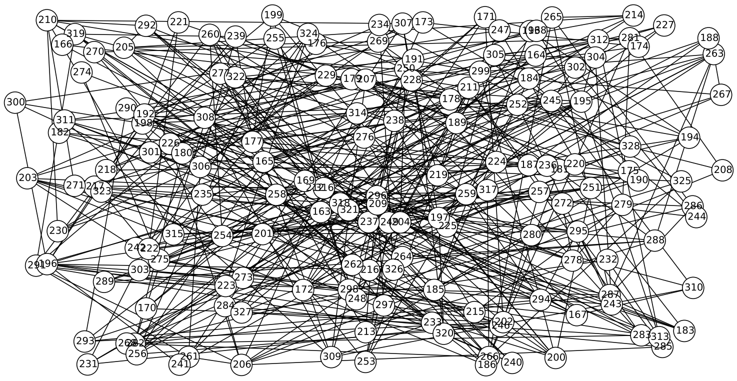

Fig. 3. Graph representation of the S-box connections for each cell identifier

breadth-first search (cf. [26]). We have depicted in Figure 3 how our framework abstracts a Verilog netlist.

From the GA, we obtain the precise arrangement of FFs that will be inserted in the new netlist in order to maximize its throughput as described in Section 4.1. Given the internal structure that we created from a Verilog netlist, now it is possible to reconstruct the circuit according to the output from the GA (Step 3 in Figure 2). Our framework first splits the binary string from the GA in different chunks according to the number of inputs of each element. Then, it associates the respective FFs to each cell input. Moreover, the required Verilog wires that connect each FF to the input/output of the cell are added. For instance, for an XOR gate with two inputs (e.g. `XOR2ELD`) and an output "`11`" from the GA, this element would be reconstructed with two FFs attached to their inputs using two wires (e.g. `ff_9_q, ff_10_q`) as:

```
QDFFCLD FF9 ( .CK(clk), .D(n180), .Q(ff_9_q) );
QDFFCLD FF10 ( .CK(clk), .D(n198), .Q(ff_10_q) );
XOR2ELD U212 ( .I1(ff_9_q), .I2(ff_10_q), .O(n201) );
```

Finally, a test bench with assertions for all the 256 possibilities of the S-box is created for the regenerated netlist. This is used in Mentor Graphics ModelSim 6.5c to guarantee the correctness of the new circuit. The resulting circuit is then synthesized using Synopsys Design Compiler in order to get pre-layout implementation results for the critical path delay and the area.

5 Results

In this section, we present our performance figures for the developed framework that analyzes and reconstructs Verilog netlist. We also show the synthesis results for our best candidates i.e. those substitution boxes that obtained the maximum throughput.

5.1 Analysis Framework

In our S-box design, based on 165 cells (Table 1), we required 0.465 seconds for generating the input for the GA using the described framework in a Intel Core i5-3230M CPU clocked at 2.60GHz (Step 1, Figure 2). The evaluation of one generation consisting of 100 individuals requires 3.5 min. on average (Step 2, Figure 2). All the experiments carried out with GAs were conducted in an Intel i5-3470 CPU equipped with 6 GB of RAM.

In Table 2 we give an overview of the total number of paths sorted into classes according to the maximum delay time. Each initial circuit represents a certain solution without FFs whereas the evolved circuit is the solution with the maximum delay time (2,793.62 as given in Table 3 for the best solution we found). Additionally, in Table 3 we present different statistics for several evolved circuits. Here, column Number of FFs represents the total number of flip-flops in evolved circuit and column Number of generations represents how long was GA running.

Table 2. Number of paths per length class

Class	Initial circuit	Evolved circuit
0 – 500	2	5,570
500 – 1,000	2,164	78,5432
1,000 – 1,500	149,944	3,751.897
1,500 – 2,000	2,026.442	2,639.751
2,000 – 2,500	3,580,150	816,636
2,500 – 3,000	1,899,675	26,411
3,000 – 3,500	361,708	0
3,500 – 4,000	3,324	0

Table 3. Statistics of evolved circuits

Max. delay time (ps)	Number of stages	Number of FFs	Number of generations
2,793.62	2	73	587
2,826.52	2	68	15
2,942.42	2	66	691
3,155.11	2	49	482
3,223.02	2	64	4452
3,247.64	2	42	1434
2,918.92	3	100	618

5.2 Synthesis

As can be seen from Table 3, the best solution the GA finds when dividing the circuit into two stages (i.e. inserting one layer of pipelining FFs) has an estimated critical path of 2,793.62 ns. The best solution with two layers of pipelining FFs to have the shortest critical path is 2,918.92 ns. Intuitively, we would expect the solution with two layers of pipelining to have the shortest critical path. However, because the number of possible solutions is much bigger for a 3-stage circuit than for a 2-stage circuit, the optimal solution found by the GA for the 3-stage circuit is worse than the optimal solution it finds for the 2-stage circuit. Nevertheless, we know that there should exist a better solution for 3 than for 2 when a longer search is performed.

We synthesized both solutions, resulting in Table 4. In order to evaluate the critical path properly, we inserted flip-flops at the inputs and outputs of all the S-boxes. As mentioned before, the implementation corresponds to the state-of-the-art design of Mentens et al. described in [3]. The netlist with 1 stage only contains these input and output flip-flops. The netlists with 2 and 3 stages contains 1 and 2 layers of pipelining flip-flops, respectively. Because of these input and output flip-flops, the netlist with only one stage is larger in area than the composite field S-boxes reported in literature (they do not contain any flip-flops). The table shows that the 2-stage S-box introduces a 50 % improvement in throughput, which is equal to the number of bits at the output (8 in our case) divided by the delay of the critical path. The increase in area is only 18 %. The synthesis results for the critical path are even slightly better than the estimate of the GA. The reason is that the synthesis tool optimizes the generated pipelined netlist again, which leads to further improvements. For the 3-stage S-box, the synthesis results are worse than the estimate of the GA. This is probably due to the fact that there is less room for optimization with two layers of pipelining flip-flops and thus less logic in between the layers.

Table 4. Pre-layout synthesis results of the netlist with 1, 2 and 3 stages.

Number of stages	Critical path (ns)	Throughput (Gbits/s)	Area (μm^2)	Gate count
1	3.9	2.05	2,450	612.50
2	2.6	3.07	2,901	725.25
3	3.2	2.50	3,433	858.25

6 Conclusion

This paper presents a methodology for pipelining composite field AES S-boxes to maximize the throughput using genetic algorithms. The best trade-off between throughput and area results in a throughput of 3.07 Gbits/s and an area of 2,901 μm^2 in a UMC 0.13 μm standard cell library. This comes down to a throughput increase of 50 % with an area overhead of 18 % in comparison to an S-box without pipelining. In order to improve the throughput even more, the design space should be increased with more composite field representations. The GA could also still be optimized, e.g. by making a more intelligent choice of the seed.

Appendix A

Here we give the details of the constants used for our composite field implementations of the AES S-box.

In [2], Satoh *et al.* made the following choices for the coefficients of the irreducible polynomials:

$$\begin{aligned} p_1 &= 1 = \{0001\}_2 \,, \\ p_0 &= \lambda = \omega y = (z+1)y = \{1100\}_2 \,, \\ q_1 &= 1 = \{01\}_2 \,, \\ q_0 &= \phi = z = \{10\}_2 \,. \end{aligned}$$

The inverse operation is implemented as

$$\begin{aligned} &\Delta = \delta_1 x + \delta_0 \in GF(((2^2)^2)^2) : \\ &\quad \Delta^{-1} = (\delta_1 x + (\delta_1 + \delta_0)) \cdot (\lambda \delta_1^2 + (\delta_1 + \delta_0)\delta_0)^{-1} \,, \\ &\delta = d_1 y + d_0 \in GF((2^2)^2) : \\ &\quad \delta^{-1} = (d_1 y + (d_1 + d_0)) \cdot (\phi d_1^2 + (d_1 + d_0) d_0)^{-1} \,. \end{aligned} \tag{3}$$

Inversion in $GF(2^2)$ requires only one addition:

$$d = t_1 z + t_0 \in GF(2^2) : d^{-1} = t_1 z + (t_1 + t_0) \,. \tag{4}$$

Appendix B

Genetic algorithms belong to the evolutionary family of algorithms where the elements of the search space S are arrays of elementary type [27]. We give a short pseudo-code for a genetic algorithm (this is also a pseudo-code for any evolutionary algorithm) in Algorithm 1.

In order to produce new individuals (solutions), the GA uses mutation and crossover operators. Mutation operators use one parent to create one child by applying randomized changes to the parent. The mutation depends on the mutation rate p_m which determines the probability that a change will occur within an individual. Crossover operators modify two or more parents in order to create an offspring via the information contained within parent solutions. Recombination is usually applied probabilistically according to a crossover rate p_c. In this work, we use only operators that work with two parents. Additionally, GAs use selection methods to choose the individuals that will continue to the next generation. We opted here for the steady-state tournament or k-tournament selection method [27]. In this selection from k randomly selected individuals, two with the best fitness values are chosen to evolve and create one offspring, replacing the worst from the tournament [25,28].

Next, we give a short description of crossover and mutation operators that we use.

Algorithm 1. Genetic algorithm

```
Input : Parameters of the algorithm
Output : Optimal solution set
$t \leftarrow 0$
$P(0) \leftarrow CreateInitialPopulation$
while $TerminationCriterion$ do
    $t \leftarrow t + 1$
    $P'(t) \leftarrow SelectMechanism\ (P(t-1))$
    $P(t) \leftarrow VariationOperators(P'(t))$
end while
Return $OptimalSet(P)$
```

One Point Crossover. When performing one point crossover, both parents are split at the same randomly determined crossover point. Subsequently, a new child genotype is created by appending the first part of the first parent with the second part of the second parent [20,25].

Uniform Crossover. Single and multi-point crossover defines cross points as places between positions where an individual can be split. Uniform crossover generalizes this scheme to make every place a potential crossover point. A crossover mask, the same length as the individual structure is created at random and the parity of the bits in the mask indicate which parent will supply the offspring with which bits. The number of effective crossing points in uniform crossover is not fixed, but will average to $l/2$ where l represents string length.

Mix Mutation. Mix (or mixing) mutation randomly chooses one area inside the individual where it will change the bits. First, in that area number of ones and zeros is counted and then random bits are set while preserving the respective number of values [20].

Simple Mutation. In simple mutation every bit is inverted with a predefined mutation probability p_m [20].

References

1. Fischer, V., Drutarovský, M.: Two Methods of Rijndael Implementation in Reconfigurable Hardware. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 77–92. Springer, Heidelberg (2001)
2. Satoh, A., Morioka, S., Takano, K., Munetoh, S.: A compact rijndael hardware architecture with s-box optimization. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 239–254. Springer, Heidelberg (2001)
3. Mentens, N., Batina, L., Preneel, B., Verbauwhede, I.: A systematic evaluation of compact hardware implementations for the Rijndael S-BOX. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 323–333. Springer, Heidelberg (2005)

4. Canright, D.: A very compact s-box for AES. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 441–455. Springer, Heidelberg (2005)
5. Fischer, V., Drutarovský, M., Chodowiec, P., Gramain, F.: InvMixColumn decomposition and multilevel resource sharing in AES implementations. IEEE Trans. VLSI Syst. **13**(8), 989–992 (2005)
6. Chodowiec, P., Gaj, K.: Very Compact FPGA Implementation of the AES Algorithm. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 319–333. Springer, Heidelberg (2003)
7. Wolkerstorfer, J., Oswald, E., Lamberger, M.: An ASIC implementation of the AES sboxes. In: Preneel, B. (ed.) CT-RSA 2002. LNCS, vol. 2271, pp. 67–78. Springer, Heidelberg (2002)
8. 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 2011. LNCS, vol. 6632, pp. 69–88. Springer, Heidelberg (2011)
9. Hodjat, A., Verbauwhede, I.: Area-throughput trade-offs for fully pipelined 30 to 70 gbits/s AES processors. IEEE Trans. Computers **55**(4), 366–372 (2006)
10. Kumar, S., Sharma, V., Mahapatra, K.: Low latency VLSI architecture of S-box for AES encryption. In: 2013 International Conference on Circuits, Power and Computing Technologies (ICCPCT), pp. 694–698 (March 2013)
11. Jhajharia, S., Mishra, S., Bali, S.: Public key cryptography using neural networks and genetic algorithms. In: Parashar, M., Zomaya, A.Y., Chen, J., Cao, J., Bouvry, P., Prasad, S.K. (eds.) IC3, pp. 137–142. IEEE (2013)
12. Sokouti, M., Sokouti, B., Pashazadeh, S., Feizi-Derakhshi, M.R., Haghipour, S.: Genetic-based random key generator (GRKG): a new method for generating more-random keys for one-time pad cryptosystem. Neural Computing and Applications **22**(7–8), 1667–1675 (2013)
13. Zarza, L., Pegueroles, J., Soriano, M., Martínez, R.: Design of cryptographic protocols by means of genetic algorithms techniques. In: Malek, M., Fernández-Medina, E., Hernando, J. (eds.) SECRYPT, pp. 316–319. INSTICC Press (2006)
14. Park, K., Hong, C.: Cryptographic protocol design concept with genetic algorithms. In: Khosla, R., Howlett, R.J., Jain, L.C. (eds.) KES 2005. LNCS (LNAI), vol. 3682, pp. 483–489. Springer, Heidelberg (2005)
15. Carpi, R.B., Picek, S., Batina, L., Menarini, F., Jakobovic, D., Golub, M.: Glitch it if you can: parameter search strategies for successful fault injection. In: Francillon, A., Rohatgi, P. (eds.) CARDIS 2013. LNCS, vol. 8419, pp. 236–252. Springer, Heidelberg (2014)
16. Clark, J.A., Jacob, J.L., Stepney, S.: The design of S-boxes by simulated annealing. New Generation Computing **23**(3), 219–231 (2005)
17. 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. Proceedings of the First Workshop on Cryptography and Security in Computing Systems, CS2 2014, pp. 13–18. ACM, New York (2014)
18. 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, pp. 80–83 (2014)
19. Faraday: Faraday Cell Library 0.13 μm Standard Cell (2004)
20. Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. The MIT Press, Cambridge (1992)

21. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Proceedings of the IEEE International Conference on Neural Networks, vol. 4, pp. 1942–1948 (November 1995)
22. Yao, X.: Optimization by genetic annealing. In: Proc. of 2nd Australian Conf. on Neural Networks, pp. 94–97 (1991)
23. Wolpert, D.H., Macready, W.G.: No Free Lunch Theorems for Optimization. IEEE Transactions on Evolutionary Computation **1**(1), 67–82 (1997)
24. Syswerda, G.: Uniform crossover in genetic algorithms. In: Proceedings of the 3rd International Conference on Genetic Algorithms, pp. 2–9. Morgan Kaufmann Publishers Inc., San Francisco (1989)
25. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer, Heidelberg (2003)
26. Knuth, D.E.: The Art of Computer Programming. Fundamental Algorithms, 3rd edn., vol. 1. Addison Wesley Longman Publishing Co. Inc., Redwood City (1997)
27. Weise, T.: Global Optimization Algorithms Theory and Application (2009)
28. Michalewicz, Z.: Genetic algorithms + data structures = evolution programs, 3rd edn. Springer, London (1996)