IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 2022, No. 3, pp. 403–447. DOI:10.46586/tosc.v2022.i3.403-447

Low-Latency Boolean Functions and Bijective S-boxes

Shahram Rasoolzadeh

Radboud University, Nijmegen, The Netherlands
firstname.lastname@ru.nl

Abstract. In this paper, we study the gate depth complexity of (vectorial) Boolean functions in the basis of {NAND, NOR, INV} as a new metric, called *latency complexity*, to mathematically measure the latency of Boolean functions. We present efficient algorithms to find all Boolean functions with low-latency complexity, or to determine the latency complexity of the (vectorial) Boolean functions, and to find all the circuits with the minimum latency complexity for a given Boolean function. Then, we present another algorithm to build bijective S-boxes with low-latency complexity which with respect to the computation cost, this algorithm overcomes the previous methods of building S-boxes.

As a result, for latency complexity 3, we present n-bit S-boxes of $3 \leq n \leq 8$ with linearity 2^{n-1} and uniformity 2^{n-2} (except for 5-bit S-boxes for whose the minimum achievable uniformity is 6). Besides, for latency complexity 4, we present several n-bit S-boxes of $5 \leq n < 8$ with linearity 2^{n-2} and uniformity 2^{n-4}.

Keywords: S-box · low-latency · gate depth complexity

1 Introduction

Studying properties of all n-bit Boolean functions is not an easy task when $n > 5$. One of the hardware properties of (vectorial) Boolean functions to study is their latency. Finding the minimum latency for hardware implementation of a Boolean function with a synthesizer is not possible if the number of inputs is high and if we only use the truth table of the function. One approach to achieve a lower latency is providing a gate-level netlist for the synthesizer to implement the Boolean function. However, finding a gate-level netlist that provides the lowest latency for implementing a Boolean function is usually not an easy task when the number of input bits to the functions is high (even for 5-bit S-boxes). Typically, *gate depth complexity* defined as the minimum length of the longest path from an input bit to an output bit within all possible implementations of a (vectorial) Boolean function, is considered to mathematically model the lowest latency for implementing that function. But, as a result of [BMP08], determining the gate depth complexity of a Boolean function is an NP-hard problem and consequently finding low-latency implementation of (larger) Boolean functions stays challenging.

Furthermore, the latency metric of a circuit is practically quite complicated, since different gates have different delays and it is dependent on several parameters that vary under different operating conditions (such as driving power, voltage, and temperature) and more importantly under different technologies that the circuit will be implemented. Even if the parameters are optimized for achieving the low-latency implementation, the latency of different logic gates covers a wide range. Therefore, it is not realistic to consider the gate depth complexity alone as a metric for the minimum latency of a Boolean function.

Received: 2022-06-01 Accepted: 2022-08-01 Published: 2022-09-09

1.1 Our Contributions

In this paper, in Section 3, we use a new metric to have a model that is closer to reality than in the case of gate depth complexity. We call it *latency complexity* which is the gate depth complexity in the basis of $\{\texttt{NAND}, \texttt{NOR}, \texttt{INV}\}$ (see Definition 8). We present a unique structure (circuit) to model the implementation of any Boolean function with the latency complexity of d and we show that the latency complexity is invariant under the extended bit permutation equivalence.

In Section 4, we study the latency complexity of (vectorial) Boolean functions. We first show that the latency complexity of (vectorial) Boolean functions stays invariant over the extended bit permutation equivalence. Then we discuss the computation cost of the search for determining the latency complexity of a given Boolean function and present several techniques to make it faster. Applying these techniques, we compute the latency complexity of all n-bit Boolean functions for up to $n = 5$. Besides, in Subsection 4.3 we present an algorithm to find all Boolean functions with low-latency complexity ($d \leq 4$) for $n \in \{6, 7, 8\}$. Furthermore, we explain several speed-up techniques on the search for computing the latency complexity of a given Boolean function. We present another algorithm to find all possible structures to implement a Boolean function with a gate depth of the same as its latency complexity. We use this algorithm to determine the latency complexity of the previously known S-boxes in symmetric cryptography.

By applying the low-latency Boolean functions, we look for the existence of bijective n-bit S-boxes with a given latency complexity and study their cryptographic properties (precisely, their linearity, uniformity, and algebraic normal form degree) in Section 5. While the previous algorithm for building S-boxes [Can07, MB19] is useful for classifying n- to m-bit Boolean functions under linear or affine equivalences, for our need (which uses the extended bit permutation equivalence) it is not efficient. Thereby, we introduce a new algorithm for building S-boxes that with respect to the computation cost, our method overcomes the previous method. As a result, for latency complexity 3, we present n-bit S-boxes of $3 \leq n \leq 8$ with linearity 2^{n-1} and uniformity 2^{n-2} (except for 5-bit S-boxes that the minimum achievable uniformity is 6). Besides, we find several n-bit S-boxes of $5 \leq n < 8$ with latency complexity 4, linearity 2^{n-2}, and uniformity 2^{n-4}.

In Section 6, we describe an approach to optimize the suggested structures produced by our algorithm to find a circuit with the lowest latency in a real ASIC hardware implementation. We apply our approach to find efficient implementations for previously known and newly introduced S-boxes that are minimized with respect to the latency and then its area.

All the results for latency complexity of Boolean functions and S-boxes are published publicly at `https://gitlab.science.ru.nl/shahramr/LowLatencySBoxes.git`.

1.2 Related Works

Designing cryptographic primitives with minimum a low latency in hardware is still a young and emergent research topic. First work in this area was [KNR12] by Knezevic, Nikov and Rombouts that compared the latency properties of multiple (lightweight) block ciphers. Immediately after, the first dedicated low-latency block cipher called `PRINCE` introduced by Borghoff et al. [BCG+12]. Designing low-latency primitives continued with block cipher `QARMA` by Avanzi [Ava17], `Gimli`: a high performance cross-platform cryptographic permutation by Bernstein et al. [BKL+17], `PRINCEv2`: an updated version of `PRINCE`, by Bozilov et al. [BEK+20], `Orthros`: a pseudorandom function (PRF) by Banik et al. [BIL+21], and `SPEEDY`: a family of block ciphers by Leander et al. [LMMR21].

There are also some works focusing on the latency of some particular cryptographic building blocks only. For instance, in [BFP19], Boyar, Find and Peralta present some techniques for finding small low-depth circuits for cryptographic functions [BFP19]. In

[LSL+19], Li et al. show how to construct involutory low-latency Maximal Distance Separable (MDS) matrices.

A recent work in the direction of determining circuit complexity of functions is [Sto16] that presents a SAT-solver-based technique to optimize the implementations of some S-boxes with respect to different criteria such as the gate depth complexity. While this technique gives *one* solution for implementing small S-boxes, it is unable to find the complexity in some cases, especially when the input size of the S-box is larger than 5.

In [BCBP03], Biryukov et al. presented an efficient algorithm to check if two functions are equivalent and another algorithm to find the representative in the linear equivalence or the affine equivalence class. Later in [MB19], De Meyer and Bilgin improved the algorithm for mappings of n- to m-bit with $m < n$. Since we will deal with the extended bit permutation equivalence and not the linear or affine equivalence, we modify these algorithms according to the properties of extended bit permutation equivalence. While these algorithms perform efficiently for linear or affine equivalence, we experienced that they are not suitable for (extended) bit permutation equivalence. Details of the modification and our solution for these problems are explained later.

In [BMD+20], Bilgin et al. present techniques to construct S-boxes with a low-latency masked variants for applying in the side-channel countermeasures that basically requires a low multiplicative depth and gate complexities. However, this is not directly related to the development of low-latency symmetric primitives in general, as the requirements are quite different and sometimes even direct opposites. While in regular cryptographic S-boxes, non-linear gates are beneficial for area and latency over the linear gates, in masked S-boxes on the other hand, linear operations are optimal and non-linear gates are the primary cost factor [BMD+20].

2 Basics and Notations

In this section, we introduce the necessary basics related to Boolean functions and their implementations based on the logic circuits. We assume that the reader has some, but not necessarily extensive, familiarity with these concepts.

We use $\mathbb{Z}_n$ to denote the finite set $\{0, 1, \ldots, n-1\}$, that is the set of non-negative integers smaller than n. By $\mathbb{F}_2$, we denote the finite field of two elements, $\{\mathtt{0}, \mathtt{1}\}$, and call it the *binary field* where the addition of this field is denoted by $\oplus$ and called *XOR*. By $\mathbb{F}_2^n$ with n being a positive integer, we denote the binary vector space of dimension n and call it the *space of n-bit vectors*.

Let $a \in \mathbb{F}_2^n$, then by $a[i]$ with $i \in \mathbb{Z}_n$, we denote the i-th element of a, i.e., $a = (a[0], \ldots, a[n-1])$. Note that in this paper, we always count starting from 0. Let $a, b \in \mathbb{F}_2^n$ be two n-bit binary vectors. We use $\langle a, b\rangle$ to denote the inner product between a and b which is defined as $\langle a, b\rangle = \bigoplus_{i=0}^{n-1} a[i]b[i]$. Also, by $\mathrm{hw}(a)$, we denote $\sum_{i=0}^{n-1} a[i]$ that is called the *Hamming weight* of a. To denote concatenation of two vectors $a \in \mathbb{F}_2^n$ and $b \in \mathbb{F}_2^m$, we use $(a\|b)$ that is $(a[0], \ldots, a[n-1], b[0], \ldots, b[m-1])$.

To make it easier and space-efficient to display a binary vector, for $a \in \mathbb{F}_2^n$, instead of displaying its all binary elements, we show it by its corresponding integer value in $\mathbb{Z}_{2^n}$. Precisely, we use the simple mapping of elements in $\mathbb{F}_2^n$ to the elements in $\mathbb{Z}_{2^n}$ that maps any $a \in \mathbb{F}_2^n$ to $\sum_{i=0}^{n-1} a[i] \cdot 2^{n-i-1}$.

2.1 Boolean Functions

The functions from the vector space $\mathbb{F}_2^n$ to the binary field $\mathbb{F}_2$ are called *Boolean functions* with n-variables or simply n-bit Boolean functions. The number of n-bit Boolean functions is 2^{2^n}, and this number is too large to study the properties of each n-bit Boolean function when $n > 5$. For this reason, determining and studying those Boolean functions satisfying the target conditions is not feasible through an exhaustive computer search.[1] Therefore, it is necessary to find solutions that make it easier to study properties of Boolean functions or find Boolean functions satisfying the target properties. In the following, we briefly explain the necessary terms and notations of Boolean functions used in this paper.

The *truth table* is the most basic way to represent a Boolean function. Let f be an n-bit Boolean function, then the truth table of f is a binary vector $T_f \in \mathbb{F}_2^{2^n}$ such that for any $x \in \mathbb{F}_2^n$, $T_f[x]$ shows the value of $f(x)$. Among the other classical representations of Boolean functions, the one most often used in cryptography is the *algebraic normal form (ANF)* which is the n-variable polynomial representation over $\mathbb{F}_2$ of the form

$$f(x) = \bigoplus_{I \in \mathbb{F}_2^n} a_I x^I = \bigoplus_{I \in \mathbb{F}_2^n} a_I \Big(\prod_{i=0}^{n-1} x_i^{I[i]} \Big),$$

where x_i is the variable corresponding to the i-th bit of x, i.e., $x[i]$. Also, x^I denotes the monomial $x_0^{I[0]} \cdots x_{n-1}^{I[n-1]}$, i.e., the corresponding monomial for x_i variables with $I[i] = \mathtt{1}$. Note that each a_I is a binary value and every coordinate x_i appears in this polynomial with exponents at most 1. It is well-known that the ANF representation is unique[2] and can be computed for the given truth table with a complexity of $n \cdot 2^n$ operations.

The Hamming weight of a Boolean function's truth table is called its *weight* which is the number of $x \in \mathbb{F}_2^n$ with $f(x) = \mathtt{1}$. *Balanced Boolean functions* are the ones whose weight is equal to 2^{n-1}, i.e., for half of $x \in \mathbb{F}_2^n$, it maps to $\mathtt{1}$, and for the other half, it maps to $\mathtt{0}$.

Definition 1 (Algebraic Degree)**.** For an n-bit Boolean function f, the algebraic degree is the maximum Hamming weight of all occurring monomials in the ANF representation of a function which we denote by $\deg(f)$, i.e.,

$$\deg(f) = \max_{I \in \mathbb{F}_2^n,\, a_I = \mathtt{1}} \mathrm{hw}(I).$$

Linear Boolean functions are those Boolean functions for which, for any $a, b \in \mathbb{F}_2^n$, we have $f(a \oplus b) = f(a) \oplus f(b)$. Each n-bit linear Boolean function can be represented as $\ell_\alpha(x) = \langle \alpha, x \rangle$ with corresponding $\alpha \in \mathbb{F}_2^n$. Note that each of these functions with $\alpha \neq 0$ are balanced.

Separating Boolean functions by their algebraic degree, the ones with degree one, two, or three are called *affine*, *quadratic*, and *cubic* functions, respectively. Affine functions, which are the extension of linear functions by a constant XOR in the output, can be displayed as $\langle \alpha, x \rangle \oplus c$ with corresponding $\alpha \in \mathbb{F}_2^n$ and $c \in \mathbb{F}_2$.

2.2 Vectorial Boolean Function

While Boolean functions map n-bit vectors to a one-bit value, *vectorial Boolean functions* map n-bit vectors to m-bit vectors. To specify the input and output bit size of these functions, we call them n- to m-bit vectorial Boolean functions, and when the input and output bit sizes are the same, we simply call them n-bit vectorial Boolean functions. Clearly, the vectorial Boolean functions include the Boolean functions which correspond to $m = 1$. In cryptography, vectorial Boolean functions are usually called *S-boxes* which provide confusion in the cipher. The S-boxes play a primary role in the key-alternating block ciphers, especially in the Substitution-Permutation-Network (SPN) ones.

[1] Consider determining if a 6-bit Boolean function has the target properties needs one millisecond (10^{-6} seconds). Then, it needs about 2^{44} seconds (about half a million years) to visit all the 6-bit Boolean functions.

[2] Precisely, the ANF polynomial in $\mathbb{F}_2[x_0, \ldots, x_{n-1}]/(x_0^2 - x_0, \ldots, x_{n-1}^2 - x_{n-1})$ is unique.

Let F be an n- to m-bit Boolean function, then the Boolean functions $f_0, \dots, f_{m-1}$ defined by $F(x) = \big(f_0(x), \dots, f_{m-1}(x)\big)$ for every $x \in \mathbb{F}_2^n$ are called *coordinate functions* of F. Also, for every non-zero $\alpha \in \mathbb{F}_2^m$, the Boolean function $x \mapsto \langle \alpha, F(x)\rangle$ is called a *component function* of F, and we denote this function by $\langle \alpha, F(x)\rangle$. In this paper, to denote the truth table of F easily, we use an array of 2^n elements in $\mathbb{Z}_{2^m}$, i.e., $\big(F(0), \dots, F(2^n-1)\big)$.

As for Boolean functions, the property of balancedness plays a crucial role in vectorial Boolean functions. An n- to m-bit Boolean function F is called balanced if it takes every value of $\mathbb{F}_2^m$ the same number of times, i.e., 2^{n-m} times. The balanced n-bit vectorial Boolean functions are the permutations on $\mathbb{F}_2^n$. It is well-known in the literature that an n- to m-bit Boolean function F is balanced if and only if its all component functions are balanced (for a proof see, e.g., [Car21]).

The algebraic degree of F is the maximum of the algebraic degrees of all coordinate functions. Hence, we use the same definition for linear, affine, quadratic, and cubic functions of the vectorial Boolean functions.

Definition 2 (Linearity and Differential Uniformity)**.** For a vectorial Boolean function $F : \mathbb{F}_2^n \to \mathbb{F}_2^m$, the *linearity* and *differential uniformity* are defined as

$$\mathrm{lin}(F) = \max_{\alpha \in \mathbb{F}_2^n,\, \beta \in \mathbb{F}_2^m \setminus \{0\}} \big|\#\{x \in \mathbb{F}_2^n \mid \langle \alpha, x\rangle = \langle \beta, F(x)\rangle\} - \#\{x \in \mathbb{F}_2^n \mid \langle \alpha, x\rangle \neq \langle \beta, F(x)\rangle\}\big|,$$

$$\mathrm{uni}(F) = \max_{\alpha \in \mathbb{F}_2^n \setminus \{0\},\, \beta \in \mathbb{F}_2^m} \#\{x \in \mathbb{F}_2^n \mid F(x) \oplus F(x \oplus \alpha) = \beta\}.$$

Definition 3 (Full-Dependency)**.** Let $F : \mathbb{F}_2^n \to \mathbb{F}_2^m$ be an n-bit to m-bit vectorial Boolean function: $F(x_0, \dots, x_{n-1}) = \big(f_0(x_0, \dots, x_{n-1}), \dots, f_{m-1}(x_0, \dots, x_{n-1})\big)$. We call F is *full-dependent* if each of its coordinate functions f_i is dependent on all the input variables, i.e., on all the x_i variables.

2.3 Equivalences

To study properties of vectorial Boolean functions, it is sometimes easier to partition them by a defined equivalence relation for which the studied properties are invariant. *Affine equivalence* and *extended affine equivalence* are the most applied ones in studying vectorial Boolean functions.

Definition 4 (Linear, Affine, and Bit Permutation Equivalences)**.** Two n- to m-bit Boolean functions F and G are called *linear equivalent* if there exist an n- to n-bit linear bijection L_{in} and an m- to m-bit linear bijection L_{out} in such a way that $F = L_{out} \circ G \circ L_{in}$.

In the same way, *affine equivalence* and *bit permutation equivalence* are defined. F and G are called affine equivalent if there exist an n- to n-bit affine bijection A_{in} and an m- to m-bit affine bijection A_{out} in such a way that $F = A_{out} \circ G \circ A_{in}$; and they are called bit permutation equivalent, if there exist a bit permutation of n bits P_{in} and a bit permutation of m bits P_{out} in such a way that $F = P_{out} \circ G \circ P_{in}$. Note that the n-bit vectorial Boolean function P is called bit permutation of n bits, if it maps $(x[0], \dots, x[n-1])$ to $\big(x[\pi(0)], \dots, x[\pi(n-1)]\big)$ where π is a permutation of $\mathbb{Z}_n$.

It is clear that if F and G are bit permutation equivalent, then they are also linear and affine equivalent. Besides, if they are linear equivalent, then they are also affine equivalent.

Similar to extending the linear equivalence to the affine equivalence, it is possible to extend the bit permutation equivalence to what is called *extended bit permutation equivalence* [LP07]. This equivalence is the most important one for our work in this paper.

Definition 5 (Extended Bit Permutation Equivalence)**.** Two n- to m-bit Boolean functions F and G are called *extended bit permutation equivalent* if there exist a bit permutation of n bits P_{in}, a bit permutation of m bits P_{out}, $\alpha \in \mathbb{F}_2^n$, and $\beta \in \mathbb{F}_2^m$ in such a way that $G(x) = \big(P_{out} \circ G \circ P_{in}(x \oplus \alpha)\big) \oplus \beta$ for all $x \in \mathbb{F}_2^n$.

Note that the affine equivalence covers all the other above-mentioned equivalences. The algebraic degree, linearity and uniformity are example properties of (vectorial) Boolean functions that are invariant over any of these equivalences.

It is common to consider the lexicographically smallest function in an equivalence class as the *representative* one. In this paper, we also use the same definition for representatives.

In [BCBP03], Biryukov, De Cannière, Braeken and Preneel presented an efficient algorithm to check if two functions are equivalent together with another algorithm to find the representative in the linear or in the affine equivalence class. Later in [MB19], De Meyer and Bilgin improved the algorithm for mappings of n- to m-bit with $m < n$. Since in this paper, we deal with the extended bit permutation equivalence and not the linear or affine equivalence, we modify these algorithms according to the properties of extended bit permutation equivalence. While these algorithms perform efficiently for the case of linear and affine equivalences, we experienced that they are not suitable for (extended) bit permutation equivalence. We discuss the details about this problem later in Subsection 5.1.

2.4 Implementation of Boolean Functions

To mathematically model the costs for implementing a Boolean circuit for some specific applications, some terms are defined that are known as the *complexity* of the implementation. In the following, we bring the general definition of these complexities. In all of these definitions, we consider that $\mathcal{G}$ is the set of all allowed gates to use, which is usually called the *basis* of implementation, e.g., all the gates with fan-in number of at most two.[3] The basis must provide *completeness property* with the meaning that based on the given type of gates in this basis, it is possible to build any (vectorial) Boolean function. The most common basis is $\mathcal{G} = \{\texttt{XOR}, \texttt{AND}, \texttt{INV}\}$; but, since `XOR` itself can be realized based on three `AND` and two `INV` gates, $\mathcal{G}' = \{\texttt{AND}, \texttt{INV}\}$ is also a complete basis.

Definition 6 (Gate Count Complexity)**.** It is the smallest number of gates required to compute the function while type of each used gate must be included in $\mathcal{G}$.

Even though different types of gates have different implementation (area) costs, this definition is typically considered the first simplified estimation for the minimum area cost for hardware implementation of a function. To achieve a reasonable estimation of the area cost, it is common to consider only the gates with a fan-in number of one or two.

Definition 7 (Gate Depth Complexity)**.** It is the minimum value for the longest path (concerning the number of gates used in the path) from any input to any output for implementing the function. Note that type of each used gate must be included in $\mathcal{G}$.

It is clear from its definition that the gate depth complexity of a vectorial Boolean function is the maximum of gate depth complexity for each of its coordinate Boolean functions.

Note that if any gate with any fan-in number is allowed, then every function can be implemented by a circuit with gate depth at most 3, e.g., by using conjunctive or disjunctive normal form expression of the function in which there are one `AND`, one `OR`, and probably one `INV` gates in each path from any input to any output. However, this can lead to a $\mathcal{G}$ that is usually not available in practice. Again, it is typical to use only gates with a fan-in of at most 2.

Similar to the gate count complexity, in the case of the gate depth complexity, even though different types of gates have different implementation (delay) costs, this definition is usually considered the first estimation for the minimum delay cost for hardware implementation of a function. But, as we discuss later in detail, this oversimplified metric does not provide an appropriate estimation for the delay cost. It is mainly because of the wide

[3]By fan-in or fan-out numbers of the gate, we mean the number of input bits or output bits in the gate.

range of delays of different types of gates (even with the same fan-in and fan-out numbers). Therefore, it needs a modification to be used as a close estimation for the delay cost of implementing a function.

Generally, any costs for hardware implementation of (vectorial) Boolean functions are invariant over bit permutation equivalence. This is because the bit permutations are realized by wiring, which means it costs a negligible value at most. Therefore, any two (vectorial) Boolean functions that are different only by a permutation of the input bits and a permutation of the output bits have the same implementation cost and the same hardware complexity in practice.

Moreover, since the cost of an inverter gate (shown by `INV` and in some literature is called the `NOT` operation) is comparably smaller than any other gates, it is reasonable to consider that the cost of implementing two (vectorial) Boolean functions which are different with a constant addition in the input or the output, is not very much different. Another reason for this consideration is that each (vectorial) Boolean function is implemented as a combinatorial circuit. Its input and output wires are usually connected to other combinatorial circuits. Hence, the `INV` gates in the input or output bits can be combined with the gates in the previous or the following combinatorial circuits. Note that if there is a layer of registers or a layer of buffers, such as in the round based or unrolled implementation architectures, respectively, the `INV` gates in the input and the output of a combinatorial circuit can be combined with the registers or the buffers (by changing the `BUF` gate to an `INV` gate). Therefore, by accepting a small tolerance, it is usually considered that the hardware implementation costs are invariant over the extended bit permutation equivalence.

As a result of [BMP08], we know that determining any of these complexities for a Boolean function is considered to be an NP-hard problem. The SAT-solver-based tool by Stoffelen [Sto16] finds a *single* solution for implementing small S-boxes. But, notice that it does not provide all the solutions for implementing the S-box, or it cannot find the complexity in the case when S-box size is larger than 5.

To use functionality of the gates in the equations, we use the signs $\wedge, \vee, \overline{\wedge}$, and $\overline{\vee}$ to denote the operation of the `AND`, `OR`, `NAND`, and `NOR` gates, respectively. Besides, we use $\neg f$ to denote the inverted value for the output of a Boolean function f, and we use $\overline{x}$ to indicate the inverted value of the input x. Moreover, for simplicity, from now on, we do not mention fan-in number of the gates, unless it is more than 2.

3 Latency Complexity of Boolean Functions

Due to the wide range of delays of logic gates provided by the applied ASIC library for implementation, it is not realistic to consider the gate depth complexity as a metric for the minimum latency of a Boolean function. For instance, in almost all the libraries, a 2-bit `XOR` or `XNOR` gate has a latency of about twice the latency for other gates with a fan-in number of 2. Another example is the difference in the latency of the gates with different fan-in numbers; the latency of the gates with a higher number of fan-ins is larger than the latency for similar gates but with fewer fan-ins. On the other hand, except for the `INV` gate, whose fan-in number is one, the 2-bit `NAND` gate and the 2-bit `NOR` gate have the minimum latency in almost all the ASIC libraries. To make a view of the latency for different gates, we list latency and area of all logic gates (with fan-out number one) in NanGate 15 nm and 45 nm Open Cell Libraries with *typical operating conditions* in Table 1.

To have a more accurate metric for the latency, we use the gate depth complexity by restricting ourselves to only `INV`, 2-bit `NAND` and 2-bit `NOR` gates. But since the `INV` gate has comparably lower latency than the other two gates, and more importantly, because of the reason explained later in Proposition 1, we do not consider the `INV` gates in the gate count of the implementation. We define the latency complexity of a (vectorial) Boolean function as in the following definition.

Definition 8 (Latency Complexity)**.** It is the minimum value for the longest path (concerning the number of only `NAND` and `NOR` gates) from any input to any output for implementing the function while the set of allowed gates to use is $\mathcal{G} = \{\texttt{INV}, \texttt{NAND}, \texttt{NOR}\}$.

Similar to the case for gate depth complexity, the latency complexity of a vectorial Boolean function is the maximum of latency complexity for each of its coordinate functions.

It is noteworthy to mention that using the basis $\mathcal{G} = \{\texttt{INV}, \texttt{NAND}, \texttt{NOR}\}$ is equivalent to using $\mathcal{G}' = \{\texttt{INV}, \texttt{NAND}\}$ or $\mathcal{G}'' = \{\texttt{INV}, \texttt{NAND}, \texttt{NOR}, \texttt{AND}, \texttt{OR}\}$. Since both `AND` and `OR` are slower than both `NAND` and `NOR`, we exclude them from the basis. However, including `NOR` makes it possible to have a more simple structure for low-latency implementation of a Boolean function that is explained in detail in Proposition 1.

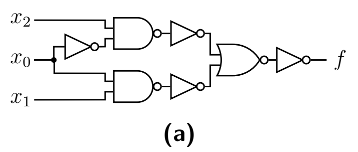
In the following example, we explain how to count the gate depth of implementation when we are computing its latency complexity.

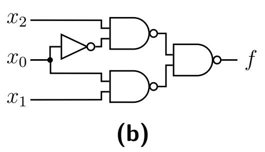
Example 1 (Latency Complexity of `MUX2`)**.** The circuit shown in Figure 1a is an instance for implementing the function $f(x_0, x_1, x_2) = (x_0 \wedge x_1) \vee (\overline{x}_0 \wedge x_2)$ using the gates in $\mathcal{G} = \{\texttt{INV}, \texttt{NAND}, \texttt{NOR}\}$. Note that f is a balanced function with the ANF representation of $x_0x_1 \oplus x_0x_2 \oplus x_2$. Besides, it represents the functionality of a multiplexer (`MUX2`) that x_0 acts as the selector to choose either x_1 or x_2.

The circuit implies that $y = \neg\big(\neg(x_0 \barwedge x_1) \,\overline{\vee}\, \neg(\overline{x}_0 \barwedge x_2)\big)$ and the length of the paths from each input to the output (by counting only `NAND` and `NOR` gates) is two. Using the equation $\neg(\overline{y}_0 \,\overline{\vee}\, \overline{y}_1) = y_0 \barwedge y_1$, one can simplify the implementation in Figure 1a to $y = (x_0 \barwedge x_1) \barwedge (\overline{x}_0 \barwedge x_2)$ as in the implementation shown in Figure 1b.

Table 1: The latency (in picoseconds) and the area (in GE) of logic gates (with fan-out number 1) in NanGate 15 nm and 45 nm open cell libraries for typical operating conditions.

Input(s)	**Gate**	**Output**	**15 nm**		**45 nm**	
			Latency	Area	Latency	Area
x_0	`INV_X1`	$\neg x_0$	1.580	0.75	22.048	0.67
	`BUF_X1`	x_0	3.068	1.25	33.557	1.00
x_0x_1	`NAND2_X1`	$\neg(x_0 \wedge x_1)$	2.031	1.00	27.886	1.00
	`NOR2_X1`	$\neg(x_0 \vee x_1)$	2.554	1.00	40.650	1.00
	`AND2_X1`	$x_0 \wedge x_1$	3.580	1.50	40.171	1.33
	`OR2_X1`	$x_0 \vee x_1$	3.644	1.50	56.414	1.33
	`XOR2_X1`	$x_0 \oplus x_1$	5.268	2.25	73.019	2.00
	`XNOR2_X1`	$\neg(x_0 \oplus x_1)$	6.788	2.25	57.604	2.00
$x_0x_1x_2$	`NAND3_X1`	$\neg(x_0 \wedge x_1 \wedge x_2)$	2.361	1.50	34.767	1.33
	`OAI21_X1`	$\neg\big((x_0 \vee x_1) \wedge x_2\big)$	2.830	1.50	32.651	1.33
	`AOI21_X1`	$\neg\big((x_0 \wedge x_1) \vee x_2\big)$	3.394	1.50	51.619	1.33
	`NOR3_X1`	$\neg(x_0 \vee x_1 \vee x_2)$	3.788	1.50	61.543	1.33
	`AND3_X1`	$x_0 \wedge x_1 \wedge x_2$	5.496	2.00	51.869	1.67
	`OR3_X1`	$x_0 \vee x_1 \vee x_2$	5.862	2.00	85.840	1.67
	`MUX2_X1`	$(\neg x_0 \wedge x_1) \vee (x_0 \wedge x_2)$	6.133	3.25	75.175	2.33
$x_0x_1x_2x_3$	`OAI22_X1`	$\neg\big((x_0 \vee x_1) \wedge (x_2 \vee x_3)\big)$	3.776	1.75	54.596	1.67
	`AOI22_X1`	$\neg\big((x_0 \wedge x_1) \vee (x_2 \wedge x_3)\big)$	4.070	1.75	57.255	1.67
	`NAND4_X1`	$\neg(x_0 \wedge x_1 \wedge x_2 \wedge x_3)$	4.659	1.75	44.487	1.67
	`NOR4_X1`	$\neg(x_0 \vee x_1 \vee x_2 \vee x_3)$	5.250	1.75	91.313	1.67
	`AND4_X1`	$x_0 \wedge x_1 \wedge x_2 \wedge x_3$	7.125	2.25	65.492	2.00
	`OR4_X1`	$x_0 \vee x_1 \vee x_2 \vee x_3$	7.683	2.25	118.592	2.00





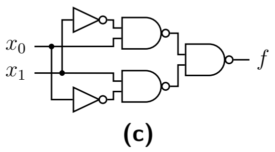


Figure 1: Low-latency implementation of a `MUX2` and an `XOR`.

Even though the latency complexity of implementing f is two, to determine it, we need to consider the length of the longest path in all possible implementations for f.

It is an interesting observation that the latency of a `MUX2` gate in the typical conditions is usually about 3 times the one for `NAND` gate (see Table 1). Hence, if the target is to lower the latency, one can use the implementation shown in Figure 1b instead of the `MUX2` gate, but then it needs more area compared to the case of using original `MUX2` gate. Precisely, depending on the ASIC technology, it needs 15% or 57% more area to reduce the latency by about 30%, which sounds to be a good trade-off.

Example 2 (Latency Complexity of `XOR`)**.** The XOR function of two variables can be represented as $f(x_0, x_1) = (x_0 \wedge \overline{x}_1) \vee (\overline{x}_0 \wedge x_1)$. Using the same approach as in Example 1, this representation changes to $f(x_0, x_1) = (x_0 \overline{\wedge} \overline{x}_1) \overline{\wedge} (\overline{x}_0 \overline{\wedge} x_1)$ with gate depth of two. The implementation of this function is shown in Figure 1c which is similar to the one in Figure 1b with an extra `INV` gate.

The latency complexity of `XOR` is also two. Considering the fact that the latency of `XOR` is always more than twice the latency for `NAND`, it is reasonable to exclude the `XOR` gate from the gate basis of the latency complexity.

Proposition 1. *Any n-bit Boolean function $f(x_0, \dots, x_{n-1})$ with latency complexity d can be implemented by a circuit of the structure shown in Figure 2. In this structure, each of $g_{i,j}$ gates with $0 < i \le d$ and $j \in \mathbb{Z}_{2^{d-i}}$ are either a `NAND` or a `NOR` gate, and the inputs to the gates in the first level is either a_i or its inverted value, $\overline{a}_i$, while each a_i with $i \in \mathbb{Z}_{2^d}$ is chosen from the set $\{x_0, \dots, x_{n-1}\}$.*

Proof. To prove the proposition, we need to show that any function with latency complexity d can be implemented by 1) using `INV` gates only in the beginning, i.e., in the depth level 0, and 2) using the output of each gate in the input of only one gate in the next depth level. To show the first part, we use the equations

$$\neg(y_0 \overline{\wedge} y_1) = \overline{y}_0 \overline{\vee} \overline{y}_1 \,, \qquad \neg(y_0 \overline{\vee} y_1) = \overline{y}_0 \overline{\wedge} \overline{y}_1 \,.$$

These equations imply that if there is an `INV` gate in the output of a `NAND` or `NOR` gate, it can be removed from the output and replaced in both inputs by changing the gate type to `NOR` or `NAND` gate, respectively. Therefore, in an implementation of a function, if there is an `INV` gate in the output of a `NAND` or a `NOR` gate at depth level i, we can replace it with two `INV` gates in the depth level of $i - 1$ by changing the `NAND`/`NOR` gate's type. Thus, starting from depth level d, we can bring all the `INV` gates to the previous depth level and update the implementation. By updating the implementation, we mean changing the corresponding `NAND` or `NOR` gate type and removing (if there exist) two repeated `INV` gates in the depth level $i - 1$.

To show the second part, assume that the output of a `NAND` or a `NOR` gate is used in the input of other gates more than once, i.e., its fan-out number is greater than 1. This can be eliminated by repeating the implementation of the corresponding sub-circuit of this output such that the output of each repetition is used only once. Besides, assume that the output of a gate in the depth level i is used in the depth level of i' with $i' > i + 1$. This also can be eliminated by using similar equations to $\neg y \overline{\wedge} \neg y = y$, i.e., repeating the

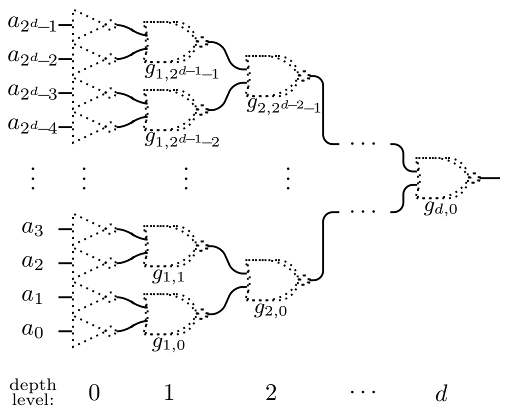


Figure 2: General structure for implementing a function with latency complexity d.

inverted circuit for y. Therefore, we ensure that the output of a gate in the depth level i is used only once in an input of a gate in the depth level of $i + 1$. $\square$

Note that to prove the second part in the above proof, we may repeat implementing a sub-circuit several times, which is not efficient area-wise. We emphasize that the structure of Figure 2 is only for a straightforward representation to implement all the Boolean functions with latency complexity d and not for an optimized low-latency implementation. As it comes later, it helps us to study the latency complexity of Boolean functions. Besides, we describe a method to find the implementation with the lowest latency of a given Boolean function in Subsection 4.4 which uses the implementation following the structure in Figure 2.

In the structure proposed in the previous works, e.g., in [Sto16, BMD+20], it is considered that the output of gates can be used in the input of several gates and in any of the following depth levels. Unlike our representation, theirs needs more variables and long-expressed equations in the SAT model, which makes solving the SAT-model harder.

As shown in Figure 2, for a low-latency implementation of a function, we only need to use the `INV` gates once and at the beginning (we call it depth level 0). Since in the unrolled implementation, it is necessary to use a layer of buffers for the input variables of each combinatorial circuit to amplify the voltage of the wires, if the input variable needs to go through an `INV` gate, there is no need to use such a buffer. This means, in the depth level 0, each input variable a_i goes through a `BUF` gate (with the output of a_i) or through an `INV` gate (with the output of $\bar{a_i}$). Besides, note that latency of an `INV` gate is lower than that of a `BUF` gate. This is the main reason why we do not count the `INV` gates in gate count of the longest path.

We emphasize neglecting latency of the depth level 0 is an oversimplification for measuring the latency of a single S-box implementation. In our structure and the ones for previous related works, we consider that for each repeated input variable, one independent wire enters the combinatorial circuit. But, in reality, there are at most two wires for each input variable entering the combinatorial circuit: one goes through a `BUF` and one through an `INV` gate, both with a higher fan-out number. This indeed affects and increases the latency of the implementation and should not be neglected. However, without this simplification, it is impossible to present a solution to model this complicated implementation parameter.

In the rest of this paper, for implementing an n-bit Boolean function f, we focus on the structure of Figure 2. By mapping each $g_{i,j}$ gate (with $0 < i \leq d$ and $j \in \mathbb{Z}_{2^{d-i}}$) to a binary value, we use $G \in \mathbb{F}_2^{2^d-1}$ to denote the type of gates, i.e.,

$$G := (g_{1,0}, \ldots, g_{1,2^{d-1}-1}, g_{2,0}, \ldots, g_{2,2^{d-2}-1}, \ldots, g_{d,0}) \, .$$

We map a `NAND` gate to `0` and a `NOR` gate to `1`. By $\alpha \in \mathbb{F}_2^{2^d}$, we denote whether a_i with $i \in \mathbb{Z}_{2^d}$ goes through an `INV` gate. More precisely, if $\alpha[i] = 1$, then $\overline{a}_i$ is used as the input of the gate in depth level 1, otherwise a_i itself. We use $\pi \in \mathbb{Z}_n^{2^d}$ to denote the choice of x_j variables by a_i variables, i.e., $a_i = x_{\pi[i]}$, and by $P : \mathbb{F}_2^n \mapsto \mathbb{F}_2^{2^d}$, we denote the corresponding mapping applied by π, i.e., $x \mapsto P(x)$ with $\forall i \in \mathbb{Z}_{2^d}$, $P(x)[i] = x[\pi(i)]$.

Besides, we use $\mathcal{I}_{i,j}$ (with $0 < i \leq d$ and $j \in \mathbb{Z}_{2^{d-i}}$) to denote the corresponding sub-circuit from the inputs $x_0, \ldots, x_{n-1}$ to the output of gate $g_{i,j}$. Therefore, each $\mathcal{I}_{i+1,j}$ (with $0 < i < d$ and $j \in \mathbb{Z}_{2^{d-i}}$) can be represented by the tuple $(\mathcal{I}_{i,2\cdot j}, \mathcal{I}_{i,2\cdot j+1}, g_{i,j})$. Thus, $\mathcal{I}_{d,0}$ is the implementation of f. Moreover, each of $\mathcal{I}_{i,j}$ can be represented by the corresponding $(G_{i,j}, \alpha_{i,j}, \pi_{i,j})$ tuple where $G_{i,j} \in \mathbb{F}_2^{2^i-1}$, $\alpha_{i,j} \in \mathbb{F}_2^{2^i}$ and $\pi_{i,j} \in \mathbb{Z}_n^{2^i}$.

To find the latency complexity of an n-bit Boolean function, by knowing that it is not smaller than d, one can try all the possibilities for the structure of Figure 2. To do so, we need to go through all 2^{2^d-1} choices for G, all 2^{2^d} choices for α, and all n^{2^d} choices for π, which ends up with a total computational complexity of about $2^{2^d\cdot(2+\log_2 n)}$. It is clear that if $d > 4$, it is impossible to do this computation in practice. Even for $d = 4$, if $n > 4$, this computation is not practical.

We applied the SAT-solver-based tool presented in [Sto16] for the gate depth optimization. We modified it to find the latency complexity of a Boolean function by replacing the previous model (with more gate types and free wiring style) with the one in the structure of Figure 2. Even though this modification makes the tool faster, it only provides a *single* solution for low-latency implementation of small Boolean functions, and it cannot find any solutions for full-dependent n-bit Boolean functions with $n > 5$.

4 Boolean Functions with Low-Latency Complexity

In this section, we first show that the latency complexity of (vectorial) Boolean functions stays invariant over the extended bit permutation equivalence. Next, we explain several speed-up techniques on the search for computing the latency complexity of a given Boolean function and also on finding all Boolean functions with a given latency complexity. Then, we present a general algorithm to find all Boolean functions with a given latency complexity. As a result, we determine the latency complexity of all Boolean functions up to 5 bits, together Boolean functions of up to 8 bits with latency complexity at most 4. Afterwards, we present another algorithm but faster one to find all possible structures to implement a Boolean function with gate depth of same as its latency complexity.

4.1 Extended Bit Permutation Equivalence

It is clear that a bit permutation or a constant addition in the input or the output of a Boolean function does not change the latency complexity. These properties are explained in detail in the following proposition.

Proposition 2. *Let f_1 and f_2 be two n-bit Boolean functions which are equivalent under extended bit permutation equivalence, i.e., $f_2(x) = f_1\big(P'(x) \oplus \alpha'\big) \oplus c$ for any $x \in \mathbb{F}_2^n$ with P' being a bit permutation function with corresponding permutation of π', and $\alpha' \in \mathbb{F}_2^n$, $c \in \mathbb{F}_2$ being constant values. Then the latency complexity of f_1 and f_2 are the same.*

Moreover, if (G, α, π) is one instantiation of implementing f_1 in the structure of Figure 2, with P being the mapping applied by π, the corresponding implementation for f_2 can be realized by $(G, \alpha \oplus P(\alpha'), \pi \circ \pi')$ if $c = 0$; and if $c = 1$ then it can be implemented by $(\overline{G}, \overline{\alpha} \oplus P(\alpha'), \pi \circ \pi')$ while $\overline{G}$ and $\overline{\alpha}$ denote the complement value of G and α, respectively.

Proof. A bit permutation can be realized in hardware by replacing the wires, and a constant addition can be realized by adding `INV` gates. Therefore, we can modify the implementation

Table 2: Number of Boolean functions up to the extended bit permutation equivalence. N_1, N_2, N_3, and N_4 denote the number of all Boolean functions, full-dependent Boolean functions, balanced Boolean functions, and full-dependent and balanced Boolean functions up to the equivalence, respectively.

n	N_1	N_2	N_3	N_4
2	4	2	2	1
3	14	10	6	4
4	222	208	58	52
5	616 126	615 904	86 603	86 545

for f_1 and use it for f_2 without changing the length of the longest path (by only counting the `NAND` and `NOR` gates). Since this is true for any implementation of f_1, the latency complexity of f_2 is the same as that of f_1.

To prove the second part, we first consider $c = \mathtt{0}$. The input of the structure in the case of f_1 for a given $x \in \mathbb{F}_2^n$ is $P(x)$, then the corresponding input of the structure for the case of f_2 must be $P(P'(x) \oplus \alpha')$. Since P is a linear mapping, then $P(P'(x) \oplus \alpha') = P \circ P'(x) \oplus P(\alpha')$. The function $P \circ P'$ can be realized by applying $\pi \circ \pi'$ mapping. The next part, $P(\alpha')$, also can be combined with the `INV` gates in the depth level of 0. To do this combination, instead of using `INV` gates corresponding to α, we use `INV` gates corresponding to $\alpha \oplus P'(\alpha')$.

In the case of $c = 1$, we implement $\neg f_2 = f_1\big(P'(x) \oplus \alpha'\big)$ as it is explained for the case of $c = \mathtt{0}$ and then insert an extra `INV` gate in the output of $\neg f_2$ to realize implementation of f_2. It is possible to replace the `INV` gate in the depth level of d with two `INV` gates in the depth level of $d-1$ by changing the gate type, i.e., $g_{d,0}$. Repeating this for d times, we end up with 2^d extra `INV` gates in the depth level 0 and changing all the gate types. This means we changed G to $\overline{G}$ and $\alpha \oplus P'(\alpha')$ to $\overline{\alpha \oplus P'(\alpha')} = \overline{\alpha} \oplus P'(\alpha')$. □

Proposition 2 suggests that instead of studying all n-bit Boolean functions, it is enough to evaluate the latency complexity of the representative Boolean functions for each equivalence class. Table 2 shows the number of n-bit Boolean functions up to extended bit permutation equivalence for $n \leq 5$. There, N_1 denotes the number of all Boolean functions up to the equivalence, N_2 denotes the number of full-dependent Boolean functions up to the equivalence, N_3 denotes the number of balanced Boolean functions up to the equivalence, and N_4 denotes the number of full-dependent and balanced Boolean functions up to the equivalence. By a full-dependent function, we mean the functions in which the output is dependent on all input variables.

To find all the representative functions, we used a technique that is based on the following lemma.

Lemma 1. *Let n-bit Boolean function $f(x_0, \ldots, x_{n-1})$ to be in the form of*

$$f = \overline{x_0} \cdot f_0(x_1, \ldots, x_{n-1}) \oplus x_0 \cdot f_1(x_1, \ldots, x_{n-1})$$

with f_0 and f_1 both being $(n-1)$-bit Boolean functions; i.e., for their corresponding truth tables, we have $T_f = (T_{f_0} || T_{f_1})$. If f is an n-bit representative Boolean function in an equivalence, then f_0 must be an $(n-1)$-bit representative Boolean function in the same equivalence. Besides, f_0 must be lexicographically smaller than the representative for f_1.

Applying this lemma, for finding all the n-bit representative Boolean functions, we need to use two $(n-1)$-bit representative Boolean functions and extend the lexicographically larger one by equivalence. Then, we need to check if the resulting n-bit function is representative. In this case, with the extended bit permutation equivalence, for finding all the n-bit representative Boolean functions, we need to consider about $|N_{1,n-1}|^2 \cdot (n-1)! \cdot 2^n$

possibilities to check if the resulting n-bit Boolean function is representative, where $N_{1,n-1}$ denotes the number of all $(n-1)$-bit representative Boolean functions. For example, to find all 5- and 6-bit representative Boolean functions, we need to check the representative-ness of about 2^{25} and 2^{51} functions, respectively.

4.2 Possible Speed-Up Techniques

To find all the n-bit Boolean functions with latency complexity d, or to find all possible implementations of a given Boolean function with latency complexity of d, one can compute all possible functions in the structure of Proposition 1. As mentioned before, the computational complexity of this search is about $2^{2^d \cdot (2+\log_2 n)}$. In the following, we explain several techniques to reduce the computational complexity of the search when we want to find 1) all the n-bit Boolean functions with latency complexity d, 2) all possible implementations with the same depth as the latency complexity for a Boolean function. Thereby, we try to remove all the redundant computations through all the possibilities.

Reduction on the Possibilities for G: Since both $(\mathcal{I}_{d-1,0}, \mathcal{I}_{d-1,1}, g_{d,0})$ and $(\mathcal{I}_{d-1,1}, \mathcal{I}_{d-1,0}, g_{d,0})$ tuples both make the same implementation for $\mathcal{I}_{d,0}$, it is enough to check one of these tuples. Furthermore, a similar reduction is valid for implementing the other smaller sub-circuits; i.e., both $(\mathcal{I}_{i,2j}, \mathcal{I}_{i,2j+1}, g_{i+1,j})$ and $(\mathcal{I}_{i,2j+1}, \mathcal{I}_{i,2j}, g_{i+1,j})$ tuples build the same implementation for $\mathcal{I}_{i+1,j}$.

This redundancy can be eliminated by limiting the possibilities for G. Precisely, if $N_{G,i}$ is the number of possibilities for each $G_{i,2j}$ and $G_{i,2j+1}$, instead of going through all $2N_{G,i}^2$ possibilities for $G_{i+1,j}$, it is enough to only check $2 \cdot \frac{N_{G,i}(N_{G,i}+1)}{2} = N_{G,i}^2 + N_{G,i}$ possibilities. Note that the multiplication by 2 is because of the number of choices for $g_{i+1,j}$. Hence, the number of reduced possibilities for G after this reduction is equal to 2, 6, 42, $2^{10.8}$, $2^{21.6}$, and $2^{43.3}$ for d to be equal to 1, 2, …, and 6, respectively.

Moreover, we know that if (G, α, π) is the corresponding implementation for function f, then we can implement $f \oplus \mathtt{1}$ function by using $(\overline{G}, \overline{\alpha}, \pi)$. Thus, if we are searching for all the Boolean functions with a given latency complexity (up to the extended bit permutation equivalence), we can use this technique to fix the type of a single gate. For simplicity, we fix $g_{d,0}$ to be a `NAND` gate. We emphasize that this reduction is only valid when we are searching for the Boolean functions with a given latency complexity.

Reduction on the Possibilities for α: Similar to reducing the possibilities for G, we can also reduce the number of possibilities for α. Since the order of inputs to a `NAND` or a `NOR` gate does not affect the output, we can reduce the computation complexity by fixing the order of the inputs. Precisely, by the notation of the structure in Figure 2, $x_{\pi(2i)} \oplus \alpha[2i]$ and $x_{\pi(2i+1)} \oplus \alpha[2i+1]$ are the inputs to the gate $g_{0,i}$ for any $i \in Z_{2^{d-1}}$. Swapping these two inputs does not change the implemented function. We can omit this redundant computation by only considering that $\alpha[2i] \leq \alpha[2i+1]$ for any $i \in Z_{2^{d-1}}$. Therefore, instead of checking for all $4^{2^{d-1}} = 2^{2^d}$ possibilities for α, it is enough to check only $3^{2^{d-1}}$ of them.

Using these two reductions on G and α, for a given n-bit Boolean function, determining the all possible implementations with latency complexity d for an n-bit Boolean function needs checking about

$$2^{11.7} \cdot n^8, \quad 2^{23.5} \cdot n^{16} \quad \text{and} \quad 2^{47} \cdot n^{32}$$

possibilities for $d = 3$, $d = 4$ and $d = 5$, respectively. It is clear that if $d > 4$, it is impossible to do this computation in practice. Even for $d = 4$, if $n > 4$, this computation is not practical.

Reduction on the Possibilities for π: To find all the Boolean functions with a given latency complexity (up to the extended bit permutation equivalence), we can still reduce the number of possibilities for π. The implementations based on (G, α, π) and $(G, \alpha, \pi \circ \pi')$ by π' being a permutation of $\mathbb{Z}_n$, build bit permutation equivalent Boolean functions. Therefore, we can reduce the possibilities for π such that if we use π, we do not check for any other $\pi \circ \pi'$ choices. This reduction reduces the computational complexity by a factor of about $n!$.

Besides, the implementations based on (G, α, π) and $(G, \alpha \oplus P(\alpha'), \pi)$ by $\alpha' \in \mathbb{F}_2^n$, build Boolean functions those are different only in a constant addition in the input. Again, we can reduce the possibilities for α such that if we use α, we do not check for any other $\alpha \oplus P(\alpha')$ choices. Note that using this reduction requires knowing the choice for π. Since we previously reduced the choices for α with $\alpha_{2i} \le \alpha_{2i+1}$ restriction for any $i \in Z_{2^{d-1}}$, this reduction reduces the computational complexity by a factor smaller than 2^n.

Altogether, even with using all these reductions on the number of possibilities, the computational complexity of the search for all the n-bit Boolean functions with a latency complexity d (up-to the extended bit permutations equivalence), we need to consider more than

$$2^{11.7} \cdot \frac{n^8}{n! \cdot 2^n}, \quad 2^{23.5} \cdot \frac{n^{16}}{n! \cdot 2^n} \quad \text{and} \quad 2^{47} \cdot \frac{n^{32}}{n! \cdot 2^n}$$

possible (G, α, π) tuples for $d = 3$, $d = 4$, and $d = 5$, respectively. For instance, finding all the 5-bit Boolean functions with latency complexity 4 and 5, we need to consider more than 2^{49} and 2^{109} functions, respectively.

Note that each of these functions built by these tuples is not a representative function. To achieve the set of representatives, we need to remove the equivalent functions. To do this, we compute the representatives for each Boolean function built by these choices and remove the duplicated representatives. We used the simplest method to compute the representative of a Boolean function in the extended bit permutation equivalence. We try all the equivalent Boolean functions by choosing one of the $n!$ bit permutation functions and one of the 2^n constants in the input of the function while we choose the output constant bit in such a way that in the resulting equivalent function, the point `0` is mapped to `0`. Hence, computing the representative for each of these Boolean functions costs about $2^n \cdot n!$ operations. Therefore, computing the set of all n-bit representative Boolean functions with latency complexity d needs the same amount of computations as for finding all possible implementations for an n-bit Boolean function with the same latency complexity; i.e.,

$$2^{11.7} \cdot n^8, \quad 2^{23.5} \cdot n^{16} \quad \text{and} \quad 2^{47} \cdot n^{32}$$

operations for $d = 3$, $d = 4$ and $d = 5$, respectively.

To remove the equivalent Boolean functions, we also tried the approach introduced in [MB19] for determining if two functions are equivalents. We modified their method for the case of extended bit-permutation equivalence instead of the affine equivalence. However, due to the difference in corresponding equivalences (that the affine equivalence is a stronger one than the extended bit-permutation), for our application in this paper, we find using the method in [MB19] slower than using the simplest method. We emphasize that their method is only slower here, where we are removing the equivalent Boolean functions up to the extended bit-permutation. In the case of removing the equivalent n-bit to m-bit vectorial Boolean functions, with a large value for n and a small value for $n - m$, modification of the [MB19] method is significantly faster than the simple approach.

4.3 Finding Boolean Functions with Low-Latency Complexity

In this subsection, we present an efficient algorithm to find all the n-bit Boolean functions with latency complexity d.

Algorithm 1: Computing $\mathcal{F}_{n,d}$, the set of all full-dependent n-bit Boolean functions with latency complexity d up to the extended bit-permutation equivalence.

```
Data: $\mathcal{F}_{n',d'}$ for all $n' \leq n$ and $d' < d$    // the sets of all full-dependent $n'$-bit
      Boolean functions with latency complexity $d'$
Result: $\mathcal{F}_{n,d}$
1  $\mathcal{F} \leftarrow \emptyset$ and $\mathcal{F}_{n,d} \leftarrow \emptyset$
2  for $n_0 \leftarrow 1$ to $n$ do
3      for $n_1 \leftarrow \max(1, n - n_0)$ to $n_0$ do
4          foreach $d_0, d_1 \in \mathbb{Z}_d$ do
5              if $\mathcal{F}_{n_0,d_0} \neq \emptyset$ and $\mathcal{F}_{n_1,d_1} \neq \emptyset$ and $(d_0 = d - 1$ or $d_1 = d - 1)$ then
6                  foreach $\pi \in \mathbb{Z}_n^{n_1}$ if $\pi$ follows the restrictions do // for $i < j$, $\pi[i] \neq \pi[j]$,
                                                          // and if $n_0 \leq \pi_1[i]$ and $n_0 \leq \pi_1[j]$, then $\pi_1[i] < \pi_1[j]$.
7                      Compute the corresponding bit-permutation function $P$.
8                      foreach $\alpha \in \mathbb{F}_2^{n_1}$ if $\alpha$ follows the restrictions do
                                                          // for each $i$ such that $n_0 \leq \pi_1[i]$, $\alpha_1[i]$ must be 0.
9                          foreach $f_0^* \in \mathcal{F}_{n_0,d_0}$ and $f_1^* \in \mathcal{F}_{n_1,d_1}$ do
10                             $\mathcal{F} \leftarrow \mathcal{F} \cup \{f_0^* \overline{\wedge} f_1^*(P(\cdot) \oplus \alpha)\}$
11 foreach $f \in \mathcal{F}$ do
12     $\mathcal{F}_{n,d} \leftarrow \mathcal{F}_{n,d} \cup \{$ComputeRepresentative$(f)\}$
13 for $n' \leftarrow 1$ to $n$ do
14     for $d' \leftarrow 0$ to $d$ do
15         if $(n', d') \neq (n, d)$ then
16             $\mathcal{F}_{n,d} \leftarrow \mathcal{F}_{n,d} - \mathcal{F}_{n',d'}$
```

To find all the n-bit Boolean functions with latency complexity d, we only need to find them up to the extended bit permutation equivalence. Assume that for each $d' < d$ and $n' \leq n$, we already have the set of all representative and full-dependent n'-bit Boolean functions with latency complexity d'. We denote each of these sets by $\mathcal{F}_{n',d'}$.

If f is an n-bit Boolean function with latency complexity d, then there exist two Boolean functions f_0 and f_1 with latency complexity d_0 and d_1, respectively, such that $f = f_0 \overline{\wedge} f_1$ or $f = f_0 \overline{\vee} f_1$ and $d_0, d_1 < d$ with at least one of d_0 or d_1 equals to $d-1$. Since we are only finding the Boolean functions up to extended bit permutation equivalence, it is enough to only consider one of the $f_0 \overline{\wedge} f_1$ and $f_0 \overline{\vee} f_1$ cases that here we use $f_0 \overline{\wedge} f_1$.

Assume that f_0 and f_1 are n_0- and n_1-bit full-dependent Boolean functions, respectively. Then we know that for each $k \in \mathbb{F}_2$, there exist $f_k^* \in \mathcal{F}_{n_k,d_k}$, bit permutation function P_k with corresponding permutation $\pi_k \in \mathbb{Z}_n^{n_k}$ such that for i and j with $0 \leq i < j < n_k$, we have $\pi_k(i) \neq \pi_k(j)$, $\alpha_k \in \mathbb{F}_2^{n_k}$, and $c_k \in \mathbb{F}_2$ which form f_k, i.e., $f_k(x) = f_k^*\big(P_k(x) \oplus \alpha_k\big) \oplus c_k$. The restriction on π is because the Boolean functions are full-dependent, and therefore π_k must be a permutation of $\mathbb{Z}_{n_k}$. As explained previously in the speed-up techniques, we can reduce the choices for both π_k permutations and both α_k constants. For simplicity, we choose $\pi_0 = (0, \ldots, n_0 - 1)$ and $\alpha_0 = 0$. Besides, we put a restriction on π_1 that for $0 \leq i < j < n_1$, if $n_0 \leq \pi_1[i]$ and $n_0 \leq \pi_1[j]$, then $\pi_1[i] < \pi_1[j]$. Moreover, we restrict α_1 in a way that for $0 \leq i < n_1$ if $n_0 \leq \pi_1[i]$, then $\alpha_1[i] = 0$.

These reductions on building $\mathcal{F}_{n,d}$, decrease the number of possibilities for π_1 to $(n_0! \cdot n_1!)/\big((n - n_0)! \cdot (n - n_1)! \cdot (n_0 + n_1 - n)!\big)$ and for α_1 to $2^{n_0+n_1-n}$. All together, if we consider all the cases for n_0, n_1, d_0 and d_1, the computational complexity of this algorithm to find all the n-bit Boolean functions with latency complexity d up to the extended bit permutation equivalence is about

Table 3: Number of full-dependent n-bit (balanced) Boolean functions with latency complexity d up to the extended bit permutation.

$d\backslash n$	2	3	4	5	6	7	8
1	1						
2	1	3	3				
3		5	54	159	170	64	20
4		2	149	131 853	20 658 457	227 737 882	?
5			2	482 072	?	?	?
6				1820	?	?	?

Balanced

$d\backslash n$	2	3	4	5	6	7	8
2	1	1					
3		2	6	8	3		
4		1	45	12 757	931 780	4 436 770	4 489 235
5			1	73 778	?	?	?
6				2	?	?	?

$$\sum_{\substack{d_0,d_1,n_0,n_1,\\ n_0\le n_1\le n \text{ and } d_0,d_1<d}} \frac{n_0!\cdot n_1!\cdot 2^{n_0+n_1-n+2}}{(n-n_0)!\cdot(n-n_1)!\cdot(n_0+n_1-n)!}\cdot \#\mathcal{F}_{n_0,d_0}\cdot\#\mathcal{F}_{n_1,d_1}\,.$$

We recall that the set of Boolean functions built by each of these choices is not the same as $\mathcal{F}_{n,d}$. To achieve $\mathcal{F}_{n,d}$, we need to remove the equivalent functions. We compute the representatives for each of these Boolean functions, which cost about $2^n\cdot n!$ operations for each function. A pseudo-code for our approach is provided in algorithm 1.

For comparison, in the case of 5-bit Boolean functions, for latency complexity 4 and 5, we need to consider about 2^{29} and 2^{48} functions, respectively. Therefore, to compute $\mathcal{F}_{5,4}$ and $\mathcal{F}_{5,5}$, we need to do about 2^{41} and 2^{60} computations, respectively. We recall that using the method in Subsection 4.1 with reducing all possible reductions, these searches need to consider more than 2^{49} and 2^{109} functions while determining only the representative functions we need about 2^{61} and 2^{121} computations, respectively.

Similarly, the computational cost of $\mathcal{F}_{6,4}$, $\mathcal{F}_{7,4}$, $\mathcal{F}_{8,4}$ and $\mathcal{F}_{6,5}$ are about 2^{49}, 2^{56}, 2^{61} and 2^{82}, respectively. Using the above algorithm, we compute all the Boolean functions in $\mathcal{F}_{n,d}$ for $n\le 5$ with all d values and for $n\le 8$ with $d\le 4$. However, in the case of $\mathcal{F}_{8,4}$, we only take the balanced Boolean functions and then compute their representatives. This makes it possible to reduce the computation cost, and we can find all the balanced Boolean functions.

The number of functions in $\mathcal{F}_{n,d}$ sets is shown in Table 3. Besides, we partition the balanced Boolean functions in $\mathcal{F}_{n,d}$ with respect to their linearity. Table 6 and Table 5 show the number of full-dependent n-bit balanced Boolean functions categorized with their linearity or ANF degree, and latency complexity.

4.4 Finding All Possible Implementations of a Boolean Function

In the following, we present an efficient algorithm to find all the possible implementations of an n-bit *full-dependent* Boolean function whose latency complexity is d, for the cases that either $n\le 5$, or $n\in\{6,7,8\}$ with $d\le 5$. Note that here, we are only interested in the implementations that have the minimum depth in the basis of $\{\texttt{NAND},\texttt{NOR},\texttt{INV}\}$. Precisely, we are only looking for the implementations with structure in Figure 2 for a depth of d. We recall that to do this, we only need to find structures up to the reductions explained in Subsection 4.2.

For simplicity, we assume that we know what is the value for latency complexity of the given Boolean function. Note that this assumption is based on the fact that we can compute representative of the given n-bit full Boolean function with $n! \cdot 2^n$ computations and then we can search if it is in $\mathcal{F}_{n,d}$. For the cases that $n \in \{6, 7, 8\}$ and $d > 4$ that we do not have the corresponding $\mathcal{F}_{n,d}$ sets, we assume $d = 5$ and try to find a structure for its implementation and if it is not possible, we conclude that d is higher than 5.

For starting the algorithm, we assume that for each $d' < d$, the set of all n-bit Boolean functions with latency complexity d' is already computed. We denote these sets with $\mathcal{F}^*_{d'}$. Note that these functions are not necessarily full-dependent. It means $\mathcal{F}^*_{d'}$ involves all equivalent Boolean functions for each representative function from $\mathcal{F}_{n,d'}$, or each function from $\mathcal{F}_{n',d'}$ extended to n bits.

If f is the given n-bit Boolean function with latency complexity d, then there exist two Boolean functions f_0 and f_1, both with latency complexity less than d and such that $f = f_0 \barwedge f_1$ or $f = f_0 \,\overline{\vee}\, f_1$. We use the following lemma to find all potentially possible f_0 and f_1 functions to build the structure of f with the minimum depth.

Lemma 2. *Let f, f_0 and f_1 be three n-bit Boolean functions such that $f = f_0 \wedge f_1$, then $f \wedge f_0 = f \wedge f_1 = f$. Similarly, if $f = f_0 \vee f_1$, then $f \vee f_0 = f \vee f_1 = f$. Converting the `AND` and `OR` operations to `NAND` and `NOR`, respectively, provides similar results; if $f = f_0 \barwedge f_1$, then $\neg f \wedge f_0 = \neg f \wedge f_1 = \neg f$, and if $f = f_0 \,\overline{\vee}\, f_1$, then $\neg f \vee f_0 = \neg f \vee f_1 = \neg f$.*

In other meaning, for the case $f = f_0 \barwedge f_1$, if there is an $x \in \mathbb{F}_2^n$ such that $f(x) = 0$, then $f_0(x) = f_1(x) = 1$; and similarly for the case $f = f_0 \,\overline{\vee}\, f_1$, if there is an x such that $f(x) = 1$, then $f_0(x) = f_1(x) = 0$.

For the first step of the algorithm, by applying the above lemma, we consider each n-bit Boolean function with depth less than d as a potential candidate for f_0 and f_1, and check if it the candidate function, g, satisfies $\neg f \wedge g = \neg f$ or $\neg f \vee g = \neg f$ conditions. If so, depending on which condition is fulfilled, we add the function to one of $\mathcal{A}_{\barwedge}$ or $\mathcal{A}_{\overline{\vee}}$ sets. We provide a pseudo-code of this approach at algorithm 2.

Since for a representative n'-bit full dependent Boolean function, there can be at most $2 \cdot 2^{n'} \cdot n'! \cdot \binom{n}{n'}$ equivalent n-bit Boolean function, the computation cost to complete each of $\mathcal{A}_{\barwedge}$ and $\mathcal{A}_{\overline{\vee}}$ sets is about

$$\sum_{\substack{d',n' \\ n' \leq n \text{ and } d' < d}} 2 \cdot 2^{n'} \cdot n'! \cdot \binom{n}{n'} \cdot |\mathcal{F}_{n',d'}| = \sum_{\substack{d',n' \\ n' \leq n \text{ and } d' < d}} 2^{n'+1} \cdot \frac{n!}{(n-n')!} \cdot |\mathcal{F}_{n',d'}| \,.$$

That is for $d = 4$, computation costs are about 2^{16}, 2^{20}, 2^{24}, 2^{28} and 2^{31}, for $n = 4$, $n = 5$, $n = 6$, $n = 7$ and $n = 8$, respectively, and for $d = 5$, these numbers are about 2^{16}, 2^{30}, 2^{41}, 2^{48} and $2^{25} \cdot |\mathcal{F}_{8,4}|$. Note that this computation cannot exceed 2^{2^n}, and this upper bound only happens for d values which are close to the maximum possible latency complexity for the corresponding n. For instance, for $n = 4$, it happens in the case of $d \in \{4, 5\}$ and for $n = 5$ it happens only if $d = 6$.

To reduce the memory usage and efficient computation of the algorithm, instead of computing and saving the $\mathcal{F}^*_{d'}$ sets, we can compute each element of these sets and check if it is a valid candidate for building up the given function f. Moreover, we do not need to compute the function from $\mathcal{F}^*_{d'}$ completely. We only need to compute the function output for the entries that are needed for the corresponding condition check and as soon as it does not fulfill the condition for one entry, we reject the function. Hence, the memory usage of this computation is only saving all of $\mathcal{F}_{n',d'}$ sets together with $\mathcal{A}_{\barwedge}$ and $\mathcal{A}_{\overline{\vee}}$ sets.

Algorithm 2: Finding all possible implementations of an n-bit Boolean function with minimum latency depth (in the basis of {`NAND`, `NOR`, `INV`}).

```
Data: $\mathcal{F}^*_{d'}$ for all $d'$        // all sets of $n$-bit representative Boolean functions (not
        necessarily full-dependent) with latency complexity $d'$
Result: $\mathcal{I}$        // all possible implementations of the given Boolean function
1  Function FindAllCircuits($f$):
2      if $f = x_i$ or $f = \neg x_i$ then        // the case for latency complexity of $d' = 0$
3          return the corresponding $x_i$ or $\neg x_i$
4      $\mathcal{I} \leftarrow \emptyset$ , $\mathcal{A}_{\overline{\wedge}} \leftarrow \emptyset$ and $\mathcal{A}_{\overline{\vee}} \leftarrow \emptyset$
5      for $d' \leftarrow 0$ to $d - 1$ do
6          foreach $g \in \{g \,|\, g$ is equivalent to $g^* \in \mathcal{F}^*_{d'}\}$ do
7              if $\neg f \wedge g = \neg f$ then
8                  $\mathcal{A}_{\overline{\wedge}} \leftarrow \mathcal{A}_{\overline{\wedge}} \cup \{g\}$
9              if $\neg f \vee g = \neg f$ then
10                 $\mathcal{A}_{\overline{\vee}} \leftarrow \mathcal{A}_{\overline{\vee}} \cup \{g\}$
11         foreach $g, h \in \mathcal{A}_{\overline{\wedge}}$ do
12             if $g \wedge h = \neg f$ then
13                 $\mathcal{I} \leftarrow \mathcal{I} \cup \{(\text{FindAllCircuits}(g), \text{FindAllCircuits}(h), \texttt{NAND})\}$
14         foreach $g, h \in \mathcal{A}_{\overline{\vee}}$ do
15             if $g \vee h = \neg f$ then
16                 $\mathcal{I} \leftarrow \mathcal{I} \cup \{(\text{FindAllCircuits}(g), \text{FindAllCircuits}(h), \texttt{NOR})\}$
17         if $\mathcal{I} \neq \emptyset$ then        // this means that the latency complexity of $f$ is $d' + 1$
18             return $\mathcal{I}$
19     return $\perp$        // it happens if the latency complexity of $f$ is higher than $d$
```

At the second step, for each two functions of f_0 and f_1 from $\mathcal{A}_{\overline{\wedge}}$, we check if $f_0 \,\overline{\wedge}\, f_1 = f$. We repeat this step for each two functions of f_0 and f_1 from $\mathcal{A}_{\overline{\vee}}$ and check if $f_0 \,\overline{\vee}\, f_1 = f$. If there is any (f_0, f_1) pair satisfying one of the equations for $\overline{\wedge}$ or $\overline{\vee}$, then we repeat the algorithm to find structure of implementing f_0 and f_1 functions with depth of $d - 1$. Otherwise, if there is no such pair of functions, we conclude that the latency complexity of f is higher than d. Note that this only happens for $d = 5$ with $n \in \{6, 7, 8\}$.

In this recursive algorithm, the bottleneck of our computations is computing $\mathcal{A}_{\overline{\wedge}}$ and $\mathcal{A}_{\overline{\vee}}$ for f function, i.e., in the depth level d. For other depth levels, even if we have several (f_0, f_1) pairs to find their structure, the computations are comparably smaller.

Note that for the given function, there might be several solutions for representing it with the structure of Figure 2 and it is not necessarily unique. Besides, the above-mentioned algorithm to find possible structures of the given function, there might be cases where the suggested solution does not follow the structure shown in Figure 2. This happens in the cases where the given function f with latency complexity d can be built with two functions f_0 and f_1 with latency complexities of $d_0 = d - 1$ and $d_1 < d - 1$, respectively. In this case, the sub-structure for f_1 will be with gate depth d_1 and shorter than the one for function f_0. Clearly, it does not follow the structure of Figure 2, but it is a simplification of the structure for implementing it.

4.5 Latency Complexity of Known S-boxes

In this subsection, we compute the latency complexity of previously introduced S-boxes in the cryptographic primitives and compare their latency complexity together with their cryptographic properties: linearity, uniformity, and algebraic degree. We listed all these

S-boxes together with the latency complexity of each coordinate function of the S-boxes in Table 7 that are categorized first by the input size of the S-boxes, and then by their uniformity and linearity. It also shows the algebraic degree (of ANF representation) for each coordinate of the S-boxes and their inverse S-boxes (in the case of bijective ones). Note that in the case of n-bit S-boxes with $n \in \{6, 7, 8\}$, if the latency complexity of a coordinate is 5 or higher, we denote it by x.

In the case of 3-bit S-boxes, all the S-boxes have a latency complexity of 3; and for 4-bit S-boxes, regardless of their cryptographic properties, their latency complexity is either 4 or 5, except for χ_4 (which is a non-bijective S-box) and `Midori-`s_0 S-box whose latency complexity are 3.

Within 5-bit S-boxes, χ_5, also known as `KECCAK` S-box, has the minimum latency complexity, 3, whose uniformity is 8 and linearity is 16. However, to achieve the minimum uniformity and linearity, such as in `Fides-5` S-box, they have a latency complexity of 5.

For the case of 6-bit S-boxes, the only previously introduced S-boxes with a low-latency complexity are the non-bijective χ_6 function with latency complexity 3, uniformity 16 and linearity 32, and `Speedy` S-box with latency complexity 4, uniformity 8 and linearity 24.

For 7-bit and 8-bit S-boxes, there is no S-box with latency complexity less than 5, except χ_7 (with uniformity 32 and linearity 64) and χ_8 (non-bijective and with uniformity 64 and linearity 128) that both have latency complexity of 4. As results show, there are very few n-bit S-boxes that are optimized for their latency, especially when $n > 4$.

With this motivation, using the Boolean functions with low-latency complexity found in the previous subsections, we build and introduce some new bijective S-boxes with a low-latency complexity in the following section.

5 Bijective S-boxes with a Low-Latency Complexity

To build an n-bit bijective S-box with latency complexity of d, as of the S-box's coordinate functions, we can use all the balanced Boolean functions equivalent to one of the representative balanced functions in one of $\mathcal{F}_{n',d'}$ sets with $d' \leq d$ and $n' \leq n$. One can restrict to only those S-boxes for which each of its coordinate functions is a full-dependent Boolean function, i.e., each output bit of the S-box is dependent on all of the input bits. Another restriction can be to put a limit on the linearity and the uniformity of the S-boxes or on the algebraic degree of the coordinate functions. Note that these restrictions make it possible to find cryptographically stronger S-boxes.

Assume that $\mathcal{F}^*$ is the set of all representative functions those are following our limits for the target S-boxes, e.g., n-bit S-boxes with latency complexity of d, linearity of at most ℓ and uniformity of at most u. Then any n-bit S-box of our target can be formed as $S = (f_0, \ldots, f_{n-1})$ such that for all $i \in \mathbb{Z}_n$, we have $f_i = f_i^*\big(P_i(\cdot) \oplus \alpha_i\big) \oplus c_i$ where $f_i^* \in \mathcal{F}^*$, P_i is a bit permutation function, $\alpha_i \in \mathbb{F}_2^n$ and $c_i \in \mathbb{F}_2$. Searching through all the possibilities for f_i, P_i, α_i, and c_i, we need to consider $\big(|\mathcal{F}^*| \cdot n! \cdot 2^{n+1}\big)^n$ cases. By fixing $\alpha_0 = 0$, P_0 to the identity function, and $c_i = \mathtt{0}$ for all $i \in \mathbb{Z}_n$, we can find all the S-boxes up to the extended bit permutation equivalence. Besides, due to the bit permutation in the output bits, we can also fix the order of the coordinate functions, e.g., for each $i < j$, we fix f_i to be lexicographically smaller than f_j. Then the computational complexity of the search is reduced to about $|\mathcal{F}^*|^n \cdot (n!)^{n-2} \cdot 2^{n^2-n}$. For instance, to build 6-bit S-boxes, the complexity of this search is about $2^{68} \cdot |\mathcal{F}^*|^6$. Even if we restrict ourselves to full-dependent S-boxes with latency complexity of 4 and linearity of 16, then $|\mathcal{F}^*| = 1546$ (see Table 6); therefore, we need to consider about 2^{131} possibilities. In the following, we present an algorithm to reduce the computational complexity of this search.

5.1 Step-By-Step Method for Building Bijective S-boxes

We use the property of the bijective S-boxes together with the definition for linearity.

Lemma 3. *For an n-bit bijective S-box S with linearity ℓ, each of its component functions, namely $\langle \alpha, S \rangle$ with $\alpha \in \mathbb{F}_2^n \setminus \{0\}$, is balanced and has a linearity of at most ℓ.*

Using Lemma 3 makes it possible to filter out some of the possibilities, only by having some of the coordinate functions. Precisely, assume that f_0^* and f_1 are already chosen, then without choosing other coordinate functions, we can check for balancedness and linearity of $f_0^* \oplus f_1$. If $f_0^* \oplus f_1$ is balanced and has a linearity at most ℓ, then we choose the third coordinate function, f_2. Again, we can check for balancedness and linearity of $f_0^* \oplus f_2$, $f_1 \oplus f_2$, and $f_0^* \oplus f_1 \oplus f_2$. Continuing in this way, after choosing the last coordinate function, f_{n-1}, we can check for balancedness and linearity of other $2^{n-1} - 1$ component functions. If these $2^{n-1} - 1$ conditions are met, then we have a bijective S-box with linearity at most ℓ, and we can compute its uniformity.

Assuming that the average probability of satisfying all $2^i - 1$ conditions over all possible choices for f_i is p_i, then the computational complexity of this search is about

$$\frac{1}{n!} \cdot |\mathcal{F}^*| \cdot N_f \cdot \Big(1 + p_1 \cdot N_f \cdot \Big(1 + p_2 \cdot N_f \cdot \big(\ldots (1 + p_{n-1})\big)\Big)\Big) \approx$$
$$\frac{1}{n!} \cdot |\mathcal{F}^*| \cdot N_f \cdot (1 + p_1 \cdot N_f + p_1 \cdot p_2 \cdot N_f^2 + \ldots + p_1 \cdot \ldots \cdot p_{n-2} \cdot N_f^{n-2})$$

where $N_f \approx |\mathcal{F}^*| \cdot n! \cdot 2^n$ and the first division by $n!$ is because of that due to the output bit permutation the coordinate functions can be ordered. Note that without using this step-by-step choosing of the coordinate functions, the complexity of the search is about $|\mathcal{F}^*| \cdot N_f^{n-1}/n!$ which is significantly larger than when we choose the coordinate functions step-by-step.

Moreover, we use the following technique to omit a dominant part of the computations. After choosing f_0^* in step 0, we compute the set of possible choices for f_1,

$$\mathcal{F}_1^\dagger(f_0^*) = \big\{ f \mid f = f^*\big(P(\cdot) \oplus \alpha\big)\, , f^* \in \mathcal{F}^*\, , \alpha \in \mathbb{F}_2^n\, , P : \text{bit permutation}\, ,$$
$$f \oplus f_0^* \text{ fulfills all the conditions} \big\}\, .$$

By fulfilling the conditions by function g, we mean that g is balanced and $\text{lin}(g) \le \ell$. We know that not only $f_1 \in \mathcal{F}_1^\dagger(f_0^*)$, but also all other coordinate functions must be in this set; i.e., for each $1 \le i < n$, $f_i \in \mathcal{F}_1^\dagger(f_0^*)$. Note that determining $\mathcal{F}_1^\dagger(f_0^*)$, for a given f_0^*, needs N_f computations and it includes $N_f \cdot p_1$ Boolean functions. Therefore, to build the S-box, in step 0, we choose $f_0^* \in \mathcal{F}^*$ and compute $\mathcal{F}_1^\dagger(f_0^*)$.

In step 1, after choosing $f_1 \in \mathcal{F}_1^\dagger(f_0^*)$, we compute the set of possible choices for f_2,

$$\mathcal{F}_2^\dagger(f_0^*, f_1) = \big\{ f \in \mathcal{F}_1^\dagger(f_0^*) \mid f \oplus f_1 \text{ and } f \oplus f_1 \oplus f_0^* \text{ fulfill the conditions} \big\}\, .$$

Note that since we only check for $f \in \mathcal{F}_1^\dagger(f_0^*)$, it already fulfills the conditions for $f \oplus f_0^*$. Again, we know that not only $f_2 \in \mathcal{F}_2^\dagger(f_0^*, f_1)$, but also all the next coordinate functions must be in this set; i.e., for each $2 \le i < n$, $f_i \in \mathcal{F}_2^\dagger(f_0^*, f_1)$. Determining $\mathcal{F}_2^\dagger(f_0^*, f_1)$, for given f_0^* and f_1, needs $N_f \cdot p_1$ computations and it includes $N_f \cdot p_1 \cdot p_2$ Boolean functions.

In step 2, after choosing $f_2 \in \mathcal{F}_2^\dagger(f_0^*, f_1)$, we compute the set of possible choices for f_3,

$$\mathcal{F}_3^\dagger(f_0^*, f_1, f_2) = \big\{ f \in \mathcal{F}_2^\dagger(f_0^*, f_1) \mid f \oplus f_2\, ,\ f \oplus f_2 \oplus f_1\, , f \oplus f_2 \oplus f_0^* \text{ and}$$
$$f \oplus f_2 \oplus f_1 \oplus f_0^* \text{ fulfill the conditions} \big\}\, .$$

Again, since we only check for $f \in \mathcal{F}_2^\dagger(f_0^*, f_1)$, it already fulfills the conditions for $f \oplus f_0^*$, $f \oplus f_1$, and $f \oplus f_1 \oplus f_0^*$. Besides, we know that not only $f_3 \in \mathcal{F}_3^\dagger(f_0^*, f_1, f_2)$, but also all the next coordinate functions must be in this set; i.e., for each $3 \le i < n$, $f_i \in \mathcal{F}_3^\dagger(f_0^*, f_1, f_2)$. Determining $\mathcal{F}_3^\dagger(f_0^*, f_1, f_2)$, for given f_0^*, f_1 and f_2, needs $N_f \cdot p_1 \cdot p_2$ computations and it includes $N_f \cdot p_1 \cdot p_2 \cdot p_3$ Boolean functions.

We continue in this way until we choose all the coordinate functions for the S-box and fulfill the conditions. Therefore, the built S-box is a bijection with linearity at most ℓ. Then, we can check for the condition on the S-box's uniformity. On average, this techniques reduce the computational complexity of building a bijective S-box to

$$\frac{1}{n!} \cdot |\mathcal{F}^*| \cdot N_f \cdot \Big(1 + p_1 \cdot (p_1 \cdot N_f) \cdot \big(1 + p_2 \cdot (p_2 \cdot N_f) \cdot \big(\ldots (1 + p_{n-1})\big)\big)\Big) \approx$$
$$\frac{1}{n!} \cdot |\mathcal{F}^*| \cdot N_f \cdot (1 + p_1^2 \cdot N_f + p_1^2 \cdot p_2^2 \cdot N_f^2 + \ldots + p_1^2 \cdot \ldots \cdot p_{n-2}^2 \cdot N_f^{n-2})$$

Clearly, the modification explained above makes the step-by-step algorithm much faster than its simpler version.

We provide a pseudo-code for our new method of building S-boxes at algorithm 3. This algorithm, in the simplest mode, needs to save all $N_f \approx |\mathcal{F}^*| \cdot n! \cdot 2^n$ Boolean functions, in the beginning, to reduce the redundant computations in the next steps. However, it is also possible to only save the Boolean functions in the set $F^\dagger(f_0^*)$ for each $f_0^* \in \mathcal{F}^*$. This way, we need to save only about $N_f \cdot p_1$ Boolean functions, significantly less than the previous way, but on the other hand, we need to repeat computing all N_f equivalent functions for $|\mathcal{F}^*|$ times. It is noteworthy that the value of p_1 is strongly related to the target properties for the S-boxes we are searching and also the functions in $\mathcal{F}^*$ which is not easy to compute.

Using the Upper Limit on the Uniformity: Similar to using the upper limit on the S-box's linearity, we can use the limit on the S-box's uniformity in the intermediate steps of the algorithm.

Lemma 4. *For an n-bit S-box $S = (f_0, \ldots, f_{n-1})$ with uniformity u, the uniformity of sub-S-box $S'_i = (f_0, \ldots, f_i)$ with $i < n$ is upper bounded by $\min\{u \cdot 2^{n-i-1}, 2^n\}$.*

Applying this lemma, in step i of the step-by-step algorithm, after choosing the coordinate function f_i, it is possible to check uniformity of the sub-S-box $S'_i = (f_0^*, f_1, \ldots, f_i)$. Note that for small i values, this condition does not filter the choices for the coordinate function. For instance, when $i = 0$, then uniformity of f_0^* is limited by $\min\{u \cdot 2^{n-1}, 2^n\} = 2^n$ that is a trivial condition and indeed we do not need to check it. For $i = 1$, the uniformity of (f_0^*, f_1) is limited by $\min\{u \cdot 2^{n-2}, 2^n\}$, and it is non-trivial if $u = 2$, i.e., it is meaningful to check this condition if we are looking for APN S-boxes.

Comparison to the Previous Method of Building S-box In [MB19], the authors used an algorithm to classify n- to m-bit quadratic balanced Boolean functions up to the affine equivalence with $n \le 6$, that this algorithm is the most applied algorithm for building S-boxes up to the applied equivalence, e.g., [Can07]. A pseudo-code of this is provided at algorithm 4.

In this algorithm, consider that $\mathcal{F}^*$ is the set of all n-bit representative Boolean functions under the equivalence that we want to use them to build bijective n-bit S-boxes. As the first step, we choose f_0^* from $\mathcal{F}^*$ and f_1 from the set of all functions that are equivalent to one of the functions in $\mathcal{F}^*$ that we denote by $\mathcal{F}$. By using these two functions together, we have an n- to 2-bit function of the form (f_0^*, f_1) that can be checked for chosen criteria (if there are any), e.g., balancedness and linearity. Trying this for all choices of f_0^* and

Algorithm 3: The new method of building n-bit bijective S-boxes.

```
Data: $\mathcal{F}^*$    // the set of all $n$-bit representative balanced Boolean functions with
      linearity of at most $\leq \ell$ (and extra criteria such as their latency
      complexity)
Result: $\mathcal{R}$   // the set of all $n$-bit representative bijective S-boxes with linearity
      of at most $\ell$ and uniformity of at most $u$

1  $\mathcal{R} \leftarrow \emptyset$ and $\mathcal{F} \leftarrow \emptyset$
2  foreach $f_0^* \in \mathcal{F}^*$ do
3      $\mathcal{S} \leftarrow \emptyset$ , $\mathcal{R}' \leftarrow \emptyset$ and $\mathcal{F}_1^\dagger \leftarrow \emptyset$
4      $\mathcal{F} \leftarrow \mathcal{F} \cup \{f \,|\, f$ is equivalent to $f_0^*\}$
5      foreach $f_1 \in \mathcal{F}$ do
6          if $f_1 \oplus f_0^*$ fulfills conditions then
7              $\mathcal{F}_1^\dagger \leftarrow \mathcal{F}_1^\dagger \cup \{f_1\}$
8      foreach $f_1 \in \mathcal{F}_1^\dagger$ do
9          $\mathcal{F}_2^\dagger \leftarrow \emptyset$
10         foreach $f_2 \in \mathcal{F}_1^\dagger$ do
11             if $f_2 \oplus f_1$ and $f_2 \oplus f_1 \oplus f_0^*$ fulfill conditions then
12                 $\mathcal{F}_2^\dagger \leftarrow \mathcal{F}_2^\dagger \cup \{f_2\}$
13         foreach $f_2 \in \mathcal{F}_2^\dagger$ do
14             $\mathcal{F}_3^\dagger \leftarrow \emptyset$
15             foreach $f_3 \in \mathcal{F}_2^\dagger$ do
16                 if $f_3 \oplus f_2, f_3 \oplus f_2 \oplus f_1, f_3 \oplus f_2 \oplus f_0^*$ and $f_3 \oplus f_2 \oplus f_1 \oplus f_0^*$ fulfill conditions
                    then
17                     $\mathcal{F}_3^\dagger \leftarrow \mathcal{F}_3^\dagger \cup \{f_3\}$
18             foreach $f_3 \in \mathcal{F}_3^\dagger$ do
19                 ...
20                 $\mathcal{F}_{n-2}^\dagger \leftarrow \emptyset$
21                 foreach $f_{n-2} \in \mathcal{F}_{n-3}^\dagger$ do
22                     if $2^{n-3}$ functions of $f_{n-2} \oplus f_{n-3}, \ldots,$ and $f_{n-2} \oplus f_{n-3} \oplus \ldots \oplus f_0^*$
                        fulfill conditions then
23                         $\mathcal{F}_{n-2}^\dagger \leftarrow \mathcal{F}_{n-2}^\dagger \cup \{f_{n-2}\}$
24                 foreach $f_{n-1} \in \mathcal{F}_{n-2}^\dagger$ do
25                     if $2^{n-2}$ functions of $f_{n-1} \oplus f_{n-2}, \ldots,$ and $f_{n-1} \oplus f_{n-2} \oplus \ldots \oplus f_0^*$
                        fulfill conditions then
26                         $S \leftarrow (f_0^*, f_1, f_2, \ldots, f_{n-1})$
27                         if $\mathrm{uni}(S) \leq u$ then
28                             $\mathcal{S} \leftarrow \mathcal{S} \cup \{S\}$
29     foreach $S \in \mathcal{S}$ do
30         new $\leftarrow$ 1
31         foreach $R \in \mathcal{R}'$ do
32             if AreTheyEquivalent$(S, R)$ then
33                 new $\leftarrow$ 0 and break the loop
34         if new = 1 then
35             $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{$ComputeRepresentativeFunction$(S)\}$
36     $\mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{R}'$
```

Algorithm 4: The previous method of building n-bit bijective S-boxes in [Can07].

```
Data: $\mathcal{R}_1$            // the set of all $n$-bit representative balanced Boolean functions
Result: $\mathcal{R}_n$          // the set of all $n$-bit representative bijective S-boxes
1  $\mathcal{F} \leftarrow \{f \mid f$ is equivalent to $f^* \in \mathcal{R}_1\}$
2  for $m \leftarrow 1$ to $n-1$ do
3      $\mathcal{S} \leftarrow \emptyset$ and $\mathcal{R}_{m+1} \leftarrow \emptyset$
4      foreach $R \in \mathcal{R}_m$ do
5          foreach $f_m \in \mathcal{F}$ do
6              $F \leftarrow (R, f_m)$
7              if IsItBalancedFunction($F$) then
8                  $\mathcal{S} \leftarrow \mathcal{S} \cup \{F\}$
9      foreach $F \in \mathcal{S}$ do
10         new $\leftarrow$ 1
11         foreach $R \in \mathcal{R}'$ do
12             if AreTheyEquivalent($S, R$) then
13                 new $\leftarrow$ 0 and break the loop
14         if new then
15             $\mathcal{R}_{m+1} \leftarrow \mathcal{R}_{m+1} \cup \{$ComputeRepresentativeFunction($S$)$\}$
```

f_1, we have a set of n- to 2-bit functions that some are equivalent to each other. We remove all equivalent ones and only keep one function within each equivalence class as a representative for that class and put it in a set called $\mathcal{R}_2$. Note that here, we do not need to consider the lexicographically smallest function as the representative one.

In the second step, we use one function from $\mathcal{R}_2$ and another function from $\mathcal{F}$. By using these two functions together, we have an n- to 3-bit function that can be checked for the criteria. Again, we try this for all choices of those two functions, and then we have a set of n- to 3-bit functions that some are equivalent to each other. By removing all the equivalents and only keeping one for each equivalence class, we build a set called $\mathcal{R}_3$.

Similarly, we do another $n-3$ steps to build $\mathcal{R}_4, \ldots, \mathcal{R}_n$ that the latest one is the set of all n-bit S-boxes up to the equivalence. Note that this algorithm can be applied for different equivalences and not only for linear or affine equivalence.

While this method is efficient for classifying n- to m-bit Boolean functions under linear or affine equivalences, for our need it is not that efficient. The main reason for this is that the applied equivalences there and here are different. Extended bit permutation equivalence is covered by affine equivalence. Hence, the number of equivalence classes in the extended bit permutation equivalence is more than the number of classes in affine equivalence. Indeed, there is a big difference in these numbers. While number of n- to m-bit Boolean functions belonging to an extended bit permutation equivalence class is about $2^{n+m} \cdot n! \cdot m!$ (note that for simplicity here, we are not considering self-equivalent Boolean functions), in the affine equivalence this number is about $2^{n+m} \cdot \prod_{i=0}^{n-1}(2^n - 2^i) \cdot \prod_{i=0}^{m-1}(2^m - 2^i) \approx 2^{n^2+m^2+n+m-2}$. Therefore, the ratio of the probability that two n- to m-bit *randomly chosen* Boolean functions are extended bit permutation equivalent to the probability that they are affine equivalent is quite small, $n! \cdot m! \cdot 2^{-n^2-m^2+2}$. For example, for $n = m = 4$, this ratio is about 2^{-20} and for larger n values, it gets smaller. We emphasize that this probability is not the case for ours, because it considers randomly chosen Boolean functions, while we are dealing with balanced Boolean functions and with a limit on their linearity, hence they are not randomly chosen, and also it does not consider the self-equivalent functions. However, even in our case, the ratio of these two probabilities is very small.

Therefore, we find the new method more suitable and efficient for our case to find bijective S-boxes with small latency complexity. Then, if there is any possible S-box to build, by applying another algorithm, we reduce the equivalent S-boxes up to the extended bit permutation equivalence. This algorithm of reducing the equivalent S-boxes is based on the one explained and given in [BCBP03, MB19] for linear and affine equivalence. We modified that algorithm for the case of extended bit permutation equivalence and made it efficient by applying properties of this equivalence that generally do not hold for the linear or affine equivalence.

For comparison, we applied both methods to build 5-bit S-boxes with latency complexity 3, linearity 16, and uniformity at most 8. For this, we used the balanced Boolean functions in $\mathcal{F}_{5,3}$, together with the extension of Boolean functions in $\mathcal{F}_{4,3}$ and $\mathcal{F}_{3,2} \cup \mathcal{F}_{3,3}$ to 5-bit Boolean functions. Altogether, up to the bit permutation equivalence, there are only 10 balanced Boolean functions with linearity 16. Using these Boolean functions and their equivalents, based on our algorithm, it is possible to build about six thousand S-boxes with linearity 16 and uniformity 6 or 8 that up to the equivalence they are only 509 S-boxes. The time to find the S-boxes takes needs 2 minutes and another 2 minutes for removing the equivalent S-boxes. We also applied the previous method to build such S-boxes which takes about 96 minutes. Note that for both computations, we used a single thread on an Intel i7-10610U CPU @ 1.80 GHz CPU.

As the example shows, our method is more suitable for the extended bit permutation equivalence. We should mention that if the number of starting coordinate functions increases (here it was 10), or we search for larger S-boxes, the efficiency of our method increases.

5.2 Results on the Bijective S-boxes with Low-Latency Complexity

To build low-latency S-boxes, we start with the lowest possible limit on the linearity or uniformity of the S-box. If it is not possible to build any S-boxes with such criteria, we increase the target linearity and uniformity and repeat the search for building S-boxes. This way, we can find the best possible S-boxes with respect to linearity and/or uniformity.

In the following, we report the *best* results with respect to their linearity and uniformity for building n-bit S-boxes. Note that the list for all of these S-boxes is presented in [Ras22]. We emphasize that we only search for the bijective S-boxes and the reported number of Boolean functions and the number of S-boxes are up to the extended bit permutation equivalence. Also, we use $\mathcal{F}_{n,d,\ell}$ as the set of all n-bit full-dependent balanced Boolean functions with a latency complexity *at most* d and linearity of *at most* ℓ.

3-bit S-boxes There is one function in $\mathcal{F}_{3,2,4}$ which is equivalent to a multiplexer (see Example 1). Using this function, it is possible to build two S-boxes with linearity 8 and uniformity 4.

For latency complexity of 3, since all the 3-bit balanced Boolean functions with linearity 4 are included in $\mathcal{F}_{3,3,4}$, we can build all the bijective 3-bit S-boxes with linearity 4 and uniformity 2 those are affine equivalent to the inversion in $\mathbb{F}_{2^3}$. Up-to the extended bit permutation equivalence, there are 7 of such S-boxes. Note that for the last S-box in this list, all the coordinate functions are equivalent up the extended bit permutation.

4-bit S-boxes: By extending the only function in $\mathcal{F}_{3,2,4}$ to a 4-bit Boolean function, it is possible to build one bijective and quadratic S-box whose linearity and uniformity, both are 16.

For latency complexity of 3, there are 4 functions in $\mathcal{F}_{4,3,8}$ with linearity 8 that algebraic degree of all these functions is 3. Using these functions, it is possible to build 152 S-boxes with linearity 8 and uniformity of 4 that both are the minimum values for a 4-bit S-box. Next, we combine $\mathcal{F}_{4,3,8}$ with extension of Boolean functions in $\mathcal{F}_{3,3,4}$ to 4-bit Boolean functions. Using these functions, it is possible to build another 129 S-boxes with a linearity 8 and uniformity of 4.

It is noteworthy to mention that the inverse of S-boxes number 73 from the first set, 32 and 38 from the second set have latency complexity of 3 and also they can produce involutive S-boxes. Besides, from the first set, coordinate functions for S-boxes number 22, 41, 42, 55, 80, 99, 136 and 147 are equivalent while there is no such an S-box from the second set with this property.

In comparison with previously known S-boxes, the only S-boxes with latency complexity 3 are `Midori`-s_0 and the non-bijective χ_4 that the first one is equivalent to the representative S-box number 32 from the second above-mentioned set. However, for the same level of latency complexity, uniformity and linearity, the new S-boxes offer a wide variety with respect to the algebraic degree of coordinate or component functions.

Moreover, since $\mathcal{F}_{4,4,8}$ includes all 4-bit balanced Boolean functions with linearity 8, therefore, within latency complexity of 4, we can build any 4-bit S-box with linearity 8 and uniformity 4, those are named as *Golden* S-boxes by [LP07].

5-bit S-boxes: By extending the function in $\mathcal{F}_{3,2,4}$ to a 5-bit Boolean function, it is possible to build 13 different bijective S-box whose linearity and uniformity, both are 32.

$\mathcal{F}_{5,3}$ includes 4 balanced Boolean functions with linearity 16 those build $\mathcal{F}_{5,3,16}$. Using these functions, it is possible to build 4 S-boxes with a linearity of 16 and uniformity of 6. Note that all the coordinate functions of these S-boxes and their inverse S-boxes have an algebraic degree of 4. Besides, all the coordinate functions of the last S-box, are extended bit permutation equivalent of each other.

Using the extended Boolean functions of $\mathcal{F}_{3,3,4}$ and $\mathcal{F}_{4,3,8}$ to 5-bit Boolean functions together with the ones in $\mathcal{F}_{5,3,16}$ does not improve the minimum achievable linearity or uniformity in the S-boxes. It only gives another 9 S-boxes with the same linearity and uniformity values.

For the latency complexity of 4, there are 93 functions in $\mathcal{F}_{5,4,8}$. Using these functions, it is possible to build 2514 different S-boxes with linearity 8 and uniformity 2 (due to the large number of these S-boxes, we only provide them in [Ras22]). It is noteworthy to mention that the coordinate functions of these S-boxes have an algebraic degree of 2 or 3, and are equivalent to only 28 functions (out of 93).

Moreover, since all balanced 5-bit Boolean functions with minimum linearity are included in $\mathcal{F}_{5,5,8}$, any 5-bit S-box with linearity 8 has latency complexity of at most 5.

In comparison with previously known S-boxes, the only known S-box with latency complexity 3 is χ_5 that is used in `KECCAK` and has uniformity 8 and linearity 16. For the same level of latency complexity, uniformity 6 or 8, and linearity 16, the new S-boxes offer a wide variety with respect to the algebraic degree of coordinate or component functions and also with a higher dependency of the coordinate functions on the input variables (while for χ function is always 3 bits).

For the case of S-boxes with uniformity 2 and linearity 8, while the previously known S-boxes all have a latency complexity of 5, our new proposed S-boxes can achieve these properties within the latency complexity of 4. It is noteworthy to mention that the algebraic degree of the coordinate or component functions of these new S-boxes is either 2 or 3; there are also some S-boxes with all quadratic or all cubic coordinates.

6-bit S-boxes: Using the extension of the Boolean function in $\mathcal{F}_{3,2,4}$ to a 6-bit Boolean function, it is possible to build 3 bijective S-box whose linearity and uniformity are 64 and 32, respectively. But these S-boxes are a combination of two parallel 3-bit S-boxes found previously in 3-bit S-boxes. Excluding this kind of S-boxes, it is possible to build 19 bijective 6-bit S-boxes whose linearity and uniformity both are 64.

$\mathcal{F}_{6,3}$ includes 3 balanced Boolean functions with 3 different linearities, 24, 32 and 40. Using these Boolean functions, it is possible to build an S-box with linearity 64 and uniformity 20 that all the coordinate functions of the S-box equivalent to the representative function in $\mathcal{F}_{6,3}$ with linearity 32 and algebraic degree 5.

Then, we use these two functions with less linearity, i.e., $\mathcal{F}_{6,3,32}$, together with extension of the Boolean functions in $\mathcal{F}_{5,3,16}$ to 6-bit Boolean functions. It is possible to build another S-box with the same linearity and uniformity that all coordinate functions for this S-box have an algebraic degree of either 4 or 5.

Next, by combining $\mathcal{F}_{6,3,32}$ and extension of Boolean functions in $\mathcal{F}_{5,3,16}$ and $\mathcal{F}_{4,3,8}$ to 6-bit Boolean functions, we could not find any S-box with the same or better linearity or uniformity. One step more, we combine the extension of the Boolean functions in $\mathcal{F}_{3,3,4}$ to 6-bit Boolean functions, with the other three above-mentioned sets, to see if there is any S-box with better uniformity or linearity. We found 49 new S-boxes (excluding the ones that are a combination of two parallel 3-bit S-boxes) with linearity 32 and uniformity 16. It is noteworthy to mention that all the coordinate functions of each last 5 S-boxes in this list, are equivalent to each other and a single function in $\mathcal{F}_{3,3,4}$. Besides, within these 49 S-boxes, there are 15 quadratic ones (i.e., for each of these S-boxes, the algebraic degree of all coordinate functions is 2) which are usually considered to be suitable for side-channel countermeasures.

For the case of latency complexity 4, we first start to build quadratic bijective S-boxes. There are only 2 quadratic Boolean functions in $\mathcal{F}_{6,4}$; one with linearity 16 and another with linearity 32. Using the one with linearity 16, it is possible to build 4 S-boxes with both linearity and uniformity 32. However, by involving the extension of quadratic 5-bit Boolean functions $\mathcal{F}_{5,4,8}$ to 6-bit Boolean functions (there are 4 such functions), it is possible to build 908 S-boxes with linearity 16 and uniformity 4. Due to the large number of these S-boxes, we only provide them in [Ras22]. It is noteworthy to mention that in this list, there are two S-boxes that coordinate functions of each S-box are equivalent to each other, namely S-boxes number 496 and 775.

Then, we try to build full-dependent S-boxes and probably with higher algebraic degree. $\mathcal{F}_{6,4}$ includes 1546 balanced Boolean functions with linearity 16 that are forming $\mathcal{F}_{6,4,16}$. Using these Boolean functions, we find another S-box with linearity 16 and uniformity 4 that algebraic degree of its two coordinate functions are 3 and for other ones are 2.

```
(00,01,02,03,04,06,3e,3c,08,11,0e,17,2b,33,35,2d,19,1c,09,0c,15,13,3d,3b,31,2c,25,
 38,3a,26,36,2a,34,1d,37,1e,30,1a,0b,21,2e,1f,29,18,0f,3f,10,20,28,05,39,14,24,0a,
 0d,23,12,27,07,32,1b,2f,16,22)
```

We emphasize that since we did not finish all the computations for this search (we only searched about 9.2 % of the whole space), we are not sure if this is the only full-dependent 6-bit S-box with latency complexity 4, linearity 16 and uniformity 4. But we know that based on the Boolean functions in $\mathcal{F}_{6,4,16}$, it is not possible to build an S-box with linearity 16 and uniformity 4 whose coordinate functions are equivalent.

In comparison with the previously known 6-bit S-boxes, the only ones with latency complexity less than 5, are χ_6 and `Speedy` S-boxes. χ_6 is a non-bijective function with latency complexity 3, linearity 32 and uniformity 16. It is comparable with our 49 bijective S-boxes with a wide variety with respect to the algebraic degree of coordinate or component functions and also with a higher dependency of the coordinate functions on the input variables.

`Speedy` S-box has a latency complexity of 4, linearity 24 and uniformity 8. For the same level of latency complexity, we presented 908 quadratic S-boxes together with a hybrid-quadratic-cubic S-box, all with linearity 16 and uniformity 4. While the new S-boxes are better than `Speedy` S-box with respect to the uniformity or linearity, they have a smaller algebraic degree. However, if the target is to have a higher algebraic degree, we found several S-boxes with linearity 24, uniformity 6 and all coordinate functions have an algebraic degree of 5. Clearly, these S-boxes have better cryptographic properties than the `Speedy` S-box, but we recall that in the design of `Speedy` S-box, it was only restricted to have two layers of `NAND` gates with fan-in number of 3 or 4 (with the same latency complexity of 4) that makes the S-box has slightly lower latency than our S-boxes.

7-bit S-boxes: Using the extension of the function in $\mathcal{F}_{3,2,4}$ to a 7-bit Boolean function, it is possible to build 92 bijective S-box whose both linearity and uniformity are 128. Three of these S-boxes are parallel combination of a 3-bit S-box with a 4-bit S-box.

We repeat a similar approach to 6-bit S-boxes with latency complexity 3 for 7-bit S-boxes. We first use extension of 6-bit functions in $\mathcal{F}_{6,3,32}$ to 7-bit Boolean functions, to see if it is possible to build 7-bit bijective S-boxes. Then, we use $\mathcal{F}_{6,3,32} \cup \mathcal{F}_{5,3,16}$, $\mathcal{F}_{6,3,32} \cup \mathcal{F}_{5,3,16} \cup \mathcal{F}_{4,3,8}$ and $\mathcal{F}_{6,3,32} \cup \mathcal{F}_{5,3,16} \cup \mathcal{F}_{4,3,8} \cup \mathcal{F}_{3,3,4}$, step by step. Except in the last two step, it is not possible to build any bijective S-boxes. For the case of $\mathcal{F}_{6,3,32} \cup \mathcal{F}_{5,3,16} \cup \mathcal{F}_{4,3,8}$ there are many S-boxes with linearity and uniformity, both to be 128. Using $\mathcal{F}_{6,3,32} \cup \mathcal{F}_{5,3,16} \cup \mathcal{F}_{4,3,8} \cup \mathcal{F}_{3,3,4}$, there are 1074 S-boxes with linearity 64 and uniformity 32 that $152 \times 7 = 1064$ of them are parallel combination of a 3-bit S-box with a 4-bit S-box those found previously in 3-bit and 4-bit S-boxes with latency complexity 3. It is noteworthy to mention that the coordinate functions of these S-boxes, all are extended bit permutation equivalent to extension of one function in $\mathcal{F}_{4,3,8}$ and two functions from $\mathcal{F}_{3,3,4}$ to 7-bit Boolean functions. Moreover, in the last two S-boxes of this list, all coordinates are extended bit permutation equivalent of each other and equivalent to the extension of one function in $\mathcal{F}_{3,3,4}$.

For the case of latency complexity 4, we only checked building quadratic bijective S-boxes. Using extension of the single quadratic 6-bit Boolean function in $\mathcal{F}_{6,4,32}$ to a 7-bit function, it is possible to build 1110 S-boxes with linearity 64 and uniformity 32. Due to the large number of these S-boxes, we only provide them in [Ras22]. One step forward, by involving the extension of four quadratic 5-bit Boolean functions $\mathcal{F}_{5,4,8}$ to 7-bit Boolean functions, it is possible to build 134 S-boxes with linearity 32 and uniformity 8. It is noteworthy to mention that coordinate functions for two S-boxes of this list (namely S-boxes number 51 and 133) are equivalent to each other.

There are only two previously known 7-bit S-boxes: χ_7 with a latency complexity 3, linearity 64 and uniformity 32, and `Wage` S-box with a latency complexity higher than 4, linearity 40 and uniformity 8. χ_7 is comparable with the new 10 S-boxes with an algebraic degree of 2 or 3 for the coordinate or component functions and with a higher dependency of the coordinate functions on the input variables.

`Wage` S-box has a latency complexity of at least 5. However, for the latency complexity of 4, we presented 134 quadratic S-boxes with linearity 32 and uniformity 8. Clearly, the new S-boxes are better than `Wage` S-box with respect to the latency complexity, uniformity or linearity, but they have a smaller algebraic degree. If the target is to have an S-box with a latency complexity of 4 and a higher algebraic degree, it should be investigated what is the best achievable linearity and uniformity.

8-bit S-boxes: Using the extension of the Boolean function in $\mathcal{F}_{3,2,4}$ to an 8-bit Boolean function, it is possible to build 221 bijective S-box whose linearity and uniformity, both are 256. 40 of these S-boxes are a parallel combination of two 4-bit S-boxes or one 3-bit and one 5-bit S-boxes and the other 181 S-boxes are provided in [Ras22].

We repeat a similar approach to 6- and 7-bit S-boxes with latency complexity 3 for 8-bit S-boxes. We use extension of functions in $\mathcal{F}_{6,3,32}$, $\mathcal{F}_{6,3,32} \cup \mathcal{F}_{5,3,16}$, $\mathcal{F}_{6,3,32} \cup \mathcal{F}_{5,3,16} \cup \mathcal{F}_{4,3,8}$ and $\mathcal{F}_{6,3,32} \cup \mathcal{F}_{5,3,16} \cup \mathcal{F}_{4,3,8} \cup \mathcal{F}_{3,3,4}$, step by step, to see if it is possible to build any 8-bit bijective S-boxes. Excluding two last steps, it is not possible to build any 8-bit S-boxes whose linearity and uniformity both are less than 256. Using $\mathcal{F}_{6,3,32} \cup \mathcal{F}_{5,3,16} \cup \mathcal{F}_{4,3,8}$, we find 11628 S-boxes with linearity 128 and uniformity 64 that all of these S-boxes are parallel combination of two 4-bit S-boxes, those found previously in 4-bit S-boxes.

Using $\mathcal{F}_{6,3,32} \cup \mathcal{F}_{5,3,16} \cup \mathcal{F}_{4,3,8} \cup \mathcal{F}_{3,3,4}$, we find another 12032 S-boxes with linearity 128 and uniformity 64 that 3563 of these S-boxes are parallel combination of a 3-bit S-box with a 5-bit S-box and 8385 of them are combination of two 4-bit S-boxes. The other 84 S-boxes are listed at [Ras22]. It is noteworthy to mention that the coordinate functions of the remaining 84 S-boxes, all are quadratic and extended bit permutation equivalent to the extension of two functions from $\mathcal{F}_{3,3,4}$ to 8-bit Boolean functions. Moreover, within these S-boxes, there are 29 S-boxes (the ones with index 19, 29, 31, 34, 35, 36, 41, 42, 49, 50, 51, 55, 56, 57, 58, 60, 61, 62, 68, 69, 72, 73, 75, 76, 79, 80, 81, 82 and 83) that all coordinates are extended bit permutation equivalent of each other and to the extension of one function in $\mathcal{F}_{3,3,4}$.

The only previously known 8-bit S-box with a latency complexity of less than 5 is χ_8 which is a non-bijective S-box with latency complexity 3, linearity 64 and uniformity 128. In `Skinny-8` and `CSS` S-boxes, and also in their inverses, there are four coordinates whose latency complexity is 3 or 4, but the other four coordinates it is higher than 4. For the latency complexity of 3, we introduced new 84 quadratic and bijective S-boxes with linearity 128 and uniformity 64.

Latency Complexity vs. Minimum Achievable Uniformity and Linearity: As reported, up to 8-bit S-boxes, there is no n-bit S-box with a latency complexity of 2 whose both linearity and uniformity are smaller than 2^n; therefore, to achieve such a property, we need to use S-boxes with a latency complexity of at least 3.

For latency complexity of 3, while there are 3- and 4-bit S-boxes with the minimum linearity and uniformity, for larger S-boxes (with respect to the input size), this is not achievable. Generally, for n-bit S-boxes with $3 \leq n \leq 8$ and latency complexity 3, the minimum achievable linearity and uniformity are 2^{n-1} and 2^{n-2}, respectively, (except for 5-bit S-boxes in which the minimum achievable uniformity is 6).

For latency complexity of 4, it is possible to achieve the minimum linearity and uniformity for 5-bit S-boxes which are 8 and 2, respectively. For 5-, 6-, and 7-bit S-boxes, the minimum achievable linearity and uniformity is 2^{n-2} and 2^{n-4}, respectively, and this is probably the case for 8-bit S-boxes and it is interesting to be investigated.

6 Hardware Implementation of a Low-Latency Structure

While the proposed structure in Figure 2 helps us to study the Boolean functions with latency complexity d, it does not promise the lowest latency in a real hardware implementation. In the following, we explain the reasons behind this statement. Then we describe our approach for optimizing the suggested structures produced by the algorithm in Subsection 4.4 to find a circuit with the lowest latency in an ASIC hardware implementation. We apply our approach to find efficient implementations for previously introduced S-boxes that are minimized with respect to the latency and then its area.

6.1 Optimizing the Low-Latency Structure for a Boolean Function

The fact that each structure produced by the algorithm in Subsection 4.4 (which usually follows the structure in Figure 2) do not promise the lowest latency in reality is because we modeled the latency, a complicated hardware parameter, with an over-simplified metric, the latency complexity. Here, we describe three main reasons that cause the latency difference between the structures suggested by the algorithm in Subsection 4.4 for a given Boolean function.

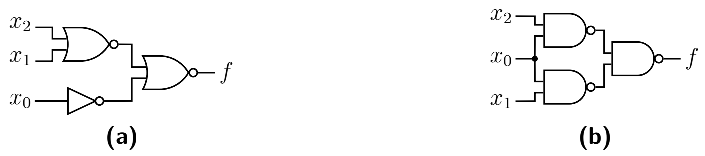


Figure 3: Two different circuits with the minimum gate depth for implementing $f = x_0 \wedge (x_1 \vee x_2)$ used in Example 3.

Different Structures for the Same Function: We recall that for the given Boolean function the structure suggested by the algorithm in Subsection 4.4 might not be unique. We emphasize that this algorithm already reduces the trivial equivalent structures that are explained in Subsection 4.2. The following example is a good instance of that different structures (with the minimum gate depth) for the same function can have different latency values.

Example 3. Let $f(x_0, x_1, x_2)$ be a 3-bit Boolean function with $f = x_0 \wedge (x_1 \vee x_2)$. The latency complexity of this function is two and it can be implemented using two different structures based on $\overline{x}_0 \mathbin{\overline{\vee}} (x_1 \mathbin{\overline{\vee}} x_2)$ and $(x_0 \mathbin{\overline{\wedge}} x_1) \mathbin{\overline{\wedge}} (x_0 \mathbin{\overline{\wedge}} x_2)$ equations. We depict the corresponding circuits for each of these structures in Figure 3.

While for the second circuit, both of the sub-circuits have gate depth 1, in the first circuit, the sub-circuits for gate depth 1, have different latency complexity; for one of them ($\overline{x}_0$) is zero and for the other one is 1. Besides, in the first circuit, the input $\overline{x}_0$ is repeated only once, while in the second circuit, the input x_0 is repeated twice. This means that in the first circuit, the `INV` gate for variable x_0 has fan-out number 1, but in the second circuit, the `BUF` gate for the same variable has fan-out number 2.

These differences in the latency of sub-circuits and in the number of fan-out numbers for the `INV` and `BUF` gates for the input variable x_0, cause that the first circuit to have a lower latency than the one for the second circuit.

In the aforementioned equations, inputs of the equations are chosen directly from inputs of the combinatorial circuit. If inputs of the equations each comes from other sub-circuits, i.e., $f = f_0 \wedge (f_1 \vee f_2)$, with f_0, f_1, f_2 and f each being an n-bit Boolean function, then the latency of two circuits based on realizing $\neg f_0 \mathbin{\overline{\vee}} (f_1 \mathbin{\overline{\vee}} f_2)$ and $(f_0 \mathbin{\overline{\wedge}} f_1) \mathbin{\overline{\wedge}} (f_0 \mathbin{\overline{\wedge}} f_2)$ can be much bigger than the case for $f = x_0 \wedge (x_1 \vee x_2)$.

The above example shows one of the biggest differences in the latency of different circuits for the same function, which is because of the shorter gate depth in one sub-circuit than in the other one. However, there might be small differences in the latency of different circuits, but with the same gate-depth value for both sub-circuits. Therefore, for a given function if there are possible structures such that the gate depth in one of the sub-circuits is smaller than the other one (such as the first structure in the above example), is preferred over other suggested structures. Otherwise, if in all of the structures, the gate depth of the sub-circuits are the same, we must consider all the structures.

Gates with Higher Fan-Out Number: In our structures, we assumed that from each variable x_i there are two wires coming to the combinatorial circuit; one goes to a `BUF` gate and the other one goes to an `INV` gate. The fan-out number of these two gates is dependent on the number of times that x_i or $\neg x_i$ are used in the input of `NAND` or `NOR` gates in the depth level 1. If in a combinatorial circuit, for some variable there is a `BUF` or `INV` whose fan-out number is high, it will increase the latency of the circuit which is in contrast with our assumption that the `INV` and `BUF` gates in the gate level 0 of the proposed structure in Figure 2 is much smaller than the latency of the rest of the circuit. Therefore, we should reduce the fan-out number of these `BUF` and `INV` gates to reach a lower latency.

The suggested structures by our algorithm, all are based on 2-bit NAND and NOR gates with fan-out number 1. In some structures, it might be possible to use such gates but with a higher fan-out number to reduce the fan-out number for BUF and INV gates. Consider the case that there are two sub-circuits with outputs of f_0 and f_1 with $f_0 = f_1$ for all input values. Then instead of having two separate sub-circuits for each of f_0 and f_1, we can keep one of these sub-circuits just by increasing the fan-out number of the latest gate in this sub-circuit. Thereby, we reduce the fan-out number of several BUF and INV gates, just by increasing the fan-out number of a single gate in a middle depth level. This kind of simplification, not only possibly reduces the latency of implementation, it reduces its corresponding area.

Note that within the suggested structures by our algorithm, there might be different ones that lead to the same updated circuit after these simplifications. In this case, we omit the repeated circuits for the next step.

Different Gate Types in ASIC Libraries: The proposed structures, all are based on only 2-bit NAND and NOR gates. However, depending on the ASIC technology used for the hardware implementation, there are other logic gates with a higher fan-in number that might be more efficient (with respect to the latency and the area) than its representation with the basis of 2-bit NAND and NOR gates. Therefore, since we exclude the gates with a higher fan-in number, the corresponding circuits for the suggested structures do not necessarily provide the lowest possible latency.

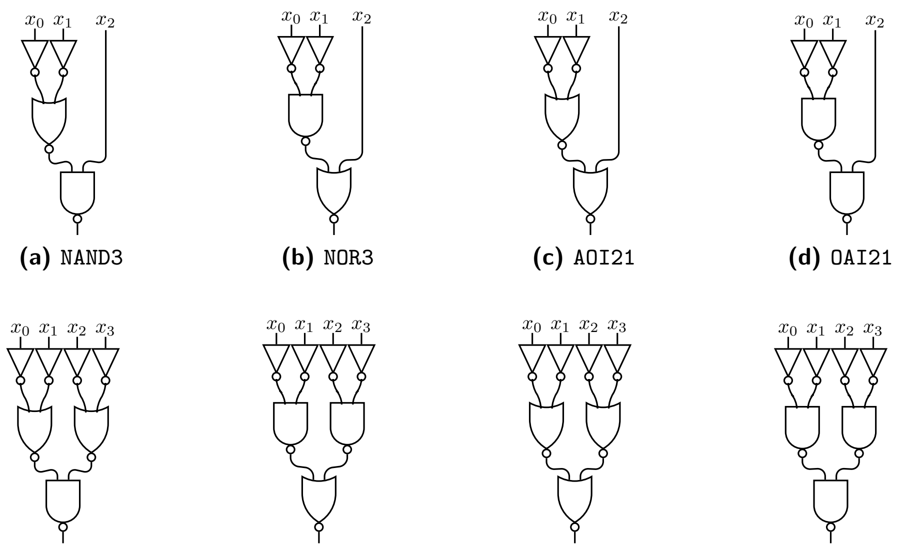
In [LMMR21, Section 2], the authors studied the latency behavior of logic gates and their combinations in the CMOS hardware. There, it is explained in detail that compared to the other gates with the same fan-in number, NAND and OAI gates are the most suitable gates to achieve a low-latency implementation. Hence, we need to adapt the circuits for each suggested structure to apply the other low-latency gates to find the implementation with the lowest possible latency for the corresponding function. We suggest not considering only those 2 gates (with the *best* latency behavior) but to consider also similar gates (with *good* latency behavior) such as NOR and AOI gates.

Thereby, for each suggested structure (remaining from the previous step), we suggest trying all possible replacements for the aforementioned gates with the corresponding sub-circuit (in the basis of NAND, NOR, and INV gates) and evaluate latency of the updated circuit. We emphasize that each of these replacements does not necessarily reduce the latency of the implementation, but for achieving the lowest latency, we need to check all combinations of these possible replacements. In Figure 4, we provide the corresponding sub-circuits (in the basis of 2-bit NAND, NOR and INV gates) for NAND, NOR, OAI and AOI gates with fan-in number of 3 and 4.

The possible improvements in this step by using the gates with higher fan-in numbers generally depend on the transistor-level design of gates and the corresponding conditions used in the given library's technology. While a single replacement of a gate with a higher fan-in number by the corresponding sub-circuit, in one technology, can improve the latency does not necessarily mean it is the same in other technology. Even in the same technology, a possible improvement by replacing an specific gate type with its corresponding sub-circuit in one circuit does not insure an improvement for replacing the same kind of gate in another circuit. Therefore, we suggest trying all possible replacements of the gates with higher fan-in numbers separately for each targeted technology.

Including XOR and XNOR Gates: In our studies, to have a better metric for the latency of a circuit, we exclude XOR and XNOR gates from the basis of defining the latency complexity. However, similar to the previous step, we can use equivalent sub-circuits for these two gates to simplify the corresponding circuits for the suggested structures.

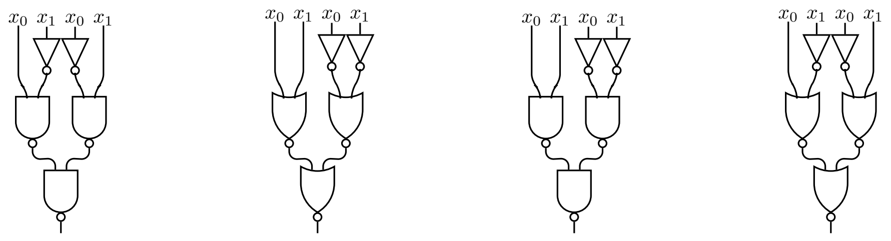
In Figure 5, we provide the corresponding sub-circuits for 2-bit XOR and XNOR gates.



(a) `NAND3` **(b)** `NOR3` **(c)** `AOI21` **(d)** `OAI21`

(e) `NAND4` **(f)** `NOR4` **(g)** `AOI22` **(h)** `OAI22`

Figure 4: The equivalent sub-circuits for `NAND`, `NOR`, `OAI` and `AOI` gates with fan-in number of 3 and 4 in the basis of 2-bit `NAND`, `NOR` and `INV` gates.



(a) `XOR` **(b)** `XOR` **(c)** `XNOR` **(d)** `XNOR`

Figure 5: The equivalent sub-circuits for 2-bit `XOR` and `XNOR` gates in the basis of 2-bit `NAND`, `NOR` and `INV` gates.

As depicted in the figures, for implementing f_0 `XOR` f_1 or f_0 `XNOR` f_1, before any of these replacements, we need to have separate sub-circuits for each $f_0, f_1, \neg f_0, \neg f_1$ functions together with 3 `NAND` or `NOR` gates. But, after the replacement, we need to have separate sub-circuits only for f_0 and f_1 functions together with an `XOR` or `XNOR` gate.

Note that latency of an `XOR` or an `XNOR` gate is higher than the latency of corresponding equivalent structures (see Example 2). But due to reducing the fan-out number of `BUF` or `INV` gates in the depth level 0, it can probably reduce latency of whole circuit. However, this replacements can reduce the area of implementation significantly.

Vectorial Boolean Functions: To find the circuit with the minimum latency for implementing a vectorial Boolean function, we can use a similar approach as for a Boolean function. First, using the algorithm in Subsection 6.1 for each coordinate function, we find all the possible structures with minimum gate depth. Then, for each combination of the structures for the coordinate functions, we repeat the aforementioned steps for replacing the gates with a higher fan-out number, or `XOR` and `XNOR` gates, or other different gate types provided by the ASIC library. In other meaning, for an n-bit to m-bit vectorial Boolean function, if there are $N_{C_0}, N_{C_1}, \ldots,$ and $N_{C_{m-1}}$ possible structures, respectively

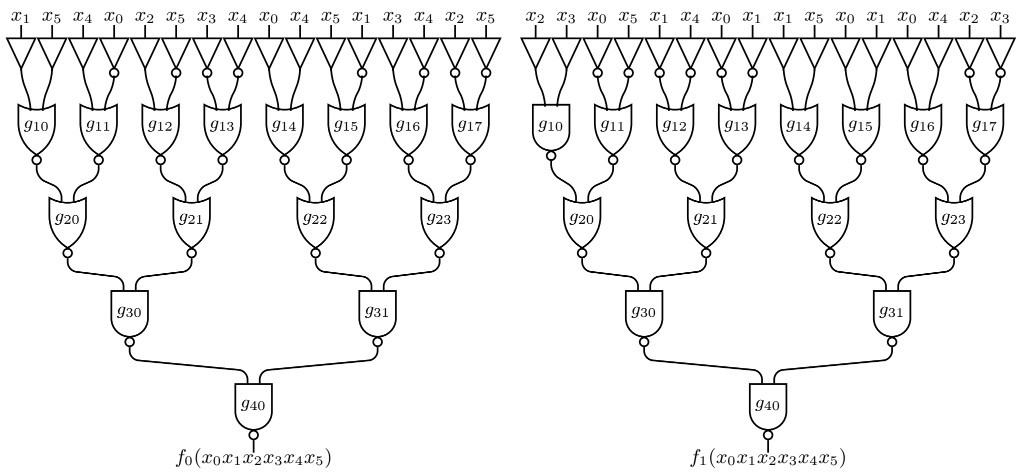

x_1 x_5 x_4 x_0 x_2 x_5 x_3 x_4 x_0 x_4 x_5 x_1 x_3 x_4 x_2 x_5
g_{10} g_{11} g_{12} g_{13} g_{14} g_{15} g_{16} g_{17}
g_{20} g_{21} g_{22} g_{23}
g_{30} g_{31}
g_{40}
$f_0(x_0x_1x_2x_3x_4x_5)$
x_2 x_3 x_0 x_5 x_1 x_4 x_0 x_1 x_1 x_5 x_0 x_1 x_0 x_4 x_2 x_3
g_{10} g_{11} g_{12} g_{13} g_{14} g_{15} g_{16} g_{17}
g_{20} g_{21} g_{22} g_{23}
g_{30} g_{31}
g_{40}
$f_1(x_0x_1x_2x_3x_4x_5)$

Figure 6: One of the possible low-latency structures for the representatives of the coordinate functions for 6-bit S-box in Example 4.

for each coordinate function, then we must try all $\prod_{i=0}^{m-1} N_{C_i}$ combinations, and check for possible replacements. Completing this approach will find a circuit for implementing the given function with minimum possible latency.

Note that all of these searches to find all possible circuits (that realize achieving the latency complexity) for implementing a (vectorial) Boolean function can be automated completely.

In the following, we consider the hybrid cubic-quadratic 6-bit S-box with latency complexity 4, linearity 16 and uniformity 4 which is presented in Subsection 5.2 as an example to show our approach for replacing the gates with good latency and higher fan-in number in the structure of Boolean functions boxes with low latency complexity.

Example 4 (6-bit S-box with latency complexity 4, linearity 16 and uniformity 4)**.** Consider $(y_0, y_1, y_2, y_3, y_4, y_5) = S(x_0, x_1, x_2, x_3, x_4, x_5)$ as this 6-bit bijective S-box with input bits x_i and output bits y_i with $0 \le i < 6$. The coordinate functions of this S-box is equivalent to only two Boolean functions, namely f_0 and f_1 presented as follows:

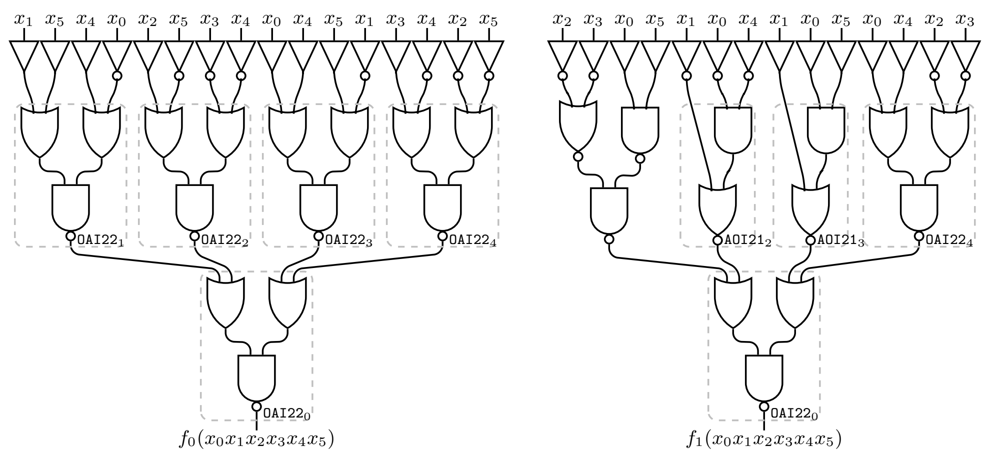
$$f_0 = (\mathsf{000000110101011010101010011111110011001111100110100110010100110000}),$$
$$f_1 = (\mathsf{0000000000001111001100110011110001010101010110101111111111110000}),$$

$$\begin{aligned} f_0 &= x_0x_4 \oplus x_1x_5 \oplus x_2x_5 \oplus x_3x_4 \oplus x_0 \oplus x_1 &&= x_0\overline{x_4} \oplus x_1\overline{x_5} \oplus x_2x_5 \oplus x_3x_4\,, \\ f_1 &= x_0x_1x_4 \oplus x_0x_1x_5 \oplus x_0x_1 \oplus x_0x_5 \oplus x_1x_4 \oplus x_2x_3 &&= \overline{x_0}x_1x_4 \oplus x_0\overline{x_1}x_5 \oplus x_0x_1 \oplus x_2x_3\,. \end{aligned}$$

Therefore, to find an optimized circuit for low latency implementation of this S-box requires finding latency-optimized circuits for each of these representative Boolean functions.

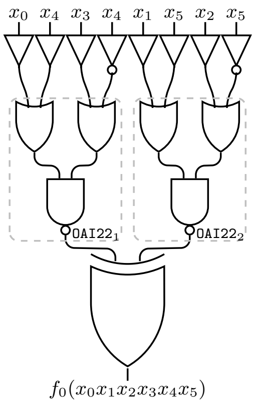
Running the algorithm in Subsection 4.4 to find low-latency structures of these functions returns 32 and 18 different simplified circuits for implementing f_0 and f_1, respectively. Note that this simplifications are only based on the basic and trivial logic equations and do not include the simplifications based on the larger (by fan-in or fan-out) and different gates. In Figure 6, we present one of these simplified circuits for each of f_0 and f_1.

About possibility of applying the larger or different gates and replacing them with their corresponding sub-circuits, considering the gates with fan-in number 3 or 4, there are 7 and 6 possible replacements for the structures of f_0 and f_1 shown in Figure 6, respectively. These replacements are the sub-circuits of $(g_{i,2j}, g_{i,2j+1}, g_{i+1,j})$ with $1 \le i \le 3$ and $0 \le j < 2^{4-i}$ excluding (g_{10}, g_{11}, g_{20}) for the structure of f_1.



x_1 x_5 x_4 x_0 x_2 x_5 x_3 x_4 x_0 x_4 x_5 x_1 x_3 x_4 x_2 x_5
$\mathtt{OAI22}_1$
$\mathtt{OAI22}_2$
$\mathtt{OAI22}_3$
$\mathtt{OAI22}_4$
$\mathtt{OAI22}_0$
$f_0(x_0x_1x_2x_3x_4x_5)$
x_2 x_3 x_0 x_5 x_1 x_0 x_4 x_1 x_0 x_5 x_0 x_4 x_2 x_3
$\mathtt{AOI21}_2$
$\mathtt{AOI21}_3$
$\mathtt{OAI22}_4$
$\mathtt{OAI22}_0$
$f_1(x_0x_1x_2x_3x_4x_5)$

Figure 7: One of the optimized circuits for the representatives of the coordinate functions for 6-bit S-box in Example 4.



x_0 x_4 x_3 x_4 x_1 x_5 x_2 x_5
$\mathtt{OAI22}_1$
$\mathtt{OAI22}_2$
$f_0(x_0x_1x_2x_3x_4x_5)$

Figure 8: Another optimized circuit for the representative function f_0 in Example 4.

Note that to find the lowest possible latency for each of these structures in an specific ASIC library, we need to consider all possible combinations for all replacements of previously mentioned low-latency gates with fan-in number of 3 or 4. In this example, we start with replacing (g_{30}, g_{31}, g_{40}) with an `OAI22` gate in both structures for f_0 and f_1. To do this, we need remove (g_{30}, g_{31}, g_{40}) and replace it with an `OAI22` gate (denoted by $\mathtt{OAI22}_0$) together with a single `INV` gate for each of the four inputs to this gate. However, we can omit these four `INV` gates by complementing all of g_{ij} gates with $0 \le i < 3$ and $0 \le j < 2^{4-i}$. Precisely, all g_{ij} gates with $i \in \{1, 2\}$ change to `NAND` gates, except g_{10} in the structure for f_1 which changes to a `NOR` gate, and all `BUF` gates in the depth level 0 change to `INV` gates and vice versa.

Again, we can replace each of $(g_{1,2j}, g_{1,2j+1}, g_{2,j})$ sub-circuits with $0 \le j < 4$ (excluding (g_{10}, g_{11}, g_{20}) for the structure of f_1) by an `OAI22` gate, or by an `AOI21` gate for $0 < j < 3$ in the structure of f_1. Note that all of these gates changed to a `NAND` gate after replacing $\mathtt{OAI22}_0$ gate. Similarly, we need to complement the g_{0j} gates with $0 \le j < 16$ (except for some of them in the circuit of f_1). This leads us to an optimized circuit shown in Figure 7.

One step forward, by considering `XOR` or `XNOR` gates, it is possible to do a further simplification on some of the corresponding suggested circuits for the representative function f_0. One of such simplified circuits is shown in Figure 8.

Table 4: Latency of the circuits for 6-bit S-box used in Example 4 using behavioral and structural modes of implementation in NanGate 15 nm and 45 nm OCLs.

	Structural	Behavioral	Ratio
45 nm	122.677	225.917	54.30%
15 nm	14.202	19.052	74.55%

We synthesized latency of this S-box in NanGate 15 nm and 45 nm OCL with typical operating conditions in two different types of behavioral and structural to show efficiency of our method for simplifying the low-latency structure. While in the structural mode, we used one of the simplified structures and simply synthesized the structure, in the behavioral mode, we used the look-up table representation of the S-box and let the synthesizer to optimize the circuit by using `compile_ultra -incremental` command for several times. We recall that the optimization in the behavioral mode is strongly related to the methods used in the synthesizer that in our case, it is a Synopsys Design Compiler.

The result of these syntheses are provided in Table 4 and as you can see the latency of the structure found by our method can have about 25% lower latency compared to the latency of the circuit found by the synthesizer itself in the behavioral mode.

It is noteworthy that we chose the structure used for the synthesis in the following way. First, in the target library, for both f_0 and f_1 Boolean functions, we tried all the possible structures and evaluate their latency. Then, for each function, we choose and used the structures with the lowest latency in the corresponding coordinate function of the S-box. Hence, this it is a local optimization and not generally optimized. Even for each chosen structure for f_0 and f_1, there are different possibilities for their input variables to realize the coordinate functions of the S-box. Precisely, there are 16 and 4 possible choices for the inputs of f_0 and f_1, respectively, to realize the corresponding coordinate. This means for the given structures for f_0 and f_1, there are 2^{20} different possibilities for the inputs of each coordinate function. We only considered the first possibility and used it in the structure given to the synthesizer.

Example 5 (7-bit quadratic S-boxes with latency complexity 4, linearity 32 and uniformity 8)**.** To provide stronger argument on the efficiency of our method for simplifying the low-latency structure, we implement all 134 7-bit quadratic S-boxes found in Subsection 5.2 in both behavioral and structural modes. Table 8 depicts the result of these syntheses. We recall that in the structural mode, we only try one of the simplified structures of the S-box which might be not the one with lowest latency. In some cases, specially in the case for 15 nm OCL, for the behavioral mode the synthesizer finds a circuit with a lower latency than the one for the circuit we used in the structural mode. In all of these cases, we checked the circuits found by synthesizer and realize that they are one of the possible simplifications of the low-latency structure for the corresponding S-box. That means if we check for all possible simplifications for the low-latency structure of the S-box, we will also meet the circuit found by the synthesizer.

We conclude this example with that trying only a *single* structure suggested by our method can improve the latency of a given S-box by 22% or 6% in average in the 45 nm and 15 nm OCLs, respectively.

7 Conclusion and Future Works

In this paper, we mathematically studied the latency of (vectorial) Boolean functions. We introduced the *latency complexity* metric to measure the latency of Boolean functions. We presented efficient algorithms for 1) finding all Boolean functions with low-latency complexity, 2) determining the latency complexity of the (vectorial) Boolean functions, and 3) finding all the circuits with the minimum latency complexity for a given Boolean function. Then, we presented another efficient algorithm to build bijective S-boxes with low-latency complexity while the previous method of building S-boxes was not suitable for our case.

As a result, for latency complexity 3, we found n-bit S-boxes with $3 \leq n \leq 8$ whose linearity are 2^{n-1} and uniformity are 2^{n-2} (except for 5-bit S-boxes that the minimum achievable uniformity is 6). Besides, we found several 5-, 6-, and 7-bit S-boxes with latency complexity 4 whose linearity are 2^{n-2} and uniformity are 2^{n-4}.

Our research has left several possible future works that we point out some of them:

- Since determining the gate depth complexity and accordingly latency complexity of a Boolean function is an NP-hard problem, presenting an efficient algorithm to find an upper-bound for the latency complexity is useful. Such an algorithm would present a low-latency implementation of the Boolean function (not necessarily with the minimum latency as of the latency complexity), which is interesting for a designer to reduce the latency of the implementation.

- Another future work is to complete our search for building 6-bit bijective with higher algebraic degree S-boxes with latency complexity 4 and repeat the search for 7- and 8-bit S-boxes. Then we can further investigate the relation between latency complexity and minimum achievable linearity and uniformity of the S-boxes within this latency complexity.

- Our algorithm for building bijective S-boxes is not only suitable for finding low-latency S-boxes. With simple modifications, this algorithm is applicable to build S-boxes with other properties and up to other equivalences. An interesting question in this area is to find all 6-bit bijective S-boxes with minimum linearity, 16, up to the affine equivalence.

Acknowledgments

I gratefully want to thank Thorben Moos from Universite catholique de Louvain for his valuable help in implementing and synthesizing the S-boxes and also for his constructive comments while preparing this work. I am also grateful to Christof Beierle and Gregor Leander from Ruhr-University Bochum for their valuable comments on an earlier version of this paper. Besides, this work described in this paper is supported by the Netherlands Organisation for Scientific Research (NWO) under TOP grant TOP1.18.002 SCALAR.

References

[Ava17] Roberto Avanzi. The QARMA block cipher family. almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. *IACR Trans. Symmetric Cryptol.*, 2017(1):4–44, 2017.

[BCBP03] Alex Biryukov, Christophe De Cannière, An Braeken, and Bart Preneel. A toolbox for cryptanalysis: Linear and affine equivalence algorithms. In Eli Biham, editor, *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, volume 2656 of *Lecture Notes in Computer Science*, pages 33–50. Springer, 2003.

[BCG$^+$12] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2012.

[BEK$^+$20] Dusan Bozilov, Maria Eichlseder, Miroslav Knezevic, Baptiste Lambin, Gregor Leander, Thorben Moos, Ventzislav Nikov, Shahram Rasoolzadeh, Yosuke Todo, and Friedrich Wiemer. Princev2 - more security for (almost) no overhead. In Orr Dunkelman, Michael J. Jacobson Jr., and Colin O'Flynn, editors, *Selected Areas in Cryptography - SAC 2020 - 27th International Conference, Halifax, NS, Canada (Virtual Event), October 21-23, 2020, Revised Selected Papers*, volume 12804 of *Lecture Notes in Computer Science*, pages 483–511. Springer, 2020.

[BFP19] Joan Boyar, Magnus Gausdal Find, and René Peralta. Small low-depth circuits for cryptographic applications. *Cryptogr. Commun.*, 11(1):109–127, 2019.

[BIL$^+$21] Subhadeep Banik, Takanori Isobe, Fukang Liu, Kazuhiko Minematsu, and Kosei Sakamoto. Orthros: A low-latency PRF. *IACR Trans. Symmetric Cryptol.*, 2021(1):37–77, 2021.

[BKL$^+$17] Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. Gimli : A cross-platform permutation. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 299–320. Springer, 2017.

[BMD$^+$20] Begül Bilgin, Lauren De Meyer, Sébastien Duval, Itamar Levi, and François-Xavier Standaert. Low AND depth and efficient inverses: a guide on s-boxes for low-latency masking. *IACR Trans. Symmetric Cryptol.*, 2020(1):144–184, 2020.

[BMP08] Joan Boyar, Philip Matthews, and René Peralta. On the shortest linear straight-line program for computing linear forms. In Edward Ochmanski and Jerzy Tyszkiewicz, editors, *Mathematical Foundations of Computer Science 2008, 33rd International Symposium, MFCS 2008, Torun, Poland, August 25-29, 2008, Proceedings*, volume 5162 of *Lecture Notes in Computer Science*, pages 168–179. Springer, 2008.

[Can07] Christophe De Cannière. Analysis and design of symmetric encryption algorithms. *Doctoral Dissertaion, KULeuven*, 2007.

[Car21] Claude Carlet. *Boolean Functions for Cryptography and Coding Theory.* Cambridge University Press, 2021.

[KNR12] Miroslav Knezevic, Ventzislav Nikov, and Peter Rombouts. Low-latency encryption - is "lightweight = light + wait"? In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012.*

Proceedings, volume 7428 of *Lecture Notes in Computer Science*, pages 426–446. Springer, 2012.

[LMMR21] Gregor Leander, Thorben Moos, Amir Moradi, and Shahram Rasoolzadeh. The speedy family of block ciphers: Engineering an ultra low-latency cipher from gate level for secure processor architectures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):510–545, Aug. 2021.

[LP07] Gregor Leander and Axel Poschmann. On the classification of 4 bit s-boxes. In Claude Carlet and Berk Sunar, editors, *Arithmetic of Finite Fields, First International Workshop, WAIFI 2007, Madrid, Spain, June 21-22, 2007, Proceedings*, volume 4547 of *Lecture Notes in Computer Science*, pages 159–176. Springer, 2007.

[LSL+19] Shun Li, Siwei Sun, Chaoyun Li, Zihao Wei, and Lei Hu. Constructing low-latency involutory MDS matrices with lightweight circuits. *IACR Trans. Symmetric Cryptol.*, 2019(1):84–117, 2019.

[MB19] Lauren De Meyer and Begül Bilgin. Classification of balanced quadratic functions. *IACR Trans. Symmetric Cryptol.*, 2019(2):169–192, 2019.

[Ras22] Shahram Rasoolzadeh. Low latency boolean functions and s-boxes. https://gitlab.science.ru.nl/shahramr/LowLatencySBoxes.git, 2022.

[Sto16] Ko Stoffelen. Optimizing s-box implementations for several criteria using SAT solvers. In Thomas Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*, pages 140–160. Springer, 2016.

A Appendix

Table 5: Number of the balanced Boolean functions in $\mathcal{F}_{n,d}$ with categorized by their ANF algebraic degree; i.e., number of full-dependent n-bit balanced Boolean functions with given degree and latency complexity up to the extended bit permutation equivalence.

n	d	algebraic degree						
		1	2	3	4	5	6	7
3	2	-	1					
	3	-	2					
	4	1	-					
4	3	-	-	6				
	4	1	10	34				
	5	-	-	1				
5	3	-	-	1	7			
	4	-	13	575	12 169			
	5	-	34	2953	70 791			
	6	1	-	-	1			
6	3	-	-	-	1	2		
	4	-	2	182	56 314	875 282		
7	4	-	-	20	19 600	758 263	3 654 887	
8	4	-	-	1	3 058	144 423	1 703 200	2 638 553

Table 6: Number of the balanced Boolean functions in $\mathcal{F}_{n,d}$ with linearity of ℓ, i.e., number of full-dependent n-bit balanced Boolean functions with linearity ℓ and latency complexity d up to the extended bit permutation equivalence.

n	d	ℓ															
		4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64
3	2	1															
	3	2															
	4		1														
4	3		4	2													
	4		34	10	1												
	5			1													
5	3				4	3	1										
	4		93	5928	5590	992	145	9									
	5		1274	46117	23771	2441	163	12									
	6							1	1								
6	3						1	-	1	-	1						
	4				1546	44505	291336	280133	200292	66971	34470	9039	2894	493	91	10	
7	4						1	7	7174	19874	255291	268255	1008535	428929	978496	241472	621429
8	4												1	-	64	95	10677

Table 7: Latency complexity of known S-boxes together with their uniformity, linearity and algebraic degree. In the columns for algebraic degree and latency complexity, in the case of bijective S-boxes, the second arrays are corresponding values for the inverse S-box.

n	S-box	uni	lin	algebraic degree	latency complexity
3	3-WAY / χ_3	2	4	(2,2,2),(2,2,2)	(3,3,3),(3,3,3)
3	CTC	2	4	(2,2,2),(2,2,2)	(3,3,3),(2,2,3)
3	PRINTcipher	2	4	(2,2,2),(2,2,2)	(3,3,3),(3,3,3)
3	Pyjamask-3	2	4	(2,2,2),(2,2,2)	(3,3,3),(3,3,3)
3	SEA	2	4	(2,2,2),(2,2,2)	(3,2,3),(3,3,2)
4	χ_4	4	8	(2,2,2,2)	(3,3,3,3)
4	CLEFIA-ss_0	4	8	(3,3,3,3),(3,2,3,3)	(4,4,4,3),(3,4,4,4)
4	CLEFIA-ss_1	4	8	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(3,3,4,4)
4	CLEFIA-ss_2	4	8	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
4	CLEFIA-ss_3	4	8	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,3,4)
4	CS-cipher	4	8	(2,2,3,3),(3,3,3,2)	(2,2,4,3),(4,4,3,4)
4	Crypton-p_1	4	8	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(3,4,4,4)
4	Elephant	4	8	(3,3,3,2),(2,3,3,3)	(4,4,4,4),(4,4,4,4)
4	Enocoro-4	4	8	(3,3,3,3),(3,3,3,3)	(4,4,3,4),(4,4,4,3)
4	Fox-s_1	4	8	(3,3,3,3),(3,3,3,3)	(3,4,4,4),(4,4,4,4)
4	Fox-s_2	4	8	(3,3,2,3),(3,3,2,3)	(4,4,4,4),(4,4,2,4)
4	Fox-s_3	4	8	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
4	GOST2-s_1	4	8	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,3)
4	GOST2-s_2	4	8	(3,3,3,3),(3,3,2,3)	(4,4,4,4),(4,4,4,4)
4	HUMMINGBIRD-s_1	4	8	(3,3,3,3),(3,3,3,2)	(4,4,4,4),(4,4,4,4)
4	HUMMINGBIRD-s_2	4	8	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
4	HUMMINGBIRD-s_3	4	8	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
4	HUMMINGBIRD-s_4	4	8	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
4	HUMMINGBIRD2-s_1	4	8	(3,3,3,3),(3,3,3,2)	(4,4,4,4),(4,4,4,4)
4	HUMMINGBIRD2-s_2	4	8	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
4	HUMMINGBIRD2-s_3	4	8	(2,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
4	HUMMINGBIRD2-s_4	4	8	(3,3,3,3),(2,3,3,3)	(4,4,4,4),(4,4,4,4)
4	ICEBERG-s_0, s_1	4	8	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
4	JH-s_0	4	8	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
4	JH-s_1	4	8	(3,3,3,3),(3,3,3,3)	(4,4,4,3),(3,4,4,4)
4	KHAZAD-p, q	4	8	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
4	KLEIN	4	8	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
4	KNOT	4	8	(2,2,3,3),(2,3,2,3)	(4,4,4,4),(4,4,4,4)
4	LBLOCK-s_0	4	8	(3,3,2,2),(2,2,3,3)	(4,3,4,4),(3,3,4,4)
4	LBLOCK-s_1	4	8	(3,3,2,2),(2,2,3,3)	(4,3,4,4),(3,3,4,4)
4	LBLOCK-s_2	4	8	(2,3,2,3),(2,2,3,3)	(4,4,4,3),(3,3,4,4)
4	LBLOCK-s_3	4	8	(3,2,3,2),(2,2,3,3)	(3,4,4,4),(3,3,4,4)
4	LBLOCK-s_4	4	8	(2,3,3,2),(2,2,3,3)	(4,4,3,4),(3,3,4,4)
4	LBLOCK-s_5	4	8	(3,2,3,2),(2,2,3,3)	(4,4,3,4),(3,3,4,4)
4	LBLOCK-s_6	4	8	(3,3,2,2),(2,2,3,3)	(4,3,4,4),(3,3,4,4)
4	LBLOCK-s_7	4	8	(3,3,2,2),(2,2,3,3)	(4,3,4,4),(3,3,4,4)
4	LBLOCK-s_8	4	8	(3,3,2,2),(2,2,3,3)	(3,4,4,4),(3,3,4,4)
4	LBLOCK-s_9	4	8	(2,2,3,3),(2,2,3,3)	(4,4,3,4),(3,3,4,4)

Table 7: Latency Complexity of Known S-boxes (Continued).

n	S-box	uni	lin	algebraic degree	latency complexity
4	LUFFA	4	8	(3,3,3,3),(3,3,3,2)	(4,4,4,4),(4,4,4,3)
	LUFFA-v1			(3,3,3,3),(3,3,2,2)	(4,4,4,4),(4,4,3,4)
	Magma-s_1			(2,3,3,3),(3,2,3,3)	(4,4,4,3),(4,4,4,4)
	Magma-s_2			(2,3,3,3),(3,3,3,3)	(3,4,4,4),(4,4,4,3)
	Magma-s_3			(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	Magma-s_4			(3,3,3,3),(3,3,3,3)	(4,4,3,4),(4,4,4,4)
	Magma-s_5			(2,3,3,3),(3,3,3,3)	(3,4,4,4),(4,4,4,4)
	Magma-s_6			(3,3,3,2),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	Magma-s_7			(2,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	Magma-s_8			(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,3)
	mCRYPTON-s_i / MIBS			(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	MIDORI-s_0 / MANTIS / CRAFT			(3,3,2,3),(3,3,2,3)	(3,3,3,3),(3,3,3,3)
	MIDORI-s_1			(3,3,3,3),(3,3,3,3)	(3,4,3,3),(3,4,3,3)
	Minalpher			(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	Noekeon			(2,2,3,3),(2,2,3,3)	(3,4,4,4),(3,4,4,4)
	Panda			(3,3,3,3),(3,3,3,3)	(4,3,4,4),(4,3,4,4)
	Piccolo / Joltik			(2,2,3,3),(3,2,2,3)	(3,3,4,4),(4,4,3,4)
	Present / CiliPadi / PHOTON-4 / Orange			(3,3,3,2),(3,3,3,2)	(4,4,4,4),(4,4,4,4)
	Pride / Prost			(2,2,3,3),(2,2,3,3)	(3,3,4,4),(3,3,4,4)
	Pyjamask-4			(2,3,2,3),(3,2,2,3)	(4,4,2,4),(3,4,4,4)
	QARMA-σ_0			(3,2,3,3),(3,2,3,3)	(3,3,3,4),(3,3,3,4)
	QARMA-σ_1 / Qameleon			(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	QARMA-σ_2			(3,3,3,3),(3,3,3,3)	(3,4,4,4),(4,3,4,3)
	Rectangle / REC^{-1}			(3,3,2,2),(3,2,2,3)	(4,4,4,4),(4,4,4,4)
	SATURNIN-s_0, s_1			(3,3,3,3),(3,3,3,3)	
	SC-2000-4				
	SERPENT-s_0			(2,3,3,3),(3,2,3,3)	
	SERPENT-s_1			(3,2,3,3),(2,3,3,3)	
	SERPENT-s_2			(3,3,3,2),(3,3,3,2)	
	SERPENT-s_3			(3,3,3,3),(3,3,3,3)	
	SERPENT-s_4			(3,3,3,2),(3,3,3,3)	
	SERPENT-s_5			(3,3,3,2),(3,3,3,3)	
	SERPENT-s_6			(3,3,2,3),(3,3,2,3)	
	SERPENT-s_7			(3,3,3,3),(3,3,3,3)	
	Skinny-4 / Remus-4			(2,2,3,3),(3,2,2,3)	(3,3,4,4),(4,4,3,4)
	SMASH256-s_1			(2,3,3,3),(3,3,2,3)	(4,4,4,4),(4,4,4,4)
	SMASH256-s_2			(3,3,2,3),(2,3,3,3)	
	SMASH256-s_3			(3,2,3,3),(3,3,3,3)	
	SPONGENT			(3,3,3,2),(2,3,3,3)	
	Spook / Clyde / Shadow			(3,3,2,2),(2,3,3,2)	(4,4,3,3),(3,4,4,4)
	TRIFLE			(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	TWINE				
	UDCIKMP			(3,2,3,2),(3,3,2,2)	(4,3,4,3),(4,4,3,3)

Table 7: Latency Complexity of Known S-boxes (Continued).

n	S-box	uni	lin	algebraic degree	latency complexity
4	WHIRLPOOL-E	4	8	(3,3,3,3),(3,3,3,3)	(3,4,4,4),(4,4,4,3)
	WHIRLPOOL-R				(4,4,4,4),(4,4,3,4)
	Yarara / Coral				(4,4,4,4),(4,4,4,4)
	Fountain-s_1	6	8	(3,3,2,2),(2,2,3,3)	(4,4,4,4),(4,4,4,4)
	Fountain-s_2			(3,3,3,3),(2,2,3,3)	(4,4,4,4),(4,2,4,4)
	Fountain-s_3			(3,3,3,3),(2,2,3,3)	(4,4,4,4),(4,2,4,4)
	Fountain-s_4			(3,3,3,2),(2,2,3,3)	(4,4,4,3),(4,4,4,4)
	GIFT / HYENA / TGIF			(3,3,2,2),(2,2,3,3)	(4,4,4,4),(4,4,4,4)
	Lucifer-s_0	6	12	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	Lucifer-s_1	6	8		(4,4,4,4),(4,4,3,4)
	BLAKE-s_1	8	12	(2,3,3,3),(3,3,3,3)	(4,4,4,4),(4,3,4,4)
	BLAKE-s_2	6	12	(3,3,3,2),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	BLAKE-s_3	6	12	(3,3,3,3),(3,3,3,3)	(3,4,4,3),(4,3,4,3)
	BLAKE-s_4	6	12	(3,3,3,3),(3,3,3,3)	(4,4,3,4),(4,4,3,4)
	BLAKE-s_5	6	12	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,3,4,4)
	BLAKE-s_6	8	12	(3,3,3,3),(3,3,3,2)	(4,4,4,4),(4,4,4,2)
	BLAKE-s_7	6	8	(3,3,2,3),(3,3,2,3)	(4,4,4,4),(4,4,4,4)
	BLAKE-s_8	6	12	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	BLAKE-s_9	6	12	(3,3,3,3),(3,3,3,3)	(4,4,3,4),(4,4,4,4)
	GOST-s_1	6	8	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,3)
	GOST-s_2	6	12	(3,3,3,3),(3,3,3,3)	(4,4,4,3),(3,4,4,4)
	GOST-s_3	6	12	(3,3,3,3),(3,3,3,3)	(4,4,3,4),(4,3,4,4)
	GOST-s_4	6	12	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,5,4)
	GOST-s_5	4	12	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	GOST-s_6	6	12	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	GOST-s_7	8	12	(3,3,3,3),(3,2,3,3)	(4,4,4,3),(3,4,4,4)
	GOST-s_8	8	12	(3,3,3,3),(3,3,3,3)	(4,4,4,3),(3,4,4,4)
	GOST-IETF-s_1	6	12	(3,3,3,3),(3,3,3,3)	(4,4,3,4),(3,4,3,3)
	GOST-IETF-s_2	8	12	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	GOST-IETF-s_3	6	12	(2,3,3,3),(3,3,3,3)	(4,4,3,4),(4,4,4,4)
	GOST-IETF-s_4	4	8	(3,2,3,3),(3,3,2,3)	(4,4,4,4),(4,4,4,4)
	GOST-IETF-s_5	8	12	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(3,4,4,4)
	GOST-IETF-s_6	6	8	(3,3,3,3),(2,3,3,3)	(4,4,4,4),(3,4,4,4)
	GOST-IETF-s_7	8	12	(3,3,3,3),(3,3,3,3)	(4,4,3,4),(4,4,4,4)
	GOST-IETF-s_8	6	12	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	Twofish-q_0-t_0	6	12	(3,3,3,3),(3,3,3,3)	(4,4,4,3),(4,4,4,4)
	Twofish-q_0-t_1	6	12	(3,3,3,3),(3,3,3,3)	(3,4,4,4),(4,3,4,3)
	Twofish-q_0-t_2	4	8	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	Twofish-q_0-t_3	6	12	(3,3,3,3),(3,2,3,3)	(4,4,4,4),(4,3,4,4)
	Twofish-q_1-t_0	6	12	(3,3,3,3),(3,3,3,3)	(4,3,3,4),(4,4,3,4)
	Twofish-q_1-t_1	6	12	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,3,4)
	Twofish-q_1-t_2	6	12	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	Twofish-q_1-t_3	8	12	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(3,4,4,4)
	DES-$s_{1,1}$	8	12	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(5,4,4,4)
	DES-$s_{1,2}$	8		(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	DES-$s_{1,3}$	8		(3,3,3,3),(3,3,3,3)	(4,4,4,4),(5,4,4,4)
	DES-$s_{1,4}$	8		(3,2,3,2),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	DES-$s_{2,1}$	6		(3,2,3,3),(3,3,3,2)	(4,4,5,4),(4,4,4,4)
	DES-$s_{2,2}$	8		(3,3,3,2),(3,3,2,2)	(5,4,4,4),(4,4,4,4)
	DES-$s_{2,3}$	8		(2,3,3,3),(2,3,3,3)	(4,4,4,4),(4,4,4,4)

Table 7: Latency Complexity of Known S-boxes (Continued).

n	S-box	uni	lin	algebraic degree	latency complexity
4	DES-$s_{2,4}$	6	12	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	DES-$s_{3,1}$	8		(3,3,3,2),(3,3,3,3)	(4,4,4,4),(5,4,4,4)
	DES-$s_{3,2}$	8		(3,3,3,2),(3,3,2,3)	(4,4,5,4),(4,4,4,4)
	DES-$s_{3,3}$	8		(3,2,3,3),(3,3,3,2)	(4,4,4,5),(4,4,4,4)
	DES-$s_{3,4}$	8		(2,3,3,3),(3,2,3,3)	(4,4,4,4),(4,4,4,5)
	DES-$s_{4,1}$	6		(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	DES-$s_{4,2}$	6		(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	DES-$s_{4,3}$	6		(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	DES-$s_{4,4}$	6		(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	DES-$s_{5,1}$	8		(3,2,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,5)
	DES-$s_{5,2}$	8		(3,3,3,3),(3,3,3,3)	(4,4,4,4),(5,4,4,4)
	DES-$s_{5,3}$	6		(3,3,3,3),(3,3,3,3)	(4,4,4,4),(5,4,4,4)
	DES-$s_{5,4}$	6		(3,3,3,3),(3,3,3,3)	(4,4,5,4),(5,4,4,4)
	DES-$s_{6,1}$	6		(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,5)
	DES-$s_{6,2}$	8		(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	DES-$s_{6,3}$	6		(3,3,3,3),(3,3,3,3)	(4,4,5,4),(4,4,4,4)
	DES-$s_{6,4}$	6		(3,3,3,2),(2,3,3,3)	(4,5,4,4),(4,4,4,4)
	DES-$s_{7,1}$	8		(3,2,3,3),(3,3,2,3)	(4,4,5,4),(4,4,4,4)
	DES-$s_{7,2}$	8		(2,3,3,3),(3,3,2,3)	(4,4,4,4),(4,4,4,5)
	DES-$s_{7,3}$	6		(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,4,4)
	DES-$s_{7,4}$	8		(3,3,3,3),(2,3,3,3)	(4,4,4,5),(4,4,4,4)
	DES-$s_{8,1}$	6		(3,3,2,3),(2,3,2,3)	(4,4,4,5),(4,4,4,4)
	DES-$s_{8,2}$	10		(3,3,3,3),(3,3,2,3)	(5,4,4,4),(4,4,4,4)
	DES-$s_{8,3}$	6		(3,3,3,3),(2,3,3,3)	(4,4,4,4),(4,4,4,4)
	DES-$s_{8,4}$	8		(3,2,3,3),(3,2,3,3)	(4,4,4,4),(5,4,4,4)
	Kuznyechik-ν_0	6	8	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,3,4,4)
	Kuznyechik-ν_1	16	12	(2,2,3,2),(3,2,2,3)	(4,4,4,4),(4,3,2,4)
	Kuznyechik-σ	6	12	(3,3,3,3),(3,3,3,3)	(4,4,4,4),(4,4,3,5)
	Kuznyechik-ϕ	8	12	(3,4,4,4),(4,3,4,4)	(4,4,4,4),(4,4,4,4)
5	Fides-5 / Primate	2	8	(2,2,2,2,2),(3,3,3,3,3)	(4,5,5,5,5),(5,5,5,5,5)
	Shamash				(5,5,5,5,5),(5,5,5,5,5)
	SC2000-5			(3,3,3,3,3),(2,2,2,2,2)	(5,5,5,5,5),(5,5,5,5,4)
	ASCON / ISAP	8	16	(2,2,2,2,2),(3,3,3,3,3)	(5,5,4,5,4),(5,5,5,4,5)
	Dry-GASCON128				(4,5,4,5,5),(5,4,5,5,5)
	KECCAK / χ_5				(3,3,3,3,3),(5,5,5,5,5)
	ICEPOLE			(4,4,4,4,4),(4,4,4,4,4)	(4,4,4,4,4),(5,5,5,5,5)
	SYCON				(4,4,5,5,5),(5,5,5,5,4)
6	APN-6	2	16	(4,3,4,3,4,4),(4,4,4,3,3,3)	(x,. . .,x),(x,. . .,x)
	Fides-6			(4,3,4,3,4,4),(4,4,4,4,3,4)	(x,. . .,x),(x,. . .,x)
	SC2000-6	4	16	(5,5,5,5,5,5),(5,5,5,5,5,5)	(x,. . .,x),(x,. . .,x)
	Speedy	8	24	(5,3,3,3,4,5),(5,4,5,4,5,5)	(4,4,4,4,4,4),(x,. . .,x)
	χ_6	16	32	(2,2,2,2,2,2)	(3,3,3,3,3,3)
7	Wage	8	40	(6,6,5,6,3,5,5),(6,5,5,6,5,6,6)	(x,. . .,x),(x,. . .,x)
	χ_7	32	64	(2,2,2,2,2,2,2),(4,4,4,4,4,4,4)	(3,3,3,3,3,3,3),(x,. . .,x)

Table 7: Latency Complexity of Known S-boxes (Continued).

n	S-box	uni	lin	algebraic degree	latency complexity
	SKINNY-8	64	128	(4,2,2,4,2,2,5,6),(2,5,2,6,2,2,3,4)	(x,3,3,x,4,3,x,x),(3,x,3,x,3,3,5,x)
	χ_8			(2,2,2,2,2,2,2,2)	(3,3,3,3,3,3,3,3)
	CSS	128	256	(4,4,2,2,4,4,2,2),(4,4,2,2,4,4,2,2)	(x,x,3,3,x,x,3,3),(x,x,3,3,x,x,3,3)
8	The latency complexity of all coordinate functions for AES, Anubis, ARIA, BelT, Camellia, Chiasmus, CLEFIA-$s_0, s1$, CMEA, Crypton0, Crypton1-s_i, CS-cipher, CSA, DBlock, E2, Enocoro-8, Fantomas, FLY, Fox, ICEBERG-8, Iraqi, Kalyna-π_i, KHAZAD-8, Kuznechik, Lilliput, MD2, NewDES, Picaro, SAFER, SEED-s_0, s_1, iSCREAM, SCREAMv1, SCREAMv3, Skipjack, SMS4, SNOW-3G, SQUARE, Twofish-p_0, p_1, TURING, WHIRLPOOL-8, ZORRO and ZUC-S-boxes are higher than 4.				

Table 8: Latency of the circuits for 7-bit quadratic S-boxes used in Example 5 using behavioral and structural modes of implementation in NanGate 15 nm and 45 nm OCLs.

S-box's Index	**Behavioral** 45 nm	15 nm	**Structural** 45 nm	15 nm	**Ratio** [%] 45 nm	15 nm
0	129.165	16.941	155.302	17.867	83.17%	94.82%
1	138.407	16.441	185.556	20.378	74.59%	80.68%
2	119.143	17.371	171.567	18.073	69.44%	96.12%
3	126.669	17.913	185.967	18.608	68.11%	96.26%
4	124.155	17.301	196.030	17.301	63.33%	100.00%
5	125.303	17.400	166.729	17.960	75.15%	96.89%
6	125.799	16.856	163.701	16.856	76.85%	100.00%
7	138.945	13.297	158.990	13.297	87.39%	100.00%
8	154.371	17.588	186.268	19.676	82.88%	89.39%
9	125.207	14.835	147.354	14.835	84.97%	100.00%
10	139.058	14.420	160.156	14.420	86.83%	100.00%
11	122.177	17.152	158.132	17.152	77.26%	100.00%
12	126.803	14.701	155.356	14.701	81.62%	100.00%
13	139.069	16.993	189.749	17.682	73.29%	96.10%
14	127.965	14.874	242.606	19.685	52.75%	75.56%
15	125.440	13.791	161.635	13.791	77.61%	100.00%
16	152.485	17.295	152.485	17.295	100.00%	100.00%
17	125.165	17.023	156.901	17.023	79.77%	100.00%
18	125.281	13.877	166.276	13.877	75.35%	100.00%
19	125.318	16.536	158.929	17.610	78.85%	93.90%
20	125.355	14.511	152.700	14.511	82.09%	100.00%
21	125.213	15.303	146.950	15.303	85.21%	100.00%
22	154.005	17.069	156.445	17.069	98.44%	100.00%
23	124.239	14.778	152.080	14.778	81.69%	100.00%
24	125.337	14.791	163.696	14.791	76.57%	100.00%
25	124.148	16.369	150.133	16.982	82.69%	96.40%
26	118.811	17.873	155.274	17.873	76.52%	100.00%
27	124.049	16.854	194.494	17.661	63.78%	95.43%
28	125.647	17.214	166.276	17.775	75.57%	96.84%
29	121.923	17.430	179.470	18.440	67.94%	94.52%
30	148.791	16.829	180.555	17.371	82.41%	96.88%
31	128.245	17.477	152.662	17.477	84.01%	100.00%
32	125.278	17.098	148.017	17.098	84.64%	100.00%
33	126.758	17.197	221.475	23.230	57.23%	74.03%
34	125.261	16.741	153.050	17.664	81.84%	94.78%
35	142.762	17.224	152.102	17.810	93.86%	96.71%

Table 8: Latency of the circuits for 7-bit quadratic S-boxes used in Example 5 using behavioral and structural modes of implementation in NanGate 15 nm and 45 nm OCLs.

S-box's	**Behavioral**		**Structural**		**Ratio** [%]	
Index	45 nm	15 nm	45 nm	15 nm	45 nm	15 nm
36	125.484	16.496	170.018	19.106	73.81%	86.34%
37	129.590	17.169	179.368	17.536	72.25%	97.91%
38	129.621	17.430	165.777	17.430	78.19%	100.00%
39	136.851	16.766	160.212	17.212	85.42%	97.41%
40	138.555	15.248	167.125	20.599	82.91%	74.02%
41	138.508	16.513	148.907	17.130	93.02%	96.40%
42	123.401	16.713	188.826	21.674	65.35%	77.11%
43	122.934	16.675	158.902	17.109	77.36%	97.47%
44	124.235	17.136	183.760	17.136	67.61%	100.00%
45	118.856	17.824	155.621	17.824	76.38%	100.00%
46	125.128	13.845	160.424	13.845	78.00%	100.00%
47	149.929	17.250	155.617	17.250	96.34%	100.00%
48	131.261	14.110	149.421	14.110	87.85%	100.00%
49	131.200	13.530	154.018	13.530	85.18%	100.00%
50	129.777	17.177	153.938	19.272	84.30%	89.13%
51	130.908	14.844	174.856	14.962	74.87%	99.21%
52	130.774	16.999	197.211	17.910	66.31%	94.91%
53	125.245	17.794	149.951	18.446	83.52%	96.47%
54	150.089	14.660	150.089	14.660	100.00%	100.00%
55	160.323	17.552	173.747	18.211	92.27%	96.38%
56	134.071	16.870	156.903	18.063	85.45%	93.39%
57	121.557	18.210	170.474	18.442	71.31%	98.74%
58	139.527	13.829	146.471	13.829	95.26%	100.00%
59	128.070	16.870	167.073	17.789	76.65%	94.84%
60	137.717	16.325	174.132	17.518	79.09%	93.19%
61	129.509	16.905	171.160	17.569	75.67%	96.22%
62	129.524	15.111	149.915	17.885	86.40%	84.49%
63	124.022	16.716	157.395	18.861	78.80%	88.62%
64	124.342	16.820	187.842	19.586	66.19%	85.88%
65	123.714	14.701	179.887	17.683	68.77%	83.14%
66	124.998	15.936	152.578	17.710	81.92%	89.98%
67	124.132	16.668	189.164	18.577	65.62%	89.73%
68	124.237	17.615	241.162	21.399	51.52%	82.31%
69	182.229	15.147	200.098	19.969	91.07%	75.85%
70	129.485	14.869	159.079	18.274	81.40%	81.36%
71	128.543	17.060	164.101	17.060	78.33%	100.00%
72	133.167	16.648	160.375	17.738	83.03%	93.86%
73	130.494	17.563	148.900	18.053	87.64%	97.29%
74	154.039	16.918	158.345	16.918	97.28%	100.00%
75	171.496	15.333	185.126	17.805	92.64%	86.12%
76	122.100	15.575	161.860	18.659	75.44%	83.47%
77	150.250	15.847	158.544	16.132	94.77%	98.23%
78	122.495	15.038	185.036	18.141	66.20%	82.90%
79	154.354	15.963	170.426	15.963	90.57%	100.00%
80	122.713	17.104	153.997	17.104	79.69%	100.00%
81	122.800	18.643	211.289	20.904	58.12%	89.19%
82	125.253	14.687	186.895	14.687	67.02%	100.00%
83	128.550	17.073	200.854	21.766	64.00%	78.44%
84	130.381	15.395	179.444	18.841	72.66%	81.71%
85	125.198	16.363	155.266	16.363	80.63%	100.00%
86	123.618	16.647	166.857	16.961	74.09%	98.15%

Table 8: Latency of the circuits for 7-bit quadratic S-boxes used in Example 5 using behavioral and structural modes of implementation in NanGate 15 nm and 45 nm OCLs.

S-box's	**Behavioral**		**Structural**		**Ratio** [%]	
Index	45 nm	15 nm	45 nm	15 nm	45 nm	15 nm
87	166.197	16.971	196.960	18.010	84.38%	94.23%
88	142.484	16.916	163.248	18.188	87.28%	93.01%
89	124.247	15.390	213.786	18.263	58.12%	84.27%
90	133.458	14.097	172.738	17.774	77.26%	79.31%
91	124.135	16.941	167.264	16.985	74.22%	99.74%
92	121.606	16.064	188.440	16.867	64.53%	95.24%
93	122.818	15.600	169.862	19.921	72.30%	78.31%
94	120.927	16.456	163.972	19.343	73.75%	85.08%
95	125.073	15.386	183.935	15.386	68.00%	100.00%
96	123.474	16.746	148.494	18.022	83.15%	92.92%
97	125.310	17.922	171.919	17.922	72.89%	100.00%
98	132.407	17.625	173.632	18.121	76.26%	97.26%
99	146.222	15.504	209.732	15.997	69.72%	96.92%
100	132.326	17.110	153.561	18.648	86.17%	91.75%
101	133.038	14.725	161.401	18.565	82.43%	79.32%
102	132.740	15.950	160.581	16.373	82.66%	97.42%
103	137.655	16.332	172.679	17.939	79.72%	91.04%
104	123.959	15.281	163.758	15.281	75.70%	100.00%
105	146.842	17.519	154.856	17.565	94.82%	99.74%
106	120.251	17.570	232.800	17.570	51.65%	100.00%
107	122.997	17.689	181.379	18.447	67.81%	95.89%
108	126.929	17.329	211.439	17.329	60.03%	100.00%
109	157.838	15.750	170.634	18.738	92.50%	84.06%
110	128.409	16.010	148.747	16.234	86.33%	98.61%
111	126.437	17.746	186.929	18.581	67.64%	95.51%
112	149.297	16.796	161.742	20.035	92.31%	83.83%
113	122.346	18.000	206.107	20.060	59.36%	89.73%
114	124.172	16.869	181.923	16.869	68.26%	100.00%
115	123.478	17.517	170.058	18.915	72.61%	92.61%
116	125.348	16.922	186.242	16.922	67.30%	100.00%
117	124.138	16.303	179.561	17.639	69.13%	92.43%
118	125.559	16.251	183.799	19.482	68.31%	83.41%
119	120.788	14.196	159.386	14.196	75.78%	100.00%
120	118.445	17.450	180.114	18.577	65.76%	93.93%
121	122.318	16.822	202.930	20.620	60.28%	81.58%
122	154.161	16.261	159.797	18.031	96.47%	90.18%
123	123.622	16.054	169.045	18.423	73.13%	87.14%
124	122.712	18.319	165.366	18.745	74.21%	97.73%
125	123.985	17.708	166.640	18.920	74.40%	93.59%
126	122.664	17.648	158.553	17.916	77.36%	98.50%
127	124.454	15.789	158.640	18.615	78.45%	84.82%
128	165.664	16.936	165.664	19.015	100.00%	89.06%
129	124.077	16.408	157.027	20.085	79.02%	81.69%
130	130.412	14.085	185.064	14.985	70.47%	94.00%
131	120.896	17.353	153.509	17.353	78.76%	100.00%
132	156.675	17.434	191.844	17.935	81.67%	97.21%
133	159.382	15.567	170.861	18.402	93.28%	84.59%