A Heuristic Proof of P ≠ NP

Ping Wang

Shenzhen University
wangping@szu.edu.cn
January 15, 2025

Abstract. The question of whether the complexity class P equals NP is a major unsolved problem in theoretical computer science. In this paper, we introduce a new language, the Add/XNOR problem, which has the simplest structure and perfect randomness, by extending the subset sum problem. We prove that P ≠ NP as it shows that the square-root complexity is necessary to solve the Add/XNOR problem. That is, problems that are verifiable in polynomial time are not necessarily solvable in polynomial time.
Furthermore, by giving up commutative and associative properties, we design a magma equipped with a permutation and successfully achieve Conjecture 1. Based on this conjecture, we obtain the Add/XOR/XNOR problem and one-way functions that are believed to require exhaustive search to solve or invert.

Keywords: P, NP, subset sum problem, Add/XNOR problem, complexity theory, polynomial time, exponential time

1 Introduction

P and NP are two central complexity classes in computational complexity theory [4]. P contains decision problems that can be solved in polynomial time, while NP contains problems where solutions can be verified in polynomial time. One of the most fundamental open questions [5,6] in computer science and mathematics is whether P = NP, that is, whether every problem that can be verified in polynomial time can also be solved in polynomial time. Resolving this question either way would have profound implications. Most computer scientists believe that P ≠ NP. A key reason for this belief is that, after decades of research on these problems, no one has been able to find a polynomial time algorithm for any of the more than 3000 important known NP-complete problems. Furthermore, we have the following definitions.

Definition 1 (PTIME (P)). *A language $L \in P$ if and only if there exists a poly($|x|$) time deterministic algorithm f, such that:*

- $\forall x \in L,\ f(x) = 1.$
- $\forall x \notin L,\ f(x) = 0.$

By $|x|$, we mean the number of bits in the binary string x. That is, P contains all decision problems that can be solved by a deterministic Turing machine using polynomial time.

Definition 2 (Nondeterministic Polynomial Time (NP)). *A language $L \in$ NP if and only if there exists a deterministic poly($|x|$) time verifier V, such that:*

- $\forall x \in L,\ \exists y,\ |y| = poly(|x|),\ V(x, y) = 1.$
- $\forall x \notin L,\ \forall y,\ |y| = poly(|x|),\ V(x, y) = 0.$

NP is the set of decision problems for which the problem instances, where the answer is "yes", have proofs verifiable in polynomial time by a deterministic Turing machine.

Here, we introduce a new language, the Add/XNOR problem, where XNOR ($\odot$) is the negation of XOR ($\oplus$).

Definition 3 (The Add/XNOR Problem (Decision Problem)). *Let m be an integer constant (e.g., $m = 1024$). Given a sequence of n integers $A_1, A_2, \ldots, A_n$, each chosen independently and uniformly at random from $\{0,1\}^m$ (i.e., each is an m-bit number), determine whether there exists a sequence of operators $O_1, O_2, \ldots, O_{n-1}$, where each $O_i \in \{+, \odot\}$ (addition modulo 2^m or bitwise XNOR), such that the sequential left-associative expression:*

$$E = (((A_1\, O_1\, A_2)\, O_2\, A_3)\ \ldots)\, O_{n-1}\, A_n \equiv \mathbf{0}\ (i.e.,\ m\ zeros). \tag{1}$$

Without loss of generality, in this paper, we will assume $m := n$ for the Add/XNOR problem for simplicity. Here, the expression E is evaluated sequentially from left to right (i.e., left-associative evaluation), and the problem asks whether there is a combination of operators $O_i \in \{+, \odot\}$ that satisfies the equation. For $i = 1, 2, ..., n-1$, let

$$x_i = \begin{cases} 0, & \text{if } O_i = +; \\ 1, & \text{if } O_i = \odot. \end{cases}$$

The problem is to determine whether there is an $(n-1)$-bit string $x = x_1 x_2 \ldots x_{n-1}$ such that equation (1) holds.

The computational Add/XNOR problem is to recover a solution x if at least one exists. It is clear that given access to an oracle that solves the decision problem, the computational problem can be solved by using $n-1$ calls to this oracle. If a solution exists, we can determine x_1 by calling oracle once. For example, we can call oracle to determine whether the sequence $A_1 + A_2, A_3, \ldots, A_{n-1}$ has a solution. If there is a solution, then $x_1 = 0$; otherwise $x_1 = 1$. After x_1 is fixed, we can determine x_2 by calling oracle once again, and so on.

In fact, the Add/XNOR problem can be viewed as a generalized version of the subset sum problem. Given a set of integers $\{A_1, A_2, \ldots, A_n\}$ and a target T, the subset sum problem is to decide whether any subset of these integers

sums to T. The problem is known to be NP-complete. The subset sum problem can be rephrased as determining whether there exist operators $O_1, O_2, \ldots, O_n$, where each O_i is either "multiplied by 0 then add" or "multiplied by 1 then add", denoted as $O_i \in \{+_0, +_1\}$, such that the following equation holds:

$$E = (O_1 A_1)(O_2 A_2) \ldots (O_n A_n) \equiv T. \tag{2}$$

In other words, the problem is to determine whether there is a not all-zero n-bit string $x = x_1 x_2 \ldots x_n$, such that: $\sum_{i=1}^{n} x_i A_i = T$.

Actually, only addition is used in the subset sum problem, which introduces specific mathematical structures to the problem, making it possible to design more efficient algorithms for solving it by exploiting the structure. For example, modulo operations (mod) can be utilized effectively to reduce the computational complexity of subset sum problems, allowing the complexity to break the square-root complexity bound for the random collision problem [9,1,2].

Therefore, we need to consider more random operations to minimize the mathematical structure or properties of the problem. As shown in the truth table of Table 1, adding, XOR, and XNOR preserve randomness among all two single-bit binary operations (see Theorem 3 for reference). Correspondingly, for n-bit operations, addition modulo 2^n, bitwise XOR, and bitwise XNOR preserve randomness, since addition modulo 2^n is equivalent to bitwise single-bit adding then modulo 2^n. Addition modulo 2^n forms the ring $\mathbb{Z}_{2^n}$, which introduces non-linearity, in contrast to XOR and XNOR.

Table 1. Truth table of $+$, $\oplus$ and $\odot$.

a	b	$a+b$		$a \oplus b$	$a \odot b$
		carry	sum		
0	0		0	0	1
0	1		1	1	0
1	0		1	1	0
1	1	1	0	0	1

– If we replace the addition in the subset sum problem with bitwise XOR, we get the subset XOR problem, where each O_i is either "multiplied by 0 then bitwise XOR" or "multiplied by 1 then bitwise XOR". The subset XOR problem can be formulated as a linear algebra problem over $\mathbb{F}_2$. Let $x_i \in \{0, 1\}$ indicate whether A_i is included in the subset. For each bit position j (from 1 to n), we have:

$$\sum_{i=1}^{n} x_i \cdot A_i^{(j)} \equiv T^{(j)} \mod 2,$$

where $A_i^{(j)}$ is the j-th bit of A_i and $T^{(j)}$ is the j-th bit of T. We have n linear equations over $\mathbb{F}_2$ with n variables x_i. We are looking for a non-trivial

solution (not all $x_i = 0$). Solving such a system can be done in $O(n^3)$ time. The subset XOR problem can be solved in polynomial time. The properties of addition modulo 2 (XOR operation) and the structure of $\mathbb{F}_2$ allow for efficient solutions that are not possible with integer addition (for the subset sum problem). The situation is the same if we replace the addition in the subset sum problem with bitwise XNOR.

– If we set $O_i \in \{+_0, +_1\}$ in the subset sum problem to $O_i \in \{\oplus, \odot\}$ (bitwise XOR or bitwise XNOR), we get the following generalized problem: Determine whether there exist operators $O_1, O_2, \ldots, O_{n-1}$ with $O_i \in \{\oplus, \odot\}$, such that the following equation holds:

$$E = A_1 O_1 A_2 \ldots O_{n-1} A_n \equiv \mathbf{0}. \tag{3}$$

The problem can also be expressed as a linear algebra problem over $\mathbb{F}_2$ since XNOR is the negation of XOR. Let $x_i \in \{0, 1\}$ indicate whether O_i is $\oplus$ ($x_i = 0$) or $\odot$ ($x_i = 1$). Since $a \odot b = \overline{a \oplus b} = a \oplus b \oplus 1$, for each bit position j (from 1 to n), we have:

$$\sum_{i=0}^{n-1} (x_i + A_{i+1}^{(j)}) \equiv 0 \mod 2,$$

where $x_0 = 0$, and $A_i^{(j)}$ is the j-th bit of A_i. We have n linear equations over $\mathbb{F}_2$ with $n-1$ variables x_i. Such a system can be solved in polynomial time. Because XOR and XNOR are complementary operations, the combination provides perfect randomness in single-bit binary operations, but the combination has the vulnerability that the problem can be solved efficiently by a system of linear equations.

– If we set $O_i \in \{+_0, +_1\}$ in the subset sum problem to $O_i \in \{+, \oplus\}$ (addition modulo 2^n or bitwise addition modulo 2 (bitwise XOR)), we need consider the sequential left-associative expression such as equation (1), since addition modulo 2^n and bitwise XOR do not satisfy "associative" law. The combination of addition modulo 2^n and bitwise XOR makes it impossible to solve the problem using a system of linear equations due to the existence of random carries in addition. However, since addition can be regarded as XOR without carry, this combination lacks randomness in single-bit operations, making it possible to design more efficient algorithms based on this property. For example, the lowest bit of the result calculated by the expression E in equation (1) is completely determined by the lowest bits of $A_1, A_2, \ldots, A_n$, regardless of the combination of O_i. That is, if the number of 1s in the lowest bits of $A_1, A_2, \ldots, A_n$ is even, then the lowest bit of the result is 0; otherwise, it is 1. Similarly, for the bits in other positions, we can determine whether an odd or even number of carries is needed in the lower position based on the parity of the number of 1s in the current position so that the result is 0.

– If we set $O_i \in \{+_0, +_1\}$ in the subset sum problem to $O_i \in \{+, \odot\}$ (addition modulo 2^n or bitwise XNOR), then we get the Add/XNOR problem as defined in Definition 3. The Add/XNOR problem can not be expressed as a

system of linear equations over the finite field $\mathbb{F}_2$ because addition modulo 2^n brings non-linearity, in contrast to the case where $O_i \in \{\oplus, \odot\}$. On the other hand, addition modulo 2^n without carry is equivalent to XOR, which is the negation of XNOR. Thus, the combination of addition modulo 2^n and XNOR provides perfect randomness in single-bit binary operations, in contrast to the case where $O_i \in \{+, \oplus\}$.

The key to proving that P $\neq$ NP is to show that there is no efficient (polynomial time) algorithm for a language in NP. For a language in NP, it is usually impossible to prove that it is not in P, because we can only claim that no better algorithm has been found so far, and there is no way to guarantee (or prove) that a more efficient algorithm does not exist. In this sense, it seems impossible to prove that P $\neq$ NP. However, in all attempts to prove P $\neq$ NP, we can still distinguish whether it is a better proof or not. For example, a problem that claims to be solvable only by the square root algorithm is better for proving P $\neq$ NP if the structure of the problem itself is simpler than a problem with the same computational complexity. On the other hand, a problem that claims to be solvable only by exhaustive search has more potential and is more convincing to prove P $\neq$ NP than a problem that claims that the best algorithm is the square root algorithm.

Problems in general have some mathematical structure, we cannot guarantee (or prove) that a more efficient (i.e., polynomial time) algorithm that makes effective use of such mathematical structure does not exist. The Add/XNOR problem we present, on the other hand, has the simplest structure among the currently known NP-complete problems. Furthermore, the nature of addition modulo 2^n and XNOR operations provides perfect randomness, which implies that the Add/XNOR problem has essentially no effective mathematical structure, making it possible for us to prove P $\neq$ NP.

Here, we outline our contributions as follows:

- Take the elliptic curve discrete logarithm problem (ECDLP) as an example. For a long time and so far, the best algorithm for solving it has been the square-root algorithm (Pollard rho algorithm [10,11,3]). However, we cannot claim that the square-root algorithm is the optimal algorithm for ECDLP, because it is almost impossible to prove that a more efficient algorithm does not exist based on the rich structure of elliptic curves over finite fields. For this reason, to make it possible for us to prove that P $\neq$ NP, we propose to find the computationally difficult problem with the simplest mathematical structure, and create new ones if necessary. The Add/XNOR problem we propose is a computationally difficult problem with minimal structure among the known NP-complete problems. Due to the simple structure or the lack of effective structure, the methods for solving these problems are extremely limited.
- We claim that the square-root algorithm is optimal for the Add/XNOR problem because it essentially has no valid mathematical structure, making it surpass the subset sum problem, which has algorithms that break the square-root complexity bound.

– By combining Boolean algebra and integer arithmetic, we define two new elegant operations $+_{\oplus\odot}$ and $+_{\odot\oplus}$, and get two commutative quasigroups. Based on these two commutative quasigroups, by giving up the commutative property, we design and obtain an algebraic structure: a magma (or groupoid) equipped with a permutation. For the first time, we combine the concepts of quasigroup, magma, and left-associative evaluation to design irreversible functions and one-way functions, and give a new direction for designing irreversible transformations and one-way functions. We achieve a pretty result: Conjecture 1, which is the first problem with exponential complexity that is claimed to require an exhaustive search to solve.
– We claim that an exhaustive search is required to solve the Add/XOR/XNOR problem under Conjecture 1, because the problem also has no effective structure and is resistant to meet-in-the-middle (MITM) attacks. This is also a computationally difficult problem with exponential complexity, which can only be solved by exhaustive search.
– Based on Conjecture 1 and the Add/XOR/XNOR problem, we designed two one-way functions that can resist the meet-in-the-middle attack. Under Conjecture 1, the computational complexity of inverting both one-way functions is $O(2^n)$.

For the language L (Add/XNOR problem) defined by Definition 3, we will show that $L \in$ NP and $L \notin$ P. Therefore, P $\neq$ NP. It is trivial to prove that $L \in$ NP. For any Yes-instance of L, given the corresponding proof, i.e., $x = x_1x_2 \dots x_{n-1}$, we can verify the instance in polynomial time using a deterministic Turing machine by checking if the equation (1) holds. Hence, $L \in$ NP.

Without loss of generality, in this paper, we will assume that $m := n$ for the Add/XNOR problem, and suppose that no x or only one x exists such that the equation (1) holds for simplicity. In section 2, we will prove that the Add/XNOR problem is NP-complete. In section 3, we will show that $L \notin$ P. In section 4, we will introduce a new language, the Add/XOR/XNOR problem, which requires an exhaustive search to solve. In section 5, we will present the construction of one-way functions from the Add/XOR/XNOR problem, and give an open problem as a challenge.

2 The Add/XNOR problem is NP-complete

Theorem 1. *The Add/XNOR problem is NP-complete.*

Proof:

The Problem is in NP: Given a certificate (a sequence of operators), we can verify in polynomial time whether the expression evaluates to zero.

Given a sequence of operators $O_1, O_2, \dots, O_{n-1} \in \{+, \odot\}$, we can compute the expression E as:

$$E = (((A_1\, O_1\, A_2)\, O_2\, A_3)\ \dots)\, O_{n-1}\, A_n.$$

Each operation involves two n-bit numbers. Both addition modulo 2^n and bitwise XNOR can be computed in $O(n)$ time. There are $n-1$ operations to perform. Hence, the total time complexity is $O(n^2)$. Verification can be done in polynomial time. Therefore, the Add/XNOR problem is in NP.

The Problem is NP-Hard: We will reduce the subset sum problem, which is known to be NP-complete, to this problem in polynomial time.

The definition of subset sum problem: Given a set of integers $S = \{s_1, s_2, \ldots, s_m\}$ (each representable by n-bit numbers) and a target integer T (also an n-bit integer), determine if there is a subset $S' \subseteq S$ such that:

$$\sum_{s_i \in S'} s_i = T.$$

Without loss of generality, we assume that all integers and the target are n-bit integers and that addition is taken modulo 2^n.

We will construct an instance of the Add/XNOR problem from a given subset sum instance such that: There exists a subset S' summing to T if and only if there exists a sequence of operators O_i such that $E = \mathbf{0}$. We have the following construction steps.

1. Initialize E:

We start by setting

$$E_0 = A_0 := 2^n - 2T \bmod 2^n.$$

This will represent our initial value of E.

2. Gadget for each s_i:

For each element s_i of the subset sum instance, we introduce four additional integers:

$$A_{4i-3} = s_i, \quad A_{4i-2} = 2^{n-1}, \quad A_{4i-1} = s_i, \quad A_{4i} = 2^{n-1},$$

where i runs from 1 to m. To clarify the indexing explicitly, for $i = 1$:

$$A_1 = s_1, \quad A_2 = 2^{n-1}, \quad A_3 = s_1, \quad A_4 = 2^{n-1}.$$

For $i = 2$:

$$A_5 = s_2, \quad A_6 = 2^{n-1}, \quad A_7 = s_2, \quad A_8 = 2^{n-1},$$

and so forth, appending four integers per element.

These four integers form a "gadget" that can be operated in two different ways:

– Include s_i (add $2 \cdot s_i$ to E_{i-1}), apply:

$$\begin{aligned} E_i &= (((E_{i-1} + A_{4i-3}) \odot A_{4i-2}) + A_{4i-1}) \odot A_{4i} \\ &= (((E_{i-1} + s_i) \odot 2^{n-1}) + s_i) \odot 2^{n-1} \\ &= E_{i-1} + 2s_i. \end{aligned}$$

– Exclude s_i (do not change E_{i-1}), apply:

$$\begin{aligned} E_i &= (((E_{i-1} \odot A_{4i-3}) \odot A_{4i-2}) \odot A_{4i-1}) \odot A_{4i} \\ &= (((E_{i-1} \odot s_i) \odot 2^{n-1}) \odot s_i) \odot 2^{n-1} \\ &= E_{i-1}. \end{aligned}$$

3. Final sequence $A_0, A_1, A_2, \ldots$:

Putting it all together, the full sequence of integers for the Add/XNOR instance is:

$$A_0 = 2^n - 2T, \quad A_1 = s_1, \quad A_2 = 2^{n-1}, \quad A_3 = s_1, \quad A_4 = 2^{n-1}, \quad A_5 = s_2, \ldots$$

Hence, we have one initial integer A_0, and for each s_i, we add four integers.

If a Subset S' with $\sum_{s_i \in S'} s_i = T$ Exists:

For each $s_i \in S'$, choose the "include" pattern of operations to add $2s_i$ to E_{i-1}. For each $s_i \notin S'$, choose the "exclude" pattern to leave E_{i-1} unchanged.

Initially:

$$E_0 = A_0 = 2^n - 2T.$$

After including all $s_i \in S'$:

$$E := E_m = (2^n - 2T) + 2 \sum_{s_i \in S'} s_i = (2^n - 2T) + 2T = 2^n.$$

Since we work modulo 2^n, therefore $E = \mathbf{0}$.

If We Achieve $E = \mathbf{0}$:

Recall we started with

$$E_0 \;=\; 2^n \;-\; 2T \pmod{2^n}.$$

After processing all m gadgets, we want

$$E_m \;=\; \mathbf{0} \pmod{2^n}.$$

But each gadget for s_i contributes either $+\,2s_i$ or $+\,0$. Hence:

$$E_m \;=\; \left(2^n - 2T\right) \;+\; 2 \sum_{s_i \in S'} s_i \pmod{2^n},$$

where $S' \subseteq \{s_1, \ldots, s_m\}$ is the set of elements chosen by the "include" gadgets. We want $E_m = 0 \pmod{2^n}$. This is equivalent to

$$E := E_m = (2^n - 2T) + 2 \sum_{s_i \in S'} s_i \equiv 0 \pmod{2^n}.$$

Since 2^n is the modulus, and $0 \leq 2^n - 2T + 2\sum_{s_i \in S'} s_i < 2^{n+1}$, the only way for this congruence to hold true is if:

$$(2^n - 2T) + 2 \sum_{s_i \in S'} s_i = 2^n.$$

This simplifies to:

$$2 \sum_{s_i \in S'} s_i = 2T \implies \sum_{s_i \in S'} s_i = T.$$

Hence, a solution to the Add/XNOR instance corresponds to a solution of the subset sum instance.

Therefore, if there is a subset S' that sums to T, we can choose the gadgets' operators so that each $s_i \in S'$ is "included" (adding $2s_i$) and each $s_i \notin S'$ is "excluded," yielding $E_m = 0$. Conversely, if there is a sequence of operators making $E_m = 0$, then exactly the subset of gadgets chosen in the "inclusion" mode determines a subset S' whose sum is T. Hence, the subset sum instance is solvable if and only if the constructed Add/XNOR instance is solvable.

By starting with $A_0 = 2^n - 2T$ and encoding each element s_i into a small sequence of integers $(s_i, 2^{n-1}, s_i, 2^{n-1})$, we have constructed a polynomial-time reduction from the subset sum problem to the Add/XNOR Problem. Since the subset sum problem is NP-complete, this implies that the Add/XNOR problem is NP-hard. Combined with the fact that the problem is in NP, we conclude that the Add/XNOR problem is NP-complete. □

The construction leverages the similarity in selecting operators (addition or XNOR) to include or exclude numbers in a cumulative operation aiming for a specific result (zero). In the complexity-theoretic, the decision version of an NP-complete problem and the corresponding functional version are considered polynomial-time equivalent. This is clearly true for the Add/XNOR Problem.

3 $L \notin \mathbf{P}$

For the problem p of language L defined by Definition 3 with $m := n$, we have the following theorems.

Theorem 2. *The problem p corresponds to a non-linear system, and solving this system is equivalent to an exhaustive search.*

Proof:

The problem p is to determine whether there exists a sequence of operators $O_1, O_2, \ldots, O_{n-1}$, where each O_i is either addition modulo 2^n (denoted by $+$) or bitwise XNOR (denoted by $\odot$), such that when these operators are applied left-associatively to a sequence of n-bit integers $A_1, A_2, \ldots, A_n$, the final result E is the zero vector:

$$E = (((A_1 \, O_1 \, A_2) \, O_2 \, A_3) \, \ldots) \, O_{n-1} \, A_n \equiv \mathbf{0}.$$

The Non-Linear Nature of the Operations:

Consider addition modulo 2^n as a function:

$$f_{\text{add}} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n, \quad f_{\text{add}}(x, y) = x + y \pmod{2^n}.$$

Over the field $\mathbb{F}_2$, bitwise XOR is linear. However, addition mod 2^n involves carrying bits, which cannot be expressed as a simple linear or even affine transformation over $\mathbb{F}_2$. The carry operation introduces a dependency between bits that

is inherently non-linear. More formally, the presence of carries means the output bit depends on combinations of input bits in a way that is not representable by a system of linear equations over $\mathbb{F}_2$.

The XNOR operation $\odot$ between two n-bit numbers a and b is defined as:

$$a \odot b = \overline{a \oplus b},$$

where $\oplus$ is bitwise XOR and $^-$ is bitwise NOT.

XOR: $a \oplus b$ is linear over $\mathbb{F}_2$ because $\oplus$ is just addition mod 2 bit-by-bit. NOT: $\overline{c} = c \oplus (2^n - 1)$. While a NOT operation on its own can be seen as affine (a linear operation plus a constant vector), when combined with addition, it does not preserve linearity in the system as a whole. Overall, XNOR can be viewed as a composition of linear (XOR) and affine (NOT) operations. This composition yields a non-linear transformation in the context where these operations are mixed with addition mod 2^n.

Therefore, when we chain these operations—Add/XNOR—in a sequence:

$$E_{n-1} = (((A_1 \; O_1 \; A_2) \; O_2 \; A_3) \cdots) O_{n-1} A_n,$$

each intermediate step can be seen as applying a non-linear transformation to an n-bit integer. The mixture of addition with carries and the XNOR's NOT operation ensures the result is a non-linear mapping from the original sequence $(A_1, \ldots, A_n)$ and the choice of operations $(O_1, \ldots, O_{n-1})$ to the final E_{n-1}.

Thus, the system of constraints:

$$(((A_1 \; O_1 \; A_2) \; O_2 \; A_3) \cdots) O_{n-1} A_n = \mathbf{0},$$

is a non-linear system. It cannot be expressed as a set of linear equations over any simple algebraic structure like $\mathbb{F}_2^n$.

Next, we will show that problem p is identical to a non-linear system of equations over finite fields $\mathbb{F}_2$. Then, we will show that solving this non-linear system is equivalent to an exhaustive search over all possible operator sequences.

Formalizing the Problem as a System of Equations:

We define the variables and notations as follows.

$A_i \in \{0,1\}^n$ for $i = 1, 2, \ldots, n$. Each A_i is represented by its bits $(a_{i,1}, a_{i,2}, \ldots, a_{i,n})$, where $a_{i,j} \in \{0,1\}$ denotes the j-th bit.

$x_i \in \{0,1\}$ for $i = 1, 2, \ldots, n-1$, where $x_i = 0$ corresponds to addition modulo 2^n (i.e., $O_i = +$), and $x_i = 1$ corresponds to bitwise XNOR (i.e., $O_i = \odot$).

$E_i \in \{0,1\}^n$ denotes the intermediate result after the i-th operation, with bits $(e_{i,1}, e_{i,2}, \ldots, e_{i,n})$ for $i = 0, 1, \ldots, n-1$. $e_{i,j}$ represents the j-th bit of the intermediate result after the i-th operation with $e_{0,j} = a_{1,j}$, for $j = 1, 2, \ldots, n$. $c_{i,j}$ represents the carry into bit j during the i-th addition operation with $c_{i,0} = 0$, for $i = 1, 2, \ldots, n-1$.

Then, we define the equations as follows. At each step $i = 1, 2, \ldots, n-1$, the operation depends on x_i.

If $x_i = 0$, then sum bits:

$$e^{+}_{i,j} = e_{i-1,j} \oplus a_{i+1,j} \oplus c_{i,j-1}, \quad \text{for } j = 1, \ldots, n;$$

and carry bits:

$$c_{i,j} = e_{i-1,j} \cdot a_{i+1,j} + e_{i-1,j} \cdot c_{i,j-1} + a_{i+1,j} \cdot c_{i,j-1}, \quad \text{for } j = 1, \ldots, n.$$

If $x_i = 1$, then result bits:

$$e^{\odot}_{i,j} = 1 - (e_{i-1,j} \oplus a_{i+1,j}), \quad \text{for } j = 1, \ldots, n.$$

We combine the two operations into a single equation:

$$e_{i,j} = (1 - x_i) \cdot e^{+}_{i,j} + x_i \cdot e^{\odot}_{i,j}, \quad \text{for } i = 1, \ldots, n-1;\ j = 1, \ldots, n.$$

Substituting $e^{+}_{i,j}$ and $e^{\odot}_{i,j}$:

$$e_{i,j} = (1 - x_i) \cdot (e_{i-1,j} \oplus a_{i+1,j} \oplus c_{i,j-1}) + x_i \cdot (1 - (e_{i-1,j} \oplus a_{i+1,j}))\,.$$

Similarly, we express the carry bits:

$$c_{i,j} = (1 - x_i) \cdot (e_{i-1,j} \cdot a_{i+1,j} + e_{i-1,j} \cdot c_{i,j-1} + a_{i+1,j} \cdot c_{i,j-1})\,.$$

Finally, we get n equations with $n-1$ unknowns $x_1, x_2, \ldots, x_{n-1}$:

$$e_{n-1,j} = 0, \quad \text{for } j = 1, \ldots, n. \tag{4}$$

From the construction of the non-linear system, we have established the degree d_i of the equations $e_{i,j}$ in terms of the operator variables x_i. The degree grows according to the recurrence relation:

$$d_i = 2 \times d_{i-1} + 1,$$

with the initial condition $d_0 = 0$. Unrolling the recurrence in the equations, we can get the degree of the final equations $e_{n-1,j}$ as a polynomial in $x_1, x_2, \ldots, x_{n-1}$ is $2^{n-1} - 1$.

Solving the Non-Linear System is Equivalent to an Exhaustive Search:

For a system of equations, the essence of all equation-solving algorithms is to reduce the degree and eliminate variables. The equations involve products of variables x_i, leading to terms of high degree (up to $2^{n-1} - 1$). The exponential degree makes the equations highly complex and non-linear. Each x_i determines the operation at step i and is involved multiplicatively in the equations. The equations are recursive, with each $e_{i,j}$ depending on $e_{i-1,j}$, x_i, and other variables. The high-degree terms involving products of x_i cannot be simplified or linearized without assigning values to x_i. Due to the multiplicative and recursive nature of the equations, isolating x_i or expressing them in terms of other variables is not feasible.

Algebraic methods such as substituting expressions may rearrange the terms but do not eliminate the high-degree monomials involving x_i. Due to the binary nature of variables and operations, the highest-degree terms cannot cancel out through algebraic manipulation. The recursive structure and carry-over computations create interdependencies that prevent isolating or simplifying the equations to reduce the degree.

Eliminating x_i means assigning it a specific value (0 or 1), thereby removing it as a variable from the equations. The degree of $e_{n-1,j}$ is fundamentally dependent on the number of x_i variables due to the recursive multiplication. Since the degree is tied directly to the presence of x_i, any reduction in degree must involve eliminating x_i.

The ultimate goal of any method for solving this nonlinear system of equations is to eliminate the variables. We will show that solving the non-linear system using elimination is equivalent to an exhaustive search.

Solving the non-linear system using elimination involves systematically reducing the system of equations by eliminating variables to solve for the unknowns. We have the following steps:

Step 1: Assign Values to Operator Variables x_i

Each x_i can be either 0 or 1. For $n-1$ operator variables, there are 2^{n-1} possible combinations.

To eliminate x_i from the equations, we need to assign it a specific value (0 or 1). This transforms the equations into a system where x_i is no longer a variable.

Step 2: Simplify the Equations Based on x_i Values

When $x_i = 0$, the operation is addition modulo 2^n. The equations simplify to:

$$e_{i,j} = e_{i-1,j} \oplus a_{i+1,j} \oplus c_{i,j-1},$$

$$c_{i,j} = e_{i-1,j} \cdot a_{i+1,j} + e_{i-1,j} \cdot c_{i,j-1} + a_{i+1,j} \cdot c_{i,j-1}.$$

When $x_i = 1$, the operation is bitwise XNOR. The equations simplify to:

$$e_{i,j} = 1 - (e_{i-1,j} \oplus a_{i+1,j}), \quad c_{i,j} = 0.$$

Step 3: Solve the Simplified Equations

Starting from $i = 1$ and $e_{0,j} = a_{1,j}$, compute $e_{i,j}$ for all j. Use the simplified equations based on the assigned x_i values. For addition operations, compute carry bits $c_{i,j}$ recursively. Finally, check whether the final result $e_{n-1,j} = 0$ for all j.

If eliminating x_i yields a simplified system of equations that is still not solvable, we can repeat the above process from Step 1 to further simplify the system of equations by eliminating more operating variables.

Since eliminating x_i effectively requires testing both possible values, it does not simplify the problem but rather shifts the complexity. Each variable eliminated halves the number of combinations and reduces the degree accordingly. That is eliminating one operator variable x_i from the non-linear system derived from the problem p is equivalent to reducing the degree of the system from

$2^{n-1} - 1$ to $2^{n-2} - 1$. This equivalence arises because each x_i contributes exponentially to the degree of the system due to the recursive doubling in the recurrence relation $d_i = 2 \times d_{i-1} + 1$; eliminating x_i removes one level of complexity, halving the degree contribution and the number of possible combinations to consider. Furthermore, reducing the degree by one exponent implies that one x_i is no longer present, as the system's complexity is directly tied to the number of operator variables.

This relationship emphasizes that variables and degree are intrinsically linked in this non-linear system, and attempts to simplify the system by reducing the degree are essentially equivalent to eliminating variables, which requires considering all possible values of x_i. Therefore, solving the system or reducing its complexity inherently involves an exhaustive search over all possible operator sequences. □

As a variant of the subset sum problem, it is not a surprise that the Add/XNOR problem does not have any efficient algebraic algorithm. Search algorithms are another type of algorithm that solves subset sum problems and are also the most efficient. We will explore the lower bound of the computational complexity required to solve p through search.

Theorem 3. *The Adding, XOR and XNOR operations preserve randomness.*

Proof:

The truth table for Adding, XOR and XNOR operations is shown in Table 1. In terms of two single-bit binary operations, XOR can be viewed as an addition without carry, and XNOR is the negation of XOR. Therefore, we will prove below that XOR preserves randomness, and the randomness of Adding and XNOR can be derived similarly.

Consider two binary variables a and b. If both a and b are independently random (each has a $1/2$ chance of being 0 or 1), the result $c = a \oplus b$ will also be random. This is because: If $a = 0$, then $c = b$, so c is equally likely to be 0 or 1, depending on b. If $a = 1$, then $c = \overline{b}$, which is also equally likely to be 0 or 1 because b is random. In both cases, c has a $1/2$ chance of being 0 or 1, which means c is uniformly random.

Now, let us consider the case where a is random, but b is fixed (either 0 or 1). If $b = 0$, then $c = a \oplus 0 = a$, so the output is exactly the same as a, which is random. If $b = 1$, then $c = a \oplus 1 = \overline{a}$, which means that the result is simply the inversion of a, but since a is random, $\overline{a}$ is also random. In both cases, the result remains random, showing that a random bit XORed with a non-random bit will result in a random bit.

Hence, if both input variables are random, the output is random; if one input is random and the other is fixed, the output remains random. This shows that XOR preserves the random characteristics of its input. □

As a corollary, for n-bit operations, addition modulo 2^n, bitwise XOR, and bitwise XNOR maintain randomness, since addition modulo 2^n is equivalent to bitwise single-bit adding then modulo 2^n. For example, the result of adding a random n-bit number to a fixed n-bit integer modulo 2^n will have the same probability of being any element in $\{0, 1\}^n$.

Theorem 4. *For all 2^{n-1} possible combinations of the operator variables x_i in problem p, each combination has the same probability of being a solution to the problem instance randomly sampled from the space.*

Proof:

Let $+$ denote addition modulo 2^n: $a + b \mod 2^n$, $\odot$ denote bitwise XNOR: $a \odot b = \overline{a \oplus b}$, where $\oplus$ denotes bitwise XOR.

Let E_i denote intermediate results with $E_0 = A_1$. For $i = 1, 2, \ldots, n-1$:

$$E_i = \begin{cases} (E_{i-1} + A_{i+1}) \mod 2^n, & \text{if } x_i = 0; \\ E_{i-1} \odot A_{i+1}, & \text{if } x_i = 1. \end{cases}$$

The final result is $E = E_{n-1}$. A combination of x_i is a solution if $E = \mathbf{0}$.

1. Both Operations Preserve Uniform Distribution

1.1. Addition Modulo 2^n

Let X and Y are independent and uniformly distributed over $\{0,1\}^n$, $Z = X + Y \mod 2^n$. For any fixed $z \in \{0,1\}^n$:

$$\Pr(Z = z) = \Pr(X + Y \mod 2^n = z) = \frac{1}{2^n},$$

because for each possible value of X, Y can be any value such that $X + Y \mod 2^n = z$, and since X and Y are independent and uniform, then $Z = X + Y \mod 2^n$ is also uniformly distributed over $\{0,1\}^n$.

1.2. Bitwise XNOR

Let X and Y are independent and uniformly distributed over $\{0,1\}^n$, $Z = X \odot Y$. The bitwise XNOR operation can be thought of as: $Z = \overline{X \oplus Y}$. Since X and Y are independent and uniform, for each bit position j, X_j and Y_j are independent and uniformly random bits. According to Theorem 3, the XOR of two independent random bits is also a uniformly random bit. The complement (NOT) of a uniformly random bit is also uniformly random. Therefore, each bit of Z is independent and uniformly random, so Z is uniformly distributed over $\{0,1\}^n$.

2. The Intermediate Results E_i Are Uniformly Distributed

We will show by induction that for any fixed operator sequence $x_1, x_2, \ldots, x_{n-1}$, and for randomly chosen A_i, each E_i is uniformly distributed over $\{0,1\}^n$.

Base Case ($i = 0$):

$E_0 = A_1$. Since A_1 is uniformly random over $\{0,1\}^n$, E_0 is uniformly random.

Inductive Step:

Assume that E_{i-1} is uniformly distributed over $\{0,1\}^n$ for some $i \geq 1$.

Case 1: $x_i = 0$ (Addition Modulo 2^n)

$E_i = E_{i-1} + A_{i+1} \mod 2^n$. Since E_{i-1} and A_{i+1} are independent and uniformly random, E_i is uniformly random by the property of addition modulo 2^n.

Case 2: $x_i = 1$ (Bitwise XNOR)

$E_i = E_{i-1} \odot A_{i+1}$. Since E_{i-1} and A_{i+1} are independent and uniformly random, E_i is uniformly random by the property of bitwise XNOR.

In both cases, E_i is uniformly distributed over $\{0,1\}^n$. By induction, $E_{n-1} = E$ is uniformly distributed over $\{0,1\}^n$ regardless of the operator sequence $x_1, x_2, \ldots, x_{n-1}$.

3. Probability That $E = \mathbf{0}$ Is $\frac{1}{2^n}$ for Any Operator Sequence

Since E is uniformly distributed over $\{0,1\}^n$, the probability that $E = \mathbf{0}$ (the all-zero vector) is:

$$\Pr(E = \mathbf{0}) = \frac{1}{2^n}.$$

This probability is the same for any fixed operator sequence.

4. All Operator Combinations Have the Same Probability of Being a Solution

There are 2^{n-1} possible combinations of the operator variables x_i. For each combination, the probability that $E = \mathbf{0}$ is $\frac{1}{2^n}$.

Therefore, for all 2^{n-1} possible combinations of the operator variables x_i in problem p, each combination has the same probability of being a solution to the problem. □

The birthday paradox is a core theory of random search. The problem of determining whether there is a person with a particular birthday in a set of people is equivalent to the problem of searching for a particular value among N random values. The problem of finding whether there are two people with the same birthday in a set of people is equivalent to the problem of finding collisions among random values. It is clear that the lower bound of the complexity of finding a particular value among N unordered independent random values is $O(N)$, and the lower bound of the complexity of the corresponding finding collisions is $O(\sqrt{N})$.

In cryptography, an attack based on the birthday paradox is called a birthday attack, which uses this probabilistic model to convert the problems of searching for a particular value into collision-finding problems, to reduce the algorithm complexity from $O(N)$ to $O(\sqrt{N})$. In fact, the square-root algorithm based on the birthday paradox is optimal for finding collisions among independent random numbers drawn uniformly from a finite set.

However, not all problems that search for a particular value among random values can be converted into collision-finding problems. For example, the unstructured data search problem cannot be converted into a collision-finding problem. It is trivial to show that solving the unstructured data search problem requires an exhaustive search and its query complexity is $O(N)$ for Turing machines (Note that this is not an exponential time algorithm.), although its quantum algorithm (Grover's algorithm [7]) complexity is $O(\sqrt{N})$. Fortunately, the Add/XNOR problem can be reduced to a collision-finding problem from a problem that searches for a particular combination among all possible combinations that produce random results, reducing its computational complexity to $O(\sqrt{N})$.

Without loss of generality, in this paper, we assume that the Add/XNOR problem either has exactly one solution or no solution for simplicity. We have the following theorem.

Theorem 5. *For the problem p, the lower bound of the complexity of the search algorithm is* $\Omega(2^{n/2})$.

Proof:

Since there are $n-1$ operations O_i, i.e., 2^{n-1} possible combinations, the time complexity of an exhaustive search is $O(2^{n-1})$.

However, for all combinations, we can check (exclude) two combinations by judging whether E_{n-2} is equal to $2^n - A_n$, or whether it is equal to the bitwise inversion (complement) of A_n. Specifically, we have the following observations.

We can compute E_{n-1} based on E_{n-2} and x_{n-1}:

$$E_{n-1} = \begin{cases} (E_{n-2} + A_n) \mod 2^n, & \text{if } x_{n-1} = 0; \\ E_{n-2} \odot A_n, & \text{if } x_{n-1} = 1. \end{cases}$$

For case 1: $E_{n-2} = 2^n - A_n$, then:

$$E_{n-1} = (E_{n-2} + A_n) \mod 2^n = (2^n - A_n + A_n) \mod 2^n = 2^n \mod 2^n = 0.$$

Hence, if $E_{n-2} = 2^n - A_n$, then $E_{n-1} = 0$ when $x_{n-1} = 0$.

For case 2: $E_{n-2} = \overline{A_n}$ (bitwise complement of A_n), then:

$$E_{n-1} = E_{n-2} \odot A_n = \overline{A_n} \odot A_n = 0.$$

Because bitwise XNOR of a number with its complement yields zero.

Hence, if E_{n-2} equals either $2^n - A_n$ or $\overline{A_n}$, then $E_{n-1} = 0$ for $x_{n-1} = 0$ or $x_{n-1} = 1$, respectively. Therefore, for each E_{n-2}, we can quickly determine whether $E_{n-1} = 0$ for either value of x_{n-1}. If E_{n-2} does not equal $2^n - A_n$ nor $\overline{A_n}$, then $E_{n-1} \neq 0$ for both values of x_{n-1}.

That is, we can determine O_{n-1} (i.e., x_{n-1}) without having to calculate E_{n-1}. This principle effectively halves the number of full evaluations, reducing the complexity from $O(2^{n-1})$ to $O(2^{n-2})$. This principle indicates that the structure of the Add/XNOR problem allows for meet-in-the-middle attacks. Specifically, the recursive application of operations can be split at the midpoint, enabling attackers to precompute partial results and significantly reduce the complexity of finding collisions.

Consider the following transformations:

$$\begin{aligned}
&(((A_1\, O_1\, A_2)\, O_2\, A_3)\, \ldots) O_{n-1}\, A_n = \mathbf{0}, \\
&(((A_1\, O_1\, A_2)\, O_2\, A_3)\, \ldots) O_{n-2}\, A_{n-1} = \mathbf{0}\ R_{n-1}\ A_n, \\
&(((A_1\, O_1\, A_2)\, O_2\, A_3)\, \ldots) O_{n-3}\, A_{n-2} = (\mathbf{0}\ R_{n-1}\ A_n) R_{n-2}\ A_{n-1}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&(((A_1\, O_1\, A_2)\, O_2\, A_3)\, \ldots) O_{\frac{n}{2}-2}\, A_{\frac{n}{2}-1} = (((\mathbf{0}\ R_{n-1}\ A_n) R_{n-2}\ A_{n-1}) \ldots) R_{\frac{n}{2}-1}\ A_{\frac{n}{2}},
\end{aligned}$$

where $a\ R_i\ b := (a - b) \mod 2^n$, if $O_i = +$; $a\ R_i\ b := \bar{a} \oplus b$, if $O_i = \odot$.

These observations suggest that an attacker can split the sequence of operations into two halves, precompute all possible intermediate states for each

half, and then match these intermediate states to find collisions. This approach reduces the time complexity from $O(2^{n-1})$ to $O(2^{n/2})$, effectively halving the computational complexity in terms of bits, at the cost of $O(2^{n/2})$ space complexity requirement.

Furthermore, because the order of addition modulo 2^n and XNOR operations cannot be exchanged, and they do not satisfy the "associative" law, the expression E of the equation (1) needs to be calculated sequentially and serially, and cannot be calculated in parallel. Therefore, the square-root complexity achieved by the strategy of meet-in-the-middle based on the birthday paradox is optimum for the Add/XNOR problem under the sequential serial calculation constraint. This yields a fundamental exponential lower bound of $\Omega(2^{n/2})$. □

The Add/XNOR problem is a variant of the subset sum (or knapsack) problem, and it is necessary to consider whether the most efficient algorithms for the subset sum problem apply to the Add/XNOR problem. First, the above meet-in-the-middle algorithm is equivalent to the algorithm proposed by Horowitz and Sahni [8] based on the birthday paradox to solve the knapsack problem. Later, Schroeppel and Shamir [12,13] proposed it is not necessary to store the full two sets of size $O(2^{n/2})$, they introduced 4-way merge algorithm to generate them on the fly using priority queues, reducing memory requirement to $O(2^{n/4})$. It should be noted that the 4-way merge algorithm does not apply to the Add/XNOR problem because the expressions on both sides of the final equation in the meet-in-the-middle algorithm need to be computed sequentially and cannot be further split into two halves. Furthermore, because the Add/XNOR problem uses a combination of addition mod 2^n and XNOR operations, it makes modulo operations inapplicable to the expression E of equation (1), making the algorithms [9,1,2] that utilize modulo operations to reduce the computational complexity of subset sum problems, allowing the complexity to break the square-root complexity bound for the random collision problem does not apply to the Add/XNOR problem.

In summary, by utilizing the meet-in-the-middle strategy, the bound of the complexity of the search algorithm for the Add/XNOR problem is reduced from $O(2^{n-1})$ to $\Omega(2^{n/2})$. For language L defined by Definition 3, we show that the lower bound of the complexity of the search algorithm is $\Omega(2^{n/2})$. Therefore, $L \notin \mathrm{P}$.

4 The Add/XOR/XNOR Problem

Based on the analysis in Theorem 5, it is clear that the key to enabling the meet-in-the-middle attack to work on the Add/XNOR problem is that addition mod 2^n and XNOR operations can be inverted, which allows us to move half of the A_i's and operations to the right-hand side of the equation to achieve a square-root speedup.

Here, we consider improving the operations while maintaining perfect randomness by making one of them irreversible. First of all, we have the following definition:

Definition 4 (The Operations of $+_{\oplus\odot}$ and $+_{\odot\oplus}$). *For $a, b \in \{0,1\}^n$, we define $+_{\oplus\odot}$ and $+_{\odot\oplus}$ as follows:*

$$\begin{aligned} a +_{\oplus\odot} b &= a + b + (a \oplus b) - (a \odot b) \mod 2^n \\ &= a + b + 1 + 2(a \oplus b) \mod 2^n. \end{aligned}$$

$$\begin{aligned} a +_{\odot\oplus} b &= a + b + (a \odot b) - (a \oplus b) \mod 2^n \\ &= a + b - 1 - 2(a \oplus b) \mod 2^n. \end{aligned}$$

Table 2. Truth table of $+$, $\oplus$, $\odot$, $+_{\oplus\odot}$ and $+_{\odot\oplus}$.

a	b	$a+b$		$a \oplus b$	$a \odot b$	$a +_{\oplus\odot} b$			$a +_{\odot\oplus} b$	
		carry	sum			borrow	carry	sum	carry	sum
0	0		0	0	1	1		1		1
0	1		1	1	0		1	0		0
1	0		1	1	0		1	0		0
1	1	1	0	0	1			1	1	1

As shown in Table 2, the operations $+$, $\oplus$, $\odot$, $+_{\oplus\odot}$ and $+_{\odot\oplus}$ preserve randomness as single-bit binary operations. It should be noted that, for n-bit a and b, $a +_{\oplus\odot} b$ is not equivalent to bitwise $(a +_{\oplus\odot} b) \bmod 2$, and $a +_{\odot\oplus} b$ is not equivalent to bitwise $(a +_{\odot\oplus} b) \bmod 2$, contrary to the fact that $(a + b) \bmod 2^n$ is equivalent to bitwise $(a + b) \bmod 2$. In fact, for n-bit a and b, bitwise $(a +_{\odot\oplus} b) \bmod 2$ is equivalent to bitwise $a \odot b$. For fixed b, $a +_{\oplus\odot} b$ and $a +_{\odot\oplus} b$ are both permutations on $\{0, \ldots, 2^n - 1\}$. More important, we have the following theorem.

Theorem 6. *The operations $+_{\oplus\odot}$ and $+_{\odot\oplus}$ preserve randomness.*

Proof:

Given two n-bit numbers a and b, the operation $+_{\oplus\odot}$ is defined as:

$$c = a +_{\oplus\odot} b = a + b + 1 + 2\,(a \oplus b) \mod 2^n,$$

where $a \oplus b$ is the bitwise XOR of a and b.

We will prove that the operation $+_{\oplus\odot}$ preserves randomness in the following two senses. First, if a and b are chosen independently and uniformly at random in $\{0, \ldots, 2^n - 1\}$, then $c = a +_{\oplus\odot} b$ is also uniformly distributed in $\{0, \ldots, 2^n - 1\}$. Second, for each fixed b, the map $f_b(a) = a +_{\oplus\odot} b$ is a permutation on $\{0, \ldots, 2^n - 1\}$. Consequently, if a is uniformly distributed, then $c = f_b(a)$ is also uniform.

To prove both statements, it suffices to show that for every fixed b, the function

$$f_b(x) \;=\; \big(x + b + 1\big) \;+\; 2\,\big(x \oplus b\big) \mod 2^n,$$

is a bijection, the key step is to show surjectivity. Then f_b is a permutation of the n-bit values. This immediately proves:

1) For each fixed b, $c = f_b(a)$ is a permutation of all n-bit values a. Hence if a is uniformly random, so is c.

2) If a and b are both uniform and independent, then for each fixed b, the image of $a \mapsto c$ is uniform. Since b was uniform and independent to begin with, c remains uniform overall.

We will prove surjectivity by explicitly constructing for each "target" a a unique solution x such that

$$x +_{\oplus\odot} b \;=\; a.$$

Fix $a, b \in \{0,1\}^n$. We want to solve

$$f_b(x) \;=\; a \iff (x + b + 1 + 2(x \oplus b)) \bmod 2^n \;=\; a.$$

Equivalently (working in integers mod 2^n),

$$x + 2(x \oplus b) \;\equiv\; a - (b+1) \pmod{2^n}.$$

Set

$$s \;:=\; a - (b+1) \bmod 2^n.$$

Hence the task is to solve

$$x \;+\; 2\big(x \oplus b\big) \;\equiv\; s \pmod{2^n}. \tag{5}$$

Write each element of $\{0,1\}^n$ in binary:

$$x = (x_{n-1}x_{n-2}\ldots x_1x_0)_2, \quad b = (b_{n-1}b_{n-2}\ldots b_1b_0)_2, \quad s = (s_{n-1}s_{n-2}\ldots s_1s_0)_2.$$

Define the bits $c_i := x_i \oplus b_i$ for $i = 0, \ldots, n-1$. Then

$$x \oplus b \;=\; (c_{n-1}c_{n-2}\ldots c_1c_0)_2,$$

and multiplying by 2 (mod 2^n) corresponds to a "left shift by 1 bit" (with wrap-around ignored since we are mod 2^n). In other words,

$$2\big(x \oplus b\big) \;=\; (\, c_{n-2}c_{n-3}\ldots c_1c_00\,)_2 \pmod{2^n},$$

i.e., the least significant bit becomes 0, and each c_i moves up one position.

We now analyze

$$x \;+\; 2\big(x \oplus b\big) \;=\; (\, x_{n-1}\ldots x_0\,)_2 \;+\; (\, c_{n-2}\ldots c_00\,)_2 \pmod{2^n}.$$

We want this sum to equal $s = (s_{n-1}\ldots s_0)_2$ bit by bit.

We will show that there is a unique way to choose the bits $x_0, x_1, \ldots, x_{n-1}$ so that equation (5) holds. We do this iteratively from least significant bit to most significant bit, keeping track of any carry that arises from lower bits.

1) Initialization. Let carry-in to bit 0 be $c_{\text{in}}(0) = 0$. We must make the 0th bit of $x + 2(x \oplus b)$ equal s_0. But in the sum, the contribution to the 0th bit is

$x_0 + 0 + c_{\text{in}}(0)$ (since the 0th bit of $2(x \oplus b)$ is 0). Exactly one choice of $x_0 \in \{0, 1\}$ will make that sum's 0th bit be s_0. Let that choice be the actual x_0. Define the resulting carry-out from bit 0 to bit 1, call it $c_{\text{in}}(1)$.

2) Inductive step $i \geq 1$. Suppose we have chosen $x_0, \ldots, x_{i-1}$ uniquely so that bits $0, \ldots, i-1$ of the sum $x + 2(x \oplus b)$ match those of s. Now, to match the i-th bit of s, we look at:

$$(\text{bit } i \text{ of } x) \ + \ (\text{bit } i \text{ of } 2(x \oplus b)) \ + \ (\text{carry-in from bit } i-1).$$

The i-th bit of x is x_i. The i-th bit of $2(x \oplus b)$ is $c_{i-1} = (\, x_{i-1} \oplus b_{i-1} \,)$. We already know x_{i-1}, b_{i-1} from the previous step, so we know the value of c_{i-1}. We know the carry-in from the previous bit, $c_{\text{in}}(i)$. Exactly one choice of $x_i \in \{0, 1\}$ will make the i-th bit of that sum equal to s_i. We pick that x_i. This also determines the carry-out to the next bit $c_{\text{in}}(i+1)$.

By continuing this process up through $i = n-1$, we obtain a unique n-bit number x. By construction, it satisfies

$$x + 2(x \oplus b) \ \equiv \ s \pmod{2^n},$$

i.e. it solves equation (5). Therefore, for each $a \in \{0, 1\}^n$, there is exactly one $x \in \{0, 1\}^n$ such that $x +_{\oplus\odot} b = a$.

Hence, for each fixed b, $f_b(x) = x +_{\oplus\odot} b$ is surjective (and therefore bijective) on $\{0, 1\}^n$, and $a \mapsto f_b(a)$ is a permutation. A permutation of a uniformly random variable is still uniformly distributed. Hence if a is uniformly random, $c = f_b(a)$ is also uniformly random.

If a and b are both uniform and independent, then conditioning on any fixed b, we get a uniform c. Since b itself was uniform over $\{0, \ldots, 2^n - 1\}$, the joint distribution of (c, b) is uniform in c and b. Marginalizing over b leaves c uniform.

Therefore, in both senses, the operation $+_{\oplus\odot}$ (or equivalently $f_b(a)$) preserves randomness.

Note that the operation $+_{\odot\oplus}$ preserving randomness can be proved equally well, since for each fixed b, the map $f_b(a) \ = \ a +_{\odot\oplus} b$ is also a permutation on $\{0, \ldots, 2^n - 1\}$. Consequently, if a is uniformly distributed, then $c = f_b(a)$ is also uniform. □

It is clear that, if $a +_{\oplus\odot} b = c$ (or $a +_{\odot\oplus} b = c$) for n-bit a, b and c, there is no way to write a as a function of b and c, i.e., $a = f(b, c)$. We cannot isolate a in a simple, unique algebraic form purely in terms of b and c because of the XOR $(a \oplus b)$ in the expression.

However, taking the operation $+_{\odot\oplus}$ as an example, we can convert the problem of computing the inverse of the following function to the problem of finding the inverse of an element in a group:

$$f_b(x) \ = \ \big(x + b - 1\big) \ - \ 2\,\big(x \oplus b\big) \mod 2^n,$$

for fixed b.

Let N denote the set of all n-bit numbers. Since there is no identity element, it is clear that $\langle N, +_{\odot\oplus} \rangle$ does not form a group. However, $G = \langle N, \overline{+_{\odot\oplus}} \rangle$ forms a

cyclic group, where the operation $\overline{+_{\odot\oplus}}$ is bitwise NOT (or bitwise complement) of the result of $+_{\odot\oplus}$. More importantly, the inverse of the function f_b with respect to x is equivalent to finding the inverse of b in the group G:

$$\begin{aligned} c = x +_{\odot\oplus} b &\iff \overline{c} = x \,\overline{+_{\odot\oplus}}\, b \\ &\iff \overline{c} \,\overline{+_{\odot\oplus}}\, b' = x \,\overline{+_{\odot\oplus}}\, b \,\overline{+_{\odot\oplus}}\, b' = x, \end{aligned}$$

where b' is the inverse of b in G with respect to the operation $\overline{+_{\odot\oplus}}$.

Similarly, for operation $+_{\oplus\odot}$, the problem of computing the inverse of the function can be converted to finding the inverse of an element in the cyclic group $\langle N, \overline{+_{\oplus\odot} + 1}\rangle$, where $a \,\overline{+_{\oplus\odot} + 1}\, b := \overline{(a +_{\oplus\odot} b + 1) \mod 2^n} = \mathrm{NOT}((a +_{\oplus\odot} b + 1) \mod 2^n)$. There exist polynomial-time algorithms to compute the inverse of elements in these groups.

Consider the algebraic structure $\langle N, +_{\oplus\odot}\rangle$ (and $\langle N, +_{\odot\oplus}\rangle$ as well), it is trivial to check that it has the following properties:

- Closed.
- Commutative (the formula is symmetric in a and b).
- Not associative.
- No identity element.

Consequently, it is not a group (no identity, no inverses in the group sense). A binary operation $\star$ on a finite set Q makes $\langle Q, \star\rangle$ a quasigroup if and only if, for every fixed $b \in Q$, the map

$$x \longmapsto x \star b$$

is a bijection (permutation) of Q. Equivalently, each row of the Cayley table is a rearrangement of Q (no repeats). By symmetry, each column is also a permutation, so the table is a Latin square.

Therefore, to prove quasigroup property for $\langle N, +_{\oplus\odot}\rangle$, it suffices to fix $b \in N$ and show that the function

$$f_b(x) \;:=\; x +_{\oplus\odot} b \;=\; \big(x + b + 1 + 2(x \oplus b)\big) \mod 2^n,$$

is a bijection $N \to N$. This is exactly what we proved in Theorem 6. Hence, let N denote the set of all n-bit numbers, we have the following theorem.

Theorem 7. *$\langle N, +_{\oplus\odot}\rangle$ (and $\langle N, +_{\odot\oplus}\rangle$ as well) forms a commutative quasigroup.*

As analyzed above, the inverse of an operation on a quasigroup can be easily converted into a problem of finding the inverse of an element of a group. Therefore, we need to consider or design a more fundamental algebraic structure, one that is simpler than the requirements of a quasigroup. We consider giving up the commutative and partial permutation properties while keeping no associative property. Hence, we have the following design.

Definition 5 (The Operation of $+\oplus\odot$). *For $a, b \in N$, let $a = a_1 a_2 \dots a_n$ be the binary representation of a. We define $+\oplus\odot$ by the following left-associative expression:*

$$a +\oplus\odot b := (((b\ O_{a_1} a)\ O_{a_2} a) \dots)\ O_{a_n} a,$$

where each $O_{a_i} \in \{+_{\oplus\odot}, \overline{+_{\odot\oplus}}\}$ with $O_0 = +_{\oplus\odot}$ and $O_1 = \overline{+_{\odot\oplus}}$.

Note that it is also appropriate to set $O_{a_i} \in \{\overline{+_{\oplus\odot} + 1}, +_{\odot\oplus}\}$, $\{\overline{+_{\oplus\odot} + 1}, \overline{+_{\odot\oplus}}\}$ or $\{+_{\oplus\odot}, +_{\odot\oplus}\}$, where $a\ \overline{+_{\oplus\odot} + 1}\ b := \overline{(a +_{\oplus\odot} b + 1)} \mod 2^n$, to define operation $+\oplus\odot$. The definition is certainly *well-defined* as we always get exactly one outcome for each $a, b \in N$. The operation $+\oplus\odot$ is explicitly a function from $N \times N$ to N. Consider the algebraic structure $\langle N, +\oplus\odot \rangle$, it is easy to verify that it has the following properties:

- Closed.
- Not commutative.
- Not associative.
- No identity element.

Therefore, $\langle N, +\oplus\odot \rangle$ is just a magma. A magma is a fundamental type of algebraic structure, consisting of a set with a single binary operation that, by definition, needs closure. No other properties are imposed. It does not satisfy the usual higher-level algebraic axioms that would make it a quasigroup.

Theorem 8. *The operation $+\oplus\odot$ preserves randomness.*

Proof:

We will show that for each fixed a, the map

$$b \longmapsto a\ +\oplus\odot\ b,$$

is a permutation of the n-bit space $N = \{0, 1, \dots, 2^n - 1\}$. Once we establish that it is a permutation for each a, it follows immediately that:

If b is drawn uniformly at random from N, then $a\ +\oplus\odot\ b$ is also uniformly distributed in N. This completes the proof of the operation $+\oplus\odot$ "preserves randomness".

We have N = the set of all n-bit unsigned integers ($|N| = 2^n$), and two "sub-operations" given by:

$$x +_{\oplus\odot} y\ :=\ \big(x + y + 1 + 2(x \oplus y)\big) \bmod 2^n.$$

This is known to be a *quasigroup* operation—indeed, for each fixed y, the map $x \mapsto x +_{\oplus\odot} y$ is a permutation of N.

$$x +_{\odot\oplus} y\ :=\ \big(x + y - 1\ -\ 2(x \oplus y)\big) \bmod 2^n.$$

By a very similar "bitwise" argument, for each fixed y, the map $x \mapsto x +_{\odot\oplus} y$ is likewise a permutation of N.

Furthermore, we define

$$\overline{x +_{\odot\oplus} y} \;:=\; (2^n - 1) \;-\; \Big[\big(x + y - 1 - 2(x \oplus y)\big) \bmod 2^n\Big].$$

Notice that $u \mapsto (2^n - 1) - u$ is also a simple "bitwise complement" bijection on N. Hence, for each fixed y,

$$x \;\longmapsto\; \overline{x +_{\odot\oplus} y}$$

is a composition of two permutations—so it, too, is a permutation in the variable x.

Finally, the operation $+\oplus\odot$ on $a, b \in N$ is defined in a left-associative manner, depending on the binary bits of a. Namely, if

$$a = a_1 a_2 \ldots a_n \quad (\text{each } a_i \in \{0, 1\}),$$

then

$$a \;+\oplus\odot\; b \;:=\; \big(\big(\big(\, b \, O_{a_1} \, a\big) \, O_{a_2} \, a\big) \ldots\big) \, O_{a_n} \, a,$$

where

$$O_0 \;=\; +_{\oplus\odot}, \quad O_1 \;=\; \overline{+_{\odot\oplus}}.$$

In words, we start with the value b on the left, and repeatedly apply either $+_{\oplus\odot}$ or $\overline{+_{\odot\oplus}}$ with a on the right, exactly n times, depending on whether each bit a_i of a is 0 or 1.

Given a fixed a, define the function

$$F_a : N \;\longrightarrow\; N, \quad F_a(b) \;:=\; a \;+\oplus\odot\; b.$$

Expanding the definition, we see

$$F_a(b) \;=\; \underbrace{\Big(\big(\big(b \; O_{a_1} \; a\big) \; O_{a_2} \; a\big) \ldots\Big)}_{\text{an iterated application of } O_{a_i}} \; O_{a_n} \; a.$$

But each sub-operation $x \mapsto x \; O_{a_i} \; a$ is, as argued above, a permutation of N in the variable x.

Indeed, if $a_i = 0$, then $x \mapsto x +_{\oplus\odot} a$ is a permutation in x. If $a_i = 1$, then $x \mapsto x \,\overline{+_{\odot\oplus}}\, a$ is also a permutation in x.

Therefore, each step of the iteration is a permutation of N in the variable "running total." A finite composition of permutations is still a permutation.

Concretely: 1) Start with the identity map $x \mapsto x$. 2) Compose it with the map $x \mapsto x \; O_{a_1} \; a$. (A permutation.) 3) Compose the result with $x \mapsto x \; O_{a_2} \; a$. (Another permutation, so overall still a permutation.) 4) Continue through all bits $a_1, \ldots, a_n$.

Hence the final map $F_a(b)$ is a permutation on b. That is, for each fixed $a \in N$, the function $b \mapsto a \,+\oplus\odot\, b$ is a bijection $N \to N$.

It is clear that if X is a random variable taking values in a finite set S, and $f : S \to S$ is a bijection, then $f(X)$ has the same distribution as X. In particular,

if X is *uniformly distributed* over S, then $f(X)$ remains uniformly distributed over S.

Applying this to our case: if b is a random variable uniformly distributed over N, then for each fixed a, the map

$$b \longmapsto a \ +\oplus\odot \ b$$

is a bijection $N \to N$. Therefore, $a \ +\oplus\odot \ b$ is also uniform on N.

That is precisely the statement that $+\oplus\odot$ preserves randomness in its right operand: "If b is uniform, then $a +\oplus\odot b$ is uniform."

We summarize the structure of the proof as follows:

1) Each sub-operation is a permutation (in the left variable for fixed right operand). $x \mapsto x +_{\oplus\odot} a$ is a known quasigroup translation. $x \mapsto x +_{\odot\oplus} a$ is similarly invertible. $x \mapsto 2^n-1-x$ is obviously invertible. Therefore $x \mapsto x \overline{+_{\odot\oplus}} a$ is also a permutation.

2) The left-associative chaining is a composition of permutations. For a fixed a, the bits $a_1, \ldots, a_n$ dictate which permutation we compose at each step. A finite composition of permutations is itself a permutation. Hence $b \mapsto a +\oplus\odot b$ is bijective.

3) Bijections preserve uniform distributions. If b is uniform on $\{0, \ldots, 2^n-1\}$, then so is the image $F_a(b)$.

Therefore, for each fixed a, $b \mapsto a +\oplus\odot b$ acts as a randomness-preserving transformation of b. □

According to Theorem 8, we have the following result.

Theorem 9. *$\langle N, +\oplus\odot\rangle$ forms a magma equipped with a permutation in the sense that for each fixed a, the map $f_a(b) = a +_{\oplus\odot} b$ is a permutation on $\{0, \ldots, 2^n - 1\}$.*

Recall that $\langle N, +_{\oplus\odot}\rangle$ forms a quasigroup. For each fixed b, the map $f_b(a) = a +_{\oplus\odot} b$ is a permutation on $\{0, \ldots, 2^n - 1\}$; for each fixed a, the map $f_a(b) = a +_{\oplus\odot} b$ is also a permutation on $\{0, \ldots, 2^n - 1\}$. Furthermore, computing the inverse of $+_{\oplus\odot}$ can be transformed into computing the inverse of an element in the group G.

By contrast, $\langle N, +\oplus\odot\rangle$ does not form a quasigroup. For each fixed a, the map $f_a(b) = a +_{\oplus\odot} b$ is a permutation on $\{0, \ldots, 2^n - 1\}$, which ensures that it preserves randomness; while for each fixed b, the map $f_b(a) = a +_{\oplus\odot} b$ is not a permutation on $\{0, \ldots, 2^n - 1\}$, which makes it hard to find the reverse. Computing the inverse of $+\oplus\odot$ can not be transformed into computing the inverse of an element in a group.

Definition 6 (The Inverse of $+\oplus\odot$ (Decision Problem)). *Given $b, c \in N$, determine whether there is an a such that $c = a +\oplus\odot b$.*

Definition 7 (The Inverse of $+\oplus\odot$ (Computational Problem)). *Given $b, c \in N$, find an a such that $c = a +\oplus\odot b$ if such an a exists.*

$\langle N, +\oplus\odot\rangle$ is a simpler structure than the quasigroup. It is a magma equipped with a permutation that makes the operation $+\oplus\odot$ both preserving randomness and ensuring computational difficulty for the inverse. Unlike the inverse of the operation $+_{\oplus\odot}$, which has exactly one solution, the inverse of $+\oplus\odot$ may have no solution or multiple solutions. This property is exactly what we need, and it is particularly suitable for designing irreversible functions or one-way functions.

Since operation $+\oplus\odot$ is composed of addition modulo 2^n, bitwise $\oplus$, $+_{\oplus\odot}$ and $\overline{+_{\odot\oplus}}$, there is no effective algebraic method to find the inverse of it. In the presence of bitwise logical operators (like XOR, XNOR) mixed with integer addition, standard algebraic tools do not help: the problem lives partly in Boolean algebra (XOR/XNOR) and partly in modular integer arithmetic (carries).

In summary, the difficulty of its inversion is reflected in the following aspects: on the one hand, because it involves Boolean algebra and integer arithmetic, there may not exist an algebraic method to find the inverse; on the other hand, its inverse may not exist, may have one solution, or may have multiple solutions; finally, the left-associative property of the expression makes it impossible to use the meet-in-the-middle strategy to find the inverse. Currently, the inverse of the operation requires an exhaustive search. Therefore, we have the following computational complexity conjecture.

Conjecture 1 *Solving the problem of the inverse of $+\oplus\odot$ requires an exhaustive search.*

Finally, take advantages of the perfect randomness of operations $\oplus$, $\odot$ and $+\oplus\odot$, as well as the non-linearity and irreversibility introduced by $+\oplus\odot$, we get the following new language.

Definition 8 (The Add/XOR/XNOR Problem (Decision Problem)). *Let m be an integer constant (e.g., $m = 1024$). Given a sequence of n integers $A_1, A_2, \ldots, A_n$, each chosen independently and uniformly at random from $\{0,1\}^m$ (i.e., each is an m-bit number), determine whether there exists a sequence of operators $O_{x_1}, O_{x_2}, \ldots, O_{x_{n-1}}$, where each $O_{x_i} \in \{\oplus, +\oplus\odot\}$, such that the sequential left-associative expression:*

$$E = (((A_1\, O_{x_1}\, A_2)\, O_{x_2}\, A_3)\, \ldots)O_{x_{n-1}}\, A_n \equiv \mathbf{0}. \tag{5}$$

Without loss of generality, in this paper, we will assume $m := n$ for the Add/XOR/XNOR problem for simplicity. Each bit x_i determines which operation is chosen at step i: if $x_i = 0$, then the i-th operation O_{x_i} is bitwise $\oplus$; if $x_i = 1$, then the i-th operation O_{x_i} is $+\oplus\odot$.

Under Conjecture 1, for the Add/XOR/XNOR problem, the meet-in-the-middle attack does not apply, and solving the problem requires an exhaustive search due to the fact that expression E needs to be computed serially as well as the operation $+\oplus\odot$ is irreversible. Therefore, the lower bound on the computational complexity of the problem is $\Omega(2^{n-1})$.

Note that, we can also define the Add/XOR/XNOR problem by setting $O_{x_i} \in \{\odot, +\oplus\odot\}$.

5 One-Way Functions (OWFs)

According to Definition 7 and Conjecture 1, based on the difficulty of inverting the operation

$$x \;\mapsto\; x \;+\!\oplus\!\odot\; b,$$

for a fixed b, we immediately get a one-way function.

Definition 9. *Let $n \in \mathbb{N}$. Let $N = \{0, 1, \ldots, 2^n - 1\}$ be the set of n-bit numbers. Given a fixed $b \in N$, define function f_b as:*

$$\begin{aligned} f_b :\; \{0,1\}^n \;&\longrightarrow\; \{0,1\}^n, \\ x \;&\longmapsto\; x \;+\!\oplus\!\odot\; b. \end{aligned}$$

The function f_b is efficient to compute. Computing $x \;+\!\oplus\!\odot\; b$ takes n steps (one for each bit of x), where each step involves simple integer addition mod 2^n, XOR, and/or a bitwise complement. So $\text{Time}(f_b) = O(n)$. It is conjectured hard to invert. Inversion means: given $y \in N$, find x such that $y = x \;+\!\oplus\!\odot\; b$. Because the specific sub-operation in each of the n left-associative steps depends on whether the corresponding bit in x is 0 or 1, naive "backtracking" requires trying 2^n possibilities. No known better-than-exponential algorithm solves this in the general case. Hence, under the complexity assumption, f_b serves as a *candidate one-way function.*

On the other hand, it is clear that there is currently no known NP-hard problem that has been proven to be solvable only by brute-force search. The problem of the inverse of $+\!\oplus\!\odot$ and the Add/XOR/XNOR problem are supposed to be the computationally difficult problems that can only be solved by brute-force search. We can also design efficient one-way functions based the Add/XOR/XNOR problem since it is also resistant to square-root attacks. According to the Add/XOR/XNOR problem, we can get a one-way function with n bits of input and n bits of output as follows.

Definition 10. *Let $x = x_1 x_2 \cdots x_n \in \{0,1\}^n$. For a fixed sequence of n-bit integers $A_0, A_1, \ldots, A_n$, we define function f as:*

$$\begin{aligned} f :\; \{0,1\}^n \;&\longrightarrow\; \{0,1\}^n, \\ x \;&\longmapsto\; (((A_0\, O_{x_1}\, A_1)\, O_{x_2}\, A_2) \cdots)\, O_{x_n}\, A_n. \end{aligned}$$

where O_{x_i} is bitwise $\oplus$ if $x_i = 0$ and $+\!\oplus\!\odot$ if $x_i = 1$.

This function is considered one-way under the hardness of the Add/XOR/XNOR problem, meaning it is easy to compute y given x, but hard to invert f to find x given y and the A_i's.

The computational difficulty of inverting this one-way function is equivalent to solving the Add/XOR/XNOR problem. More concretely: Given a fixed sequence of n-bit integers $A_0, A_1, \ldots, A_n$, and a target n-bit value y, determine whether there exists an n-bit sequence $x \in \{0,1\}^n$ (defining a pattern of Add/XOR/XNOR operations) such that the following equation holds:

$$E = ((((A_0\, O_{x_1}\, A_1)\, O_{x_2}\, A_2) \cdots)\, O_{x_n}\, A_n) \;\oplus\; y = \mathbf{0}.$$

Here, inverting the function f means finding x (if it exists) that satisfies the above equation for the given y. Computing the inverse of the given one-way function is computationally equivalent to solving the Add/XOR/XNOR problems. Hence, f serves as a *candidate one-way function*. Furthermore, it is also possible to design hash functions from the proposed one-way functions based on the Merkle–Damgård construction.

In practice, we can consider the binary representations of the mathematical constants such as π and e as pseudo-random numbers to generate A_i's. Mathematically, they're like every other irrational number — infinite strings of 0s and 1s (with no discernible pattern). Naturally, we have n-bit π or e Add/XOR/XNOR problem. Consider the infinite binary expansion of π. Let π be represented in base 2 as a (non-terminating) bit string:

$$\pi = b_1 b_2 . b_3 \cdots ,$$

where each $b_i \in \{0, 1\}$.

For a given positive integer n, extract the first n^2 bits of π. That is, consider the substring $b_1 b_2 \cdots b_{n^2}$ from the binary expansion of π. Partition these n^2 bits into n consecutive n-bit integers. Formally, let:

$$A_1 = (b_1 b_2 \cdots b_n)_2, \quad A_2 = (b_{n+1} b_{n+2} \cdots b_{2n})_2, \quad \ldots,$$

$$A_n = (b_{(n-1)n+1} b_{(n-1)n+2} \cdots b_{n^2})_2.$$

Here, $(b_j b_{j+1} \cdots b_{j+n-1})_2$ denotes the integer formed by interpreting the n-bit sequence as a binary number.

Definition 11 (The n-bit π Add/XOR/XNOR problem). *Given the sequence of n-bit integers $A_1, A_2, \ldots, A_n$ constructed as above from the first n^2 bits of π, determine if there exists a sequence of $n-1$ operations $O_{x_1}, O_{x_2}, \ldots, O_{x_{n-1}}$ with each $O_{x_i} \in \{\oplus, +\oplus\odot\}$, such that when applied in a left-associative manner:*

$$E = (((A_1 \; O_{x_1} \; A_2) \; O_{x_2} \; A_3) \cdots) \; O_{x_{n-1}} \; A_n = \mathbf{0}.$$

In other words, the n-bit π Add/XOR/XNOR problem is the decision problem of whether a particular sequence of n-bit integers derived directly from the binary expansion of π can be transformed into the all-zero n-bit vector by some combination of $n-1$ Add/XOR/XNOR operations. The corresponding computational problem is to recover a solution if at least one exists. Similarly, we can construct one-way functions and hash functions based on the n-bit π or e Add/XOR/XNOR problems, which have the advantage that π and e are public constants and can resist meet-in-the-middle attacks.

Finally, we leave an open question as a challenge: find a solution for the n-bit π Add/XOR/XNOR problem with $n \geq 1024$.

6 Conclusion

We introduce the Add/XNOR problem, which has the simplest structure and perfect randomness. Choosing addition modulo 2^n and XNOR makes us give

up the "associativity" property, but allows us to reap the benefits of the limitations of sequential computation, making the square-root algorithm based on the birthday paradox the optimal algorithm. We show that the new language of the Add/XNOR problem is in NP, but not in P. Therefore, it is proved that P $\neq$ NP.

Furthermore, by giving up the commutative property, we design a magma equipped with a permutation from two commutative quasigroups and left-associative evaluation, and achieve Conjecture 1. Based on this conjecture, we obtain the Add/XOR/XNOR problem and one-way functions, which are believed to require exhaustive search to solve or invert.

Acknowledgements

I would like to thank Prof. Antoine Joux for pointing out a mistake in the earlier version of the paper.

References

1. A. Becker, J.-S. Coron, and A. Joux. Improved generic algorithms for hard knapsacks. In *Advances in Cryptology – EUROCRYPT 2011*, pages 364–385. Springer Berlin Heidelberg, 2011.
2. X. Bonnetain, R. Bricout, A. Schrottenloher, and Y. Shen. Improved classical and quantum algorithms for subset-sum. In *Advances in Cryptology – ASIACRYPT 2020*, pages 633–666. Springer International Publishing, 2020.
3. R. P. Brent. An improved monte carlo factorization algorithm. *BIT*, 20(2):176–184, 1980.
4. S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.
5. L. Fortnow. The status of the p versus np problem. *Communications of the ACM*, 52(9):78–86, 2009.
6. L. Fortnow. *The Golden Ticket: P, NP, and the Search for the Impossible*. Princeton University Press, 2013.
7. L. K. Grover. A fast quantum mechanical for database search. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 212–219, 1996.
8. E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of the Association for Computing Machinery*, 21(2):277–292, 1974.
9. N. Howgrave-Graham and A. Joux. New generic algorithms for hard knapsacks. In *Advances in Cryptology – EUROCRYPT 2010*, pages 235–256. Springer Berlin Heidelberg, 2010.
10. J. M. Pollard. A monte carlo method for factorization. *BIT*, 15(3):331–335, 1975.
11. J. M. Pollard. Monte carlo methods for index computation mod p. *Mathematics of Computation*, 32:918–924, 1978.
12. R. Schroeppel and A. Shamir. A $ts^2 = o(2^n)$ time/space tradeoff for certain np-complete problems. In *20th Annual Symposium on Foundations of Computer Science*, pages 328–336, 1979.
13. R. Schroeppel and A. Shamir. A $t = o(2^{n/2})$, $s = o(2^{n/4})$ algorithm for certain np-complete problems. *SIAM Journal on Computing*, 10(3):456–464, 1981.