A Heuristic Proof of P $\neq$ NP

Ping Wang

Shenzhen University
wangping@szu.edu.cn
December 22, 2024

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 $\neq$ 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.

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 [3]. 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 [4,5] 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 $\neq$ 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 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 $x = x_1 x_2 \dots x_n$, 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 [6,1,2].

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. 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: Determining 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 4. 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. Whereas 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.

For the language L (Add/XNOR problem) defined by Definition 4, 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., $x = x_1 x_2 \ldots x_{n-1}$, 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 $O_1, O_2, \ldots, O_{n-1} \in \{+, \odot\}$, we can compute the expression E as:

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

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 $E = \mathbf{0}$:

We start at:

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

If after processing all the gadgets (one for each s_i) using the available patterns (which either add $2s_i$ or do nothing) we want E_i to end up at $\mathbf{0}$, it means:

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

Since 2^n is the modulus, and $0 \le 2^n - 2T + 2\sum_{s_i \in S'} s_i < 2^{n+1}$, 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.

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).

3 $L \notin \mathbf{P}$

For the problem p of language L defined by Definition 4 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: $\bar{c} = c \oplus (2^n - 1)$. 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. □

Since solving the system of equations corresponding to problem p is essentially equivalent to an exhaustive search, 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. 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.*

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 $\frac{1}{2^n}$. Hence, each operator combination has the same probability of being a solution, specifically $\frac{1}{2^n}$.

Therefore, given a sequence of n integers $A_1, A_2, \ldots, A_n$, each chosen independently and uniformly at random from $\{0,1\}^n$, each of the 2^{n-1} possible combinations of the operator variables x_i in the Add/XNOR problem has the same probability of being a solution to the problem. □

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

Corollary 1. *For problem p: if there is a solution, the algorithm needs to verify half of the combinations on average to get to the result; if there is no solution, the algorithm needs to verify all of the combinations to get to the result.*

Proof:

There are $n - 1$ operator variables x_i, each taking values in $\{0,1\}$. Total possible combinations of x_i is $N = 2^{n-1}$. The algorithm systematically tests different combinations of x_i to compute E and checks if $E = \mathbf{0}$. The order of testing combinations can be arbitrary (e.g., ascending or descending order, random).

The expected number of trials $\mathbb{E}[X]$ is the average number of combinations the algorithm needs to test to find the solution. The solution (if it exists) is equally likely to be any one of the N combinations, due to the uniform randomness of A_i and the operations' properties.

For the Case When a Solution Exists (Exactly One Solution)

Since there is exactly one solution, according to Theorem 4, the probability that any given combination is the solution: $p = \frac{1}{N} = \frac{1}{2^{n-1}}$. The solution is equally likely to be in any position from 1 to N in the sequence of combinations tested by the algorithm.

The expected number of trials to find the solution is the average position of the solution:

$$\mathbb{E}[X] = \frac{1}{N}\sum_{k=1}^{N} k = \frac{N(N+1)}{2N} = \frac{N+1}{2} \approx \frac{N}{2}.$$

Therefore, on average, the algorithm needs to verify half of the N combinations to find the solution when exactly one solution exists.

For the Case When No Solution Exists

Given that no solution exists, there is no combination of x_i that results in $E = \mathbf{0}$. Without prior knowledge, the algorithm cannot conclude that no solution exists until it has exhausted all possibilities since each of the 2^{n-1} possible combinations of the operator variables x_i has the same probability of being a solution to the problem p according to Theorem 4. Therefore, the algorithm must verify all N combinations to conclude that no solution exists. □

The birthday paradox is a core theory of random search. The problem of finding 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, although its quantum algorithm (Shor's algorithm [7]) complexity is $O(\sqrt{N})$. Note that the problem is not NP-hard. 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 $O(\sqrt{N})$.

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 (MITM) 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:

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

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 $\Omega(2^{n/2})$. $\square$

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 4, we show that the lower bound of the complexity of the search algorithm is $\Omega(2^{n/2})$. Therefore, $L \notin$ 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. More precisely, we have the following definitions:

$$a +_{\odot\oplus} b := \begin{cases} a + b + (a \odot b) - (a \oplus b), & \text{for single-bit } a \text{ and } b; \\ (a + b + (a \odot b) - (a \oplus b)) \bmod 2^n, & \text{for } n\text{-bit } a \text{ and } b. \end{cases}$$

$$a +_{\oplus\odot} b := \begin{cases} a + b + (a \oplus b) - (a \odot b), & \text{for single-bit } a \text{ and } b; \\ (a + b + (a \oplus b) - (a \odot b)) \bmod 2^n, & \text{for } n\text{-bit } a \text{ and } b. \end{cases}$$

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

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

As shown in Table 2, the operations $+$, $\oplus$, $\odot$, $+_{\odot\oplus}$ and $+_{\oplus\odot}$ preserve randomness for single-bit binary operations. Correspondingly, for n-bit operations, addition modulo 2^n, bitwise $\oplus$, bitwise $\odot$, $+_{\odot\oplus}$ modulo 2^n and $+_{\oplus\odot}$ modulo 2^n preserve randomness.

The elegance of the definition of operation $+_{\odot\oplus}$ (and $+_{\oplus\odot}$) is that it is a combination of addition mod 2^n, XOR and XNOR, and it maintains perfect randomness according to the truth table. On the other hand, if $a +_{\odot\oplus} b = c$ (or $a +_{\oplus\odot} b = c$) for n-bit a, b and c, there is no way to write a as a function of b and c, i.e., $a = f(b, c)$. The operation is not invertible in a simple closed form that would let us uniquely express a as a function of b and c.

Because of the XOR ($a \oplus b$) in the expression, we cannot isolate a in a simple, unique algebraic form purely in terms of b and c. The operations:

$$c \;=\; a \;+_{\odot\oplus}\; b \;=\; \big(a + b + (a \odot b) - (a \oplus b)\big) \bmod 2^n,$$

$$c' \;=\; a \;+_{\oplus\odot}\; b \;=\; \big(a + b + (a \oplus b) - (a \odot b)\big) \bmod 2^n,$$

simplifies to

$$c \;=\; a + b - 2\,(a \oplus b) + (2^n - 1) \;=\; a + b - 1 - 2\,(a \oplus b) \bmod 2^n,$$

$$c' \;=\; a + b + 2\,(a \oplus b) - (2^n - 1) \;=\; a + b - 1 + 2\,(a \oplus b) \bmod 2^n,$$

and that non-linear dependence on $a \oplus b$ prevents writing a single closed-form expression for a. The definition mixes addition (which involves carries) and XOR (which does not), leading to a non-injective mapping. Because XOR can cause collisions, for certain (b, c) pairs, there may be multiple valid a or perhaps none at all. Hence, in general, we cannot write a explicitly as a function of b and c. Thus, we cannot cleanly isolate a given b and c, i.e., operations $+_{\odot\oplus}$ and $+_{\oplus\odot}$ are not invertible.

It should be noted that there is a surprising fact that $a+b \bmod 2^n$ is invertible while $a +_{\odot\oplus} b \bmod 2^n$ is not, which contradicts the intuition given in truth Table 2. The crux of this is that $a +_{\odot\oplus} b \bmod 2^n$ is not equivalent to bitwise single-bit $+_{\odot\oplus}$, contrary to the fact that $a + b \bmod 2^n$ is equivalent to bitwise single-bit $+$.

Finally, according to the truth Table 2, the combination of $\oplus$ and $+_{\odot\oplus}$ (or $+_{\oplus\odot}$) can achieve perfect randomness, and the operations $+_{\odot\oplus}$ and $+_{\oplus\odot}$ introduces non-linearity and irreversibility. As a result, we get the following new language.

Definition 4 (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 $O_{x_i} \in \{\oplus, +_{\odot\oplus}\}$, 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 $+_{\odot\oplus}$ modulo 2^n.

Therefore, 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 $+_{\odot\oplus}$ 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 $O_{x_i} \in \{\oplus, +_{\oplus\odot}\}$, since the operation $+_{\oplus\odot}$ is also not invertible.

5 One-Way Functions (OWF)

Since the Add/XOR/XNOR problem is resistant to square-root attacks, we can design efficient one-way functions based on it. According to the Add/XOR/XNOR problem, we can get a one-way function with n bits of input and n bits of output. Consider a fixed sequence of n-bit random integers $A_0, A_1, \ldots, A_n$. We define a function f that maps an n-bit input string, which represents a choice of "$\oplus$" or "$+_{\odot\oplus}$" operations, to an n-bit output.

The input to f is a binary string $x = x_1 x_2 \cdots x_n \in \{0,1\}^n$. Each bit x_i determines which operation is chosen at step i: if $x_i = 0$, then the i-th operation is bitwise $\oplus$; if $x_i = 1$, then the i-th operation is $+_{\odot\oplus}$ modulo 2^n.

Using the bits of x as instructions, we apply a sequence of n operations to the fixed sequence $A_0, A_1, \ldots, A_n$ in a left-associative manner:

$$y = (((A_0\, O_{x_1}\, A_1)\, O_{x_2}\, A_2) \cdots)\, O_{x_n}\, A_n,$$

where O_{x_i} is bitwise $\oplus$ if $x_i = 0$ and $+_{\odot\oplus}$ modulo 2^n if $x_i = 1$. After performing all n operations, we obtain a final n-bit result $y \in \{0,1\}^n$.

Formal definition:

$$f : \{0,1\}^n \to \{0,1\}^n, \quad y = f(x) := (((A_0\, O_{x_1}\, A_1)\, O_{x_2}\, A_2) \cdots)\, O_{x_n}\, A_n.$$

Here, the integers $A_0, A_1, \ldots, A_n$ are fixed and publicly known. The function f takes as input the binary string x representing the sequence of operations, and outputs the final n-bit result y after applying those operations.

This function is considered "one-way" under the hardness related to 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. Hence, computing the inverse of the given one-way function is computationally equivalent to solving the Add/XOR/XNOR problems. Furthermore, it is also possible to design hash functions from the proposed one-way function 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 5 (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 $O_{x_i} \in \{\oplus, +_{\odot\oplus}\}$, 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 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 law of "associativity", 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. Choosing the combination of addition modulo 2^n, XOR and XNOR makes inversion and meet-in-the-middle attacks impossible, making the exhaustive search a necessity. 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.

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. 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.
4. L. Fortnow. The status of the p versus np problem. *Communications of the ACM*, 52(9):78–86, 2009.
5. L. Fortnow. *The Golden Ticket: P, NP, and the Search for the Impossible*. Princeton University Press, 2013.
6. 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.
7. P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.