一种用于查找维度不超过 9 的二次向量布尔函数轻量级（与、异或）实现的新工具

Marie Bolzer, S´ebastien Duval和 Marine Minier

arXiv:2601.08368v1 [cs.AR] 13 Jan 2026

摘要——**寻找能够实现给定函数的最小电路，是电子学中历史最悠久的问题之一。众所周知，这是一个 NP难问题。尽管如此，仍有许多工具可用于查找实现函数的次优电路。在电子学中，这类工具称为综合器。然而，这些综合器旨在实现非常大的函数（整个电子芯片）。在密码学中，关注的是小型函数，因此有必要为小型函数开发专用的新工具。**

目前已有若干用于实现小型函数的工具。它们在算法方法上各有不同（有些基于 Ullrich 于 2011年提出的深度优先搜索，有些基于 SAT求解器，例如 Stoffelen 于 2016 开发的工具；还有一些专用工具采用子域分解），优化标准也各不相同（有些优化电路大小，有些优化电路深度，还有些针对侧信道防护实现进行优化）。然而，这些工具仅适用于输入少于 5位的函数；对于二次函数，有时可处理 6 位函数；或者仅适用于非常简单的函数。其局限在于计算时间过长。

我们提出了一种新工具[1]，可在与门深度为1的条件下实现最多9位的二次函数，同时将与门数量降至最低。该工具比以往工具更加高效，能够探索比其他工具更大的6 位或更小规模的实现，并且可达到更大的规模，最多支持9位。

索引词——**实现工具、S盒、轻量级密-码学、掩码、乘法复杂度**

一、引言

A. 动机

在任何领域，寻找轻量级实现都十分重要，但与其他领域相比，密码学面临的环境略有不同。尤其是，密码学需要在软件和硬件中都具备高效性，既要适用于服务器，也要适用于小型片上系统。密码学是日常通信中一项开销高昂的负担，因此，数十年来，轻量级密码学一直受到广泛关注。

密码学领域的一个特点是，需要实现的函数广为人知、经过充分研究，并且经过专门设计以降低成本。具体来说，我们知道，大部分电路成本都集中在那些被重复使用多次的小型函数上。这些函数称为 S 盒，通常处理 3 到 8 比特。另一个特点是，默认情况下，密码函数的实现必须能够抵御侧信道攻击，因此二进制异或门（以及任何线性门）的成本远低于与门，这与更传统的情况不同。S 盒是分组密码中唯一的非线性部分，而且会被重复使用多次，因此优化密码函数时，它们是主要关注对象。

具体而言，保护异或等线性门成本低，而保护与等非线性门成本高 [1]。因此，S 盒优化的主要目标是减少与门的数量。次要目标是降低与门深度，它决定了硬件防护实现的延迟 [2]。最后，减少异或门的数量不那么重要，但仍会对成本产生一定影响。

用于优化 S 盒实现的工具分为两类：（i）不断生成电子电路，直到相应的 S 盒具备所需的数学性质；（ii）搜索实现给定 S 盒的最佳电路。这两种方法彼此相关：（ii）是（i）的一个特例，其中所需性质是与给定 S 盒相等。此外，对所有满足所需性质的 S 盒应用（ii）类工具，就能得到一个（i）类工具（利用 S 盒分类会更容易 [3]）。

尽管目前的趋势主要是使用较小的 S 盒（3 比特或 4 比特），但我们认为较大的 S 盒更值得采用。确实，（i）较大的 S 盒可以构建更强的非线性层（由多个 S 盒并行组成）；（ii）使用常见技术（如宽轨策略 [4]）将分组密码的安全性归约到 S 盒时，较大的 S 盒所需的近似更少；（iii）较大的 S 盒可能有助于在成本与安全性之间实现更好的权衡 [5]。采用较大 S 盒的主要缺点实际上是其实现成本较高，因为当前用于复杂 S 盒的工具给出的结果并非最优。另请注意，与这一趋势相反，NIST 最新的对称密码标准 Keccak [6]和 ASCON [7], 使用的是 5 位 S 盒。

a) 先前的工具：已有少数先前的工具，大多仅限于实现 4 比特排列 [8], [9],[10], [11], [12]。一个值得注意的例外是 Stoffe- len 的工具 [13]，它适用于少数 5 比特函数；该工具在 [5] 中得到增强，可处理所有 5 比特函数以及部分 6 比特二次函数。该工具基于黑盒 SAT 求解器，因此改进空间不大。SAT模型

本研究部分得到法国国家研究署资助，项目编号为ANR-22-CE39-0015。
Marie Bolzer、S´ebastien Duval和Marine Minier均任职于法国洛林大学、法国国家科学研究中心、Loria和Inria（电子邮箱：marie.bolzer@loria.fr、sebastien.duval@loria.fr、marine.minier@loria.fr）。[1]该工具随附于通过 CodeOcean发布的电气电子工程师学会论文中，也可在 https://github.com/seduval/implem-quad-sbox获取。本研究已提交至电气电子工程师学会，等待可能的发表。版权可能在未另行通知的情况下转让，届时本版本可能不再可访问。

最近在 [14] 和 [15] 中对其进行了改进，取得了类似的结果，但速度更快，并且能够处理最多 8 比特的非常简单的 S 盒（例如 Keccak χ 映射）。

除了以往用于 S 盒实现的工具外，还有一系列工具用于解决按仿射等价进行分类的问题，这与综合器有相似之处。实际上，我们的工具类似于 Biryukov等人 [3], 提出的仿射等价算法，其归结为求解一个大型线性系统。一个值得注意的工具是 De Meyer 和 Bilgin [16],提出的工具，它专注于二次函数，并结合二次分解以处理更高次数的函数。

B. 关于该问题的高层次考量

S 盒实现的研究至少已有 30 年，但大型 S 盒仍然难以实现。我们建议退一步，重新审视问题本身。

a) 问题定义：给定一个 S 盒，以其查找表（LUT）或代数正规形（ANF）表示。目标是输出一个电路，使其在给定成本度量（等效门数、与门数量、电路深度等）下，以最优方式实现该 S 盒。

b) 问题的难度：这个问题究竟有多难？我们知道它是 NP难 [10], ，但这只针对无限大的 S 盒。对于 4 位 S 盒呢？对于 10 位 S 盒呢？为了估计这个问题的难度，我们做了一个小实验：取一个 S 盒的代数正规形，用笔、纸和咖啡来查找轻量级实现。

我们对 [5] 中的所有 S 盒（即4 位和 5 位 S 盒，以及二次 6位替换盒）进行了实验，目标是优化与门数量和与门深度。令我们惊讶的是，手工得到的实现与工具得到的一样好。更重要的是，我们手工完成每个实现大约需要 4 小时，速度与工具相近（尽管更好的模型 [14], [15]能在几秒内运行）。对于 7 位 S 盒，这种方法就不切实际了。

看来，对于实际规模的问题，这个问题其实没那么难。由此引出了一个问题：计算机是强大的工具，为什么在处理这个问题时效率不够高?

c) 算法方面的考虑：我们认为这是由算法选择造成的。最初的工具使用特设算法 [8]：不断构造随机电路，直到某个电路与 S 盒匹配。在庞大的搜索空间中，这种做法具有很强的随机性。较新的工具使用 SAT求解器，它们依赖良好的通用数学模型，但这样的通用模型很难定义（尽管 [15] 已在这一方向上取得了一些进展）。

d) 纸笔算法：然而，我们认为很自然的纸笔算法却大不相同：从代数正规形出发，寻找出现在多个输出位中的模式，并迭代地对代数正规形进行因式分解。原则上，这解释了为什么计算机无法高效执行这种算法：人脑擅长模式匹配，而计算机由于采用顺序处理方式，并不擅长。

e) 研究方向：基于这些观察，我们认为有几个值得探索的研究方向：（i）寻找更好的问题数学表示，以便更准确地对其建模 [14], [15]，（ii）寻找比随机搜索更适合计算机的新算法（可与（i）结合），以及（iii）尝试利用计算技巧实现模式匹配。

在这项工作中，我们着手研究第三种方法。实现模式匹配时可以想到几种思路：（a）机器学习技术已针对这一目的进行了调优，但成本似乎很高；（b）利用图形处理器等大规模并行计算能力，或许能简化模式匹配；（c）尝试预先计算模式匹配所需的工作。

我们选择采用方法 (c) 研究问题 (iii)，并取得了一些新的、令人鼓舞的成果。不过，未来还有许多值得探索的新方向。此外，我们从二次 S 盒入手，因为对于更高次数的情形，预先计算所有模式本身就是一个难题（我们计划在未来的工作中解决这一问题）。

C. 我们的设定

与 [5], 类似，我们着重减少与门的数量和与门深度，以降低安全实现的成本。事实上，自 20 世纪 90 年代以来，[17], [18], [19], 我们就知道，密码实现必须默认受到保护[2]。通常的保护方式是掩码 [20], [21],，它将每个秘密拆分为 d 个份额，这意味着与门电路的成本非常高，且运行速度会随着与门深度线性变慢 [1], [22]。

最初，要达到安全阶数 t，需要 $d = 2t + 1$个份额 [1], ；但自从 [23], [24] 我们知道，只需$d = t + 1$ 个份额即可实现，而且高阶掩码是必要的（作者研究了最高达 d 的情况）。根据 [1], 我们还知道，理论上，保护一个与门的成本是 $\mathcal{O}$倍（d^2），而保护一个异或门的成本是 $\mathcal{O}$倍（d）。仍有待确定的是，取何值 t 时实现才是安全的。为解决这一问题，人们提出了挑战赛：根据 [25],我们知道，$d = 2$ 可在 5 分钟内通过单条迹线破解。根据 Spook 挑战赛[3] [26] 我们知道，$d = 8$ 可通过 70，$000 \sim 2^{16}$ 条迹线破解，仍远未达到安全要求。根据 [27]我们知道，$d = 32$ 是安全的。粗略来说，攻击 d 个份额似乎需要 2^{2d} 条迹线，这表明需要 $d > 20$，而 $d = 32$ 似乎足以在留有充足余量的情况下实现实用安全性。综合来看，理论上掩蔽一个与门的成本大约是 $32^2 = 1024$ 倍。也有实际测试。在 [28], 中，作者研究了运行时间：他们表明，在 ARM 32 位处理器（小型工业设备中常见的处理器）上，一个经过高度优化、采用 $d = 10$ 的 AES S 盒层需要 300，000 个周期，而线性层需要 315 个周期；整个 AES [29] 需要 $1906.5d^2 + 10972.5d + 7712$。对于 PRESENT， [30], S 盒层需要 30，000 个周期，线性层需要 160 个周期。请注意，这里采用的是 $d = 10$，因此甚至并不安全。在更强大的处理器上， [23] 显示，采用$d = 20$ 的 AES 在英特尔至强处理器上需要 1 分钟，比未受保护的实现慢 769 倍；而 Keccak [6] 需要 3 分钟。

由此看来，轻量级安全实现需要尽可能减少与门数量和与门深度，而异或门数量和异或深度的影响可以忽略不计。

为了尽可能降低与门深度，我们将实现限定为与门深度等于 1 的实现，这对于二次函数而言是最优的。

[2]例如，NIST 轻量级密码学征集要求：“AEAD算法及可选哈希函数算法的实现应便于采取针对各种侧信道攻击的对策。”[3]spook.dev

接下来要减少与门的数量，这正是我们的工具所关注的重点。

我们关注使用二元门（与、异或）的电路[4]。由于我们的目标是实现支持二元门并行性的位切片实现，因此将范围限定为二元门。我们将门限定为与门和异或门，因为这样既有非线性门（与门），也有线性门（异或），而且任何二元非线性门都可以用一个与门实现（也就是说，任何其他门都可以在不额外消耗非线性门的情况下获得）。

我们的工具仅适用于二次 S 盒。原因是，对于次数大于 2 的大型（超过 6 位）S 盒，内存复杂度很高。不过，支持次数高于二次的 S 盒也是一个有价值的目标，我们计划在未来着手解决。但需要注意的是，高效实现二次 S 盒本身已是一项有价值的成果，因为实践中使用了许多二次 S 盒。最近的两项 NIST 标准 Keccak [6] 和 ASCON [7]，就是典型例子，它们都是定义在 5 位上的二次 S 盒。

a) 组织结构：本文安排如下：第二节通过一些小例子对问题进行形式化；第三节介绍我们的算法及其在工具中的嵌入；第四节给出二次情形下获得的结果，并将我们的运行时间与先前的工作进行比较。最后，第五节给出结论。

II. 问题形式化

我们的主要思路是模仿手工操作的过程，将问题分解。计算机本质上不擅长查找模式，因此我们预先计算并存储所有模式。于是，查找 S盒 S 的实现就归结为：针对每个输出位 y_i，查找若干已存储模式的线性组合，使其等于 y_i。

示例 *1*：设 y 是我们想要实现的输出位之一，$y = x_0x_1 \oplus x_1x_2 \oplus x_2x_3$。假设我们存储了模式 $q_0 = x_0x_1 \oplus x_1x_2 = x_1(x_0 \oplus x_2)$, $q_1 = x_1x_2 \oplus x_2x_3 = x_2(x_1 \oplus x_3)$, $q_2 = x_0x_1$, $q_3 = x_1x_2$ 和 $q_4 = x_2x_3$，那么问题就归结为求解一个关于变量 q_i 的线性系统。设 $Q = (q_0, q_1, q_2, q_3, q_4)^T$。我们要寻找一个向量 I，使得 $I \times Q = y.$。一个示例解为 $I = (10001)$，它给出 $y = q_0 \oplus q_4$。

我们遇到的第一个问题是需要存储的模式数量。为减少模式数量，我们将问题分为两部分：线性部分和非线性部分。实际上，线性运算只影响异或门的数量，而这属于次要目标。我们首先优化非线性部分，使与门的数量最少。这一步会输出实现非线性部分所需的线性函数列表。第二步中，我们使用这些线性函数，查找将它们共同实现的方法，以减少异或门的数量。

非线性部分的输入由从线性部分中截断的变量构成：我们从模式 q_i 和输出位 y_i中截去线性部分。

例*2*（使用 $n = 3$进行说明）：考虑 S 盒

[4]请注意，此设置隐式包含非门，因为 NOT $x = x$ 异或 1。

采用代数正规形：

$$y_0 = x_0 \oplus x_1 \oplus x_2 \oplus x_1x_2$$
$$y_1 = x_1 \oplus x_0x_1 \oplus x_0x_2$$
$$y_2 = x_0x_1 \oplus x_2$$

列出从线性部分中截断的所有非线性运算，可得矩阵 Q:

x_0x_1，x_0x_2，x_1x_2，$x_0x_1 \oplus x_0x_2$，例如，可表示为：$x_0($
$x_1 \oplus x_2)x_0x_1 \oplus x_1x_2$，例如，可表示为：$x_1(x_0 \oplus x_2)$
$x_0x_2 \oplus x_1x_2$，例如，可表示为：$x_2(x_0 \oplus x_1)$
$x_0x_1 \oplus x_0x_2 \oplus x_1x_2$，例如，可表示为：$(x_0 \oplus x_2)(x_0 \oplus x$

1)

$$Q = \begin{pmatrix} x_0x_1 \\ x_0x_2 \\ x_1x_2 \\ x_0x_1 \oplus x_0x_2 \\ x_0x_1 \oplus x_1x_2 \\ x_0x_2 \oplus x_1x_2 \\ x_0x_1 \oplus x_0x_2 \oplus x_1x_2 \end{pmatrix}.$$

考虑表示代数正规形的矩阵 Y，去除其线性部分后：我们有

$$Y = \begin{pmatrix} x_1x_2 \\ x_0x_1 \oplus x_0x_2 \\ x_0x_1 \end{pmatrix}.$$

非线性部分的每种实现都是方程 $IQ = Y$ 的一个解，其中 I 的大小为（输出位的数量）$\times$（Q 的大小）。

$$I_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

是给出实现的一个解：

$$q_1 = x_0x_1$$
$$q_2 = x_1(x_0 \oplus x_2)$$
$$y_0 = q_1 \oplus q_2$$
$$q_3 = x_0x_2$$
$$y_1 = q_1 \oplus q_3$$
$$q_4 = x_1x_2$$
$$q_5 = (x_0 \oplus x_2)(x_0 \oplus x_1) \oplus x_0$$
$$y_2 = q_3 \oplus q_4 \oplus q_5$$

$$I_2 = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

是另一个给出实现的解：

$$
\begin{aligned}
q_1 &= x_1 x_2 \\
y_0 &= q_1 \\
q_2 &= x_0(x_1 \oplus x_2) \\
y_1 &= q_2 \\
q_3 &= x_0 x_1 \\
y_2 &= q_3
\end{aligned}
$$

一个关键观察是，与门的数量与 I中的非零列数量相同。因此，查找“最优”实现意味着查找零列最多的解。接下来进行线性步骤，为了得到最终实现，我们需要补上缺失的线性表达式。对于解 I_2，最终实现的一个示例如下：

$$
\begin{aligned}
l_0 &= x_1 \oplus x_2 \\
l_1 &= x_0 \oplus l_0 \\
q_1 &= x_1 x_2 \\
y_0 &= q_1 \oplus l_1 \\
q_2 &= x_0 l_0 \oplus x_1 \\
y_1 &= q_2 \\
q_3 &= x_0 x_1 \\
y_2 &= q_3 \oplus x_2
\end{aligned}
$$

a) 模式数量：在基于手的方法中，我们所说的“模式”实际上包含了所有可通过单个与门获得的布尔函数。尽管截去了线性部分，这类单与门函数的数量仍然很大。我们计算了 3 至 11 比特 S 盒的精确数量，列于表 I 的第 2 列。通过对单与门运算进行线性组合，我们可以实现任何次数小于或等于 2 的函数（二次函数）。请注意，这一数量增长得非常快。总执行时间在很大程度上取决于这个数量。

bits	二次多项式数量（截断）由单个与门获得的数量
3	7
4	35
5	155
6	651
7	2667
8	10 795
9	43 435
10	172 846
11	692 775

表 I：每种大小的模式数量。

III. 工具描述

本节首先在第 III-A 节从算法角度介绍我们的工具，然后在第 III-B 节从实现角度进行介绍。

A. 所用算法

1) 算法的高层描述：我们的工具分两步运行。首先，从所有表达式中截去线性部分，并对非线性部分进行优化，从而得到一个实现输出位非线性部分的二次运算集合。然后，我们根据这个集合生成另一个集合，其中包含实现线性部分所需的线性运算。最后，将所有内容组合起来，得到一个用于 S 盒的电路。该工具的输出是一个实现 S 盒的 C程序。

非线性部分使用两个集合。操作集合：通过恰好 1 个与运算可获得的所有运算的集合（例2中的矩阵 Q ）。Op selec：非线性部分的输出。它起初为空，算法会将实现所需的二次运算填入其中。

我们设计了一个专用工具，用于求解与非线性部分相关的矩阵系统。其思路是逐个实现输出位，但并非彼此独立：处理第 i 个比特时，我们将其布尔函数表示为操作集合中元素的异或，并优先使用已用于实现前面比特的运算。在步骤 i结束时，Op selec包含实现前 i 个输出位所需的运算。

该算法的核心在于如何更新Op selec 。我们要寻找Op selec 和操作集合中元素的异或和，使其等于第 i 个输出位。随后，将该异或和中操作集合 的每个元素添加到更新后的 Op selec 中，并为电路增加 1 个与门。因此，我们要在异或和中尽可能少地添加操作集合 的元素。

本质上，我们逐个输出位地连续求出局部最优解，从而达到全局次优解，这是一种贪心算法。在计算上可行时，我们会穷尽所有局部最优解的连续组合，再穷尽所有次优解，从而找到全局最优解。

这是一种分支定界算法：我们遍历一棵树，第 i 层包含 i 个输出位的所有可能实现[5]。我们根据与门数量设定界限，当 #Op selec $>$ nb and max 时便立即剪枝，其中nb and max 是用户设定的界限。实际上，我们会更早剪枝：通过计算实现剩余输出位所需与门数量的下界（从而得到一个 A^* 启发式方法 [31],[32]）。

线性部分的算法类似，只是搜索不那么穷尽，因为我们寻找的是不那么最优的解。

III-A2节详细介绍非线性算法（其伪代码见算法 1）。III-A3节详细介绍线性算法（其伪代码见算法 2）。

2）非线性部分的算法：*a*）算法说明：我们从以下内容开始：

- 操作集合：通过恰好使用 1 个与门可以得到的所有操作的集合，
- Op selec：“已选定”操作的集合：其中将包含实现所需的操作。随着算法推进，该集合会不断更新，
- nb_and_max：与门数量的上限（用于限定搜索空间）。

[5]因此，树的大小相对于输出位的数量呈指数增长。

在每个阶段 $1 \le i \le n$，我们向 Op selec 添加实现输出位 y_i所需的非线性运算。为了尽量减少添加的运算，我们依次检查是否需要添加：

- 0 与门：对应于这样的情况： y_i 可以仅使用我们已有的操作（即 选定的操作Op selec）来实现，
- 1 与门：对应于这样的情况：我们只需向选定操作集合中添加 1 个操作（来自操作集合）即可实现 y_i，
- 2与门等。

具体而言，考虑Opselec$_{i-1}$ 这个操作集合，用于实现 y_0、.....、 y_{i-1}，其基数为 C。我们最多允许向Op selec添加 $m = \text{nb and max} - C$个操作。然后，我们遍历所有 $0 \le j \le m$ 和 $0 \le k \le C$，并测试由 j 个Op selec$_{i-1}$ 元素和 k 个操作集合元素组成的异或和是否等于 y_i。通过将操作集合 中的 k个元素添加到Op selec$_{i-1}$中，得到更新后的Op selec$_i$ ，因此，添加 y_i的代价是 k 个与门。下文将查找等于 y_i 的异或和的函数记为**test xor**(Op selec y_i， j， k)。

b) 新增操作的多重性：首先请注意，循环返回的操作集合并不唯一。

示例 *3*：考虑代数正规形：

$$y_0 = x_0x_2$$
$$y_1 = x_0x_1 \oplus x_2x_3$$
$$y_2 = x_0x_3 \oplus x_1x_2 \oplus x_2x_3$$

假设我们已经处理了 y_0，并且算法已选定以下操作：

$$\text{Op_selec} = \{q_1 = x_0x_2\}$$

我们无法仅使用Op selec 得到 y_1，也无法只通过添加一个操作得到它。但我们可以通过添加两个操作得到它，并且有几种选择：

- 我们可以将 $q_2 = x_0x_1$ 和 $q_3 = x_2x_3$ 添加到Op selec中。这样，我们得到 $y_1 = q_2 \oplus q_3$，Op selec $= \{q_1, q_2, q_3\}$.
- 我们可以将 $q_4 = x_0x_1 \oplus x_1x_2$和 $q_5 = x_1x_2 \oplus x_2x_3$ 添加到Op selec中。

然后，我们有 $y_2 = q_4 \oplus q_5$, Op selec $= \{q_1, q_4, q_5\}$.

在这个阶段，我们无法知道哪个选项会是最佳选择，因此如果想找到最优解，就需要存储所有选项并逐一测试。在本例中，第二个选项只需添加一个运算（x_0x_3）即可得到 y_2，而第一个方案则做不到，因此最终第二个选项会是最佳选择。

c) 输出位处理顺序的影响：处理输出位的顺序会影响能否得到解。

例 *4*：假设我们以输出位的顺序（y_3 - y_0 - y_2 - y_1）对一个示例 S盒运行算法，并得到具有 7 个非线性运算的实现 q_i：

$$y_3 = q_0 \oplus q_4 \oplus q_5$$
$$y_0 = q_5 \oplus q_6$$
$$y_2 = q_2 \oplus q_3$$
$$y_1 = q_0 \oplus q_1$$

现在，假设比特 y_0 也可以通过执行$q_1 \oplus q_3 \oplus q_4$ 来获得，而我们改用 (y_1 - y_2 - y_0 - y_3) 的顺序。在获得 y_0 的步骤中，由于算法会寻找代价最低的实现，因此它将返回$y_0 = q_1 \oplus q_3 \oplus q_4$；这只需要增加 1 个非线性运算，因为 q_1 和 q_3 此前已经用于实现 y_1 和 y_2，所以可以免费使用。方案 $y_0 = q_5 \oplus q_6$ 需要额外增加 2 个非线性门，因为 q_5 和 q_6 都不能免费使用，因此不会被选中。最终实现将包含 6 个非线性运算：

$$y_1 = q_0 \oplus q_1$$
$$y_2 = q_2 \oplus q_3$$
$$y_0 = q_1 \oplus q_3 \oplus q_4$$
$$y_3 = q_0 \oplus q_4 \oplus q_5$$

因此，如果我们想确保获得最佳实现，就需要尝试输出位的所有可能排列。这代价高昂：对于 $n = 10$，输出位有 10! $= 3628800$种排列，因此算法的运行操作数会增加 3628800 倍。不考虑所有排列，可以更快地获得实现，但我们无法提前准确判断哪些排列对实现成本有利或不利，因此这样做会导致次优结果。

为了降低这一成本，我们选择以初始化的Op selec启动算法。为此，我们寻找“成本最低”的输出位，即所需非线性运算最少的输出位。实现该输出位的不同方式将对应于Op selec的初始化。这样，我们只需测试 $(n-1)!$ 种排列，而不是 $n!$ 种，同时也有利于代码并行化。

d) 算法： 非线性部分的优化总结于算法 1 中。（为便于阅读，test xor的返回值视为唯一。）

3) 线性部分的算法： 第二步的目标是实现我们所需的全部线性表达式。这一步建立在前一步结果的基础上，因为所需线性表达式的集合由以下部分组成：

1) 代数正规形的截断部分；2) 用于获得前一步计算出的非线性运算的线性运算；3) 非线性运算的截断部分。

例*5*： 考虑 $q = x_0x_1 \oplus x_1x_2 \oplus x_0x_2$，我们无法单独得到它：仅用 1 个与门无法实现它。例如，可以计算 ($x_0 \oplus x_1$)($x_1 \oplus x_2$)，得到 $x_0x_1 \oplus x_0x_2 \oplus x_1 \oplus x_1x_2$。因此，我们需要将 x_1 添加为类型3线性表达式，并且需要将 $x_0 \oplus x_1$和 $x_1 \oplus x_2$ 作为类型2线性表达式。

构造好这个集合后，基本原理与上一步相同。我们依次处理各个表达式，并逐步扩充选定操作集合。

a) 算法说明：我们从以下内容开始：

- L：所需的线性表达式集合 l_i ，
- Op selec：一组“已选定”的运算（类似于前一个算法中的运算），初始时包含所有线性单项式x_i。它包含实现所需的线性运算，并会随着算法的推进而不断扩充。

算法 1：非线性部分的优化。

输入：

- $\{y_0, ..., y_{n-1}\}$：S盒的输出位，由线性部分截取得到，
- 操作集合：通过 1 个与门可获得的所有操作组成的固定集合，
- 操作选择：操作集合，初始为空（其大小记为 C），
- 最大与门数：整数，由用户给出的与门数量上限。

输出：

- Op selec：实现S盒所需的操作集合。

```
1  y_k ← 成本最低的输出位
2  对于所有初始化集合 S 以获得 y_k 执行
3      Op selec ← S
4      对于 {y_0, ...., y_{n-1}} \ {y_k} 每个排列执行
5          nb 与 ← nb 与 max- S.size()
6          对于 i 从 0 </style}到 执行
               // 遍历所有输出位
7              for j 从0 到 nb 与 do
                   // 尽可能减少非线性
                       运算
8                  for k 从 0到 C do
                       // 最小化总数
                           操作数量
                           used
9                      S = test xor(Op selec y_i, k, j)
                           // 测试是否存在
                           .....的线性组合
                           j 操作集合的元素
                           以及 k 元素
                           Opselec 等于 y_i
10                     如果 S = ∅ 那么
11                         继续
12                     else
13                         nb 和 ← nb 和 − j
                               // 更新
                               与门数量
14                         Op selec ← Op selec ∪ S
                               // 更新
                               实现
15                         转到 (6)// 返回
                               第 6 行，  移至
                               下一个 y_k
16                 继续// 没有符合 j 的解
                       来自 操作集合 → 尝试 j + 1
17             中断// 没有解 → 尝试
                   下一个排列
18         返回Op selec
```

- 异或门数量上限：所需异或门数量的上界。

具体来说，考虑Opselec$_{i-1}$ 这一组用于实现 $l_0, \ldots, l_{i-1}$ 的运算，其基数为 C。我们最多允许向 Op selec添加 $m =$ nb xor max $- G$ 个运算。然后，我们遍历所有 $0 \leq j \leq m$，并测试是否存在由Op selec 中的 j 个元素构成的异或和，等于 l_i。我们将元素 l_i添加到Op selec中，并添加一个任意的中间变量集合，使 l_i 成为这 j 个元素的和（见下文示例）。总的来说，Op selec$_i$ 比Op selec$_{i-1}$多耗费 j 次异或运算。

b) 非线性表达式的分解重数：请注意，有多个线性运算集合，其与运算结果对应于给定的非线性运算。

例*6*：考虑非线性运算 $q_1 = x_1x_2 \oplus x_1x_3 \oplus x_2x_3$。

用于实现 q_1 的线性运算集合如下：$x_1 \oplus x_2$（类型2）和 $x_1 \oplus x_3$（类型 3），$x_1 \oplus x_2$（类型 2）和 x_2（类型 3），$x_1 \oplus x_3$（类型2）和 x_3（类型3）.

查找最优实现需要测试所有选项，才能确定最终成本最低的选项。但情况数量呈指数级增长，而且线性最优性并非我们的目标，因此我们决定任意选择一个选项[6]。

c) 线性分解中的多重性：类似地，一个线性表达式可以有多种分解方式，也就是说，有多种方式可以增加Op selec。

示例 *7*：考虑线性表达式$l = x_0 \oplus x_1 \oplus x_2 \oplus x_3$，并假设除了集合中选定操作里的线性单项式（$x_0$，$x_1$，$x_2$，$x_3$）之外，我们还有 $l_0 = x_0 \oplus x_2$。以相同成本获得 l 的方式如下：

组合 1。
$l_1 = x_1 \oplus x_3$, $l = l_0 \oplus l_1$Op selec
$= \{x_0, x_1, x_2, x_3,\ x_0 \oplus x_2,\ x_1 \oplus x_3,\ x_0 \oplus x_1 \oplus x_2 \oplus x_3\}$

组合 2。
$l_2 = l_0 \oplus x_1$, $l = l_2 \oplus x_3$Op selec
$= \{x_0, x_1, x_2, x_3,\ x_0 \oplus x_2,\ x_0 \oplus x_2 \oplus x_1,\ x_0 \oplus x_1 \oplus x_2 \oplus x_3\}$

组合 3。
$l_3 = l_0 \oplus x_3$, $l = l_3 \oplus x_1$Op selec
$= \{x_0, x_1, x_2, x_3,\ x_0 \oplus x_2,\ x_0 \oplus x_2 \oplus x_3,\ x_0 \oplus x_1 \oplus x_2 \oplus x_3\}$
为了保证最优性，我们必须测试所有可能性，但出于与上述相同的原因，我们每次只保留找到的第一个解。

d) 线性表达式处理顺序的影响：处理 l_i 的顺序会影响结果。但请注意，l_i 的数量很大。

[6]这种任意选择从不选取涉及 $\oplus 1$（非门）的线性表达式。

示例 *8*：设第一步结束时得到的集合为：

$q_0 = x_0x_1 \oplus x_1x_2 \oplus x_1x_3 = x_1(x_0 \oplus x_2 \oplus x_3)$,

$q_1 = x_1x_3 \oplus x_3x_4 \oplus x_1x_4 = (x_1 \oplus x_3)(x_3 \oplus x_4) \oplus x_3$,

$q_2 = x_0x_3 \oplus x_1x_3 \oplus x_0x_4 \oplus x_1x_4 = (x_0 \oplus x_1)(x_3 \oplus x_4)$。

我们希望实现的输出位为：

$$y_0 = x_0x_1 \oplus x_1x_2 \oplus x_1x_4 \oplus x_3x_4 \oplus x_0 \oplus x_1 \oplus x_2$$
$$= q_0 \oplus q_1 \oplus x_0 \oplus x_1 \oplus x_2,$$
$$y_1 = x_0x_1 \oplus x_0x_3 \oplus x_1x_2 \oplus x_1x_3 \oplus x_0x_4 \oplus x_1 \oplus x_3$$
$$= q_0 \oplus q_1 \oplus q_2 \oplus x_1 \oplus x_4.$$

*L*将包含：

$l_0 = x_0 \oplus x_2 \oplus x_3$ (from q_0)

$l_1 = x_1 \oplus x_3$ (from q_1)

$l_2 = x_3 \oplus x_4$ (from q_1)　　$l_3 = x_0 \oplus x_1$ (from q_2)

$l_4 = x_0 \oplus x_1 \oplus x_2$ (from y_0)

$l_5 = x_1 \oplus x_4$ (from y_1)

请注意，我们没有在 L 中添加 x_3 （来自 q_1），因为它是一个单项式。

此示例针对一个 2 比特 S 盒。对于 8 比特的情况，我们观察到 L 包含约 25 个元素（$25! > 2^{83}$）。我们可以单独处理仅涉及 2 x_i 的 l_i ，因为它们必然通过一次异或得到：我们直接将这些 l_i添加到Op selec 。对于所有其他 l_i，我们必须考虑处理顺序，但它们数量太多，无法测试所有排列，因此只能限定为固定数量的随机排列。

*B. C++*语言算法实现

1) 实现选择：代码使用 C++语言编写，以便精细调整内存和时间。

a) 多项式的表示：我们使用两种不同的多元多项式表示方法。第一种可表示完整的代数正规形（ANF），第二种针对内存空间进行了优化：它表示固定次数（1 或 2）的多项式。

在第一种表示方法中，多元多项式 p 是由 2^n 个布尔值组成的向量，以数组 p 的形式存储，元素类型为 uint32 t。 $p[j]$ 中比特 i 的值对应单项式 x^{32j+i}的存在与否，其中 $x^u = \prod_i (x_i^{u_i})$， u的各比特为 u_i。例如，$5 = (101)_2$, , 因此单项式 x^5表示 $(x_0)^1.(x_1)^0.(x_2)^1 = x_0x_2$。

n 比特函数 S 的代数正规形由 n 行组成，每行分别对应坐标函数S_i 的代数正规形（即第 i 个输出位的多项式）。

示例 *9*（代数正规形表示）：对于 3 比特，有$2^3 = 8$ 个单项式。多项式 $1 \oplus x_0 \oplus x_1x_2 \oplus x_0x_1x_2$编码如下：

x^0	x^1	x^2	x^3	x^4	x^5	x^6	x^7
1	x_0	x_1	x_0x_1	x_2	x_0x_2	x_1x_2	$x_0x_1x_2$
1	1	0	0	0	0	1	1

第二种表示法适用于次数为 1 或 2 的多项式。二次多项式被编码为 32 或 64 比特的正整数。整数中的每个比特表示是否存在

算法 2： 线性部分的优化。

输入：

- $L = \{l_0, ..., l_m\}$：要实现的线性表达式集合，由算法 1 生成，
- Op selec：选定操作的集合 Op selec，初始化为 $\{x_0, ..., x_{n-1}$ 以及类似 $x_i \oplus x_j\}$ 的表达式，这些表达式由算法 1 导出，
- nb xor max：整数，即用户给定的异或门数量上限。

输出：

- 生成线性表达式集合的一系列指令。

1 对于 $\{l_0, ..., l_m\}$ 的一些随机排列，执行 2 nb xor ← nb xor max 3 对于 i 从 0 到 m 执行 // 遍历所有线性表达式 4 对于 j 从 0 到 nb xor 执行 // 尽可能少地添加异或操作s　　S = test xor lin(y_i, j) // 测试是否存在由 j 个元素组成的线性组合等于 l_i6 如果 $S = \emptyset$ 则 7 继续 8 否则 9 nb xor ← nb xor − 1 // 更新可用操作的数量 10 Op selec ← Op selec ∪ S// 更新所有选定操作 11 跳转到 (3) // 返回第 3 行，处理下一个表达式 12 跳出 // 如果没有找到解，则转到下一个排列 13 返回 Op selec

每个二次 单项式是否存在。单项式按字典序排列。

示例*10*（次数为 *2* 的表示）：超过5位时，有 $\binom{5}{2} = 10$ 个二次单项式，因此，多项式可以存储为 10 位，每一位表示一个单项式是否存在。以下是多项式 $x_0x_4 \oplus x_1x_3 \oplus x_1x_4 \oplus x_2x_4$的一种编码：

x_0x_1	x_0x_2	x_0x_3	x_0x_4	x_1x_2	x_1x_3	x_1x_4	x_2x_3	x_2x_4	x_3x_4
0	0	0	1	0	1	1	0	1	0

该多项式表示为整数 $(0001011010)_2 = (90)_{10}$。

线性多项式的编码与此类似，每个比特都对应一个单项式 x_i。

示例 *11*（一次表示）：超过5位， $x_0 \oplus$

$x_2 \oplus x_3$ 具有编码：

x_0	x_1	x_2	x_3	x_4
1	0	1	1	0

因此，它表示为整数 $(10110)_2 = (22)_{10}$。

b) 预计算：为了优化时间复杂度，我们决定预先计算一些所有执行过程都通用的数据（对于给定的 S盒输入大小）：

- 操作集合包含所有可通过 1 个与运算获得的操作，
- 异或映射包含操作集合中元素两两异或所得的所有结果。它可用于测试一个表达式是否等于操作集合中一对元素的异或结果，并给出这一对元素。

示例 *12 (*异或映射*)*：在 7 比特上，由 $(393237)_{10}$ 编码的操作表示 $x_0x_3 \oplus x_0x_4 \oplus x_3x_5 \oplus x_4x_5 \oplus x_5x_6$。它是异或映射中的一个元素，由以下元素对得到：

第一个操作	编码	第二个操作	编码
x_5x_6	1	$(x_0 \oplus x_3)(x_4 \oplus x_5)$	393236
$x_6(x_3 \oplus x_4 \oplus x_5)$	11	$(x_0 \oplus x_5 \oplus x_6)(x_3 \oplus x_4)$	393246
$x_5(x_3 \oplus x_4 \oplus x_6)$	21	$x_0(x_3 \oplus x_4)$	393216
$(x_3 \oplus x_4 \oplus x_5)(x_5 \oplus x_6)$	31	$(x_0 \oplus x_6)(x_3 \oplus x_4)$	393226
x_0x_6	32768	$(x_0 \oplus x_5)(x_3 \oplus x_4 \oplus x_6)$	426005
$x_6(x_0 \oplus x_3 \oplus x_4)$	32778	$(x_0 \oplus x_5 \oplus x_6)\times$ $(x_3 \oplus x_4 \oplus x_6)$	426015
$x_5(x_0 \oplus x_6)$	65537	$(x_0 \oplus x_5)(x_3 \oplus x_4 \oplus x_5)$	458772
$x_5(x_0 \oplus x_3 \oplus x_4 \oplus x_6)$	65557	$x_0(x_3 \oplus x_4 \oplus x_5)$	458752
$x_0(x_5 \oplus x_6)$	98304	$(x_0 \oplus x_5)\times$ $(x_3 \oplus x_4 \oplus x_5 \oplus x_6)$	491541
$(x_5 \oplus x_6)(x_0 \oplus x_3 \oplus x_4)$	98334	$(x_0 \oplus x_6)\times$ $(x_3 \oplus x_4 \oplus x_5 \oplus x_6)$	491531

与 393237 关联的异或映射行将包含：
$\{1, 393236\}$ $\{11, 393246\}$ $\{21, 393216\}$ $\{31, 393226\}$ $\{32768, 426005\}$
$\{32778, 426015\}\{65537, 458772\}\{65557, 458752\}$ $\{98304, 491541\}\{98334, 491531\}$

存储此映射有助于加快testxor 函数的运行速度。实际上，要测试操作 x 是否可以通过将操作集合中的两个操作相加得到，我们可以使用算法4，而不是算法3。

算法 3： 不使用testxor 的示例
异或映射。

```
对于 i 从 0 到 操作集合大小 执行
    元素 ← x ⊕操作集合[i]
    索引 ← 操作集合.查找(元素)
    解答 ← 解答 ⋃ {操作集合[i], 操作集合[索引]}
返回解答
```

算法4： 示例：test xor使用异或映射。

```
索引 ←异或映射.find(x) for i 从 0 到 异或映射[索引].大小 执行 解 ← 解 ⋃ {异或映射[索引][i][0], 异或映射[索引][i][1]} 返回解
```

遗憾的是，我们无法对操作集合存储 3 个元素的所有异或结果，因为三元组的数量实在太大。表 II 给出了操作集合和异或映射的大小详情，以及操作集合三元组的假设映射大小（称为三元组）。当操作集合和异或映射覆盖所有二次函数时，不给出三元组的值。

Bits	#操作集合	#异或映射 键的数量	#异或映射 大小（字节）
4	35	35	1 KB
5	155	868	17 KB
6	651	18 228	356 KB
7	2667	330 708	6 MB
8	10 795	5 622 036	107 MB
9	43 435	92 672 916	2 GB
10	172 846	>1 505 005 887	> 28GB
11	692 775		

Bits	#三元组 键的数量	#三元组 大小（字节）
4	-	-
5	-	-
6	13 328	43 MB
7	1 761 110	6 GB
8		
9		
10		
11		

表 II：预计算数据大小详情。

2) 数据结构：

a) 操作集合和 异或映射 运行期间：操作集合 和 异或映射规模庞大，但已预先计算并固定。因此，在工具运行期间，它们以静态数组的形式存储。这些是最重要的数据结构：它们规模最大，访问也最频繁。能够预先计算并以静态方式存储它们，是我们最有力的数据结构优化手段。

b) 数据结构问题的形式化：静态数组非常适合优化运行期间的大部分数据，而且不需要深入了解上下文，但这些数据必须先经过预计算。在预计算期间，我们无法预测操作集合 和异或映射的内容，甚至无法预测它们的大小，因此数组并不可行。我们必须使用其他数据结构，而这正是我们的工具仅适用于少于 *10* 比特的 *S* 盒的限制所在。接下来的段落将说明这个问题及可能的解决方案，重点讨论异或映射，它是需要计算的最大数据结构。

c) 预计算情况：我们考虑 4 种情况。情况1是理想情况：为所有 n 比特多项式构建集合和映射（以实现任意 n 比特S盒）。情况2是为所有二次 n 比特多项式（不含线性部分）构建集合和映射；情况3适用于通过 *1* 次与可获得的二次 n 比特多项式；情况4适用于所有二次单项式。

实际运行时，我们使用情况3；预计算时使用它的一个变体。不过，其他选项有助于理解为什么这一选择很重要。

d) 我们想存储什么？：我们想知道哪些多项式的组合可以生成哪些多项式。这可以用一个图来表示，其中节点是多项式，边表示“节点 x 可以通过一次异或生成节点 y ”这一关系。不过，这种表示方式已经存在一个问题：二元门（如异或）是三个变量（两个输入和一个输出）之间的关系，而不是像图中的一条边那样只涉及两个变量。有一种（不太理想的）方法可以在图中存储这种关系：复制所有边。为此 $x \oplus x' = y$，我们存储两条带标签的边（$x \xrightarrow{x'} y$）和（$x' \xrightarrow{x} y$）。

e) 存储图：为了了解每种情况下数据结构的大小，我们来做一些基本的组合计数。设 n 为 S盒的输入大小，N_i 为情况 i需要存储的多项式数量。N_i 的取值汇总于表III。

	N_1	N_2	N_3	N_4
			参见表 II ，	
公式	2^{2^n}	$2^{n(n-1)/2}$	“#异或映射”	$n(n-1)/2$
$n=7$	2^{128}	2^{21}	330 708	21
$n=10$	2^{1024}	2^{45}	>1 505 005 887	45

表III：多项式数量。

由于我们关注的是以比特为单位的实际数据大小，补充说明一下，存储情况 i 的 1 个多项式需要 log_2（N_i）比特（实际存储时会向上取整到 32 的倍数）。

f) 以邻接矩阵存储：图的一种经典存储方式是使用邻接矩阵。根据上述带有重复边的图表示，矩阵的一个单元格对应一条边，该单元格存储边的标签（即一个输入值，因此是一个多项式/单项式）。

那么矩阵中有 N_i^2 个单元格，每个单元格占用 log_2（N_i）比特，因此整个矩阵占用 log_2（N_i） $\times N_i^2$ 比特。

如表 IV 所示，只有情况4的最小预计算能够在 10 比特上实现。情况2最多可实现到 $n=7$，此时矩阵可存储为 uint32_t，占用约 2^{47} 比特内存，相当于约 16 TB。

g) 以邻接向量存储：存储图的另一种经典方式是使用多个邻接向量：对于图中的每个节点（即每个多项式），我们存储一个链表，其中列出它在图中的后继节点。形式上，考虑 x 一个多项式，我们构建一个链表，列出它的后继节点 y （即通过一个异或门 x 可到达的多项式）。然后，每个向量单元存储一个对 $\{y, next\}$，其中 $next$ 用于实现链表（即列表中下一个元素的地址）。

假设 64 位地址大小，每个单元格在内存中占用 $S = log_2(N_i) + 64$ 比特。请注意，实际上，对于较大的 n，$log_2(N_i) = 64$ （较小的 n则为 32），因此 $S \simeq 2^7$。尤其要注意，存储 $next$ 会使内存消耗翻倍。

只有在邻接矩阵足够稀疏时，才有必要仅用向量存储。因此，平均而言，一个多项式 x 经过一个异或门可以到达多少个多项式 y？

有趣的是，在实际中，对于情况3（可通过 1 个与门得到的二次多项式），我们观察到，任何 x 可到达的 ys 数量都是常数 10，适用于任意 n。我们没有进一步深入理论证明这一点，因为这只是我们工作中的一个细节，但我们猜想这个数量始终等于 10。我们预计可以利用编码理论证明这一点，因为这个问题可以转化为二次 Reed-Solomon 码的问题，并在所有一次多项式上进行缩短。

根据这一猜想，我们将在研究中加入这个常数 $M = 10$ ，并推断该矩阵非常稀疏。请注意，这只是对情况3的良好估计，并且将我们的工具限制在 9 比特 S 盒。

内存消耗总结见表 IV。总体而言，与使用矩阵相比，我们使用的内存少得多，而数据访问耗时增加很少。对于 $n = 10$，仍然可行：$2^{41} \simeq$ 250GB 可用于由 1 个与运算得到的二次多项式。

	矩阵				向量			
	N_1	N_2	N_3	N_4	N_1	N_2	N_3	N_4
$n=7$	2^{263}	2^{47}	2^{41}	2^{11}	2^{138}	2^{31}	2^{28}	2^{15}
$n=10$	2^{2058}	2^{96}	2^{76}	2^{14}	2^{1034}	2^{55}	2^{41}	2^{17}

表 I：图存储大小（数量级）。

请注意，实际上我们知道向量的大小为 M 个单元格，其中 $M = 10$ 为常数，因此我们使用由 M 个单元格组成的数组来实例化这些向量（并保持数组有序）。这样可以缩短访问时间：使用向量时，平均需要访问 $M/2$ 个单元格才能找到一个元素；使用数组时，则大约需要访问 $log_2(M)/2$ 个单元格。更重要的是，向量的内存不连续，因此平均而言，每次访问 $M/2$ 个单元格都会导致缓存未命中；而数组在内存中连续存储，因此我们进行 $log_2(M)/2$ 次访问，其中只有第一次会发生缓存未命中。总的来说，在我们的邻接向量中访问元素只需 1 次 RAM 访问（以及少量缓存访问）：这与邻接矩阵的情况类似，但不会浪费内存。

此外，与使用向量相比，使用数组时，每个单元格只包含 $\{y\}$，而不包含下一个向量元素的地址，因此内存大小大约减少一半。

h) 从理论到实践：然而，在预计算期间，我们不能直接使用情况3，因为我们事先不知道映射的内容或大小（我们还没有找到这些大小的理论估计）。我们可以使用情况2，这样最多可以处理 $n = 9$ （需要约 8TB）。

另一种做法是不使用静态数组来存储数据，而是使用动态数据结构，在首次预计算这些元素时将其插入其中。在这种情况下，我们需要 (1) 确保元素唯一，以及 (2) 对元素排序，以便高效地执行“查找”和“插入”操作。总的来说，我们需要有序集合（或有序映射，也就是由若干对构成的有序集合）。

然而，实际使用 [有序] 集合时存在一个问题：我们找不到任何高效的实现。事实上，标准 C++ 集合在内存使用方面效率极低：

它以二叉搜索树的形式存储，因此每个节点都包含一个 32 位或 64 位多项式，以及两个 64 位地址，分别指向左子节点和右子节点。这意味着，存储一个 32 位多项式需要 160 比特，存储一个 64 位多项式则需要 192 比特，内存开销增加了 *3* 到 *5* 倍。

唯一的替代方案是自行重写一个高效的有序集合实现，但这本身就是一项艰巨的工作，而且并非本文的主要议题。因此，我们继续使用 C++ set，但内存占用会增加 3 倍，这带来了很大的限制（对于 $n = 10$，内存下限从 250GB 增至 750GB）。

i) 总结：我们使用情况3在运行期间存储操作集合和异或映射，并将其保存为静态数组，总内存复杂度为 $N_3 \times M \times S$ 比特。然而，在预计算期间，我们受限于必须使用有序集合，这会使内存用量至少增加 3 倍。

3) 并行化：代码中的若干部分使用 OpenMP 进行了并行化。

最初的想法是，在遍历输出位的所有排列时进行并行化，主要是出于实现方面的考虑。处理每个排列的方式完全相同，因此这种方案易于实现，也能减少线程之间的交互。线性部分采用了这一方案。并行化在算法 2 的第 1 行进行。非线性部分则采用不同的设置。原因在于，当穷举代价高昂时，尝试搜索搜索空间中相距较远的区域，以覆盖更多可能不同的实现，会更有意义。在初始化Op selec后遍历这些排列。因此，在此处进行并行化并以不同集合启动搜索，可以让我们找到不同的局部最优解，从而使最终解更加多样。这种配置似乎比另一种方案对获得解的影响更大。并行化在算法 1 的第 2 行进行。

示例 *13*：令 $y_i = x_0x_3 \oplus x_0x_4 \oplus x_3x_5 \oplus x_4x_5 \oplus x_5x_6 = (393237)_{10}$ 为一个待实现的输出位。每个线程都将从示例 12 中详述的某个二元组开始搜索，并将 Op selec 初始化为该二元组。

4) 非穷举搜索的搜索空间限制：

a) 限制因素：从 8 比特起，存在无法仅通过异或三个单与门布尔函数得到的二次布尔函数。因此，搜索必须使用四元组进行初始化，但要得到某个给定运算，所需的不同四元组数量约为 350,000。这一数量太大，无法穷举，因此不再可能进行穷举搜索。

此外，请记住，我们需要穷举输出位的所有可能排列，因为处理顺序很重要。然而，测试一个排列所需的时间会随着比特数以及初始化搜索的元组大小而增加。表 V 给出了测试一个排列的平均处理时间详情。正如我们所见，从 8 比特起，也无法穷举输出位的所有排列。

位数	平均处理时间 单次时间 排列已初始化 含有一个三元组	平均处理时间 单次时间 排列已初始化 含有一个四元组
7	1 秒	-
8	10 小时	18 小时
9	30 小时	$>$ 60小时

表 V：置换的平均处理时间详情。

b) 搜索空间界：由于从 8 比特起无法进行穷举搜索，我们转而考虑搜索空间中的一个随机有限部分。

我们选择只使用固定数量的随机四元组。NB QUAD MAX的默认值为 20,000。

我们只考虑固定数量的随机输出比特排列。对于每种配置，NB PERM MAX 由程序头文件中的宏以不同方式定义。由于我们的实验设置仅允许处理最多 9位的 S 盒，因此对于大小大于 9位的情况，我们未定义任何限制。表 VI 中给出的值为默认值。

位数	NB PERM MAX 对于一个二元组	对于一个三元组	对于一个四元组
8	50	10	5
9	50	5	2

表 VI：NB PERM MAX的默认值。

c) 关于次优结果：从 8 比特起，我们便无法再保证结果的最优性，因为搜索不再是穷举的。第四节中给出的 8 比特和 9 位实现是我们找到的最佳结果，但可能还存在更好的实现。

5) 如何使用该工具：我们的工具以给定 S盒的查找表和最大运行时间为输入；如果找到解，则返回实现该 S盒的 C程序。

还可以添加几个其他参数：

--solmax：期望解的数量；--andmax：与门数量的上限；--precomputation_files 0：若不想使用预计算文件。

默认情况下，程序将使用预计算文件，并返回其能查找到的所有最优解。由于这些文件较大，代码中不包含它们，但提供了一个 Makefile 来计算这些文件。文件大小见表 VIII。请注意，计算机的 RAM 容量必须大于这些文件的大小，才能运行该工具。

			实现（与 - 异或）运行时间（以秒为单位）						
S盒	δ	$\mathcal{L}$ Bij	[5]	[14]	[15]	Ours	[14]	[15]	Ours
χ_5	8	16 是	-	5 -	5 - 5	5 - 10	1.2	19.81	0.05
ASCON	8	16 是	-	5 -	5 -	5 - 15	1.4	26.48	0.1
SYCON	8	16 是	-	-	5 - 17	5 - 15		26.72	0.1
FIDES	2	8 是	7 - 29	7 - 7 - 27		7 - 20	3.6	140.31	0.2
X^3	2	8 是	7 - 29	-	-	7 - 19			< 1
X^5	2	8 是	7 - 26	-	-	7 - 21			< 1
S_51	4	16 是	-	-	-	5 - 20			< 1
S_52	4	16 是	-	-	-	5 - 24			< 1
S_53	4	16 是	-	-	-	5 - 24			< 1
S_54	8	16 是	-	-	-	6 - 20			< 1

表 VII：5 位 S 盒的结果。

位数	Size
4	20 KB
5	88 KB
6	2216 KB

位数	Size
7	51 MB
8	1.1 GB
9	22 GB

表 VIII：预计算文件的大小。

示例 *14*：设 S 是一个由以下查找表（小端序）定义的 S盒：

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F				
0	1	4	5	3	2	7	6	C	D	8	9	F	E	B	A				

如果我们希望最多获得 3 个使用 4 个与门实现 S 的方案，等待时间不超过 10 分钟，且不使用预计算文件，那么执行程序的命令行如下：

./implem --solmax 3--andmax 4 --lut 0,1,4,5,3,2,7,6,12,13,8,9,15,14,11,10 --timemax 10 --precomputed_files 0

IV. 结果

我们给出了一些针对最多 9 比特 S 盒的实验结果。我们选择了涵盖不同安全级别的 S 盒。值得注意的是，我们使用了幂函数及幂函数的异或，以获得良好的密码学性质，因为这些函数已得到充分研究。我们将自己的实现与 [5], [14] 和 [15]中给出的实现进行了比较。我们还将其与使用 ABC [7] 工具获得的实现进行了比较，以便与非密码学方法进行对比。条件允许时，我们会报告执行时间。请注意，工具本身的运行时间是次要目标（因为这些算法每秒会被调用数十亿次）。运行时间主要可用来判断该方法能否合理扩展，因为对于较大的输入规模，执行时间是主要限制因素之一。

a) 实验设置：该工具在两种不同类型的机器上使用：

- 配置 1：64位 x86，搭载 2.3GHz Intel Core i7，配备 12 个核心和 32GB 内存。
- 配置2：64位 x86，搭载 AMD EPYC 处理器，频率为 2.8 GHz ，配备64个核心和128GB内存。

我们选择了两种配置，分别代表普通个人电脑（配置1）所能实现的条件，以及使用更昂贵设备（配置 2）所能实现的条件。

需要注意的是，我们的硬件环境可能与其他论文所采用的条件不同。在 [5],中未说明硬件环境。在 [15], 中，实验使用了一台主频为2.1 GHz、配备8个核心的Intel Xeon CPU。在 [14], 中，实验使用了一台主频为3.4 GHz的Intel Core i7，以及一台主频为2.9GHz的AMD 3990X CPU，核心数量未说明。

b) 内存使用量：在配置2（128 GB 内存）下，对于最多 8 比特的情况，所用内存不到 1%。对于 9位，程序需要约 14% 的可用内存。

c) 结果表：表 VII、IX、X、XI、XII 详细列出了若干 S 盒的结果，包括其差分均匀度（δ）、线性度（$\mathcal{L}$）、是否为双射以及工具运行时间。对于大小为 5 和 6 的情况，由于两种配置下的运行时间非常相近，因此仅给出配置1下的结果。未指定与门深度，因为它始终为 1。

所有对应的查找表和实现均可在补充材料中找到。χ_i 个$5 \leq i \leq 9$ 比特的 S 盒对应于例如 Keccak 中使用的变换，其定义见 [33]。

请注意，对于 χ_i，我们的结果使用的异或数量是 [33] 的两倍。这实际上源于线性表达式的选择：我们选择不考虑非门（见脚注 6）。

A. 超过*5*位（表 *VII*）

S 盒 S 51 至 S 54 是随机生成的。幂映射使用域多项式 $X^5 + X^2 + 1$。

B. 超过 *6* 位（表 *IX*）

S 盒 Q2256 至 Q2263 是具有最佳差分均匀度和线性度的排列的 8 个扩展仿射等价类的代表 [16]。S 盒 BL ECCZ 1 至 BL ECCZ 14 是二次 APN 函数的 13 个 CCZ 等价类的代表 [34]。幂映射使用域多项式 $X^6 + X^4 + X^2 + X + 1$。S 盒 S 61 和 S 62 是随机生成的。

[7] https://github.com/berkeley-abc/abc

S盒	δ	$\mathcal{L}$	Bij	实现（与 - 异或）运行时间 [5]	[15]	Ours	（分钟）
χ_6	16	32	否	-	6 -	6 - 12	< 1
Q2256	4	16	是	8 -	35	-	8 - 26	< 1
Q2257	4	16	是	8 -	41	-	8 - 27	< 1
Q2258	4	16	是	8 -	38	-	8 - 25	< 1
Q2259	4	16	是	-	-	8 - 28	< 1
Q2260	4	16	是	8 -	40	-	8 - 29	< 1
Q2261	4	16	是	-	-	8 - 26	< 1
Q2262	4	16	是	-	-	8 - 27	< 1
Q2263	4	16	是	8 -	38	-	8 - 30	< 1
X^3	2	16	否	9 -	43	-	8 - 37	< 1
X^9	8	64	否	-	-	6 - 28	< 1
BL_ECCZ 1	2	16	否	-	-	9 - 23	< 1
BL_ECCZ 8	2	16	否	-	-	8 - 34	< 1
BL_ECCZ 11	2	16	否	-	-	8 - 29	< 1
BL_ECCZ 14	2	16	否	-	-	9 - 23	< 1
S_61	16	32	否	-	-	6 - 17	< 1
S_62	8	32	否	-	-	7 - 24	< 1

表 IX：6位替换盒的结果。

S盒	δ	$\mathcal{L}$	Bij	实现（与 - 异或）我们的运行时间（以**分钟**为单位） [5]	[15]	Ours	配置 1	配置2
χ_7	32	64	是		7 -	7 - 14	< 1	< 1
X^3	2	16	是	15 - 79	-	11 - 45		3
X^5	2	16	是	-	-	11 - 56	4	1
X^9	2	16	是	-	-	11 - 53	3	2
X^{17}	2	16	是	-	-	11 - 54		1
X^{33}	2	16	是	-	-	11 - 58		2
X^{65}	2	16	是	-	-	11 - 52		2
$X^3 \oplus X^5$	4	32	否	-	-	10 - 57		68
S_71	8	32	否	-	-	10 - 51		2
S_72	8	64	否	-	-	10 - 48	120	10
S_73	4	32	否	-	-	11 - 50		2
S_74	32	64	否	-	-	7 - 30	< 1	< 1
S_75	16	64	否	-	-	8 - 31	< 1	< 1
S_76	8	32	否	-	-	10 - 43	< 1	< 1

表 X：7 位 S 盒的结果。

C. 超过 *7* 比特（表 *X*）

S 盒 S 71 至 S 76 是随机生成的。幂映射使用域多项式 $X^7 + X + 1$。

D. 超过 *8* 比特（表 *XI*）

S盒 S A81 是约 13,000 个 CCZ 等价的二次 APN 函数类中的一个代表，这些函数类已在 [35] 中分类。S 盒 S 81 至 S 83 是随机生成的。 X^3 truncated 表示在 9位时截断第 8 比特（最高有效位）。幂映射使用域多项式 $X^8 + X^4 + X^3 + X^2 + 1$。

这些结果是我们得到的最佳结果，但可能还存在更好的实现。我们尝试在将运行时间限制在几天的情况下减少一个与门，但未能成功。

E. 超过 *9* 位（表 *XII*）

幂映射使用域多项式 $X^9 + X^4 + 1$。

S盒	δ	$\mathcal{L}$	Bij	实现 我们的运行时间（与 - 异或）（以**分钟**计） [15]	Ours	配置2
χ_8	64	128	否	8 -	8 - 16	< 1
X^3	2	32	No	-	14 - 77	115
X^5	4	64	No	-	12 - 71	3
X^9	2	32	No	-	14 - 75	900
X^{17}	16	256	否	-	10 - 48	1
X^{33}	2	32	No	-	14 - 76	120
X^{65}	4	64	No	-	12 - 68	1
X^{129}	2	32	No	-	14 - 81	660
S A81	2	32	否	-	14 - 68	120
S_81	8	32	No	-	14 - 70	5
S_82	8	64	No	-	11 - 48	2
S_83	8	64	No	-	10 - 46	8
X^3已截断 4		32	No	-	14 - 70	80

表 XI：8 位 S 盒的结果。

S盒	δ	$\mathcal{L}$	Bij	我们的实现 我们的运行时间（与 - 异或）	配置2（单位：**分钟**）
χ_9	128	256	是	9 - 18	< 1
X^3	2	32	是	19 - 94	5
X^5	2	32	是	20 - 109	4
X^9	8	64	是	18 - 100	5
X^{65}	8	64	Yes	18 - 97	12
$X^3 \oplus X^5$	4	64	No	18 - 100	390
$X^3 \oplus X^9$	8	64	No	18 - 93	480
$X^3 \oplus X^5 \oplus X^{17}$	8	256	否	18 - 96	4

表 XII：9 比特 S 盒的结果。

这些结果是我们获得的最佳结果，但可能存在更好的实现。我们尝试在将运行时间限制为几天的情况下，减少一个与，但未能成功。

F. 使用 *ABC* 工具获得的结果

我们在一些 S 盒上测试了 ABC 工具。我们使用了 dc2 指令，然后依次执行 balance、rewrite、refactor 和 resub，并重复该流程 5 次以提升结果。相应结果见表 XIII。这表明通用实现工具并不适用于密码学用途，也说明了为什么需要更专用的工具。

S盒	实现（与）运行时间（秒） ABC	Ours	ABC	Ours
ASCON	36	5	< 1	0.05
X^3 超过5位	61	7	< 1	< 1
X^3 超过 6 位	88	8	< 1	< 1
X^3 超过7比特	268	11	< 1	180
X^3 超过8比特	454	14	< 1	6900
X^3 超过9比特	676	19	< 1	300

表 XIII：使用 ABC 工具得到的结果。

V. 结论

我们提出了一种新工具，用于计算二次 S 盒的低 AND 实现。我们的算法基于预先计算 1-AND 门二次布尔函数并求解大型线性系统，同时谨慎控制搜索空间大小和内存存储。我们的工具在运行时间和找到的实现方面都优于现有的所有 AND 优化工具。这些出色的运行时间表明，要扩展到更大的规模还需要投入多少工作。具体而言，我们预计规模 10 已经可以实现，但需要比配置 2 更好的设置。对于其他指标（例如 [13], [15] 中的指标），如门等效复杂度或位切片门复杂度，我们的工具并非最优，也不易调整。

我们计划将这些结果扩展到更高次的 S 盒，仍然仿照手工操作的方式，但采用更具启发性的方法。附录 A 介绍了为何本文所述方法不易适用于更高次情形，以及如何应对这一问题的详细内容。

致谢

本研究部分得到了法国国家研究机构项目 ANR-22-CE39-0015 的资助。

参考文献

[1] Y. Ishai、A. Sahai 和 D. A. Wagner，“私有电路：保护硬件免受探测攻击”，载于 *CRYPTO 2003，2003* 年 *8* 月 *17—21* 日，美国加利福尼亚州圣巴巴拉，论文集，计算机科学讲义系列，第 2729 卷，2003 年，第 463—481 页。[2] B. Bilgin，“阈值实现：作为高阶差分功耗分析的对策”，博士学位论文，比利时鲁汶大学与荷兰特温特大学，2015 年。[3] A. Biryukov、C. D. Canni`ere、A. Braeken 和 B. Preneel，“密码分析工具箱：线性与仿射等价算法”，载于 密码学进展——*EUROCRYPT 2003*，密码技术理论与应用国际会议，*2003* 年 *5* 月 *4—8* 日，波兰华沙，论文集，计算机科学讲义系列，E. Biham 编，第 2656 卷，2003 年，第 33—50 页。[4] J. Daemen 和 V. Rijmen，“宽轨迹设计策略”，载于 密码学与编码，第八届 *IMA* 国际会议，*2001* 年 *12* 月 *17—19* 日，英国赛伦塞斯特，论文集，计算机科学讲义系列，第 2260 卷，2001 年，第 222—238 页。[5] B. Bilgin、L. D. Meyer、S. Duval、I. Levi 和 F. Standaert，“低与门深度与高效逆变换：低延迟掩码的 S 盒指南”，*IACR* 对称密码学汇刊，第 2020 卷，第 1 期，第 144—184 页，2020 年。 [在线]。可访问：https://doi.org/10.13154/tosc.v2020.i1. 144-184[6] G. Bertoni、J. Daemen、M. Peeters 和 G. V. Assche，“从 Panama 经 radiogat´un 到 Keccak 的道路”，载于 对称密码学，*2009* 年 *1* 月 *11—16* 日，达格施图尔研讨会论文集系列，H. Handschuh、S. Lucks、B. Preneel 和 P. Rogaway 编，第 09031 卷，2009 年。[7] C. Dobraunig、M. Eichlseder、F. Mendel 和 M. Schl¨affer，“Ascon v1.2：轻量级认证加密与哈希”，密码学杂志，第 34 卷，第 3 期，第 33 页，2021。 [在线]。可访问：https://doi.org/10.1007/ s00145-021-09398-9[8] M. Ullrich、C. De Canniere、S. Indesteege、 ¨O. K¨uç¨uk、N. Mouha 和 B. Preneel，“查找 4× 4 比特 S 盒的最优位切片实现”，载于 *SKEW 2011* 对称密钥加密研讨会，丹麦哥本哈根，2011 年，第 16—17 页。[9] B. Gladman，“为 Serpent S 盒及其逆变换查找高效的布尔函数分解”，访问日期：2025 年 6 月。 [在线]。可访问：http://brg.a2hosted.com//oldsite/cryptography technology/serpent/anal1.cpp [10] J. Boyar、P. Matthews 和 R. Peralta，“应用于密码学的逻辑最小化技术”，密码学杂志，第 26， 2， 280 312, 卷，第 2013 [ ] 10 1007 00145 012 9124 7 26， 2， 280 312, 期，第 – 页。2013 [ ] 10 1007 00145 012 9124 7 在线。可访问：https://doi.org/ 。 /s - - - –

[11] J. Jean、T. Peyrin、S. M. Sim 和 J. Tourteaux，《轻量级构件实现的优化》，IACR对称密码学汇刊，第2017卷第4期，第130–168页，2017年。 [在线]。获取地址：https://doi.org/ 10 13154 2017 4 130 168 。/tosc.v .i . - Z. Bao、J. Guo、S. Ling 和 Y. Sasaki，《PEIGEN——用于评估、实现和生成S盒的平台》，IACR对称密码学汇刊，第2019卷第1期，第330–394页，2019年。 [在线]。获取地址：https://doi.org/ 10 13154 2019 1 330 394 。/ tosc.v .i . - K. Stoffelen，《使用SAT求解器针对多项标准优化S盒实现》，载于《快速软件加密——FSE 2016》，德国波鸿，2016年3月20日至23日，计算机科学讲义丛书第9783卷，2016年，第140–160页。[14] F. Zhang 和 Z. Huang，《使用SAT求解器优化S盒实现：再探》，IACR密码学电子印本档案，第1721页，2023年。 [ ] 2023 1721 在线。获取地址：https://eprint.iacr.org/ / J. Feng、Y. Wei、F. Zhang、E. Pasalic 和 Y. Zhou，《通过SAT求解器实现轻量级密码S盒的新型优化方案》，电气电子工程师学会电路与系统汇刊第一部分：常规论文，第 71， 1， 334 347, 期，第–页。在线。2024 [ ] 10 1109 2023 获取地址：https://doi.org/ 。/TCSI. . 3325559

[16] L. De Meyer 和 B. Bilgin，《平衡二次函数的分类》，IACR密码学电子印本档案，第2018卷，第113页，2018年。 [ ] 2018 113 在线。获取地址：http://eprint.iacr.org/ / S. Chari、C. S. Jutla、J. R. Rao 和 P. Rohatgi，《探索抵御功耗分析攻击的可靠方法》，载于CRYPTO ’99，美国加利福尼亚州圣巴巴拉，1999年8月15日至19日，论文集，计算机科学讲义丛书第1666卷，1999年，第398–412页。[18] L. Goubin 和 J. Patarin，《DES与差分功耗分析（“复制”方法）》，载于《密码硬件与嵌入式系统》，第一届国际研讨会，CHES’99，美国马萨诸塞州伍斯特，1999年8月12日至13日，论文集，计算机科学讲义丛书，C。K. Koc. 和 C. Paar 编，第1717卷，1999年，第158–172页。[19] S. Micali 和 L. Reyzin，《物理可观测密码学（扩展摘要）》，载于《密码学理论》，第一届密码学理论会议，TCC 2004，美国马萨诸塞州剑桥，2004年2月19日至21日，论文集，计算机科学讲义丛书，M. Naor 编，第2951卷，2004年，第278–296页。[20] A. Shamir，《如何共享秘密》，《美国计算机协会通讯》，第22卷第11期，第612–613页，1979年。 [在线]。获取地址：https://doi.org/10.1145/359168。359176[21] G. R. Blakley，《保护密码密钥》，载于1979年国际需求知识管理研讨会，MARK 1979，美国纽约，1979年6月4日至7日，1979年，第313–318页。 [在线]。获取地址：https://doi.org/ 10 1109 1979 8817296 。/MARK. . D. Bozilov、M. Knezevic 和 V. Nikov，《优化阈值实现：为低能耗、低延迟应用保护密码加速器》，《密码工程期刊》，第 12， 1， 15 51, 期，第–页。在线。2022 [ ] 10 1007 13389 021 00276 获取地址：https://doi.org/ 。/s - - - 5

[23] ¨ ´ G. Barthe、S. Belaıd、F. Dupressoir、P. Fouque、B. Gregoire、P. Strub 和 R. Zucchini，《强非干扰与类型引导的高阶掩码》，载于2016年ACM SIGSAC计算机与通信安全会议，奥地利维也纳，2016年10月24日至28日，2016年，第116–129页。 [在线]。获取地址：https://doi.org/ 10 1145 2976749 2978427 。/ . D. Goudarzi、T. Prest、M. Rivain 和 D. Vergnaud，《通过输入输出分离实现探测安全性，并重新审视拟线性掩码》，IACR密码硬件与嵌入式系统汇刊，第2021卷第3期，第599–640页，2021年。 [在线]。获取地址：https://doi.org/10.46586/tches. v2021.i3.599-640[25] O. Bronchain、G. Cassiers 和 F. Standaert，《给我5分钟：用单条侧信道轨迹攻击ASCAD》，IACR密码学电子印本档案，第817页，2021年。 [在线]。获取地址：https://eprint.iacr.org/ 2021/817[26] D. Bellizia、F. Berti、O. Bronchain、G. Cassiers、S. Duval、C. Guo、G. Leander、G. Leurent、I. Levi、C. Momin、O. Pereira、T. Peters、F. Standaert、B. Udvarhelyi 和 F. Wiemer，《Spook：基于海绵结构的抗泄漏认证加密与掩码可调分组密码》，IACR对称密码学汇刊，第2020卷特刊1，第295–349页，2020年。 [在线]。获取地址：https: // doi.org/ 10 13154 2020 1 295 349 。/tosc.v .iS . - A. Journault 和 F. Standaert，《超高阶掩码：高效实现与安全性评估》，载于《密码硬件与嵌入式系统——CHES 2017——第19届国际会议》，中国台北，2017年9月25日至28日，论文集，计算机科学讲义丛书，W. Fischer 和 N. Homma 编，第10529卷，2017年，第623–643页。 [ ] 10 1007 978 3 319 66787 4 30 在线。获取地址：https://doi.org/ 。/ - - - - D. Goudarzi 和 M. Rivain，《软件中的高阶掩码能有多快？》，载于《密码学进展——EUROCRYPT 2017——第36届密码技术理论与应用国际年会》，法国巴黎，4月30日 - –

*2017*年*5*月*4*日，论文集，第一部分，计算机科学讲义丛书，J. Coron 和 J. B. Nielsen 编，第10210卷，2017年，第567–597页。 [在线]。获取地址：https://doi.org/ 10 1007 978 3 319 56620 7 20 。/ - - - - J.

[29] Daemen 和 V. Rijmen，《*Rijndael* 的设计：高级加密标准（*AES*）》，信息安全与密码学丛书，2002年。

[30] [ ] 10 1007 978 3 662 04722 4 在线。获取地址：https://doi.org/ 。/ - - - - A. Bogdanov、L. R. Knudsen、G. Leander、C. Paar、A. Poschmann、M. J. B. Robshaw、Y. Seurin 和 C. Vikkelsoe，《PRESENT：一种超轻量级分组密码》，载于《密码硬件与嵌入式系统——*CHES 2007*》，奥地利维也纳，*2007*年*9*月*10*日至*13*日，论文集，计算机科学讲义丛书第4727卷，2007年，第450–466页。[31] P. E. Hart、N. J. Nilsson 和 B. Raphael，《最小代价路径启发式确定的形式基础》，电气电子工程师学会系统科学与控制论汇刊，第4卷第2期，第100–107页，1968年。 [在线]。获取地址：https://doi.org/ 10 1109 1968 300136 。/TSSC. . C. Moustakas，《启发式研究：设计、方法论与应用》，1990年。[33] J. Daemen、A. Mehrdad 和 S. Mella，《计算非线性映射 χ上的差分分布》，载于《安全、隐私与应用密码工程——第*11*届国际会议，*SPACE 2021*》，印度加尔各答，*2021*年*12*月*10*日至*13*日，论文集，计算机科学讲义丛书，L. Batina、S. Picek 和 M. Mondal 编，第13162卷，2021年，第 3 21 [ ] 10 1007 978 3 030 95085 9 1 –页。在线。

[32] [34] 获取地址：https://doi.org/ 。/ - - - - M. Brinkmann 和 G. Leander，《五维以内几乎完美非线性函数的分类》，设计、编码与密码学，第49卷第1–3期，第273–288页，2008年。 [在线]。获取地址：https://doi.org/10.1007/ s10623-008-9194-6

[35] C . Beierle，和《G 二次 APN 函数的新实例》，电气电子工程师学会信息论汇刊，第 68 卷，第 1 期，670–678 页，2022 年。[在线]。网址：https://doi.org/10.1109/TIT.2021.3120698



Marie Bolzer 目前正在洛林大学和 LORIA 实验室攻读博士学位。她的研究兴趣包括算法学，以及开发用于构造和分析对称密钥密码组件的自动化工具。



S´ebastien Duval 于 2018 年获得索邦大学计算机科学博士学位。自 2021 年起，他担任洛林大学和 LORIA 实验室的副教授。他的研究兴趣包括轻量级对称密钥密码学的设计，以及密码学的现实世界安全性。



Marine Minier 于 2002 年获得利摩日大学博士学位，并于 2012 年获得里昂大学的法国博士后资格。2005 年，她加入里昂国立应用科学学院和 CITI 实验室，担任助理教授。自 2016 年起，她在洛林大学和 LORIA 实验室担任教授。她的研究兴趣包括对称密钥密码学和无线传感器网络安全性。

附录 A
高次次数的局限性

为了将这里介绍的类似方法用于 $d > 2$ 次情形，我们希望像二次情形那样将问题转化为一个线性系统。因此，我们希望将任意 d 次函数表示为 操作集合中元素的异或和。让我们从 $d = 3$ 开始，并将 L_n 记作 n 比特上线性函数的集合。为了得到 n 比特上的任意 3 次函数，我们可以先考虑形如 $l_1 \times l_2 \times l_3$ 的元素的异或和，其中 $l_i \in L_n$。

示例 *1*： 以这种方式获得的每个元素的代价为 2 个与，除非 l_i 中有一个等于 1；在这种情况下，这些元素对应于操作集合 中的元素，代价为 1 个与。请注意，与所有操作都需要 1 个与的二次情形不同，元素不再具有相同的代价。因此，在向Op selec中添加操作时，必须调整test xor 函数，使其将该代价纳入考量。

我们将这个新集合记为操作集合$_3$。该新集合中的元素数量大得多，但对于较小的大小仍可能保持合理（表 XIV 的第 3 列）。然而，为了优化乘法复杂度，我们应考虑其他类型的分解方式，如下例所示。

Examp 第 *15* 页： 考虑 $y = x_0x_1x_3 \oplus x_2x_3$。我们希望将 y 表示为操作集合$_3$中元素的异或和。y 不在操作集合$_3$中，因此我们希望了解操作集合$_3$ 中的 2 个元素的异或和是否等于 y；这样得到的实现所需代价将严格大于 2 个与。于是我们得到 $y = c_1 \oplus c_2$，其中 $c_1 = x_0x_1x_3 \in$ 操作集合$_3$ ，以及 $c_2 = x_2x_3 \in$操作集合$_3$。所得实现需要$2 + 1 = 3$ 个与。

然而，我们可以看到 $y = ((x_0 \times x_1) \oplus x_2) \times x_3$ ，并且这种分解只需 2 个与。因此，只需 2 个与的分解并不都能仅通过 $l_1 \times l_2 \times l_3$ 形式的元素实现，其中 $l_i \in L_n$。

因此，我们还考虑 $(l_1 \times l_2 + l_3) \times l_4$ 形式的元素，其中 $l_i \in L_n$。将这些分解加入 操作集合$_3$ 后，我们得到表 XIV 第 4 列（v2）的基数。

同样地，$((l_1 \times l_2) \oplus (l_3 \times l_4) \oplus l_5) \times l_6$ 形式的元素，其中 $l_i \in L_n$ ，也可以加入，因为它们的代价为 3 个与，而只有通过 $(l_1 \times l_2 \times l_6) \oplus ((l_3 \times l_4) \oplus l_5) \times l_6$ 形式的分解才能得到这些元素，而这种分解的代价为 4 个与。这样，我们就得到表 XIV 第 5 列（v3）的基数。

比特数	#操作集合	#操作集合$_3$ v1	#操作集合$_3$ v2	#操作集合$_3$ v3
4	35	141	561	561
5	155	1241	14 261	28 149
6	651	10 417	283 387	2 033 725

表 XIV：3 次运算集合的基数

总之，对于 $d > 2$ 次，我们无法继续采用这种高效方法，因为它需要存储过多元素。一种合理的方法是使用上文给出的公式分解给定函数，因为这些公式给出了可能的因式分解及其对应的与门成本。