中国科学院研究生院

博士学位论文

GNoMoN 密码计算平台的设计与实现

作者姓名：李昕

指导教师：林东岱 研究员

中国科学院软件研究所

学位类别：

学科专业：

培养单位：中国科学院软件研究所

2012 年 4 月

The Design and Implementation of

Gnomon Cryotgraphic Computing Platform

A Dissertation Submitted to
Graduate University of Chinese Academy of Sciences
In partial fulfillment of the requirement
For the degree of
Doctor of Cryptography and Information Security

Institute of Software

Chinese Academy of Sciences

April, 2012

独创性声明

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果.尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果.与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明.

签名：李[illegible]　　日期：2012-6-6

关于论文使用授权的说明

本人完全了解中国科学院软件研究所有关保留、使用学位论文的规定，即：中国科学院软件研究所有权保留送交论文的复印件，允许论文被查阅和借阅；中国科学院软件研究所可以公布论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存论文.

（保密的论文在解密后应遵守此规定）

签名：李[illegible]　导师签名：[illegible]　日期：2012-6-6

摘 要

信息安全的关键内容是密码理论，而密码理论的核心则是密码算法。众所周知，密码算法的发展和其依赖的计算工具和计算平台是分不开的。它对提高密码算法的设计、分析和应用能力将有重要的意义。本论文的研究目标主要包含以下三个内容：一是 Gnomon 密码计算平台的设计，具体包括分布式计算环境的设计以及底层数学库结构的设计，另外由于 Groebner 基是符号计算及密码分析的基本工具之一，所以本文的第二个主要研究内容是 Gnomon 密码计算平台中的 Groebner 基算法的设计实现与优化以及其在密码分析中的应用。最后本文还研究了对 Trivium 流密码的区分器攻击，所取得的主要成果如下：

1. 设计得到了一种针对一般性密码计算的基础系统模型,该模型内置的通用分布式引擎可帮助密码学研究人员高效开发自身所需的分布式计算实现。另外通过深入分析密码计算的现实特点,设计得到了一种针对一般性密码计算的数学库。目前,基于该数学库开发的密码计算基础平台 Gnomon 已取得一系列现实应用成果。

2. 针对当前最先进的非线性方程组求解算法 F5,分析了其在超定环境下的缺陷,即未能充分利用超定条件,导致大量计算冗余;进而分析了 F5 算法的矩阵版本（MatrixF5）在超定环境下的优势及不足。在二者基础上,我们设计得到了一种求解非线性超定方程组的新方法，我们称为 F5D 算法。F5D 算法完全继承了 F5 和 MatrixF5 算法在方程组求解问题上的优点,同时分别避免了 F5 算法带来的计算冗余和 MatrixF5 算法过大的存储需求。实验结果表明,在超定环境下,F5D 算法可被用作 F5 算法和 MatrixF5 算法的替代。

3. 设计一种在布尔方程组求解过程中的一种新的表示方法，我们称为 BanYan。与 BDD 和系数矩阵等基于项的传统布尔多项式表示相比,BanYan 可以降低在计算过程中的空间需求,从而显著提升布尔方程组求解算法的现实求解能力。

4. 研究了对 Bivium 流密码算法猜测部分变元,再进行求解分析的方法。给出了对于猜测部分变元后子系统平均求解时间的估计模型,提出了基于动态权值以及静态权值的猜测变元选则方法和面向寄存器的猜测方法。在计算 Groebner 基的过程中,对变元序的定义采用了 AB, S, S-rev,SM,DM 等十种新的序。同时,提出了矛盾等式的概念,这对正确分析求解结果以及缩小猜测空间有重要作用。

5.研究了对 CTC 分组密码在通过最简约化消除中间变元,获得只含有初始密钥变元的方程的分析，与传统的包含中间变元的方程描述做了攻击对比。同时，把差分方程引入 CTC 密码系统,对在差分方程作用下的仅含密钥变元的方程系统做了进一步的分析。另外对 Present 分组密码算法进行了代数攻击与错误攻击，积分攻击，及侧信道（冷冻）攻击的一些结合性研究。

6.在对 Trivium 流密码的区分器的寻找中，进行了贪心算法及遗传算法的实验，实验结果表明：遗传算法能够在更小的 IV 权重下，区分更多的初始轮。当

前我们的最好结果是找到了一个权重为 24 的 IV 变元集{ 4, 7, 10, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46, 49, 52, 55, 58, 61, 64, 67, 70, 73, 76, 79 }，可区分 929/1152 的初始轮，这比贪心算法搜到 IV 权重为 29 时，只找到 921 轮的区分器来说有不少改进。

关键词 分布式计算，密码库，Groebner 基，布尔方程组，代数攻击，区分器

Abstract

The key problem of information security is the cryptography theory, in which the core content is cryptographic algorithm. It's well known that the development of cryptographic algorithm relies on computing tools and platforms. They have great significance for improving the ability of cryptographic algorithm's design, analysis and application. In this paper, we focus on three problems, one is the design of Gnomon cryptographic computing platform, including the distributed computing environment and the mathematical library. Because the Groebner base is one of the basic tools in the symbolic computation and cryptanalysis. So in Gnomon cryptographic computing platform, the Groebner base algorithm's design and optimization and its application in cryptanalysis are the second main research contents. In the end, we focus on the Trivium distinguisher. We have obtained the following principle achievements:

1.We presented a foundational system model for general cryptographic computations. This model provides an open framework, in which all implementations are naturally merged and shared with each other. Furthermore, there is a built-in engine in the model for distributed computing, which greatly helps researchers in developing distributed implementations. Finally, based on this model, we developed GNoMoN, a foundational platform for cryptographic computations.

2.We analyzed the defects of the F5 algorithm which is the most advanced one in existence for solving over-determined non-linear equations ,namely not making full use of all the equations, which makes the computing redundancy according to the order of polynomials. Then we analyzed the advantages and limitations of MatrixF5 algorithm. On these basis, we designed a kind of new method ,named F5D algorithm. F5D algorithm inherits the advantages of F5 algorithm and MatrixF5 algorithm. Meanwhile, it avoid to bring computing redundancy of the F5 algorithm and the large storage requirements of MatrixF5 algorithm respectively. Experiment finds that, for solving nonlinear over-determined equations, F5D algorithm would be better choice than the other two.

3.The contradiction of limited computer storage space and solving demand growth of existing algorithm is the major bottleneck for get more progresses. We presented a high efficient boolean polynomial representation, BanYan. BanYan is designed for boolean equations solving algorithms based on leading-term eliminating. Analysis and experiments showed that, compared with traditional representations based on terms, BanYan can greatly reduce the space requirement and then improve the solving ability.

4.Guess-and-determine algorithm is usually used to estimate computational

complexity of algebraic attack. We introduce a model to estimate average time in solving subsystems more accurately, and propose some criteria on selecting specific guessed variables to speed up the solving efficiency, which based on static weight and dynamic weight etc. For computing Groebner bases, we use several variable order which are AB,S,S-rev etc. Meanwhile, we introduce the concept of conflicting equations, and show the importance for correct analysis and narrow guessing space. In the end, we estimate the time of attacking Bivium.

5.We described the equations of the CTC block cipher with the lexicographical order, then after eliminating all the intermediate variables via reduced normal form reductions, we can get the resulting system in initial key variables only, denoted by K_0 equations. Experiments showed that, solving the K_0 equations of the CTC block cipher is more efficient than direct solving the equations with all the intermediate variables. Adding differential equations to the equations of the CTC block cipher can bring more low degree K_0 equations, which will improve the solving efficiency. Besides we combine algebraic attack ,fault inject attack , integral attack and the side channel (frozen) attack for the Present block cipher algorithm.

6. We use the greedy algorithm and genetic algorithm to test the Trivium stream ciphers distinguisher. Experimental results show that the genetic algorithm can distinguish more initial round with the smaller IV weights. At present, our best result is to find a IV variable set {4, 7, 10, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46, 49, 52, 55, 58 , 61, 64, 67, 70, 73, 76, 79}, which can distinguish between 929 of the initial 1152 round. It is better than the greedy algorithm.

Key words: Disributed computing, cryptographic library, Groebner bases, boolean equations , algebraic attack, distinguisher

目 录

图表目录

第一章 引言

§1 通用密码计算平台简介

现代密码理论的基础是数学，密码运算的实现大都要用到一些极其专业的数学理论与算法，如数论、有限域、代数几何等。其中大整数的运算、多项式的运算、有限域上的相关运算等等是密码计算的基石。近些年来，随着计算机技术的发展，符号计算技术水平得到了很大程度的提高，出现了许多新理论和新算法，如 Groebner 基理论、格基约化算法、数学机械化方法等。这些理论和方法反映到密码计算领域，大大提高了密码计算的效率，如 XL 算法[3]、F_4[10]、F_5[11]算法等等。然而，对密码分析与设计者而言，常常需要花大量的时间和精力在底层数学算法的实现及其优化上，这在很大程度上为密码算法的设计、分析以及高效实现增加了难度，从而增加密码算法的设计开发周期。因而，利用新的计算技术和方法，设计和实现开放、高效、可扩展的密码库将有利于密码算法的研究和分析，对提高密码算法的设计、分析和应用能力以及缩短研发周期将有重要的意义。

在此背景下，我们设计并实现了 Gnomon 密码计算基础平台，该平台主要包含密码计算基础算法库、可编程语言解释环境、应用开发环境以及分布式计算环境（分布式密码计算网格系统）。在设计上，Gnomon 密码计算基础平台的计算功能涵盖了密码学的主要研究领域，具有良好的用户界面和应用开发环境，我们希望 Gnomon 密码计算基础平台作为一个专为密码计算服务的通用计算平台，能够为密码学研究和教学提供一个方便的工作实验环境和应用开发环境，为密码学的研究提供必要的技术支撑，以提高相关方面的开发能力与开发效率。

Gnomon 密码计算基础平台的底层数学库结构提供了诸如整数、小数、分数、有限域、多项式、布尔函数、矩阵、向量、列表、椭圆曲线等丰富的代数结构和代数运算，实现了多精度整数、有限域、线性代数、多项式、布尔函数、序列、椭圆曲线、密码算法等多个模块，其计算功能涵盖了密码学的主要研究领域。

该模型内置的通用分布式引擎可帮助密码学研究人员高效开发自身所需的分布式计算实现。我们组建了实验性的 GNoMoN 分布式计算环境。基于该环境，目前已取得的代表性成果包括使用 Rainbow 攻击方法和 5 台 Core Duo 2.4G PC，GNoMoN 能够以超过 99%的概率在 5 分钟内破解任意长度在 14 字符以内的 Windows XP 用户口令。进一步地，我们设计了素因子分解的分布式算法，实验结果表明，GNoMoN 分布式计算的加速比接近线性。

§2 密码分析中的多项式求解问题与代数密码分析

早在 1949 年， Shannon[1]在一篇著名的论文中便提出:破译一个好的密码系统所做的工作等价于对含有大量未知变元方程组的求解。即我们可以将密码系统视作多变元的方程系统，密码破译的难点转化为通过寻找和求解某类特定的方程组，从而得到原始密钥。这是最早出现的代数攻击的思想。

从广义上讲，把一个密码系统与一个代数方程组对应，把对这个密码系统的攻击问题归结为求解与之对应的代数方程组，这样的攻击都可以称做是代数攻击。很长时间里，密码工作者往往关注于密码函数本身的性质和弱点而忽视了对代数攻击的研究。导致他们不愿意从解方程组的角度去分析密码系统的另一个原因就是，计算复杂度理论告诉我们求解一个系数随机选取的非线性代数方程组是一个 NP 完全问题。但是对于一个精心设计的密码系统，它所对应的代数方程组必然不是随机的，那么这种“精心设计”就有可能使得它所对应的代数方程组容易求解。

1999 年，Shamir 提出了复线性化(relinearization)方法[2]，他们认为认为对于足够超定的方程系统，算法复杂度将是多项式时间，并对 HFE 公钥密码进行了分析。这可以说是近年来人们对代数攻击研究的开端。2000 年 Courtois 等人对 Shamir 的复线性化方法进行了扩展，提出了 XL(eXtended Linearizations)算法，表示扩展线性化[3]， XL 算法对流密码的攻击取得了一系列的漂亮成果。

2002 年 Courtois 用 XL 算法对 Toyocrypt 密码系统进行了分析[4]，并声称在 2^{92} 个 CPU 时间内，用 51Kbytes 的密钥流就可以将 Toyocrypt 破解。2003 年 Courtois 发表了重要的文章“Algrbric Attacks on Stream Ciphers with Linear Feedback” [5]。该文章声称：可以在 2^{49} 个 CPU 时间内，仅用 20Kbytes 的密钥流将 Toyocrypt 破解;可以在 2^{57} 个 CPU 时间内成功的将 Nessie 的候选算法 LILI-128 破解。并且文章还在更广泛的意义上指出:如果一个布尔函数只涉及关于状态比特的一个较小的集合，那么这个密码系统就容易被攻击。Courtois 对代数攻击的研究把人们对代数攻击的关注推向了顶峰，以前被人们认为并没有明显弱点的 Toyocrypt 和 LILI-128，在代数攻击下就显得有些脆弱。于是密码学家们普遍认为需要新的指标来衡量密码函数的安全性。2004 年 Meier 提出了布尔函数代数免疫度的概念[6]，并指出了布尔函数代数免疫度的上界。XL 算法似乎只对那些以具有低代数免疫度的布尔函数作为滤波函数的流密码是有效的。于是寻找具有较高代数免疫度的布尔函数就显得尤其重要。2007 年，李娜[7]给出了具有奇数变元代数免疫度达到上界的布尔函数的构造方法。这似乎抵挡住了 XL 算法给流密码带来的冲击。

XL 算法在分组密码攻击方面并没有取得太多进展，值得一提的是在 2002 年的亚密会上 Courtois 等人提出了 XSL(eXtended Sparse Linearizations)的攻击方法[8]。XSL 是对 XL 算法的改进，是一种针对分组密码代数结构的分析方法，适用于 S 盒可以用一个超定的代数方程系统描述的分组密码算法。但之后学术界对 XSL 攻击产生线性独立方程数量的估计有广泛的争议，对攻击的有效性表示质疑[9]。

与此同时，早在 1999 年 Faugere 提出了一种有效的计算 Groebner 基的方法，称作 F4 算法[10]（F4 实际上是 Buchberger 算法的矩阵版本），并且在 2002 年提出了 F4 的改进版本 F5[11]。F4，F5 最初并没有引起太多人的注意，直到 2003 年的美密会上，Faugere 发表了他对 HFE 公钥密码系统的分析结果[12]，声称他

用 C 语言实现的 F5 算法可以在 2 天的 CPU 时间内破解了 HFE Challenge 1。同时，Steel 在 2004 年使用其自己实现的 F4 算法花费了 25.4 个小时，在一台 750MHz 主频，15G 内存的机器上同样破解了 HFE Challenge 1。至此，XL 算法和 F4，F5 系列算法成为代数攻击的主流算法。F4，F5 有完善的代数几何理论作支持，Sugita 和 Ars 等研究小组考查了 XL 算法与 F4 算法之间的关系[13]，结果表明 XL 算法可以用 F4 算法的冗余版本来表示，因此，它的效率并不比 F4 算法更快。2008 年，Bettale 等人[14]基于猜测变元与 F5 算法的复杂度理论的基础上，提出了一种先猜测后求解的方法 Hybrid。Hybrid 方法把原来对一个复杂度很大的系统的求解化简为对若干更容易求解的小的系统的求解。该方法对于 TRMS 以及 UOV 签名方案[15]进行了分析，给出了理论上的最优猜测变元数。如对于 UOV 签名方案（60 变元，20 方程），在仅猜测一个变元的情况下，其复杂度由直接使用 F5 算法求解的 $2^{82.51}$ 减少为 $2^{66.73}$，所需内存为 139G。在猜测两个变元的基础上，尽管其复杂度略有提升，为 $2^{67.79}$，但仅需要 12G 内存。

在 2008 年的 SCC 会议上，丁金泰等人提出了 Mutant XL 算法[16]，并使用该算法对 MQQ 密码系统[17]进行了分析，其文章声称该算法比 F4 算法有更高的执行效率，并且占用更少的内存。同时，丁金泰指出，mutant 的概念同样可以用于 F4、F5 算法以加速 Groebner 基的求解，并把该算法称为 MF4GB 算法。在同年的 PQCrypto 2008 会议上，丁金泰等人又提出了 Mutant XL 算法在 GF(2)上的优化版本，称为 MutantXL2[18]，并针对 HFE 系统与 F4 算法进行了对比，声称其计算结果同样在算出 Groebner 基时，MutantXL2 比 F4 在计算过程中要处理的矩阵要小的多。接着，2009 年丁金泰等人又提出了 MXL3 算法[19]，该算法主要针对零维理想计算 Groebner 基，并同样破解了 HFE Challenge 1，并且声称使用了更少的内存。与 F5 算法在破解 HFE Challenge 1 的最后生成矩阵是 293287*1666981 相比，其最后的生成矩阵是 268840*1666981。2010 年，Enrico 等人在其文章中也表明对于 HFE 系统而言，MutantXL 系列算法确实优于 F4 算法。在以上工作的基础上，2010 年的非密会上，丁金泰等人提出了 MGB 算法[20]，并且使用 30G 的内存，在 2.3 天里计算出了 32 变元，32 方程的 GF(2)上的随机多项式系统。他们声称目前该方程系统并没有其他求解工具可以求解。

众所周知，布尔多项式对于密码学而言有很重要的作用。Brickenstein 领导的团队于 2007 年设计了一个专攻于布尔多项式环上的的计算系统——PolyBori[21]，该系统有很出色的表现。它使用零压缩二元决策图的数据结构（ZBDD），通过观察在 Groebner 基计算过程中，绝大多数的多项式操作都是两个相似的多项式（仅有少量项不同）相互运算，它设计了精巧的内存池以及 cache 机制以最大限度的利用曾经出现的多项式，在消去序下通过递归的加法和乘法之后不仅能保持 ZBDD 的结构，并且能保持不生成多余的子树以尽量节省空间。在其优良的底层多项式运算结构之上，PolyBori 的 Groebner 基算法选择的是 slimgb 算法的变形，称之为 symmgbGF(2)算法。其实验结果表明，与 Magma 中的 F4 算法相比，PolyBori 在 GF(2)上的表现更为高效，占用更少的内存。在与

SAT 求解器(如 MinSat)的对比中，尽管 PolyBori 的内存使用高于 MinSat，但其求解效率在很多密码学的应用实例中依然略高于 MinSat。在 HFE 的求解测试中（25-45 变元），Magma 的 HFE 专用求解工具效率最高，这是因为 PolyBori 的设计是针对上百变元进行优化的。在变元较少时，高斯消元的效率起决定作用。2010 年，Tobias Eibach 等人使用 PolyBori 对 Bivium 流密码进行了分析[22]，其所获得的求解时间估计是 $2^{39.12}$ 秒，这是目前对 Bivium 流密码进行直接求解攻击的最好结果。

Raddum 等人 2007 年提出了一种新的代数攻击思路，称为 MRHS 方法[23]。与以往的把密码系统转化为非线性方程组相比，MRHS 把密码系统转化为一个多维右侧列向量线性方程组（Multiple Right Hand Sides Linear Equations），通过对该方程组的求解来找到相应的密钥。该方法对 AES 的简化版相当有效，其求解效率与 PolyBori 相当，为科研人员进行代数攻击的相关实验提供了新的可供选择的工具。

Bard 等人于 2010 年提出了基于图分割的方程求解策略[24]，这里不再是对 Groebner 基算法本身的优化，而是对待计算的方程系统进行预处理。如果待计算的方程系统足够稀疏，以至于可被表示成无连通图的话（变元作为图中的点，在同一多项式中的变元用边相连），则相当于把对整个大系统的求解转化为分别对每个子系统的求解。显然，实际应用中很难有如此稀疏的方程系统，Bard 等人建议找到一种最优策略，通过删除尽可能少的点，把原方程系统所代表的图分解为点数均衡的两个子图。而这种策略在图论中即平衡点分割问题。而点的删除相当于对相应的变元进行猜测。通过这种方法对 Bivium 流密码进行了分析，对于 399 变元的 Bivium-B 流密码系统，需要删除 122 变元（称之为分离变元）才能使原方程系统分解为点数分别为 150 及 127 的两个子系统。由于 Bivium-B 流密码系统仅含有 80 位的密钥，分割所要进行的猜测已经大于了穷举攻击。然而，Bard 等人指出，可以选择分离变元的一个子集进行猜测，尽管不能把原方程系统分割，但选择分离变元进行猜测远优于对随机变元进行猜测。该方法同时对密码系统的设计以启发，即密码系统的设计在用图来表示时，最好是完全图。

2010 年，王明生等人提出了一种新的计算 Groebner 基算法 GVW[25]，该算法利用 syzygy 模的概念来减少无用的关键对计算，并声称其算法比 F5 算法有 4-10 倍的加速。Antoine 等人提出了 F4 算法的改进版本，称为 F4Remake[26]，该算法适用于需要反复计算具有相似密码系统结构的方程，如 Bettale 等人的 Hybrid 方法，在计算出第一个方程系统后，其余的方程系统计算会有显著加速。

仅管解方程算法的不断优化，新的工具不断出现。但对于一个实际应用中的密码系统，尤其是分组密码系统，至今没有任何工具可以直接求解出一个实际应用中的全轮的分组密码系统。解方程与其他攻击方法的结合成为未来发展的一大趋势。Courtois 等人在 2008 年提出对 KeeLoq 分组密码的滑动-代数攻击[27];Martin[28-30]则在他的一系列文章中研究了解方程技术与差分攻击、线性攻击、高阶差分攻击的结合，尤其是其对 PRESENT 分组密码的代数差分攻击，取

得了比直接差分攻击更好的结果。代数攻击与侧信道攻击相结合也成为当前的一大热点，如 Renauld[31]对 AES 密码差分能量攻击;Hojsik[32]对 trivium 流密码的差分错误攻击等，解方程技术都在其中扮演着重要作用。

纵观代数攻击仅十余年来的发展，我们可以总结为其三大发展方向：一是寻求更优的解方程算法或提高已有算法效率。以 Groebner 基算法为例，从 1965 年 Buchberger 在其博士论文中给出了最原始的计算 Groebner 基的算法，被称为 Buchberger 算法[33]，到当今的 F4，F5 算法，以及在 F4，F5 算法上的各种变形，如 F4Remake，Hybrid，GVW 等。同时一些专用的求解工具也在不断涌现，如 PolyBori，MRHS，MiniSat 等。寻求更好的求解算法与工具对密码分析有至关重要的意义。二是寻求密码系统的更丰富的表示方式，以便于找到更好的求解方法。Weinmann 等人提出了可以不经计算直接把 AES 密码系统表式为 Groebner 基的方法[34];Raddum 等人则把 AES 密码系统转化为一个多维右侧列向量线性方程组[23]; Borghoff 等人则把对 Bivium 流密码描述成一个整数规划问题[35]。这些工作为密码分析提供了一些新的思路。三是方程求解技术与其他技术的结合，借助侧信道、差分方程等技术以获得更稀疏、更易求解的方程系统。

我们的工作方向也即沿袭着代数攻击的发展方向展开。我们同样实现了 F4，F5 等算法，进一步的优化是我们下一步的工作重点;同时对于密码系统的其他求解工具，其技术原理与适用范围是值得深入研究的。我们的最终研究目标是实现基于变元猜测的方程自动求解工具，包括单机版与分布式版；建立一套利用 Groebner 基理论分析密码系统的方程的模型与方法，并实现一个套基于代数攻击与侧信道、线性、差分、高阶差分等攻击相结合的密码分析平台。

§3 论文的研究内容与组织

本文的研究内容主要分三部分，一是 Gnomon 密码计算平台的设计，具体包括分布式计算环境的设计以及底层数学库结构的设计，具体内容在第二、三章论述。

由于 Groebner 基是符号计算及密码分析的基本工具之一，所以本文的第二个主要研究内容是 Gnomon 密码计算平台中的 Groebner 基算法的设计实现与优化以及其在密码分析中的应用，具体内容在四至八章论述。

最后本文还研究了对 Trivium 流密码的区分器攻击，具体内容在第九章论述。

本文具体组织如下：

第二章分析了 Gnomon 密码计算平台的设计框架，并介绍了已经取得的分布式密码的计算成果。

第三章主要介绍 Gnomon 密码计算数学库的总体架构和设计思想，要解决的核心问题，如不同数据对象之见互操作问题、多项式系数的表示问题、有限域的设计等，并给出了解决方案。

第四章介绍了 Gnomon 密码计算平台中的 Groebner 基算法（F5）的设计实现与改进。分析了 F5 算法及 F5 算法的矩阵版本（MatrixF5）在超定环境下的优势及不足。在二者基础上，我们设计得到了一种求解非线性超定方程组的新方法——F5D 算法。实验结果表明，在超定环境下，F5D 算法可被用作 F5 算法和 MatrixF5 算法的替代。

第五章介绍了一种在布尔方程组求解过程中的一种新的表示方法——BanYan，与 BDD 和系数矩阵等基于项的传统布尔多项式表示相比，BanYan 可以降低在计算过程中的空间需求，从而显著提升布尔方程组求解算法的现实求解能力。

第六章介绍了对 Bivium 流密码算法猜测部分变元，再进行求解分析的方法。给出了对于猜测部分变元后子系统平均求解时间的估计模型，提出了基于动态权值以及静态权值的猜测变元选则方法和面向寄存器的猜测方法。在计算 Groebner 基的过程中，对变元序的定义采用了 AB，S，S-rev，SM，DM 等十种新的序。同时，提出了矛盾等式的概念，这对正确分析求解结果以及缩小猜测空间有重要作用。

第七章介绍了对 CTC 分组密码在通过最简约化消除中间变元，获得只含有初始密钥变元的方程的分析，与传统的包含中间变元的方程描述做了攻击对比。同时，把差分方程引入 CTC 密码系统，对在差分方程作用下的仅含密钥变元的方程系统做了进一步的分析

第八章探讨了代数攻击与错误攻击，积分攻击，及侧信道（冷冻）攻击的一些结合，对 Present 密码算法进行了分析。

第九章在对 Trivium 流密码的区分器的寻找中，进行了贪心算法及遗传算法的实验对比分析。

最后第十章是全文的总结与展望。

第二章 Gnomon 分布式计算平台

§1 密码计算概述

首先需要强调，本章所讨论的密码计算，并非传统意义上仅以应用为背景的狭义的密码计算，而是涵盖了密码设计、密码分析、密码应用乃至密码学基础理论研究可能涉及的所有计算对象和计算行为。事实上，在研究性密码计算、尤其是密码分析相关计算中，密码计算的特点有更加充分的体现。

1.1 计算对象的特点

现代密码体制的设计要求和理论基础决定了其计算对象在类型上具有集中性。例如为方便软硬件实现的考虑，序列密码体制和分组密码体制的直接计算对象分别为二进制位和二进制分组，相关研究则主要基于二元有限域(F_2)及其扩域上的序列、多项式等。公钥密码体制普遍建立于数论、代数几何等领域的计算困难问题之上，导致其计算对象高度集中于整数环、有限域和多变元多项式环等代数结构。根据当前的密码学发展趋势，这种集中性在未来较长一段时期内依然不会改变。不过，由于与这些类型相关的大规模计算需求往往也仅集中在密码学相关领域，使得其中相当一部分并非当前主流数学计算系统的内置类型。表 2-1 列出了密码计算主要的基本数据类型，及其在密码学研究人员目前常用的各种计算库和计算系统中的支持情况。

表 2-1 密码计算的基本数据类型

	任意精度整数	比特结构		有限域			多变元多项式			椭圆曲线
		比特分组	比特序列	素域	F_2扩域	一般扩域	整系数	有限域系数	布尔多项式	
Crypto++ 5.6.0	√	√	√	√	-	-	-	-	-	√
CoCoA 4.7.5	√	-	-	√	-	-	√	*	-	-
Macaulay 2	√	-	-	√	-	-	√	*	-	-
Magma 2.14	√	-	√	√	√	*	√	*	-	√
Maple 12	√	-	-	*	-	-	√	-	-	-
NTL 5.4.2	√	-	-	√	√	-	√	-	-	-
SINGULAR3.10	√	-	-	√	-	-	√	*	-	-

说明: √ 表示支持 - 表示不支持 * 表示部分支持

由于各种计算库和计算系统均具备不同程度的可扩展性，因此在各自基础上实现其不完全支持的数据类型是可能的。但这些数据类型结构复杂，实现效率又攸关各种高级密码计算实现的最终性能。为得到一种理想的基本类型实现，往往需要长期的开发优化过程。以最基础的任意精度整数为例，目前广泛使用的 GMP

实现[36]由专门开发组维护，自首次发布至今已近二十年，仍每年都有改进版推出。相比之下，密码研究人员根据临时需求自行开发得到的实现则很难保证质量。进一步地，各种基本数据类型之间并非孤立存在，而是彼此关联，构成一个复杂的体系。为得到某种高级数据类型(例如某种多变元多项式环)，与其相关的低级数据类型(例如其系数环或系数域)也不得不同时实现。这两个因素决定了密码计算的各种基本数据类型适合由基础密码计算系统统一提供和维护。

与类型上的集中性形成对照的是，密码计算中的计算对象在表示上却具有多样性的特点。对于同一数据类型，不同计算方法往往基于完全不同的数据表示形式。例如布尔多项式的常用表示包括真值表、项集、BDD 和系数向量等。对数据类型表示形式的要求在计算系统中直接体现为对该类型实现方式的要求。如果使用与计算方法要求不符的实现方式，将严重影响计算实现的最终性能。此外，即使对于同一计算方法，根据具体问题和计算环境的不同，也可能需要使用不同的数据类型实现。例如在问题规模很大、存储空间紧张时，为完成计算往往不得不使用空间利用率高的紧缩表示。而当问题规模较小、空间相对充裕时，则应使用空间利用率较低、但计算速度更快的一般性表示。总之，对于基础密码计算系统，仅为每种数据类型分别提供一种实现是远远不够的。

1.2 计算行为的特点

和计算对象类似，密码计算中的计算行为在问题类型和解决方式上同样分别具有集中性和多样性的特点。一方面，基本数据类型的基本运算在所有密码计算问题中居于核心地位，几乎所有密码计算问题最终都可归结为这些基本运算。因此其实现效率直接关系到绝大多数计算实现的性能，适合在基础计算系统中统一提供和维护。另一方面，密码计算涉及的计算问题往往不存在单一的最优解决，而是对于不同的具体情况(例如计算对象的实现方式，以及硬件条件等)有不同的适用方法。作为基础密码计算系统，应支持这些方法在系统中的有机共存。

进一步地，密码分析中计算行为的特点尤其突出。由于密码分析涉及大量计算困难问题，其中不乏 NP 完全问题，因此相关计算需要的时间往往很长，中间数据规模可能很大，但相比之下源数据量却较为有限，易于通过网络分发。此外，密码分析相关的传统计算实现普遍基于穷举模式，使得其计算过程较为松散，耦合度很低，易于分割为大量独立的子问题。其中最具代表性的例子是各种密码体制的暴力分析，其源数据一般仅为少量明密文对，同时对整个密钥空间的穷举工作可以直接分拆为对各个子密钥区间的穷举分别进行。这两个特点决定了密码分析中的计算非常适合以分布式方式进行[37]。但需要指出，随着密码分析技术的进步，密码分析中的计算有不断复杂化的趋势，分布式实现的设计难度也在增加。

1.3 相关研究简介

长期以来，虽有为数众多的优秀科学计算系统不断涌现，但对于密码计算而

言，局限于单台计算机的运行环境决定了 Magma，Singular 等计算系统只能被用于完成一些 Toy 计算，同时 Maple 等少数支持分布式计算的计算系统又存在基本数据类型严重不足的问题。另一方面，将分布式计算技术引入密码计算的尝试虽开始较早，但相关研究均仅针对一个或一类相近的问题。早在上世纪 80 年代末，A.K.Lenster 等就通过 Email 在多台计算机间通讯，利用当时的计算条件成功分解了一个 100 位整数[38]。1990 年，S.R.White 研究了利用计算机病毒在网络中散发密码计算程序、实施分布式计算的可能性[39]。同时，中国抽彩(Chinese Lotto)[40]、并行碰撞[41]等针对密码计算问题的分布式计算方法也相继被提出。在此之后，密码计算取得的绝大部分重要成果都离不开分布式计算技术的支持，代表性的例子包括 ECCp-97[42]和 DES-III 挑战的被攻破。在国内，Wu 等于 2005 年提出了使用分布式技术计算吴特征列的模型和方法[43][48]。这些研究均针对具体问题，其计算实现不可重用。而由于涉及通信协议、资源调度算法等一系列技术问题，分布式计算实现的设计和开发对于密码学研究人员存在较高门槛。2005 年，Jiang 等设计了一种网格环境下的密码计算模型[37][48]，利用统一的分割/拼接接口解决穷举式和概率穷举式密码分析计算的分布式执行问题，并提出了相应的调度算法和资源分配方案[44]。

密码计算相关研究近年来进展迅速，随之也产生了一些新的问题。首先，即使是多项式相乘等最基本的运算，也不断有新算法和新实现出现。新开发的计算实现如不能充分利用未来可能出现的优秀成果，将很难具有生命力。同时，密码计算研究的不断深入也导致密码学研究人员对不属自身领域的计算问题很难有充分了解，面对多年来积累的众多计算方法和计算实现难以作出合理选择。这些新问题构成了密码计算技术继续进步的现实障碍。

1.4 基础系统模型的设计要求

综上所述，根据密码计算的实际特点，并参考相关研究长期以来积累的经验和遇到的问题，我们认为，密码计算基础系统模型的设计应满足下列要求。

完备性:若干数据类型及其基本运算在密码计算中居于核心地位，其实现工作重复性强、性能要求高。作为基础计算系统，应对其提供完全和高效的支持，把密码学研究人员从这部分工作中完全解放出来。

开放性:密码计算的前沿性决定了其计算问题的解决不会是一劳永逸的，而只能是一个逐渐进步、不断更新的长期过程。这就要求基础系统模型必须具备高度的开放性，保证新的计算实现能够不断融入现有体系。

包容性:对于密码计算中的同一数据类型或问题类型，仅支持单一计算实现无法满足密码计算的现实需要。基础系统模型应提供一种包容性的框架，使从属同一数据类型或问题类型的不同计算实现能够有机共存，并根据问题具体情况协助使用者作出合理选择。

高效性:对于基础系统模型，高效性包括两部分含义。第一，作为通用计算系统，为了整合众多密码计算实现，满足开放性和包容性的要求，不可避免地会

在具体问题的处理效率上付出一定代价，这部分代价应该被尽可能地降低。第二，密码计算天生适合分布式进行，基础系统模型应为常规密码计算的分布式解决提供一种通用方案。

易用性:最后，作为基础计算系统最主要的使用者，密码学研究人员更容易接受符合数学习惯的使用方式。这一点在系统模型的设计中也应被充分考虑。

§2 形式化分析

本节将为计算对象、计算行为、计算任务等密码计算中的核心元素建立形式化描述，在此过程中引出基础系统模型的主要设计思想。

2.1 计算对象:数据类型及其实现

首先，作为密码计算的计算对象，计算数据的类型及其计算机实现是两个不同的概念。在传统计算系统中，每种数据类型有且仅有一种计算机实现，导致两个概念往往混为一谈。事实上，**数据类型**(简记为 DT)是抽象的数学概念。每种 DT 背后都存在一个范畴明确的数学元素集合 S。这些集合一般有天然无歧义的数学术语名称，如"integer"(整数)、"boolean polynomial"(布尔多项式)等。直接使用这些名称作为 DT 的唯一标识(记为 DT_UNAME)对于计算系统的易用性有重要意义。这样，DT 在计算系统中可以形式化地表示为一个二元组:

DT: (DT_UNAME， S)

数据类型实现(简记为 DTI)是在 DT 和计算系统的物理状态之间人为建立的一种对应关系。每个 DTI 总是从属于某种 DT，但同一种 DT 可以拥有多个不同的 DTI。与 DT 不同，DTI 不是数学概念，一般也没有公认的术语名称，需由计算系统为其分配唯一标识(记为 DTI_UID)。

总之，DTI 在计算系统中可以形式化地表示为一个三元组:

DTI: (DTI_UID， DT_UNAME， DT_Implemention)

其中 DT_UNAME 表明了该 DTI 所属的 DT，DT_Implemention 代表 DTI 的程序体。

2.2 计算行为:问题类型及其实现

类似地，**问题类型**(简记为 PT)也是抽象的数学概念，其背后存在一个有明确定义域和值域的数学映射 f。该映射严格刻画了问题参数和正确结果之间的对应关系，在数学中一般有被广泛接受的术语名称，例如"factor"(分解)、"log"(求对数)、"order"(求阶)、"degree"(求次数)等。

出于易用性的考虑，该名称同样应被计算系统用作 PT 的标识(记为 PT_NAME)。注意多种 PT 可能有相同的 PT_NAME，例如"factor"在数学中既被

用于表示整数的因子分解，也被用于表示各种多项式的因式分解。但结合定义域之后，PT_NAME 的含义即唯一确定。这样，PT 在计算系统中可以形式化地表示为一个四元组:

$$PT: (PT\_NAME,\ (DT\_UNAME)^{+},\ DT\_UNAME,\ f)$$

其中$(DT_UNAME)^{+}$对应的 DT 列表表明了 f 的定义域，DT_UNAME 对应的 DT 表明了 f 的值域。注意 PT 依赖于 DT 而与 DTI 无关。

计算问题实现(简记为 PTI)是 PT 在计算系统中的现实解决方案，与 DTI 统称**计算实现**。

每个 PTI 总是从属于某种 PT，但同一 PT 可以拥有多个不同的 PTI。此外，PTI 对计算对象一般有具体的 DTI 要求。总之，PTI 在计算系统中可以形式化地表示为一个五元组:

$$PTI: (PTI\_UID,\ PT\_NAME,\ (DTI\_UID^{+},\ DTI\_UID,\ PT\_Implemention)$$

其中，PTI_UID 是计算系统为 PTI 分配的唯一标识，(DTI_UID^{+})表明了该 PTI 对计算对象的 DTI 要求，DTI_UID 表明了该 PTI 计算结果所属的 DTI，PT_NAME 结合 DTI 要求表明了该 PTI 所属的 PT，PT_Implemention 代表 PTI 的程序体。

2.3 计算任务

现实中每个具体的密码计算需求，上至一次复杂的 Groebner 基计算，下至一次简单的整数相加，都可视作一个**计算任务**(简记为 TASK)。TASK 可以形式化地表示为一个二元组:

$$TASK: (TaskType,\ TaskParameters)$$

任务的类型 TaskType 和参数 TaskParameters 分别表明了该 TASK 的计算行为和计算对象。传统计算系统一般仅支持使用 PTI_UID 作为 TaskType，同时 TaskParameters 所属的 DTI 也必须和该 PTI 的要求相吻合。这样，对 TASK 的处理即简单地归结为对对应 PTI 的调用。但对于密码计算，现实中存在两类不同的计算需求，分别使用不同的 TaskType 类型:

TaskType = PT_NAME:用户仅指定 TASK 所属的 PT，希望计算系统尽可能高效地计算出正确结果，而不关心、很可能也不了解应使用哪种 PTI。现实中大部分计算需求都属于这一类型。

TaskType = PTI_UID:用户在 TASK 中明确指定应使用哪种 PTI。此时用户可

能确定该 PTI 是最佳选择，或是仅关心该 PTI 本身(例如为实验其性能)。

基础计算系统应能处理这两种类型的 TASK。其中为处理前者，必须解决 PT_NAME 和 PTI 的匹配问题。此外，DTI 的多样性导致 TaskParameters 基于的 DTI 很可能不符合将调用的 PTI 的要求，此时势必要解决 TaskParameters 在不同 DTI 间转换的问题。

2.4 非原子 PTI 与分布式 PTI

现实中的各种 PTI 往往互相依赖，例如与有限扩域相关的各种 PTI 时常会调用其基域相关的 PTI。对此可理解为，PTI 本身也包含计算需求。这些来自 PTI 的需求同样可用 TASK 表示(记为 subTASK)。据此可将所有 PTI 分为两类:不包含 subTASK 的原子 **PTI** 和包含 subTASK 的非原子 **PTI**。一般来说，对于同一 PT，非原子 PTI 开发更容易，性能也更有保证。除基本数据类型的某些基本运算外，现实中绝大部分 PTI 都是非原子 PTI。更重要的是，PTI 的 subTASK 属于典型的第一类 TASK——只关心结果的正确性和计算效率，而不关心为处理该 TASK 将使用哪种 PTI。每个 PTI 包含的 subTASK 本身具有稳定性，如果计算系统支持 PTI 使用第一类 TASK 的方式描述其 subTASK，则该 PTI 将不需随其 subTASK 相关 PTI 的更新做任何改变。这一点对系统的开放性有重要意义。

进一步地，各种分布式计算实现可被看作是全部或部分 subTASK 彼此独立的一类特殊非原子 PTI。例如[37]提出的分割/拼接模式可形式化地表述如下:分割/拼接式分布式 PTI 的 PT_Implemention 由分割器 Divider 和拼接器 Combiner 两部分组成，Divider 可根据 TaskParameters 分割出一系列彼此独立的 subTASK，Combiner 则将这些 subTASK 的计算结果拼接为最终结果，如图 2-1(a)所示。其中深色节点代表 subTASK，D 节点和 C 节点分别代表 Divider 和 Combiner。

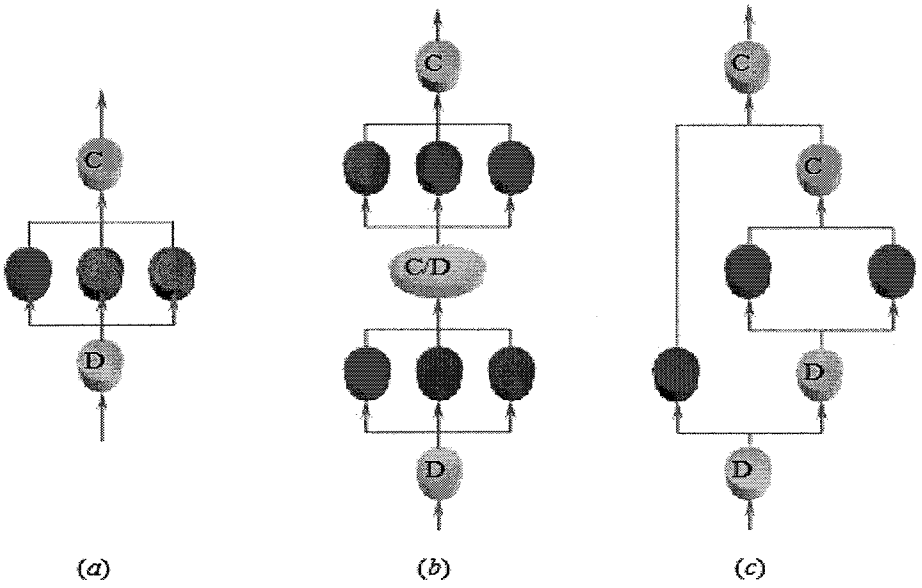

图 2-1 分布式 PTI 的各种计算流程

图 2-1(a)的单次分割/拼接适用于各种密码体制的暴力攻击，但更高级的分布式 PTI 则往往并不遵循这一模式，例如 DES 穷举攻击的分布式实现涉及两阶段分割/拼接，如图 2-1(b)。事实上，借助 subTASK 的概念，可将任意非原子 PTI 的计算过程表示为一个以分割/拼接计算和 subTASK 为节点的有向无环图(简称 DAG)，如图 2-1(c)所示。根据该 DAG 可直接确定计算过程可行的分布式执行方式。例如 DAG 中出度大于 1 的节点对应分割计算、入度大于 1 的节点对应拼接计算、无相互依赖关系的 subTASK 可并行执行、subTASK 的计算次序可归结为 DAG 节点的排序等。据此，可将基于 DAG 描述的非原子 PTI 作为分布式 PTI 使用。这一点对基础系统模型中分布式开发接口的设计有重要指导意义。

§3 系统模型设计

3.1 总体结构

基础系统模型的总体结构分为客户端和服务器两部分，如图 2-2 所示。其中，客户端既是来自用户的计算需求的接收者和直接处理者，也是分布式计算的基本工作单元。相应地，服务器是分布式计算的管理者，同时也维护着网络中所有的计算实现。

客户端包括密码计算库、解释环境和用户界面三个主要部分。其中，密码计算库由客户端拥有的全部计算实现组成，包括一个默认的公共基础库和若干可自由装卸的独立扩展包。作为密码计算库的最小集，公共基础库由计算系统的开发者负责实现和维护，其内容涵盖了密码计算中基本数据类型的 DTI 和基本运算相关的 PTI。公共基础库的存在保证了基础计算系统的完备性。独立扩展包则由来自独立开发者的若干扩展计算实现组成，其内容可以是新的 DTI/PTI，也可以是新的 DT/PT 定义。各个扩展包乃至公共基础库在地位上完全等同，内容互不干扰，统一接受 TASK 解析器的选择和调度，从而实现了系统的包容性。

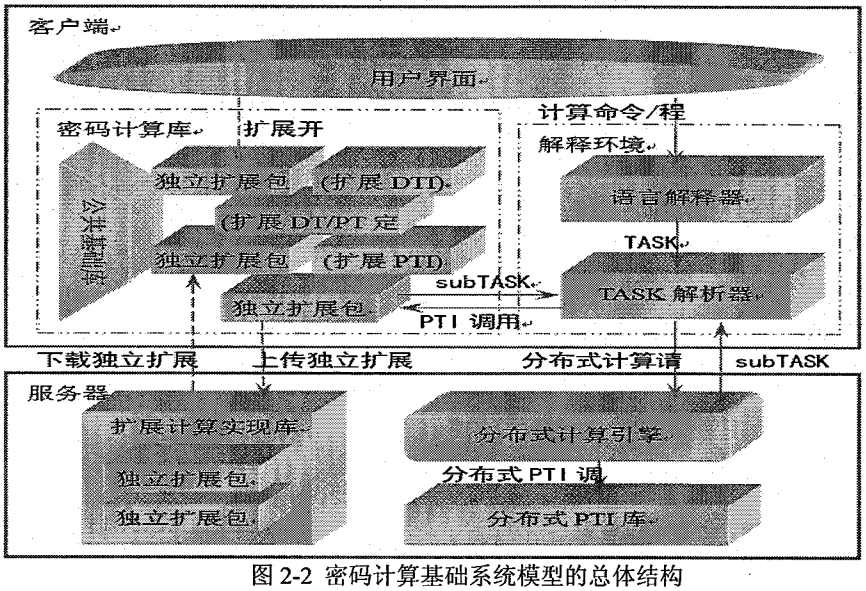

图 2-2 密码计算基础系统模型的总体结构

基础系统模型的总体结构分为客户端和服务器两部分，如图 2-2 所示。其中，

客户端既是来自用户的计算需求的接收者和直接处理者，也是分布式计算的基本工作单元。相应地，服务器是分布式计算的管理者，同时也维护着网络中所有的计算实现。

客户端包括密码计算库、解释环境和用户界面三个主要部分。其中，密码计算库由客户端拥有的全部计算实现组成，包括一个默认的公共基础库和若干可自由装卸的独立扩展包。作为密码计算库的最小集，公共基础库由计算系统的开发者负责实现和维护，其内容涵盖了密码计算中基本数据类型的 DTI 和基本运算相关的 PTI。公共基础库的存在保证了基础计算系统的完备性。独立扩展包则由来自独立开发者的若干扩展计算实现组成，其内容可以是新的 DTI/PTI，也可以是新的 DT/PT 定义。各个扩展包乃至公共基础库在地位上完全等同，内容互不干扰，统一接受 TASK 解析器的选择和调度，从而实现了系统的包容性。

解释环境接受用户界面接收到的计算命令和计算程序，之后由语言解释器提取出其中的 TASK，再由 TASK 解析器根据密码计算库的现有内容将 TASK 解析为合理的 PTI 调用并加以执行，计算结果最终通过用户界面返回给用户。在此过程中，PTI 包含的 subTASK 会被返回至 TASK 解析器再次解析。TASK 解析器支持用户和 PTI 以第一类 TASK 的方式表述其计算需求，使得各种计算实现不再与其使用者直接关联，可以被自由地更新和替换，从而保证了系统的开放性。图 2-2 中的实线箭头表明了可能的 TASK 处理步骤。

如果用户要求，或 TASK 解析器决定采用分布式计算方式，客户端将通过网络把该 TASK 发送至服务器。之后服务器的分布式计算引擎将在分布式 PTI 库中寻找可用的分布式 PTI，利用其把 TASK 分割为一系列 subTASK，分发到网络中的不同客户端分别计算。这些 subTASK 同样要经过客户端中 TASK 解析器的解析，其计算结果最终被上传至服务器由分布式 PTI 进行拼接，并将完整结果发回客户端。分布式计算引擎为分布式 PTI 的开发提供了通用的分割/拼接接口，同时也可将基于 DAG 描述的非原子 PTI 直接作为分布式 PTI 使用。

此外，网络中的所有服务器还共同负责全部计算实现的管理和维护。如果 TASK 解析器遇到根据密码计算库现有内容无法解析的 TASK 类型，将通过服务器自动检索和下载相关的独立扩展包。同时，用户也可通过客户端将自行开发的计算实现以独立扩展包的形式上传至服务器供他人下载使用。图 2-2 中的虚线箭头表明了计算实现可能的传递路径。

接下来的小节将分别介绍基础系统模型各个部分的关键设计，包括密码计算库中 DT 和 DTI 的组织方式、TASK 解析器的解析规则、分布式引擎的分布式 PTI 开发接口等。

3.2 DT 的组织

根据数学概念的天然层次关系和面向对象设计思想，基础系统模型以派生树的形式组织所有 DT，图 2-3 展示了其中代表性的部分。

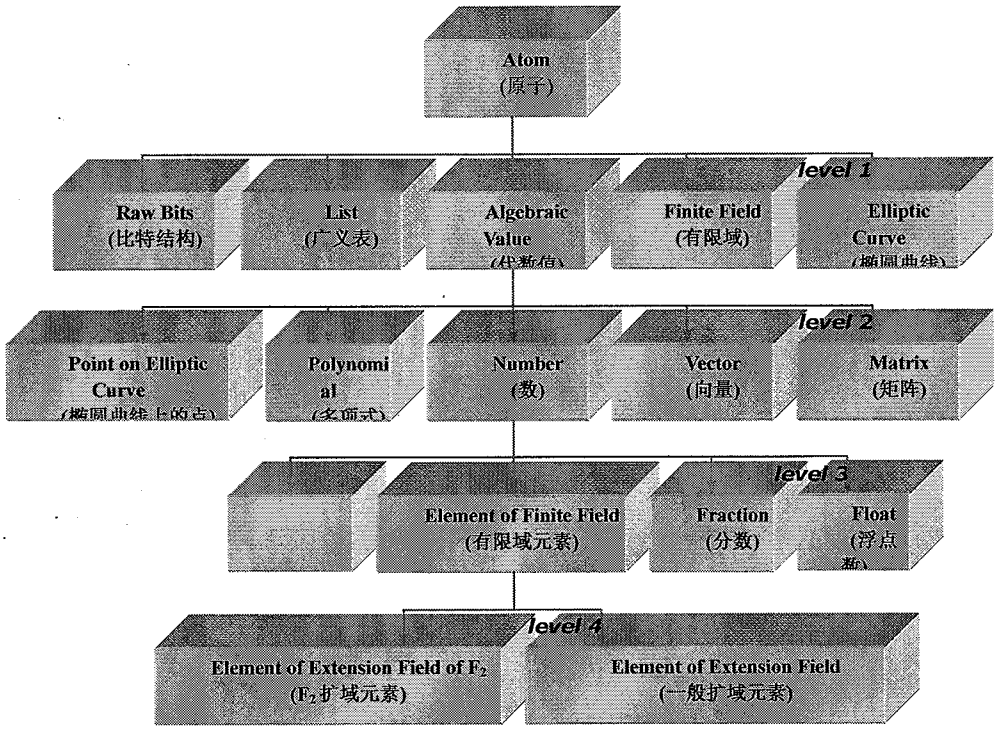

图 2-3 DT 的层状结构

树中由根到叶子每条派生路径上的 DT 对应的数学概念逐步明确。位于非终端节点的 DT 均为笼统的抽象概念，称为**虚 DT**，虚 DT 没有属于自己的 DTI。对于定义于虚 DT 之上的 PT，其 PTI 直接以该虚 DT 代替具体 DTI 作为参数要求。在 PT 的定义中使用虚 DT，将使其 PTI 可供更多实 DT、乃至未来由该虚 DT 派生的新 DT 使用。同时，位于终端节点的 DT 则已完全具体化，称为**实 DT**，实 DT 在计算系统中拥有至少一种 DTI。最后，为定义新的 DT，开发人员只需指定新 DT 在派生树中的父 DT，并提供一种主 DTI(参见 3.3 小节)及父 DT 要求的若干基本 PTI。

需要指出，PTI 的通用性与高效性之间存在矛盾。定义在虚 DT 上的 PTI 虽对其派生的所有 DT 可用，但对具体的某种实 DT 完全可能存在其他更高效的 PTI。例如有限域相关的 PT 大多定义于一般有限域元素类型之上(参见图 2-2)，但在密码学尤为关心的 F_2 扩域上，某些 PT 可能存在独特的解决办法。对此，基础系统模型支持扩展开发者根据实际需要在任意层次上定义 PT，而在 TASK 的处理过程中根据 DT 差异度(参见 3.4 小节)自动作出选择。

3.3 DTI 的组织

DTI 的多样性导致计算数据在不同 DTI 间的转换不可避免。传统计算系统通常硬性地将不同 DTI 视作不同 DT，或要求 PTI 自行完成 DTI 间的转换。前种做法人为割裂了 DTI 之间的天然联系，导致基于同一 DT 的计算实现不能互通。后

种做法则影响了 PTI 开发的独立性，同时带来大量不必要的转换。对此，基础系统模型采用星型拓扑组织同一 DT 的各种 DTI，以尽可能减少 DTI 转换带来的性能损失和附加开发要求，如图 2-4 所示。

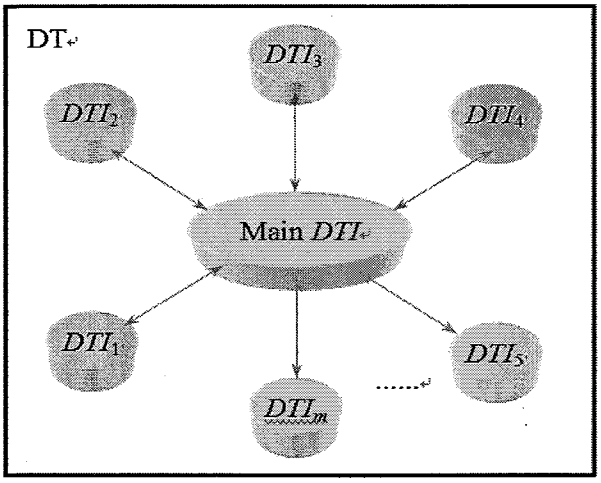

图 2-4 从属同一 DT 的 DTI 的组织

基础系统模型要求开发者在定义新的实 DT 时，必须为其提供一种 DTI 作为所有 DTI 转换的桥梁(称为主 **DTI**)。主 DTI 应提供完备的访问和构建接口。之后定义该 DT 的其他 DTI 时，只需解决其与主 DTI 间的相互转换，而不需顾及其他非主 DTI 的存在。同时，以主 DTI 为过渡，每次数据转换至多只涉及两次 DTI 间转换。事实上，对大多数 DT 而言，其在数学中最常用的表示方式对应的 DTI 即为主 DTI 的理想选择，大部分常规计算都适合在该 DTI 上进行。最后，数据在 DTI 间的转换遵循“必要时转换”的原则，仅在对当前 DTI 不存在理想 PTI 时进行(详见 3.4 小节的 TASK 解析规则)。

3.4 TASK 的解析规则

基于统一的解析规则，TASK 解析器对来自用户、非原子 PTI 和分布式计算引擎的 TASK 进行解析。对于第一类 TASK，通过 TaskType 和 TaskParameters 所属 DT 的匹配(可能涉及虚 DT)即可判断出任一 PTI 对该 TASK 是否可用。但如果同时存在多个可用的 PTI，关于进一步如何选择则涉及多种因素，例如各 PTI 来源的可靠性、版本信息、关于该 TASK 的 DT 差异度等。一个合理的选择策略应该是这些因素的量化综合。接下来以 DT 差异度为例，给出 TASK 的解析规则。

DT 差异度的定义如下:设 TASK 的 TaskParameters 各元素所属 DT 分别为 d_1， ...， d_n，

PTI 的 DTI 要求对应的 DT 为 d'_1， ...， d'_n，则该 PTI 关于该 TASK 的 **DT** 差异度为，

$$\max_k(level(d_k)\text{-}level(d'_k))$$

其中 *level(d)*表示 *d* 在 DT 派生树中的层数。PTI 关于 TASK 的 DT 差异度更小，意味着其定义在更具体的 DTI 层次上，对 DT 的特性也就利用得更充分，性能上的先天优势也就更大。

第一类 TASK 的解析规则如下:

(A). 排除密码计算库现有 PTI 中，PT_NAME 不同于 TaskType 的部分;

(B). 排除剩余 PTI 中 DTI 要求与 TaskParameters 所属的 DTI 不匹配的部分;

(C). 若剩余 PTI 为空，则从服务器下载可与该 TASK 匹配的 PTI;

(D). 计算剩余每个 PTI 关于该 TASK 的 DT 差异度，选取差异度最小的 PTI 集合;

(E). 从剩余 PTI 中选取 DTI 转换要求最少的一个;

(F). 根据该 PTI 的要求进行 DTI 转换，调用该 PTI。

第二类 TASK 的解析则简单得多:

(A). 检查当前密码计算库中是否存在 PTI_UID=TaskType 的 PTI;

(B). 若无，则联系服务器下载该 PTI;

(C). 根据该 PTI 的要求进行 DTI 转换，调用该 PTI。

作为开放性和包容性的代价，对第一类 TASK 的解析会带来额外的性能开销。但考虑到密码计算的规模特点，这一开销相较 TASK 本身的计算开销并不显著，我们的系统实现证实了这一点(参见第 4 部分)。

3.5 分布式 PTI 的开发接口

分布式计算引擎将\cite{grid1}设计的分割/拼接接口推广为更一般的形式。对于非穷举式的计算过程，分割和拼接操作往往交错进行，此时 Divider 和 Combiner 不再相互独立，必须统一到同一接口之下。此外，非原子 PTI 中的 subTASK 天然构成分布式计算的基本单元。对于以 DAG 方式描述的非原子 PTI，分布式计算引擎提供了内置的 Divider&Combiner 接口实现。在其作用下分布式计算引擎可直接将非原子 PTI 作为分布式 PTI 使用，从而大大简化了分布式 PTI 的开发过程。下面展示了统一的 Divider&Combiner 接口及其 DAG 通用实现。

SubmitTask(TASKParameters) ---提交 TASKParameters 1.生成 DAG 的初始部分(所有节点初始状态为“未分配”)
(subTASK , subTASKid) GetSubTASK() ---分割出一个 subTASK 1.返回前驱节点均为“计算完毕”状态的某一“未分配”节点对应的 subTASK; 2. 将该节点状态改为“计算中”.
divideFinished() ---检查分割是否结束 1.检查当前是否存在处于“未分配”状态的节点.
SubmitSubTASKResult(subTASKresult , subTASKid) ---提交一个 subTASK 的

结果	
	1.将该 subTASK 对应的节点状态改为"计算完毕";
	2.根据新得到的结果完成可能的拼接和新 DAG 部分的生成.
combineFinished()	---检查拼接是否结束
	1.检查是否当前所有节点都处于"计算完毕"状态.
TASKResult GetTASKResult()	---获取 TASK 结果
	1.返回最终结果.

§4 模型实现及性能表现

基于设计得到的基础系统模型，我们开发了 GNoMoN 密码计算基础平台(主页 http://www.is.iscas.ac.cn/gnomon，以下简称 GNoMoN)。各部分的开发环境分别为标准 C/C++(密码计算库和解释环境)、Visual Studio 2008(用户界面)和 Eclipse 3.0(分布式引擎部分)，代码量总计约 25 万行。

在密码计算库部分的开发中，我们主要采用继承、模板等面向对象设计技术批量地开发了包含到公共基础库中的 DTI 和 PTI。同时，使用反射(reflect)技术实现了独立扩展包的自由装卸。目前，GNoMoN 的公共基础库已汇集密码计算常用的近百种基本 DTI 和千余种 PTI，并通过独立扩展包提供了百余种分组密码、序列密码、公钥密码和散列算法的应用和分析计算功能。密码计算库迄今为止的积累过程证明，基础系统模型的开放式设计初步经受住了实践的考验。解释环境部分的核心是使用面向对象版本的 LEX/YACC 工具开发的一种面向密码计算的符号计算语言。下面展示了使用该语言编写的，AES 分组密码算法中字节替代操作的符号计算代码。

```
ByteSub := proc(input, e){
    A := matrix(8, 8, [[1, 0, 0, 0, 1, 1, 1, 1], [1, 1, 0, 0, 0, 1, 1, 1], [1, 1, 1, 0, 0,
0, 1, 1], [1, 1, 1, 1, 0, 0, 0, 1], \
        [1, 1, 1, 1, 1, 0, 0, 0], [0, 1, 1, 1, 1, 1, 0, 0], [0, 0, 1, 1, 1, 1, 1, 0], [0,
0, 0, 1, 1, 1, 1, 1]]);
    C := matrix(8, 1, [[1], [1], [0], [0], [0], [1], [1], [0]]);
    result := [];
    for (i:=1; i<=dim(input)[1]; i:=i+1) {
        for (j:=1; j<= dim(input)[2]; j:=j+1) {
            x := input[i][j]^254 ;
        X := vector(coor(x), 0) ;
        Y := A * X + C ;
        output := e^7 * Y[1][1] + e^6 * Y[2][1] + e^5 * Y[3][1] + e^4 * Y[4][1] +\
                e^3 * Y[5][1] + e^2 * Y[6][1] + e * Y[7][1] + Y[8][1];
        result := append(result, output);
```

```
        }
      }
      return matrix(dim(m)[1], dim(m)[2], result);
  }
```

该语言支持 MPI-like 的并行语法。开发者可直接使用其编写基于 DAG 描述的非原子 PTI，供分布式引擎使用。

TASK 解析器借助数据库技术维护密码计算库内容的实时信息，以实现 PT_NAME 和 PTI 的高速匹配。目前，TASK 的解析开销和计算开销相比微乎其微，对最终计算性能不产生实质影响。但考虑到解析开销主要源于可用 PTI 间的比较和选择，而目前密码计算库的内容尚不十分丰富，对于给定 TASK 往往只存在一个或少数几个可用的 PTI，因此对 TASK 解析器工作效率的检验有待密码计算库的长期积累。

分布式引擎部分的开发基于目前最有影响力的网格基础平台 Globus Toolkit 4(以下简称 GT4)。GT4 的体系结构保证了分布式引擎能最大限度地收集和利用各种异构计算资源。同时，分布式引擎利用 GT4 提供的安全基础设施 GSI 实现了资源、TASK 和用户的认证和授权。借助 GT4 的 OGSA 框架，分布式引擎按功能实现为一系列网格服务，服务之间采用 SOAP 协议和 XML 消息传输(下面展示了 TASK 的 XML 描述的例子)。由于服务之间的接口是标准化和平台无关的，分布式引擎的不同功能可以分布于不同的网络节点上，从而保证了分布式引擎的开放型。

由于密码计算需传输的数据量一般不大，我们选择了简单快速的 Base64 编码和 HEX 编码解决数据的二进制表示和信息隐藏问题。

分布式计算引擎以 P2P 方式组织计算网络中的所有服务器，并使用 DHT 技术(Distributed Hash Table)自适应地处理服务器的加入、离开和 PTI 的网络搜索等。DHT 技术使整个计算网络具有良好的健壮性和自组织能力。

```
<TaskData>
    <TaskInfo>
        <TaskType>
            <PTName> factor</PTName>
        </TaskType>
        <TaskParameters>
        <Item>
            <DT>"integer"</DT>
            <Data>694BF1F32AB9FD466CD</Data>
        </Item>
        </TaskParameters>
    </TaskInfo>
```

```
    <RelatedData>
        <TaskName>Test</TaskName>
        <SubmitedTime> 2009-08-19 17:03:41.14</SubmitedTime>
        <TaskID> AD69446BBFD1F3C29F6FF1ADA6BF64AB34FED3AF</TaskID>
        <RootTaskID> NULL</RootTaskID>
    </RelatedData>
</TaskData>
```

与国内外的分布式密码计算平台相比，如椭圆曲线离散对数的攻击，大整数因子分解， 破译有密码保护的文件的 DNA (Distributed Network Attack)平台，Gnomon 分布式密码计算平台更为通用，能够接受各类密码问题的问题实例。其采用 DHT 技术的环状拓扑结构具有耐攻击、高容错的优点。基于 Globus Toolkit 所提供的安全机制如身份验证、代码审核等进一步扩展了系统的安全性。同时，实现了一些基于 Gnomon 脚本语言的分布式库，来方便密码分析与设计人员的使用。

由于分布式计算的完整执行过程涉及计算系统的全部环节(基础计算库的性能、TASK 解析的效率、分布式引擎的工作效率等)，因此其性能直接反映了整个计算系统的运作效率。为此，我们组建了实验性的 GNoMoN 分布式计算环境。基于该环境，目前已取得的代表性成果包括，使用 Rainbow 攻击方法[46]和 5 台 Core Duo 2.4G PC，GNoMoN 能够以超过 99\%的概率在 5 分钟内破解任意长度在 14 字符以内的 Windows XP 用户口令。

进一步地，表 2-2 展示了素因子分解分布式计算实验的详细结果，从中可看到 GNoMoN 分布式计算的加速比接近线性。

表 2-2 QFRS PTI(大素数乘积分解[47])的性能测试

问题规模 (bit)	1 PC	2 PCs		5 PCs		16 PCs	
	时间 (s)	时间(s)	加速比	时间(s)	加速比	时间(s)	加速比
340	408.3	212.6	**1.92**	92.7	**4.4**	29.58	**13.8**
350	6,036.4	3,280.6	**1.84**	1,341.4	**4.5**	428.3	**14.1**
360	645,850	386,736	**1.67**	157,524	**4.1**	60,929	**10.6**

最后，GNoMoN 在密码学研究中已得到实际应用。目前基于 GNoMoN 已完成或开展中的独立密码研究课题包括:吴方法在密码分析中的应用，对序列密码算法的代数攻击，对散列函数的差分攻击等。

§5 小结

为满足密码学研究日益增长的的计算需求，我们设计了一种面向常规密码计

算的基础系统模型。该系统模型的设计充分针对密码计算的多样性等特点，致力于为常规密码计算提供一种完备和高效的解决方案。目前，基于该模型开发的密码计算基础平台 GNoMoN 的性能表现和应用现状初步证明了系统模型设计的开放性和高效性。

第三章 Gnomon 密码计算数学库

本章主要介绍 Gnomon 密码计算数学库的总体架构和设计思想.首先分析了在设计系统时要考虑的问题，如不同数据对象之见互操作问题、多项式系数的表示问题等，并给出了解决方案.第二部分介绍了密码计算基础平台的最核心的部分，多项式系统的设计.具体内容包括两种不同结构的多项式，分别用于不同情形的应用。最后描述了有限域系统的设计，最大的突破和进展在于实现了任意域的扩张和任意域上的多项式和元素的运算。这是其它数学软件所没有的功能。

§1 数学库对象的封装性与可扩展性设计

我们给出 Gnomon 数学库的核心类图，如图 3-1 所示：

1.1 对数据对象的封装性设计

1) 公共操作的封装——CAtom

抽象出每个子类都需要的一些基本操作，如显示（show），拷贝（copy），类型获取（type）等。由面向对象的程序设计思想可知，这些最基本的操作应该放在整个系统的基类，作为纯虚函数供子重载。我们定义 CAtom 类为所有类的父类，处于系统类图的最顶层。为所有的子类提供最基本的功能来继承，这些功能几乎所有的子类都可用到。

2) 对不同类型数据对象的封装——CAlgValue

CAlgValue 类是所有基本代数元素的父类。其子类具体包括数系统、有限域元素、多项式等。在 CAlgValue 类里定义了两个 CAlgValue 类对象的加、减、乘、除、模、幂、比较等基本运算，这些运算以虚函数实现.为了支持不同类型的数据之间的运算，在 CAlgValue 类下面的每个实体类都要重载这些虚函数。由于虚函数的重载机制，会自动对应到系统最底层具体的数据元素之间的运算。

3) 多项式的系数与常数项的封装——CElement

如何解决多项式的系数与常数项问题？我们知道，对一个多项式而言，其系数可以是整数、分数、小数；也可以是有限域元素；同时也可以是一个多项式。所以必须抽象出一种数据类型来包含这三种数据对象。

我们定义了 CElement 类，其主要包含递归多项式类（CRPoly）、有限域元素类、数系统类。因为这三种类型既可以作为多项式的常数项，也可以作为多项式的系数。所以我们把这三种类型单独拿出作为 CElement 类的子类。

CElement 类的成员函数主要是对 CAlgValue 类中虚函数的封装。这样当两个具体的 element 元素相互操作时候，由于虚函数的重载机制，会自动对应到系统最地层具体的数据元素之间的运算。

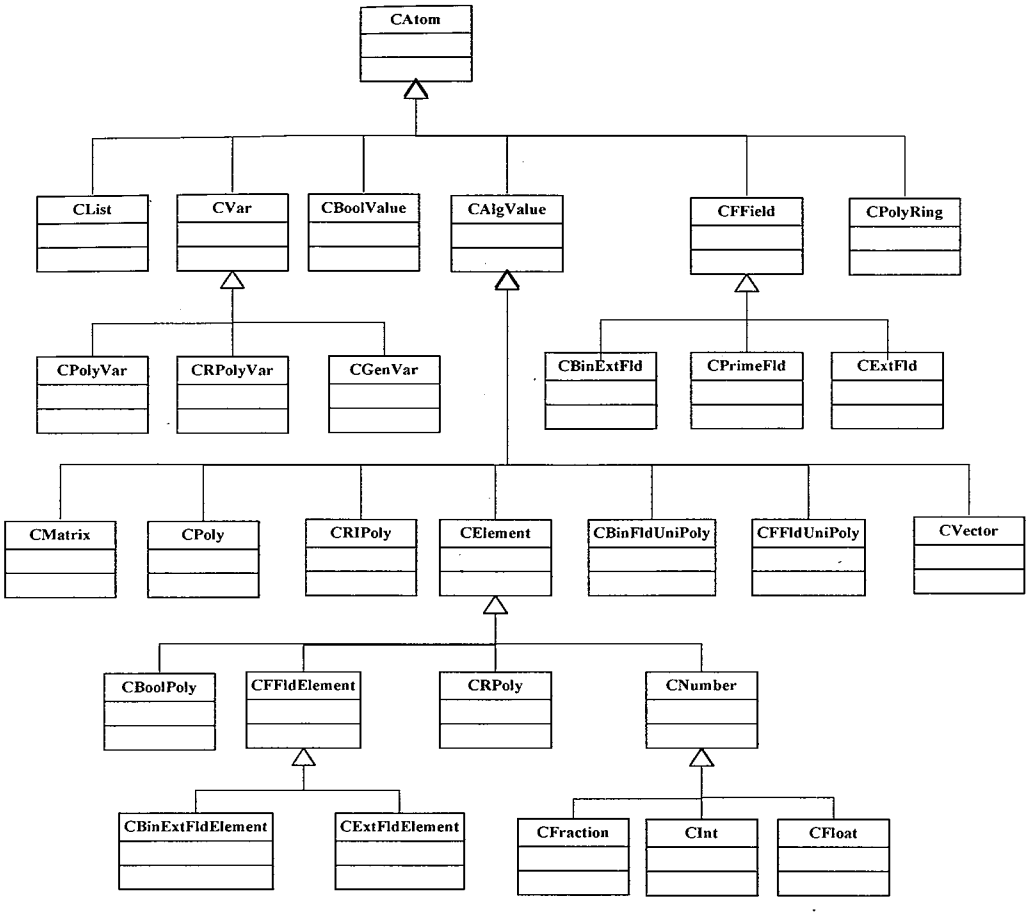

图 3-1 Gnomon 数学库的核心类图

4) 其余数据对象的封装

具体包括对有限域的封装(CFField)，对数系统得封装(CNumber)，对域元素的封装(CFFldElement)，对变元的封装(CVar)等。

1.2 不同数据对象混合运算的可扩展性设计

1)面向过程的设计方案

在一个具有多种数据对象（如整数、分数、小数、有限域元素、多项式等）的系统里，如何解决多种数据对象的相互运算问题？在面向过程的语言设计里，解决方案是定义一些全局函数，代码片段如下：

代码片段 1. 面向过程的解决方案

```
CAlgValueObj * mul(CAlgValue * lhs,  CAlgValue * rhs)
{
    if (lhs->type == CInt)
{
        if (rhs->type == CInt)
        {
            .../*Do something to process CInt * CInt*/
        }
        else if (rhs->type == CFraction)
        {
            .../*Do something to process CInt * CFraction*/
        }
else if
.....
}
```

这个解决方案的优点是浅显直接，容易理解。不过也存在巨大的缺陷:任何新的类型（如多项式）加入系统都需要更改这些 if...else...语句，即使只是忘记了一处，程序都将有错误，而且难于发现。这样的程序在 C 语言中已经很久一段历史了，本质上是没有可维护性的。它违背了开放封闭原则（OCP），即软件实体（类，模块，函数等）应该是对于扩展是开放的，对于修改是封闭的。依据 OCP 原则，以后再进行同样的改动时，只需要添加新的代码，而不必改动已经正常运行的代码。在很多方面，OCP 都是面向对象设计的核心所在，遵循这个原则可以带来巨大好处，如灵活性、可重用性以及可维护性。

2)虚函数与条件语句的结合

假设我们在实现系统时，先实现的是大整数乘法，在 CInt 类中，我们重载了 CAlgValue 类中的 mul 函数。首先判断 other 指针的类型，如果是大整数，则直接运算；对于其他类型，因为还尚未实现，直接调用 other->mul(this，true)，相当于用其他类型的数据对象去乘大整数，这其实是为后期加入的不知道的类型提供了“接口”，同时 reversed 设置为 true；这里 reversed 布尔变量有两个作用：一是为了支持某些类型（如矩阵乘）不满足乘法交换律的情况；二是为了作为调用的出口。同时，它也遵循 OCP 原则，即后期再增加新的类型，无需更改上面代码。

下面当我们实现 CFraction 类里的 mul 函数时，CInt 类中的 mul 不用修改，只要在 CFraction 类里的 mul 函数中加入对已实现的类型 CInt 的处理，代码片段如下：

代码片段 2. 虚函数和 if...else 语句混用

```
class CAlgValuel
{
public:
    virtual CAlgValuel * mul(CAlgValuel * other,  bool reversed = false) =
0;
    virtual ~CAlgValuel(){}
};
CAlgValuel * CInt ::multply(CAlgValuel * other,  bool reversed)
{
  if (typeid(*other) == typeid(CInt))
  {
      .../*Do something to process CInt * CInt*/
  }
  else if (reversed == true)
        throw UnSupportedOp("*");
  else
        return other->mul(this, true);
}
```

```
CAlgValuel * CFraction::mul(CAlgValuel * other,  bool
reversed)
  {
    if (typeid(*other) == typeid(CInt))
    {
      .../*Do something to process CFraction * CInt*/
    }
    else if (typeid(*other) == typeid(CFraction))
    {
      /*Do something to process CFraction*CFraction */
    }
    else if (reversed == true)
        throw UnSupportedOp("*");
    else
        return other->mul(this, true);
  }
```

下面是加入 CStr 和 CPoly 类型的代码片段：我们根据类型定义规则，CStr 和 CInt 可以做乘法运算其结果是相当于 CStr 的连接，CStr 和 CFraction 不可运算；CPoly 和 CInt 以及 CFraction 都可做运算，但不可与 CStr 做运算。

这里使用 typeid 即运行时类型识别(RTTI)的技术做动态类型判断，但 RTTI 的效率不高，首先一个 typeid 调用要做几次整型比较和一次取址操作，可能还有两次整型加法，这依赖于不同的编译器实现；再者就是类型重复判断，即如果没有找到 A×B 相匹配的处理，反转时 A 的类型信息丢失，需要在处理 B 时再次判断。其二就是每个类都必须知道他的同胞类，这在多层的继承体系中是很困难的。

代码片段 3. 虚函数和 if...else 语句混用

```
CAlgValue * CStr::multpily(CAlgValue * other,  bool reversed)
{
  const type_info & objectType = typeid(*other);
  if (objectType == typeid(CInt))
  {  ...
     /*Do something to process CStr * CInt*/
  }
  else if (reversed == true)
        throw UnSupportedOp();
```

```
CAlgValue * CPoly::multpily(CAlgValue * other, bool reversed)
{
  const type_info & objectType = typeid(*other);
  if (objectType == typeid(CInt))
  {  .../*Do something to process CPoly * CInt*/      }
  else if (objectType == typeid(CFraction))
  {  .../*Do something to process CPoly * CFraction*/  }
  else if (reversed == true)
        throw UnSupportedOp();
```

```
    else
        return other->multpily(this，true);
}
```

```
    else
        return other->multpily(this，true);
}
```

这种 if...else 过时的技巧在 C 语言中常导致错误，在 C++中它们也是很丑陋的。为了支持那种不能处理的未知类型，我们抛出了一个异常，但无法想象调用者对这个异常处理能够好多少。

3)对虚函数的模拟

注意到编译器通常创建一个函数指针数组（虚表）来实现虚函数，并在虚函数被调用时进行下标索引，这样就避免使用 if...else 链。如果我们自己实现一套这样的虚表，则使得比基于 RTTI 的代码效率更高，也限制了 RTTI 的使用范围。代码片段 4 中第一层使用编译器的虚函数，第二层使用自己模拟虚表的方法。

代码片段 4. 模拟虚表的方法

```
class CAlgValue
{
public:
    virtual CAlgValue * mul(CAlgValue * other) = 0;
    virtual ~CAlgValue(){}
};

class CInt : public CAlgValue
{
    typedef CAlgValue * (CInt::*MulFuncPtr)(CAlgValue * other);
    typedef map<const type_info*，MulFuncPtr> MulMap;

    static MulMap initMap();

    static MulFuncPtr lookMulFuncPtr(CAlgValue * other);
public:
    CAlgValue * mul (CAlgValue * other)
    {
        MulFuncPtr mptr = lookMulFuncPtr(other);
        if (mptr)
            return (this->*mptr)(other);
        else
            throw UnSupportedOp();
    }
    CAlgValue * mulInt(CAlgValue * other);
```

```
class CFraction : public CAlgValue
{
    typedef CAlgValue * (CFraction::*MulFuncPtr)(CAlgValue * other);
    typedef map<const type_info*，MulFuncPtr> MulMap;

    static MulMap initMap();

    static MulFuncPtr lookMulFuncPtr(CAlgValue * other);
public:
    CAlgValue * mul (CAlgValue * other)  {...}
    CAlgValue * mulInt(CAlgValue * other);
    CAlgValue * mulFraction(CAlgValue * other);

Fraction::MulMap Fraction::initMap()
{
        MulMap map;   map[&typeid(Int)] = &mulInt;
        map[&typeid(Fraction)] = &mulFraction;
    return map;
}
Fraction::MulFuncPtr Fraction::lookMulFuncPtr(Symbol * other)
{
    static MulMap mulmap(initMap());
```

```
    CAlgValue * mulFraction(CAlgValue * other);
    CAlgValue * mulStr(CAlgValue * other);
};
```

```
MulMap::iterator funcEntry =
mulmap.find(&typeid(*other));
if (funcEntry == mulmap.end())
    return NULL;
return funcEntry->second;
}
};
```

和方案 2 基于 RTTI 的方法类似，CAlgValue 只有一个处理乘法运算的函数，它实现了第一层虚函数，而每种乘法都有一个独立的函数处理，这里使用了不同的函数名字，是因为乘法运算有着相同的参数类型，必须使用名字来区分他们。为了实现第二层虚函数表，每个 CAlgValue 的具体实现类为 mul( )运算增加模拟虚表、模拟虚表初始化函数以及模拟虚表查询函数。

mul 函数需要将参数 other 的动态类型映射到一个成员指针，通过查询模拟虚表 mulMap 完成。initMap 函数用于初始化模拟虚表，仅需要在第一次查询模拟虚表时初始化。Mul 函数调用 lookMulFuncPtr 函数查找运算的实际处理函数。限于篇幅，省略了 CInt 类的 initMap()，mul()，lookMulFuncPtr()的实现。

上面仅以乘法运算为例，若将其他运算加入，只需要另外增加一个虚函数，为每一个数据类型增加一个与运算对应的模拟虚表即可。方案三程序可控性和可维护性都得到了保证，满足 OCP 设计原则，且计算效率高，灵活性强，是一种比较理想的方案。

§2 数系统的设计

在数系统的设计中，首先实现了三种基本的数对象。即任意精度整数（简称大整数）、分数、小数。大整数是数系统的核心。因为分数、小数其实都是在大整数的基础上实现的。为了节省内存，对于大整数必须引入引用记数。

1)大整数——CInt

对于任意精度整数计算，目前广泛使用的是 GMP[36]实现。很多知名数学软件的大整数运算引擎都是基于 GMP 来实现，如 Singular[50]、Trip [51]等。同样，Gnomon 的底层也是对 GMP 运算库进行封装，实现了大整数的各种数学运算、位运算、各种进制的转换等功能。

2)分数——CFraction

分数记录为分子(nu)和分母(de)，它们都是大整数.它的值等于 nu/de。由于分数不一定是最简的，所以我们在构造分数的时候，能化简的需要化间。在分数做运算的时候，有时两个分数的运算结果会是整数，这就需要在返回值类型上做相

应的处理。

3)小数——CFloat

小数由两部分组成，有效数字(mant)和以 10 为底的幂的指数(exp)，它的值等于 $mant\times10^{exp}$。为了方便定义实数的有效数字位数，可以通过全局变量 Digits 来实现，初始化 Digits=15，即有效数字 15 位。

4)数类——CNumber

有时我们需要操作一个数对象，这个数对象可以是大整数、分数、小数的任意一种。所以我们有必要实现一种虚结构来包含这三种对象。并提取一些公用函数放入该类中。我们定义了 CNumber 类来实现这个功能，包含一些如取上界（ceil）、下界（floor）等虚函数。

在数系统的基础上，实现了基于大整数的欧几里德算法、扩展的欧几里德算法、中国剩余定理、素数的产生与判定等数论算法；大整数，分数，小数的混合运算等。

§3 多项式系统的设计

3.1 设计目标

多项式系统是整个系统中最为重要的部分，因为很多高层算法如分解因式、Groebner bases、吴方法[52]求特征列等都依赖于底层多项式的高效实现。我们的目标是支持整数环、有理数域和任意有限域上的多变元多项式环，从效率上考虑，一种多项式结构显然不能满足上述目标。

为了便于描述，我们称多项式中带系数的项为 **term**；去掉系数的项为 **mono**。多项式运算本质上分两部分：一是计算每一个 term 的结果，二是对计算出的 term 进行排列。第一部分的主要运算是系数运算以及变元次数的加减(多项式乘/除)；第二部分的主要运算包括变元次数的比较。所以有必要针对不同的系数做独立的多项式结构；同时对于 mono 间的比较有必要用特殊的数据结构来加速。

另外，不同的算法对多项式的底层结构有不同的要求。比如，对于吴方法而言，其求解过程主要以消元为主，这样很自然的会想到用递归结构去存储多项式，类似的数学软件有 Trip、Pari[53]等；而对于 Grobner 基求解，在运算中会大量的用到求一个多项式的首项，然后算 s 多项式消去彼此的首项。显然这种应用更适合用展开结构去存储多项式。类似的数学软件有 Singular、 Magma[54]等。

在有限域中，我们最长用到的两种多项式是二元域上的单变元多项式类和素域与一般扩域上的多项式，在这两类多项式中，有很多基本数学运算，多项式求阶，多项式求根，多项式分解，求本原多项式等。由于有限域的特殊性，如果我们实现两种特定的数据结构去存储这两类多项式，必然会大大提高算法的效率。同时，密码学中最为常用的布尔多项式由于其结构的特殊性，更宜单独设计。

针对以上分析，我们设计了：

表 3-2 五类多项式

名称	描述
CPoly	非递归多项式，适合计算 grobner 基等应用.
CRPoly	递归多项式.适用于吴方法等应用.
CBoolPoly	布尔多项式
CBinFldUniPoly	二元域上的单变元多项式
CFFldUniPoly	素域和一般扩域上的多项式

后两个多项式类，我们在讨论有限域的设计时再详细论述。

3.2 多项式环结构——CPolyRing

为了有效提高多项式系统的运算速度，必须把系数域确定下来考虑。即我们首先要确定多项式所在的环，然后所有的运算都是在具体的多项式环上的运算。该环确定了多项式的变元，序、所在的域。所有特定的多项式多项式：整系数、有理系数多项式、素域上多项式、二元扩域多项式都由该环生成。这样特定系数的多项式运算应该由特定的算法来优化。

具体来说，我们定义了小素数域多项式类、大素数域多项式类、有理数系数多项式类；

其他域系数多项式类等，它们皆作为 CPoly 或 CRPoly 的子类。

系数分离的优点非常明显：比如小整数系数也用 CInt 类存储，效率必然差于系统类型 int；又如对特征为 0 的多项式而言，运算复杂性主要在系数算法上了[55]，故可引入中国剩余定理进行相关优化。

我们定义了多项式环结构——CPolyRing，环中最重要的成员是域、变量列表以及多项式的序。如定义 CPolyRing R=(GF_5，[x_1，x_2，x_3，x_4，x_5]，"lex")，则表示有限域 GF_5 上的含 5 个变元的多项式环，环中的多项式都是字典序的。

3.3 展开多项式——CPoly

1）CPoly 的总体结构

多项式由若干 term 的和组成，而 term 又是由系数乘 mono 组成。所以，我们首先确定 mono 的结构 。

一般来说，对于密码学应用，不会出现次数太大的变元次数，所以一个变元次数用一个字（unsigned int）来存，既浪费空间，又无益于效率的提高。所以我们希望把一个单项式存储在一个或多个字中，尤其在 64bit 机器中。如对于单项式 $x^iy^jz^k$，其变元序 $x>y>z$，设一个字为 32bit，我们可以如下方式存贮它：

$i+j+k$	i	j	k

图 3-2　mono 的存储方式

之所以记录变元的次数和，是因为在做次数字典/反字典等序比较 mono 时更高效。这样做的好处是 mono 间的比较与基本运算只需要一条机器指令即可完成。一个机器字中存储的 次数向量越长越好。这样更节省内存，而 cache 命中也会更高，一条机器指令会操作更多的变元次数。

代码片段 5. mono 间的乘法

```
CTMono *CTMono::mulMono (const CTMono *right, int expSize)
const
{

    int *result= new int[expSize];

    int *leftexp=this->expArray;
    int *rightexp=right->expArray;

    for (int i=0; i<expSize; i++)
        result[i] = leftexp[i] + rightexp[i];

    return CTMono(result);
}
```

对于一个变元的次数，分配多少位来存储是需要仔细考虑的，否则计算会导致位溢出错误。一般在计算前，可预先估计在计算过程中所出现的变元的最高次数，同时依变元数选择合适的位数来存储该变元的次数。同时，还需做溢出检查，一旦发现溢出，发出警告。然后换更多的位数来存储变元次数重新计算。如对于含有 10 个变元的多项式，若其变元均只在 GF_5 上取数值，则每个变元的次数不会超过 5。所以，我们可以用 5 个比特来存储每个变元的次数，14 个比特来存储 10 个变元的次数和。这样一个 mono 只用两个 32 位字便可存贮。

对于多项式来说，可以用 term 数组或 term 链表来表示。Singular 符号计算软件中的多项式结构即是 term 链表，其优点是访问首项快。缺点是整个多项式在内存中非连续存储，在运算过程中容易出现 cache miss。伴随着大量 term 的生成与释放，内存会出现大量碎片。所以使用 term 链表来存储多项式必须同时实现高效的内存池。

若把链表改成数组，则顺序访问多项式的每一项会很大程度上避免 cache miss。同时，整个多项式的 term 数组空间可一次性分配。且对多项式 term 数组空间的回收也可在 $O(1)$时间内完成。

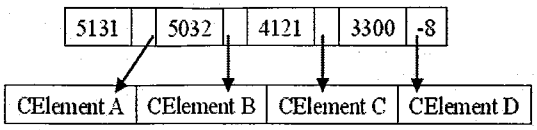

图 3-3a 非小整数系数多项式的结构

对于小整数系数（32bit），如多项式 $9xy^3z-4y^3z^2-6xy^2z-8x^3-5$，我们把系数与次数向量用连续的字节来存贮，整个多项式以一块连续的内存来存贮，见图 3-3b。而对于系数是大整数、二元扩域、一般扩域时，如多项式 $Axy^3z-By^3z^2-Cxy^2z-Dx^3$，这里 A~D 分别代表上述类型系数。系数为指向相应系数值的指针，所有的系数值也以数组来存放，由于操作系统在访问数据的时候一般会预取相邻位置的数据，系数数组更有效的利用了空间局部性。

5131	9	5032	-4	4121	-6	3300	-8	0000	-5

图 3-3 b 小整数系数多项式的结构

2）多项式相乘法算法

a.直接乘法

对于多变元多项式的乘法来说，简单乘法是这样的：设多项式 f, g 分别由 n, m 个项。对于 $f\times g$，我们通过计算 $\sum_{i=1}^{n} f_i g$ 得到结果。每次把 $f_i\times g$ 加到中间结果，需要 mono 比较 im-1 次，于是总共需要 $\sum_{i=2}^{n} im-n+1$ 次 mono 比较，即其时间复杂度为 $O(n^2m)$，这里时间复杂度以 mono 的比较次数来衡量，同时需要 O(mn) 的空间复杂度。传统的多项式乘法有两大缺点[56]：一是每次相加均产生中间多项式 $f_i\times g$，即使大部分中间多项式的项可以合并；二是每次计算均需要对多项式遍历，若其太大，无法全存入 cache，会更多的造成 cache miss。但对于 $m\times n$ 小于 100 的两个多项式，用一般乘法即可。

b.“分而治之”乘法

采用分而治之的方法，设多项式 f、g 分别有 n、m 个项。对于 $f\times g$，我们通过递归的计算 $\sum_{i=1}^{n} f_i g$，即先加其前 $n/2$ 项，再加其后 $n/2$ 项。而对两个子部分也分别递归的相加。于是，算法的 mono 比较次数 $C(n)\leqslant 2\,C(n/2)+mn-1\in O(mnlogn)$。这种方法的时间复杂度比直接乘低一个数量级。因为它只对大小相当的两个多项式合并。对于小的多项式相乘，有较高效率，但对于大的多项式，由于其递归次数增多，空间复杂度太大，为 $O(mnlogn)$，影响了计算效率。

c.“geobucket”乘法

目前公认的最快的算法是 geobucket[57]算法和 heap[58]算法。geobucket 算法的思路尽量用相同尺寸的多项式去运算。对于多项式 $f_1+f_2+f_3$，若 $\#f_1 \gg \#f_2 = \#f_3 = 1$。则从左向右加会有 $2\#f_1+1$ 次 mono 比较；从右向左加仅需要 $\#f_1+2$ 次 mono 比较。CoCoA 与 Singular [15]等软件中使用了该算法。

对于计算 $\sum_{i=1}^{n} f_i g$ 来说，实际是 n-1 次的多项式相加。不失一般性，设其某次多项式为 $p+q$，对于多项式 p 用若干“桶”的和来表示，即 $p=b_1+b_2+\cdots+b_k$，$\#b_i \leqslant 2^i$，即“桶” b_i 至多 2^i 项。对于 $p+q$，则用 p 的第 j 个“桶”去加 q，其中 $2^{j-1} \leqslant \#q \leqslant 2^j$。若其结果大于 2^j 项，则把 g 加到第 $j+1$ 个“桶”，直到其结果满足对“桶”长的限制为止。其时间复杂度是 $O(nmlognm)$， 空间复杂度是 $O(nm)$。

geobucket 算法非常适合在 Grobner 基的计算中的约化步骤，由于在约化过程中，中间多项式膨胀的越来越大，相应的每次处理的子多项式相对会较短，于是用桶算法去平衡每次两个操作对象的长度，会大大提高计算效率[57]。

d.“堆”乘法

本质上多项式乘法的速度更取决于 cache miss 而不是 CPU。内存的速度远慢于 cache，所以能用 cache 就用 cache。cache miss 所导致的运算延迟远大于操作数量的影响[59]。如对于多精度数的乘法而言，计算出系统得时间一般 50~100 个指令周期，远小于在内存中寻找合适的位置去存储它的时间 150~200 个指令周期[58]。L1，3 个指令周期; L2，20 个指令周期，所以尽量使多项式在 cache 里运算。由于堆算法更有效的使用了 cache，所以对一些大的多项式运算，其效率最优秀。

其计算过程如下：首先把 $f_i \times g$ 的首项存入堆中，然后从堆中抽取最大的 mono $f_i \times g_j$，若 $f_i \times g_j$ 被从堆中抽取，则我们向堆中插入 $f_i \times g_{j+1}$，如下例所示：对于多项式 f，g，我们首先把 f_1g_1， f_2g_1， f_3g_1 存入堆中，然后抽取最大的 mono，即 f_1g_1，计算其相应的系数，然后把结果存到 result 数组中。f_1g_1 被抽取后，向堆中插入 f_1g_2，此时堆中的最大元素是 f_2g_1，于是该元素被抽取，计算相应的系数后存到 result 数组，同时向堆中插入 f_2g_2。现在堆中的元素正是 f_1g_2， f_2g_2， f_3g_1，正如下图所描述。该算法直至把所有的项 $f_i \times g_j$ 插入堆中，并累加到 result 数组中后结束。

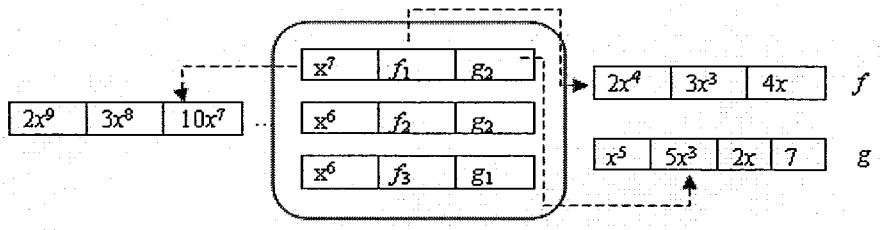

图 3-4　使用堆算法进行相乘运算的两个多项式

由于堆中仅含有 min(m，n)个元素，所以只需要 $O(log^{\min(m,\ n)})$的时间复杂度去获得堆中的最大元素；对于对于分别含有 m，n 项的多项式 f，g 相乘，其计算复杂度为 $O(mn\log^{\min(m,\ n)})$ 。heap 算法乘算法需要较少的“工作空间”，也即所有的中间 term 都是在堆中产生，最耗时的 mono 比较运算基本可在堆中完成。对于分别含有 m，n 项的多项式，其计算结果以空间 $O(mn)$为上限，而其工作空间仅需要 $O(\min(m, n))$，当 $nm<10^9$ 时，整个算法的计算过程可在 cache 中完成

[60] [61]。

3.4 递归多项式——CRPoly

1）多项式结构

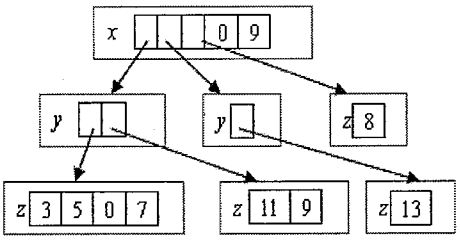

图 3-5a 递归多项式的稠密表示

递归稀疏表示不利于变元很少但次数很大的多项式，如 $y+(1+x)^{1000}$，递归稠密表示的优点是快速定位特定次数变元的系数，缺点是不利于表示次数很高但很多系数为 0 的多项式，如 $1+x^{1000}$

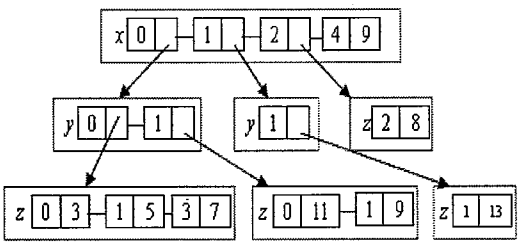

图 3-5b 递归多项式的稀疏表示

2）递归多项式相乘

FFT 算法应用到递归多项式乘，$a(x_1,\ldots,x_v)$，$b(x_1,\ldots,x_v)\in R[x_1,\ldots,x_v]$，可把多项式看作 $a(X)\in R[x_1,\ldots,x_{v-1}][x_v]$，即多项式的不定元是 x_v，而其余 n-1 个变元为系数变元。当两个递归多项式相乘时，遇到两个系数相乘时，就递归调用乘算法自身，直到系数为环 R 中的元素为止。由于递归表示，再也不需要 mono 的比较。

令 $C(n,v)$为两个稠密递归多项式相乘的步骤数，n 为每个多项式的次数，v 为多项式的变元数。则 $C(n,v)=(n+1)^2$ 次系数多项式乘+ n^2 次系数乘积相加 $=(n+1)^2C(n,v\text{-}1)+n^2(2n+1)^{v-1}$，这里 $C(n,0)=1$。则 $C(n,v)$以 $O(n^{2v})$为上界。

对于递归多项式相乘，一种常用的加速方法是 Kronecker 算法。它把乘法中的两个多变元多项式转化为单变元多项式后，利用 FFT 或 Karatsuba 等快速单变元多项式相乘算法去算出结果，然后再把结果转化为多变元多项式。

举例来说，设 $a(x,y)=(2y+1)x+(-y+2)$，$b(x,y)=(y+3)x+(4y-3)$，由于在 $a\times b$ 中，y 出现的最高次数 2，则令 $x=y^3$，于是，得到 $a'(y)$和 $b'(y)$，其中 $a'(y)=2y^4+y^3-y+2$，$b'(y)=y^4+3y^3+4y-3$，

于是可以用 FFT 算法去计算 $a'(y)\times b'(y)=2y^8+7y^7+3y^6-7y^5-3y^4+3y^3-4y^2+11y-6$，用该结果反复除 y^3，直到商为 0，则最终获得：

$(2y^2+7y+3)x^2+(7y^2-3y+3)x+(-4y^2+11y-6)$，这也即 $a\times b$ 的结果。对于 Kronecker 算法，同样令 $C(n, v)$为两个稠密递归多项式相乘的步骤数，n 为每个多项式的次数，v 为多项式的变元数。则 $C(n, v)$以 $O(vn^v log^n)$为上界[62]。

同样，对于不同的系数域，应采用不同的数据对象来分别实现。

§4 有限域系统的设计

有限域系统是密码库中最重要的部分之一。整个有限域系统的设计分为域的设计、域元素的设计、域上多项式的设计。

为了提高运算效率，对于二元扩域和素域使用了专门的数据结构进行实现。在这两种有限域的基础上，实现了任意域的扩张和任意域上的多项式和元素的运算。 扩域用基域上的不可约多项式和定义元来确定。扩域的元素使用多项式基表示，有限域上的多项式用系数元素的数组表示。

4.1 域的设计

素域和二元一次扩域是密码学中最常用的两种域，同时因为这两种域和一般域相比有较简单的表示形式。那么很自然的应该把这两种域单独进行设计。这样不仅简化了设计， 而且容易采用一些针对特殊结构的优化以加速常用的运算。除此之外，还有在这两种有限域的基础上的扩域，这样这三种域便组成了完备的有限域体系。

为了能对这三种域进行抽象，我们还设计了三种域类的基类。

针对以上分析，我们设计了二元一次扩域（CBinExtFld）、素域类（CPrimeFld）、一般扩域类（CExtFld）以及上述三种域类的基类（CFField）。结构如图 3-6

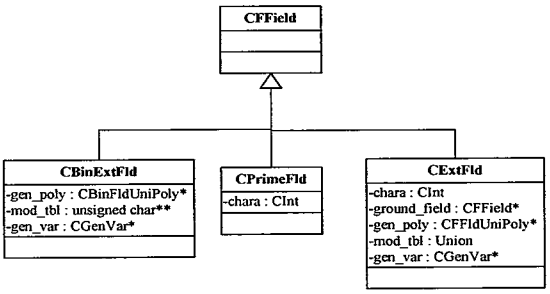

图 3-6　Gnomon 有限域的继承关系

1）二元一次扩域类 CBinExtFld

二元扩域类 CBinExtFld 由三个数据成员组成。为了生成二元一次扩域，首先要定义该域上的不可约多项式 gen_poly，其类型是 CBinFldUniPoly，将在 5.3 节介绍。同时需要该域的定义元 gen_var，类型是有限域上定义元 CGenVar类的指针。最后根据给定的不可约多项式计算模计算表 mod_tbl，用于加速域上的运算。因为域上的任意两个元素相乘，其都要经过模该域上的不可约多项式这一步。模计算表中存贮的是在模运算中会用到的一些多项式，对二元扩域来说，这些多

项式用位数组来存贮。

2）素域类 CPrimeField

特征为大整数的素域类 CPrimeField 中仅一个数据成员，即大整数 chara，表示素域特征。这里在生成素域时，要对大整数的素性进行判定。

3）一般扩域类 CExtFld

一般扩域类 CExtFld 由五个数据成员组成。首先是素域的大整数特征 chara；接着是扩域的直接基域 groundField，类型为抽象基域 CFField 的指针，即表示直接基域可以是素域、二元一次扩域或一般扩域的任一种。然后是扩域相对于直接基域的生成多项式 genPoly，为基域上的首一不可约多项式。同时给出该域的定义元 gen_var。

同样，为了加速域上的运算，我们生成了该域的不可约多项式生成的模计算表 modTable。模计算表的类型是 CFldElementArray 将在 4.2 节介绍。

4）三种域类的抽象基类 CFField

CFField 是上述三种域类的基类，通过提取各种子域类的公共特性得到，在该类中实现各种子域类所共有的操作。这部分所列的操作基本上是在每一种子域类中都提供的操作，用虚函数的机制实现，如求域的特征、域元素个数、子域判断等。

4.2 有限域上的元素

有限域元素可分为素域元素、二元一次扩域上的元素和一般扩域上的元素。之所以把二元一次扩域上的元素单独进行设计，是从效率上的考虑。 对于素域上的元素，也即是整数。而对于一般扩域的元素，其结构具有递归性，因为其元素可以从基域一层层扩展而来，所以可用一种递归的数据结构来描述。而最基域必定是素域和二元一次扩域。

通过提取各种子类的公共特性得到上述两种元素类的基类，在该类中实现各种子类所共有的操作。针对以上分析，我们设计了二元一次扩域上的元素类（CBinExlFldElement ）、素域和一般扩域上的元素类（CExtFldElement）以及有限域上元素的抽象基类（CFFldElement）结构如图 3-7：

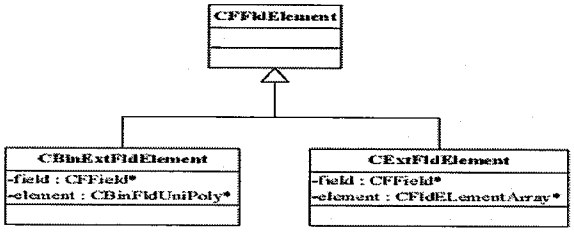

图 3-7 Gnomon 域元素的继承关系

1）二元一次扩域上的元素——CBinExtFldElement

二元一次扩域上的元素 CBinExtFldElement 由两个数据成员组成。首先是代表域元素的多项式(使用多项式基) element，类型即为二元域上的单变元多项式类

CBinFldUniPoly 的指针，同时给出该元素属于的域 field，类型为指向二元扩域的指针。

我们可以用 0、1 串来构造二元一次扩域上的元素，例如：

代码片段 6. 用 0、1 串来构造二元一次扩域上的元素

```
CBinFldUniPoly *p = new CBinFldUniPoly("111"); // p is defined as x^2+x+1
CGenVar *var = new CGenVar("x"); // var is the generate element
CBinExtFld *fld = new CBinExtFld(var,  p);
CBinExtFldElement e( "11",  fld); // e is set to  "x+1" };
```

首先生成该二元一次扩域的不可约多项式 p：x^2+x+1，然后以该多项式以及生成元 x 定义二元一次扩域 fld。最后使用 0、1 串构造一个二元域上的多项式 $x+1$。如果构造出的多项式的次数大于生成域的不可约的多项式次数， 则使用不可约多项式对它进行约化。

2）素域和一般扩域上的元素——CExtFldElement

CExtFldElement 可以表示素域上的元素和一般扩域的元素，由两个数据成员组成。首先是元素所在的域（为奇素域或一般扩域）的指针 field。其次是元素的值 element，类型为元素值类 CFldElementArray 的指针。CFFldElementArray 是一个具有递归性的数据结构。

一般扩域上元素为定长稠密数组表示，数组长度为元素所在域的生成多项式的次数。一般扩域上的元素应该用元素所在域基域上的次数小于生成多项式的多项式表示。元素的值若为素域上元素，则为大整数指针强制转换为元素值类指针；否则为代表基域上元素值类的指针。CFldElementArray 的结构如下图：

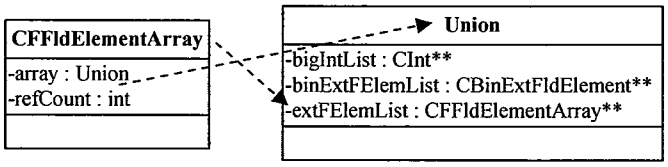

图 3-8　一般扩域上元素的递归关系

CFFldElementArray 为一般扩域上的元素值类。为扩域元素去掉域信息的部分得到，既可用于表示一般扩域元素的值，又可作为多项式的系数，也是一般扩域模乘法表的组成元素。其中，array 为联合结构，其成员或为 bigIntList，表示大整数指针数组，用于奇素域上的多项式或一次扩域上的元素的内部表示；或为 binExtFElemList，表示二元一次扩域上的元素指针数组，用于二元一次扩域上的多项式或二元二次扩域上的元素的内部表示；或为 extFElemList 指针数组，用于一般扩域上的元素或多项式的内部表示，实现了有限域的任意次扩域。

代码片段 7. 用 Gnomon 脚本语言生成 GF(11)上的二次扩域元素

```
>>F:=field(11);
"F" is set to FIELD: GF(11).

>>p:=randirdc(3，x，F);
"p" is set to x^3+3*x^2+7*x+4.

>>F1:=field<a>(F，p);
"F1" is set to FIELD: GF(11)<a^3+3*a^2+7*a+4>.

>>p1:=randirdc(3，x，F1);
"p1" is set to x^3+(2*a^2+2*a+8)*x^2+(10*a^2+2*a+10)*x+3*a^2+5*a+8.

>>F2:=field<b>(F1，pp);
"F2" is set to FIELD:GF(11)<a^3+3*a^2+7*a+4>
<b^3+(2*a^2+2*a+8)*b^2+(10*a^2+2*a+10)*b+3*a^2+5*a+8>.

>>e:=rand(F2);
"e" is set to (10*a^2+3*a+8)*b^2+(2*a^2+6*a+10)*b+4*a^2+9*a+2.
```

如代码片段 7，首先定义域 F 为 GF_{11}，然后生成 F 上的 3 次不可约多项式 p，以定义元 a 和不可约多项式 p 生成 F 的扩域 F_1。接着生成 F_1 上的 3 次不可约多项式 p_1，以定义元 b 和不可约多项式 p_1 生成 F_1 的扩域 F_2。最后在 F 的二次扩域 F_2 上，生成随机域元素 e。元素 e 的计算机表示见图 3-9：

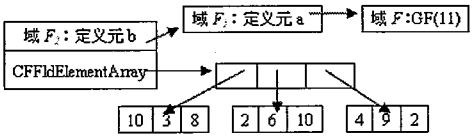

图 3-9 元素 e 的结构图

3）有限域上元素的抽象基类 CFFldElement

上述两种元素类的基类，派生自 CElement 类。实现对域元素的封装。

4.3 有限域上的多项式

与域元素的设计相对应，域上的多项式包括二元域上的单变元多项式（CBinFldUniPoly），素域和一般扩域上的多项式（CFFldUniPoly）。结构如图 3-10

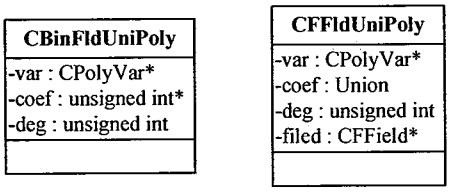

图 3-10　域上的多项式

1）二元域上的单变元多项式——CBinFldUniPoly

数据成员中 degree 记录多项式的次数；coef 记录系数的定长数组，其长度为不可约多项式的次数；var 记录多项式的变元；ref_count 引用计数变量记录有几个多项式共享同一个系数数组。

该域上单变元多项式使用比特数组表示。数组中的每一个比特表示多项式中的一项。这种表示有两个优点：首先这种表示节约空间，一个 1023 次的多项式只需要长度为 32 的 unsigned int 型的数组。其次，这种表示将大部分代数运算归结为位运算(移位，异或等)，从而提高了运算速度。

2）素域和一般扩域上的多项式类 CFFldUniPoly

数据成员中 field 为多项式所在的域指针（为奇素域、二元一次扩域或者一般扩域)；var 记录多项式的变元；degree 记录多项式的次数；coef 为 Union 类型，是记录系数的变长稠密数组。数组长度为多项式次数加一。

一般扩域多项式与一般扩域元素的表示类似，不同之处在于元素数组长度为扩展多项式的次数，若生成该元素的多项式长度不足者前面用零元补齐；而多项式数组的长度为多项式次数加一。

§5 小结

本章把Gnomon系统底层数学库中最重要的模块部分作了较为细致的说明，给出了系统的体系架构及其关键数据结构的设计思想。下一步的工作主要从两个方面入手:一是进一步提高算法的效率，具体包括多项式运算算法，有限域运算算法等；二是利用Gnomon密码计算数学库所提供的功能进行一些相关密码研究应用。目前基于Gnomon已完成或开展中的独立密码研究课题包括:吴方法在密码分析中的应用，对序列密码算法的代数攻击，对散列函数的差分攻击等。

还有部分模块如线性代数模块、布尔多项式模块、椭圆曲线模块、线性递归序列模块等没有进行说明。一方面因为这些模块在我们整个数学库的上层，自成体系，可以单独进行描述；另一方面由于这些模块我们还在继续开发，功能在不断增加。我们会在未来不断完善 Gnomon 系统的功能与性能，希望能够为密码学研究和教学提供一个方便的工作实验环境和应用开发环境。

第四章 基于F5算法的超定方程组Groebner基计算的改进

非线性超定方程组的求解技术对于密码分析具有重要的现实意义。目前，现有求解算法巨大的空间需求是制约该技术取得更多成果的主要瓶颈。本章首先针对当前最先进的非线性方程组求解算法 F5，分析了其在超定环境下的缺陷，即未能充分利用超定条件，导致大量计算冗余;我们进而分析了F5算法的矩阵版本（MatrixF5）在超定环境下的优势及不足。在二者基础上，我们设计得到了一种求解非线性超定方程组的新方法——F5D 算法。F5D 算法完全继承了 F5 和MatrixF5算法在方程组求解问题上的优点，同时分别避免了F5算法带来的计算冗余和 MatrixF5 算法过大的存储需求。实验结果表明，在超定环境下，F5D 算法可被用作F5算法和MatrixF5算法的替代。

§1 引言

在密码学领域中，由于密码体制的破解者应被假设能获取大量的明密文对，从而拥有比破解密钥所需要的更多的信息量，此时的代数攻击相当于在超定环境下执行方程组求解，因此超定方程组的求解技术对于密码分析具有重要的现实意义。2002年，J.-C.Faugère 提出的F5算法[11]是目前最先进的非线性方程组求解算法。F5算法通过以多项式递增的顺序计算各方程的Groebner基完成方程组的求解。其在计算过程中，通过Rewrite准则和F5准则来确保不会引入无效计算(即约化为零的关键对)。然而由于这种计算顺序并不能利用超定条件，导致该算法并不适合超定方程组。

2003年，J.-C.Faugère 又提出了F5算法 的Matrix版本（简称MatrixF5）[63]。 MatrixF5 算法按照次数递增的顺序进行计算，从而可以较好地利用方程组的超定性质。但其缺点同样明显：MatrixF5算法无法如F5算法般预知未来的计算内容，而只是简单地计算所有可能性，所以会包含大量冗余，使得算法空间性能表现不佳。

本章采用了 MatrixF5 算法的总体思路，即按次数递增的顺序进行计算，但在计算过程中采用了类似F5算法的基于关键对的计算模式。从而既充分利用了方程组的超定性，又避免了MatrixF5算法空间膨胀过快的缺点，我们称之为F5D算法。为此，如何实现 Rewrite 准则和 F5 准则的等价平移是本章要解决的主要问题。

§2 现有Groebner算法对于超定方程组求解问题的不足

2.1 F5算法不足

对于一个方程组系统，一般用求解过程中涉及的所有多项式的最高次数来衡量求解复杂度。F5 算法不适合求解非线性超定方程组，其缺陷在于，没有充分

利用所有的多项式，因为它是按照多项式递增的顺序去计算 Groebner 基。比如对于超定方程组$\{f_1, \ldots, f_m\}$，在计算前三个多项式 f_1，f_2，f_3的时候，假设它们的解不是零维的，其 Groebner 基会膨胀得很大，导致中间多项式次数升得很高。而在加入后续多项式时，有可能所有方程的解是零维的，Groebner 基逐渐缩小。这样，本该提前约化的多项式却没有约化，造成了 Groebner 基计算过程中多项式的次数升得过高，增加了求解复杂度。

举例来说，用 F5 算法求解 10 变元 20 方程的 HFE 公钥方程组，假设该 HFE 公钥方程组的中心方程的次数为 40，由 J.-C. Faugère 对 HFE 公钥方程组的求解分析中可知[63]，其次数达到 4 便可以求解出该方程组的所有根。而 F5 算法没有充分利用超定的特性，其次数上升远超过 4。具体参见表 4-1，其中 f_1，…，f_{10}为域方程，首项两两互素，不会产生新的多项式，所以，从 f_{11}开始。

表 4-1 F5 算法对 HFE10 求解过程中的次数变化图

加入 fi 后	11	12	13	14	15	16	17	18	19	20
次数上升至	6	8	9	10	11	10	7	7	7	7

实际上，对求解出方程组系统 Groebner 基所要达到的最低次数的分析是建立在 MatrixF5 上的，也就是说，对 HFE 方程的求解用的是 MatrixF5 算法。由于 F5D 算法是由 MatrixF5 结合原始 F5 算法演变而来，所以下面分析 MatrixF5 算法。

2.2 MatrixF5 算法的分析

MatrixF5 的思想来源于 D. Lazard 在 1983 年的工作[64]。

定义 1[64] 设 K 为有限域，$P=K[x_1, x_2, \cdots, x_n]$为一多项式环，$f_1, \cdots, f_m$为 P 中的齐次多项式。构建矩阵 $M_{d,m}$:其行为按给定项序从大到小排列的所有次数为 d 次的单项式的系数，而这些系数是这样形成的：对每一行的 f_i（$1 \leqslant i \leqslant m$)乘所有的次数为 $d-d_i$的单项式，其中 d_i表示 f_i的次数。该矩阵被称为 Macaulay 矩阵。

引理 1[64]以次数递增的顺序对 Macaulay 矩阵 $M_{d,m}$进行高斯消元（$d>0$），当 d足够大时可计算出 G 得到 Groebner 基。

如果仅仅根据引理 1 去计算 Groebner 基，一个明显缺点就是当方程组生成高次 Macaulay 矩阵的时候，没有利用之前得到的低次 Macaulay 矩阵的消去结果。比如当方程组生成 $M_{d,m}$时，对于 $M_{d-1,m}$中所有次数为 $d-1$ 的多项式，在其乘以变元 x，y，z 等后，其中若 $x*f_i$被约化为 f_j;而在生成 $M_{d+1,m}$时候，f_i依然要乘以 $x\hat{}2$，$x*y$，$x*z$ 等二次单项式，显然，对于 $x\hat{}2*f_i$来说，不如 $x\hat{}2*f_j$能更充分利用 $M_{d,m}$的计算结果。

定义 2[65] 对于 Macaulay 矩阵 $M_{d,m}$中的多项式 $mj*f_k$(mj 为任意单项式，1

$\leqslant k \leqslant m$) 如果约化为 f_j，则在生成 $M_{d+1,m}$ 时，$x_i * m_j * f_k$（$1 \leqslant i \leqslant n$）可被 $x_i * f_j$ 取代。即 $x_i * m_j * f_k$ 不必再添加到矩阵里，该准则被称为 Rewrite 准则。

特别的，对于非正规的多项式输入，若 $M_{d,m}$ 中的多项式乘以一个变元使次数升一，在高斯消元后，有约化为零的多项式，则可直接将其删除。这样，在生成 $M_{d+1,m}$ 时，这些多项式就不会再参与计算。

上述方法虽然利用了低次 Macaulay 矩阵的计算关系，但是却引入了另外一个缺点。比如对 $M_{d-1,m}$ 中的 f_i，在生成 $M_{d+1,m}$ 时，f_i 可能会出现先乘 y 再乘 x 和先乘 x 后乘 y，从而形成了冗余的行。为了避免这种情况，必须通过某种机制以记录当前多项式是从哪个多项式乘哪个单项式而来。

定义 3[66] Macaulay 矩阵中的每一行有二元组（m，i），其中 m 为单项式，i 为多项式的标签，表明该行是由 $m * f_i$ 而来，称（m，i）为该行所代表的多项式的签名。

在 MatrixF5 中，通过使 f_i 按照变元序从小到大逐个乘所有变元来解决上述问题。这样冗余的多项式便不会加入到矩阵中。这种一次乘一个变元使多项式次数升高的策略实则为 Rewrite 准则的体现。

引理 3[66] 对于 Macaulay 矩阵 $M_{d-di,\ i-1}$ 中的多项式 f_j（$j<i$），如果其首项为 t，则在 Macaulay 矩阵 $M_{d,i}$ 中签名为（t，i）的多项式是冗余的。该准则被称为 F5 准则。

定义 4[67] 在使用 MatrixF5 算法计算 Groebner 基的过程中，为得到 Groebner 基所必须达到的最小次数记为 D_{reg}。

在具体应用该算法时，可以先估算 D_{reg} 的值。这样，算法计算到 D_{reg} 次时即可得到 Groebner 基。

下面给出了完整的 MatrixF5 算法。

算法 1. MatrixF5(***F***，***D***)

输入:环上多项式序列 F，要算到的次数 D;

输出:F 的 Groebner 基

```
1.basis = null
2.For   d=1 to D
3.    For f_i in F
4         If   deg(f_i)==d
5.              add_signature( (1， f_i)， f_i) ， M[d].append(f_i)
6               Continue
7.        For f in M[d-1] with get_signature(f)==(*， f_i)
8.              m， fi =get_signature(f)
```

```
9.          For x in variables
10.             If x< max(variables(m))   Continue
11.                 If t in M[d-deg(f_i)]   with LM(t)==x*m
12.                 m_2, f_j=get_signature(t)
13.                     If j<i   Continue
14.             Add_signature((x*m, fi), x*f)
15.                 M[d].append(x*f)
16.     M[d]=" M[d]高斯消元后所有非 0 多项式"
17. basis=" M[d]中所有多项式", 1≤d≤D
18. Return basis
```

2.3 MatrixF5 算法的不足

MatrixF5 算法的优点在于按次数递增的顺序去计算 Groebner 基，尽可能的对低次的多项式优先进行约化，使得使其求解出 Groebner 基时算法所必须上升的最小次数比 F5 算法低得多。对于文献[66]中的例子，如表 2 所示，F5、F4 算法为得到结果，次数均要升到 18，而 MatrixF5 在次数升到 12 便可得到 Groebner 基。

MatrixF5 算法虽然也利用了 F5 的 Rewrite 和 F5 准则，但还是引入了大量无用的对。主要原因在每次生成新的 Macaulay 矩阵时，因为无法预知在约化过程中究竟要用到哪些多项式，所以 MatrixF5 算法把所有可能在约化过程中用到的多项式都放入了矩阵，所以 MatrixF5 算法的空间需求增长很快，其空间效率远不如基于关键对的 F4 算法。如表 2 所示，当 MatrixF5 计算至次数为 11 时，矩阵已经膨胀到 1，349*1，362，远大于使用 F4 算法中的最大矩阵 43*52。而 F4 在约化过程中仅考虑关键对，其缺点是有很多关键对最终会约化为零，造成了不必要的计算。这也是 F5 算法的 Rewrite 和 F5 准则取得的进展。尽管 F5 算法也只考虑关键对，但如 2.1 节中所分析，它不适合密码学里求解大规模超定方程组的计算。这里 F5 算法的矩阵是在约化中引入的 F4 算法产生的。很自然的想到在 MatrixF5 里只进行关键对的处理应该有更好的结果，这是本章要解决的问题。

表 4-2 三种算法的在不同的幂次下所产生的矩阵的大小

	MatrixF5		F4		F5	
d	matrix dim	#zero red.	matrix dim.	#zero red	matrix dim	#zero red

1	--	--	3 × 9	0	--	0
2	4 × 11	0	14 × 25	0	--	0
3	20 × 32	0	39 × 44	7	10 × 21	0
4	54 × 67	0	55 × 53	14	17 × 29	0
5	110 × 123	0	56 × 55	13	16 × 28	0
6	194 × 207	0	44 × 50	6	18 × 30	0
7	314 × 327	0	41 × 50	3	19 × 31	0
8	479 × 492	0	40 × 49	3	24 × 36	0
9	699 × 712	0	41 × 50	3	19 × 31	0
10	985 × 998	0	42 × 51	3	20 × 32	0
11	1349 × 1362	0	43 × 52	3	15 × 27	0
12	1804 × 1817	0	.....	.....	.....	.....

§3 F5D算法设计

3.1 F5D算法

F5D算法的思想如下：把输入多项式组 f_1，…，f_m按照次数从小到大排序，然后以次数为横坐标（设为 d），多项式为纵坐标（设为 i）建立计算表格。d从输入多项式的最小次数开始，从左到右，按照多项式递增的顺序，遍历每个方格去计算相应的关键对;计算完一行， d的值增加1，进入下一行计算。直至 d达到某个事先给定的 D，而这时所有次数为 D 的关键对都已经约化为零。(通过不断增加 D的值直至其达到 D_{reg}，因此F5D本质上是一种试探性算法)。最终，第 m 列所有方格内的元素（多项式的序号）就是要求的 Groebner 基。而对于每个 d（$1 \leqslant d \leqslant D_{reg}$），第 m列 d行之前的所有方格内的元素即为 d次 Groebner 基。

为便于说明，下面给出所用到的关键存储结构。

表 4-3　　存储结构说明

关键存储结构	用途说明
标签多项式数组（L[ ]）	动态数组，存放在整个F5算法中生成的所有标签多项式.
多项式序号数组(G[ ][ ])	每个方格 $G[i][j]$存放相应的多项式的序号. $i \in [1, m]$， $j \in [\text{mindeg}(f_i), D_{reg}]$
关键对队列数组(P[ ])	每纵列有一个关键对数组($P[i]$, i∈[1, m]). 整个运算过程会用到 m个关键队列.
规则链数组($Rules$[ ])	每纵列对应一个规则链（$Rules[i]$， $i \in [1, m]$). 用于相应关键对的 Rewrite 判
前序多项式序号数组	$G_{prev}[i]$表示对于多项式 f_i，其上一轮（d-1，i-1）算出的 d-1 次 Groebner 基

当前多项式序号数组	$G_{curr}[i]$表示对于当前列，已经产生的所有多项式的序号列表 $i\in[1, m]$
最新多项式序号数组	$G_{new}[i]$里的是每列新产生的和 $deg(f_i) = d\_curr$ 的多项式序号列表. $i\in[2, m]$.

注：

①F5D 算法在每一列上的计算都等同于原始 F5 算法. 原始 F5 算法需要 1 个关键对数组和 m 个规则数组，而 F5D 算法则各需要 m 个。

②对于前序多项式序号链表数组（$G_{prev}[]$），有如下表示：

$$\begin{cases} G_{prev}[i] = G[d][i-1], if(d == d\_\min), i \in [2, m] \\ G_{prev}[i] = \sum_{d=d\_\min}^{d\_curr-1} G[d][i-1], if(d > d\_\min), i \in [2, m] \end{cases} \quad (1)$$

这里 d 代表每次按次数递增的变量，每次增 1，也就是整个网格的横坐标。从第 2 个多项式，也即第 2 列才开始有 $G_{prev}[i]$，所以，i 从 2 开始。$d\_min=min(deg(f_1, \cdots, f_m))$，也即是次数最小的多项式。$d\_curr$ 表示当前的次数。如果 d 等于 d_min，表示在网格的第一行，$G_{prev}[i]$就是其左边相临的方格的多项式序号。如果 d 大于 d_min，则 $G_{prev}[i]$为左边相临一列的，从 d 减一直到 d_min 的所有方格中的多项式序号。

③当前多项式序号链表数组（$G_{curr}[]$）：$G_{curr}[i]=\sum_{d=d\_\min}^{d\_curr} G[d][i]$

④最新多项式序号链表数组（$G_{new}[]$）。对于每一列的多项式序号，其来源可以归结为三类：

1. 从其相临的左侧方格拷贝而来；
2. 若$deg(f_i) == d_curr$，则加入$G[d][i]$；
3. 在运算过程中，约化产生的最新多项式

我们把条件 2、3 中产生的多项式同时加入 $G_{new}[i]$，$G_{new}[i]$的用处在于当进入一个方格，而该方格中已经有多项式序号，而且当前的 $G_{new}[i]$不为空时，要与方格中的多项式组成新的关键对。之所以用 $G_{new}[i]$，是因为条件 1 中的多项式已经在上一列组过关键对了。

3.2 算法流程与分析

算法 2. F5D(***F***，***D***)

输入:环上多项式序列 F;要算到的次数，D.

输出:F 的 Groebner 基.

1. *sort（F）// F* 按次数从小到大排序;

2. *d=d_min; m=# F //* m 为多项式个数

3. 生成 f_1的标签多项式后加入标签多项式数组 L
4. $G[d][1].append(\#L)$;//#L 为数组 L 的长度，代表多项式的标号
5 While$(d<=D)$ do
6. $i=2$;
7. While$(i<=m)$ do
8. $Incremental\_gb(fi, i, d)$
9. If$(has_one())$
10. return {1}
11. i++
12. End While
13. d++
14. End While
15. Return $G_{curr}[m]$ // $G_{curr}[m]$中的多项式即为 Groebner 基

算法 1 是 F5D 算法的主算法，d 的初始值为 F 中次数最小的多项式，即 f_1 的次数。首先生成 f_1的标签多项式后加入标签多项式数组 L，同时把其序号①放入 $G[d][1]$，也就是第一个方格。接着算法以 d递增开始循环，d每次递增 1；而内层以多项式递增循环，因为第一个多项式已经处理过，所以内层循环从 i=2 开始。第 7 行 $Incremental\_gb$ 函数根据输入的多项式 f_i以及当前的 d、i 数值，动态的计算出当前的 $G[d][i]$会产生哪些新的多项式序号，有哪些新的关键对要处理等。第 8 行判断如果 $G[d][i]$中的多项式有常数 1，则直接返回 Groebner 基为 1。该算法在次数为 D时终止，Bard 在文[6]中给出了 D的估算公式。

算法 3.incremental_Groebner 基(f_i, i, d)

输入: f_i,, i, d 表示在次数 d 计算加入 f_i 后的 Groebner 基

输出:无

1. 根据 d, i 的数值设置 $G_{prev}[i]$
2. If($G[d][i-1]$不空) Then
3. 把 $G[d][i-1]$中的多项式序号拷贝到 $G[d][i]$
4. If ($G[d][i]$ 不空) Then
5. If($G_{new}[i]$中有元素) Then
6. $G[d][i]$与 $G_{new}[i]$中有元素组成关键对并加入关键队列 $P[i]$
7. $G[d][i]$中的多项式序号加入 $G_{curr}[i]$
8. Else
9. $G[d][i]$中的多项式序号加入 $G_{curr}[i]$
10. If(deg(f_i)==d) Then
11. 生成 f_i 的标签多项式后加入标签多项式数组 L
12. $curr\_id$=#L
13. G_{new}[i]->append($curr\_id$)

14. $G[d][i]$->append($curr\_id$)
15. f_i 与 $G_{curr}[i]$中的多项式形成关键对并加入关键队列 $P[i]$
16. $G_{curr}[i]$->append($curr\_id$)
17. 把 $P[i]$中关键对次数为 d 次的所有关键对存为 P_d
18. If (P_d 不空) Then
19. $P[i]= P[i] \setminus P_d$//除去 Pd 中的关键对
20. $S=Spoly(P_d)$ //计算 s 多项式
21. $R=Reduction(S,\ G_{prev}[i]\ G_{curr}[i],\ d)$
22. For(k in R)
23. f_k 与 $G_{curr}[i]$中的多项式形成关键对并加入关键队列 $P[i]$
24. $G_{curr}[i]$->$append(k)$
25. G[d][i] ->$append\ (k)$
26. $G_{new}[i]$->$append(k)$

算法 2 首先判断当前方格左边方格 $G[d][i-1]$的元素是否为空，这里我们分别讨论两种情况：

1）如果 $G[d][i-1]$不空，则拷贝其元素到当前方格 $G[d][i]$。接着看 $G_{new}[i]$中的元素，如果不空，则与 $G[d][i]$中的元素组成关键对。同时把 $G[d][i]$中的元素添加到 $G_{curr}[i]$里。如果 $G_{new}[i]$为空，则直接把 $G[d][i]$中的元素添加到 $G_{curr}[i]$里即可。

2）如果 $G[d][i-1]$为空，则判断当前列对应的多项式的次数是否等于当前算到的次数 d，如果相等，则首先把 f_i添加到标签多项式数组，同时把该多项式的序号添加到 $G_{new}[i]$和 $G[d][i]$，然后该多项式与 $G_{curr}[i]$中的多项式两两组成关键对加入队列 $P[i]$。最后把该多项式的序号添加到 $G_{curr}[i]$。

3）从关键队列 $P[i]$中取出所有次数为 d的关键对放入 P_d，如果 P_d为空，则算法退出，到下一方格去处理。否则，计算 S 多项式，然后约化。如果约化的结果不空，则新产生的多项式与 $G_{curr}[i]$中的多项式两两组成关键对加入队列 $P[i]$，然后把其序号分别加入 $G_{curr}[i]$，$G_{new}[i]$，$G[d][i]$。

4）算法中有三处生成关键对，而 F5 准则的使用，就在 $crit\_pair$ 函数里。以 22 行为例，关键对列可表示为 $p[i]+= \bigcup_{j \in Gcurr[i]} crit\_pair(k,j,i,Gprev[i])$。

5）*spoly*，*reduction* 等函数类似 F5 基本算法，所不同的只是在每纵列的关键队列上都会有对这两个函数的调用。

§4 举例分析

以 J.-C. Faugère 在文献[11]中的例子，分析 F5D 算法的执行流程。

先把输入多项式按次数从小到大排序。从 $G[d][i]=G[3][1]$开始，f_1的序号①加入 $G[3][1]$，其标签多项式加入标签多项式数组 L，这里为了方便理解，简

记为 $L=\{①\}$。接着进入下一方格，首先把 $G[3][1]$中的多项式序号拷贝到 $G[3][2]$，由于 f_2的次数也是 3，所以同时 f_2的序号②也要加入 $G[3][2]$，同时 $G_{curr}[2]=\{①, ②\}$，$G_{new}[2]=\{②\}$，因为②不是拷贝来的。$G_{prev}[2]=\{①\}$。最后<1，2>组成关键对，由于其关键对的次数上升到 5，所以当 d等于 5，在 $G[5][2]$时，从关键对列中取出再做处理。接着进入 $G[3][3]$，拷贝 $G[3][2]$中的{①，②}，同时设置 $G_{curr}[3]=\{①, ②\}$，$G_{new}[3]=\{\ \}$，$G_{prev}[3]=\{①, ②\}$。

接着算法进入 d=4，由于 f_3的次数是 4，所以首先把 f_3的序号③加入 $G[4][3]$，其标签多项式加入标签多项式数组 L。同时与 $G_{prev}[3]$中{①，②}组成关键对，次数分别为 5，6。所以当 d 等于 5，6 时，分别在 $G[5][3]$、$G[6][3]$中处理。因为③不是拷贝来的，$G_{new}[3]=\{③\}$。

在 d等于 5 时，首先处理 $G[5][2]$中的关键对<1，2>，其约化后形成新的多项式④，添加规则(z^2，④)后，④与 $G_{prev}[2]$中的①以及 $G_{new}[2]$中的②分别成关键对<4，1>、<4，2>。其中<4，1>到 d 等于 7 时判断，而<4，2>被 F5 准则所排除。当进入 $G[5][3]$时，首先从 G[5][2]拷贝到④，因为 $G_{new}[3]$此时并不空，所以④与 $G_{new}[3]$中的③成关键对<4，3>，次数为 8。所以当 d 等于 8，在 G[8][3]中处理。而<3，1>对生成了新的多项式⑤，⑤先与 $G_{curr}[3]$中的①~④形成关键对，然后分别把⑤加入 $G_{curr}[3]$，$G_{new}[3]$，$G[5][3]$。

直到 d为 8，m为 3 时，关键对列 $P[1]$至 $P[3]$中的所有关键对都处理完时，算法结束。返回 $G_{curr}[3]=\{①~⑩\}$即为 Groebner 基。

表 4-4　F5D 对 Faugère 文中算例的分析

次数	f_1=x*z^2-y^2*t	f_2= x^2*y-z^2*t	f_3=y*z^3-x^2*t^2
3	①　L={①}	拷贝①，加入② L={①，②} <1，2>:deg==5 $G_{curr}[2]=\{①, ②\}$ $G_{new}[2]=\{②\}$ $G_{prev}[2]=\{①\}$	拷贝①② $G_{curr}[3]=\{①, ②\}$ $G_{new}[3]=\{\ \}$ $G_{prev}[3]=\{①, ②\}$
4			加入③　L={①,②,③} $G_{prev}[3]=\{①, ②\}$ $G_{new}[3]=\{③\}$　$G_{curr}[3]=\{①, ②, ③\}$ <3，1>:deg==5　<3，2>:deg==6
5		<1，2>形成关键对生成④ {[z^2，2]，x*y^3*t-z^4*t} L= {①，②，③，④} Add_rule(z^2，④) Reduction:④更新为 x*y^3*t-z^4*t $G_{curr}[2]=\{①, ②, ④\}$ $G_{new}[2]=\{②, ④\}$ $G_{prev}[2]=\{①\}$ <4，1>:deg==7 <4，2>:top_reduce	拷贝④　④与 $G_{new}[3]$中的③成关键对 <4，3>:deg==8 <3，1>形成关键对生成⑤ Add_rule(x，⑤) L= {①，②，③，④，⑤} {[x，3]，y^3*z*t-x^3*t^2} Reduction:　⑤　更　新　为 y^3*z*t-x^3*t^2 <5，1>:deg==7　<5，2>:deg==7 <5，3>: top_reduce <5，4>deg==6 $G_{prev}[3]=\{①, ②\}$ $G_{new}[3]=\{③⑤\}$ $G_{curr}[3]=\{①②③④⑤\}$

d从 6 到 8 的详细计算过程见附录。

§5 算法正确性与终止性证明

命题 1 F5D 算法处理了在计算 Groebner 基过程中所产生的所有关键对。

证明：F5D 算法中有三处生成关键对，一处是在算法的 15 行，由于每一格中新生成的多项式序号都会从左向右拷贝，这样保证了输入的 m 个多项式两两都会组成关键对。而对于约化后新生成的多项式 f_i 会立刻与当前方格中的所有多项式组成关键对，这是在算法的 23 行实现的。但当前方格（若非最右一列）中的多项式序号并不完整，因为其右边一格中有可能有先于 f_i 约化出的多项式和次数为当前次数的刚加入该方格的原始多项式，而这些多项式会被保存在 G_{new} 中，算法的第 6 行保证了这种情况下的关键对组合。这样，通过这三处关键对的处理，保证了 F5D 算法处理了在计算 Groebner 基过程中的所有关键对。

命题 2 F5D 算法在运算过程中不会产生约化为零的关键对。

证明：1）F5D 算法在列方向上的计算相当于原始 F5 算法，但原始 F5 算法只需要一个关键对数组，而这里共需要 m 个关键对数组，自然的把 Rewrite 的判断分布到 m 个关键队列上。

2）MatrixF5 的 F5 准则中，对每个次数，按多项式标签从小到大去生成新一轮的多项式。在生成的过程中，只有标签相同的多项式才会从 $d-1$ 次去乘变元升到 d 次。假定当前多项式标签为 i，要用所有标签小于 i 的多项式去判断约化关系。这相当于在 F5D 中 *crit_pair* 函数中使用 G_{prev} 去判断约化关系。因为由(1)中 G_{prev} 的定义可知，标签小于 i 即相当于左边前面一列去判断；而 G_{prev} 中的多项式同样是最大 $d-1$ 次的。

这样，由 F5 算法中的 Rewrite 准则与 F5 准则保证了 F5D 算法运算过程中不会产生约化为零的关键对。

定义 5[10] 设 G 是多项式环 P 的某个理想 I 的子集，称 G 为理想 I 的 d 次 Groebner 基，当且仅当对于所有的多项式 f，g 属于 G，其相应的关键对的次数不大于 d，且 S 多项式 $spoly(f, g)$ 约化 G 为零。则 G 被称为 **d 次 Groebner 基**。

命题 3 F5D 算法在次数为 d 时的计算结果为输入的 d 次 Groebner 基。

证明：Groebner 基的计算过程就是生成所有的关键对，计算 s 多项式，并进行约化，约化后的非零多项式重新生成关键对，如此递归，直到所有的关键对全部约化为零，则生成的多项式集合就是 Groebner 基。F5D 的输入是齐次多项式，整个计算过程严格按照次数递增的顺序进行。而对于次数为 d 时，所有的次数等于 d 的关键对皆要相对于 G_{prev} 约化。新产生的次数大于 d 的关键对或被 Rewrite 准则与 F5 准则消除，相当于全部约化为零;要么会在更高处约化。所以，f5D 在次数为 d 时，得到的是 d 次 Groebner 基。

推论 1 F5D 算法与 MatrixF5 算法在同一次数，即 D_{reg} 终止。

证明：当 d 达到 D_{reg}，而这时所有次数为 D_{reg} 的关键对都已经约化为零。最终，第 m 列所有方格内的元素（就是多项式的序号）就是所要求的 Groebner 基。其终止性是由 D_{reg} 的性质保证的。而由命题 1,2,3，显而易见 F5D 算法与 MatrixF5 算法的 D_{reg} 相等。

§6 实验结果

我们在一台主频 2.5GHz、4G 内存的多核计算机上对于 GF2、GF3、GF5 上的 HFE 多变量系统进行实验，对比了 MatrixF5、F54D、F5D 三种方法的实际表现。其中 F54D 是为了达到时间和空间的更好折衷，在 F5D 的约化过程中，采用了类似 F4 的符号预处理，把大矩阵的高斯消元分解为若干小矩阵的高斯消元，以提高 F5D 的约化速度，同时也为了能从产生最大矩阵的维数形成更直观的对比。

HFE 多变量系统由中心方程 $F(X)=\sum_{i=0}^{r_2-1}\sum_{j=0}^{i}a_{ij}X^{q^i+q^j}+\sum_{i=0}^{r_1-1}b_iX^{q^i}+c$ 和变元数确定。参数 r_1，r_2 越大，变元数越多，HFE 多变量系统越难求解。对于 GF2 上的多项式系统，由表 4-5 可看到，其在算到 D_{reg} 为 4 时候 F54D 所生成最大矩阵的维数比 MatrixF5 均约减少一半。Ding， J 建议使用更大些的素数域来生成 HFE 多变量系统[68]，这样可以很少的变元以达到更高的安全性。计算结果显示，求解 GF3 上 12 变元系统（r_1=2， r_2=2）所生成的矩阵与 GF2 上 24 变元（r_1=4， r_2=4）的系统相当。同样，F54D 生成最大矩阵的维数比 MatrixF5 均约减少一半。

表 4-5 GF2 与 GF3 上的 HFE 方程求解

GF2 上的 HFE 方程				MatrixF5	F54D
#VAR	D_{reg}	r1	r2	Rows *Cols	Rows *Cols
18	4	4	4	6210 x 7315	3491 x 5860
20	4	4	4	8460 x 10626	4693 x 9424
22	4	4	4	11198 x 14950	7224 x 11913
24	4	4	4	14472 x 20475	8626 x 19462

GF3 上的 HFE 方程				MatrixF5	F54D
#VAR	D_{reg}	r1	r2	Rows *Cols	Rows *Cols
10	6	2	2	8875 x 8008	4368 x 7048
11	6	2	2	13387 x 12376	7956 x 10274
12	6	2	2	19576 x 18564	12958 x 16274
13	6	2	2	27124 x 26232	15986 x 24278

表 4-6 GF5 上的 HFE 方程求解

GF5 上的 HFE 方程				2G 内存内能算到的幂次		
#VAR	D_{reg}	r1	r2	MatrixF5	F54D	F5D
12	10	1	1	6	7	10
13	10	1	1	6	7	10
14	10	1	1	5	6	10
15	10	1	1	5	6	10

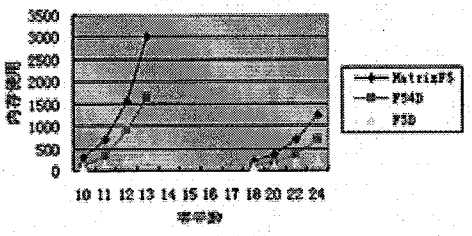

图 4-1 GF2 与 GF3 上的 HFE 方程求解内存对比

对于 GF5 上的 HFE 多变量系统，Magma 在 D_{reg} 为 10 时才可以解出整个系统[69]。而 MatrixF5 算法随着次数的升高，矩阵不断膨胀，在 win32 程序 2G 内存空间的限制下，其中 12、13 变元的 HFE 系统（r1=1，r2=1）只能算到次数为

6;14、15 变元的 HFE 系统算到次数为 5。而 F54D 算法是时间与空间的折衷，在同样的条件下，只能多算一个次数。F5D 算法由于每次只选择有价值的关键对去约化，并且所选择的关键对分布在多条不同的关键对列上，内存使用量远低于 MatrixF5，可以算到次数为 10，从而最终解出方程。

最后，给出 GF2 与 GF3 上三种算法在不同变元数下所使用的内存对比。这里需要说明的是，对矩阵的存储并未采用压缩存储，每个矩阵元素用整型(int)来存储。从图 4-1 中可看出在 GF3 上求解从 10 到 13 变元以及 GF2 上从 18 到 24 变元的 HFE 方程的内存消耗增长变化。两个不同系数域上 MatrixF5 算法的内存增长最快，F5D 算法内存增长比较平缓。同时在 GF3 上求解 HFE 方程的内存增长明显快于 GF2 上的增长，这也从某种程度上印证了 Ding， J 关于非 GF2 素数域上 HFE 方程的变元增长与求解难度成指数关系[68]。

§7 小结

本章首先分析了 F5 算法和 MatrixF5 算法对于求解非线性超定方程组的缺陷，设计了一种求解非线性超定方程组的新方法，即 F5D 算法。F5D 算法避免了 MatrixF5 算法在运算过程中产生过大矩阵的缺陷，同时为了达到时间和空间的良好折衷，在 F5D 的约化过程中，采用了类 F4 的符号预处理技术以提高约化速度。实验结果表明，在超定环境下，F5D 算法可被用作 F5 算法和 MatrixF5 算法的完美替代。

下一步的工作，主要进行 F5D 算法对非齐次多项式的处理方面的研究。对于非齐次多项式，在按照幂次递增计算的过程中，有可能出现“降次”的情况，从而导致关键对与相关准则的处理更加复杂。同时计划仿照从 FXL[70]算法，先对若干变元“猜值”，以增加方程组超定性。方程越超定，其求解复杂度越低[71]，这里具体猜几个变元，猜哪些变元是值得深入研究的问题。

附录：

6			<5，4>形成关键对生成 ⑥{[x^2，3]，z^5*t-x^4*t^2} Add_rule(x^2，⑥) L={①，②，③，④，⑤，⑥} Reduction: ⑥更新为 z^5*t-x^4*t^2 G_{prev}[3]={①，②，④}G_{new}[3]={③⑤⑥} G_{curr}[3]={①②③④⑤⑥} <6，1>:deg==7 <6，2>~<6，5>: top_reduce <3，2>: rewrite
7		<4，1>形成关键对生成⑦ {[z^4，2]，-z^6*t+y^5*t^2} L={①，②，③，④，⑤，⑥，⑦} Add_rule(z^4，⑦) Reduction: ⑦更新为-z^6*t+y^5*t^2 <7，1>:top_reduce <7，2>:top_reduce <7，4>:top_reduce G_{curr}[2]={①②④⑦} G_{new}[2]={②④⑦} G_{prev}[2]={①}	拷贝⑦ <7，3>:deg==8 <7，5>: top_reduce <7，6>: deg==7 <7，6>形成关键对生成⑧ {[x^2*z，3]，y^5*t^2-x^4*z*t^2} L={①，②，③，④，⑤，⑥，⑦，⑧} Add_rule(x^2*z，⑧) <6，1>形成关键对生成⑨ {[x^3，3]，-x^5*t^2+y^2*z^3*t^2} L={①，②，③，④，⑤，⑥，⑦，⑧，⑨} Add_rule(x^3，⑨) Reduction: ⑧更新为 y^5*t^2-x^4*z*t^2 Reduction: ⑨更新为-x^5*t^2+z^2*t^5

			G_{curr}[3]={①~⑨} G_{new}[3]={③⑤⑥⑧⑨} G_{prev} [3]={①，②，④} <8，1>:top_reduce <8，2>:rewrite <8，3>:top_reduce <8，4>:rewrite <8，5>:top_reduce <8，6>:top_reduce <8，7>:top_reduce <9，1>～<9，8>:top_reduce <5，1>，rewrite　　<5，2>，rewrite
8			<7，3>形成关键对生成⑩ {[z^3*t，3]，y^6*t^2-x^2*z^3*t^3}L= {①，②，③，④，⑤，⑥，⑦，⑧，⑨，⑩} Add_rule(z^3*t，⑩) Reduction: ⑩更新为 y^6*t^2-x^2*z^3*t^3 <4，3>:rewrite G_{curr}[3]={①~⑩}G_{new}[3]={③⑤⑥⑧⑨⑩} G_{prev} [3]={①②④⑦} <10，1>~<10，9>::top_reduce

Rules[1]= ∅; *Rules[2]*= {（z^2，④），（z^4，⑦）}; *Rules[3]*= {(x，⑤)，(x^2，⑥)，(x^2*z，⑧)，(x^3，⑨)，(z^3*t，⑩)}

第五章 Groebner 基计算中布尔多项式的高效表示

布尔方程组求解技术对于密码分析具有重要的现实意义。然而，在众多求解算法的实际计算过程中的，难以抑制的空间需求增长与计算机系统有限的存储能力之间的矛盾，正是当前制约布尔方程组求解技术取得更大成果的最主要瓶颈。本章针对基于消项的求解算法，分析了该矛盾的产生根源，提出了解决途径，进而设计了一种全新的布尔多项式计算机表示，称之为 BanYan。 BanYan 适用于基于首项约化的求解算法，如 F4，F5，XLs 等算法。与 BDD 和系数矩阵等基于项的传统布尔多项式表示相比，BanYan 可以降低在计算过程中的空间需求，从而显著提升布尔方程组求解算法的现实求解能力。

§1 引言

众多密码算法的安全性都可以归结为有限域上非线性方程组的求解问题，这正是当今密码学研究热点代数攻击的出发点。代数攻击在理论上适用于当前大部分分组密码和流密码算法，以及基于多变量理论的公钥算法等。尤其是近年来，出于方便软件实现和理论分析的考虑，密码算法的设计愈加注重代数表示的简单化，这一趋势使得非线性方程组求解问题在密码分析中的应用前景尤其令人期待。

具体而言，由于绝大多数分组密码和流密码算法都以二进制位或二进制分组作为基本数据单元，使得这些算法的直接代数表示往往具有二元有限域上非线性方程组的形式。进一步地，由于密码分析总是只关心方程组的有理解，故最终可将其视为二元有限域上多项式环的剩余环 $GF(2)[x_1, x_2, \ldots, x_n]/\langle x_1^2-x_1, x_2^2-x_2, \ldots, x_n^2-x_n\rangle$ 上的方程组，即布尔方程组。在密码分析中，如果能够成功求解某密码算法根据已知信息建立的布尔方程组，就意味着该算法被完全攻破。

不过，布尔方程组求解本身是计算困难问题，对其高效算法的探索一直在进行中。自上世纪末以来，相关研究已经取得了丰富的理论成果，各种求解算法被不断设计出来。根据求解途径的不同，这些算法主要可归类为本质上等同于 Groebner 基理论的消项算法[71-77]，基于吴特征列的消元算法[52]，以及把布尔方程组求解问题转化为 SAT 等其他计算困难问题的转化方法等[78]。其中，基于消项思想的 F_4，F_5，XLs 等算法尤其值得关注，它们已经分别取得了对于 HFE，SFlash 等现实密码体制的一系列攻击成果[79-86]。

但目前，各算法对于规模较大的随机(稀疏)布尔方程组，尤其是对于大部分主流密码体制的代数表示依然无能为力。其中，首当其冲的障碍并非是理论层面的，而是各求解算法及其实现普遍缺乏对计算空间规模的认知和控制，最终表现为中间结果的存储需求大大超出计算机系统的实际处理能力[87]。这一症结尤其突出地体现在布尔多项式的计算机表示问题中。虽然密码算法内非线性部分的规模十分有限，但经验表明，在消元或消项等各类求解算法的实际工作过程中，经

由不断的迭代和组合，非线性因素的项数规模往往以指数速度膨胀。这一隐患在 BDD 或系数矩阵等基于项的传统多项式表示下，即表现为多项式表示的空间需求很快超越计算机系统的存储能力，导致计算被迫中止[88]。

总之，布尔多项式的计算机表示问题，已经成为当前制约布尔方程组求解技术发展的首要障碍。对此，本章在第二部分介绍必要的背景知识和数学符号，以及在第三部分介绍传统的布尔多项式表示及其规模膨胀问题之后，将在第四部分中针对基于消项的求解算法，分析中间结果膨胀问题的产生根源及其解决途径。之后的第五部分将给出基于该思想设计得到的布尔多项式计算机表示，BanYan。最后，第六部分将展示 BanYan 在模拟实验中的实际表现。

§2 背景知识与数学符号

这里，我们将采用集合的方式表述布尔方程组消项解法的相关定义，定理和算法。与基于多项式环剩余环和 Groebner 基理论的传统表述方式相比，我们认为集合的方式不仅更为简洁和直接，而且更能体现问题本质。

- 本章使用 x_1，x_2，…，x_n 作为变元，并定义变元序 $x_1>x_2>\ldots>x_n$。
- 称任意变元集 $t\subseteq\{x_1, x_2, \ldots, x_n\}$ 为一个项。并定义项间的乘法 $t_A\times t_B=t_A\cup t_B$ 和除法 $t_A/t_B=t_A-t_A\cap t_B$。注意 $(t_A\times t_B)/t_B=t_A$ 当且仅当 $t_A\cap t_B=\Phi$。
- 定义 $T=\{x_1, x_2, \ldots, x_n\}^*$，即由全部项组成的集合，易知 $|T|=2^n$。并定义 T 上的(字典)序

$$t_A > t_B \Leftrightarrow \max_{x\in t_A/t_B} x > \max_{x\in t_B/t_A} x$$

易知该序为全序，且有以下结论成立：

定理 1. 对任意 $t_A>t_B$，和任意 $t_C\in T$ 满足 $t_A\cap t_C=\Phi$，均有 $t_A\times t_C>t_B\times t_C$。

- 称任意项集 $p\subseteq T$ 为一个多项式。并定义多项式间的加法 $p_A+p_B=p_A\cup p_B-p_A\cap p_B$，以及项与多项式的乘法 $t\times p=\Sigma_{t'\in p}\{t'\times t\}$。
- 对于多项式 $p\neq\Phi$，称 $\max_{t\in p}t$ 为 p 的首项，记作 $lt(p)$。
- 定义 $R=T^*$，即全部布尔多项式组成的集合，易知 $|R|=2^{2^n}$。
- 对任意多项式集 $p\subseteq R$，称 $\{lt(p)|p\in P\}$ 为 P 的首项集，记作 $LT(P)$。
- 对任意多项式集 $p\subseteq R$，称 $\{\Sigma_i t_i\times p_i \mid t_i\in T,\ p_i\in P\}$ 为 P 的生成闭包，记为 $<P>$。

基于消项的求解算法的目标为，对于给定多项式集 P，希望求得其生成闭包中关于集合包含关系 $\supseteq$ 的所有极小首项成员组成的集合 G。亦即，希望求得 $G\subseteq<P>$，满足 $LT(G)$ 为 $LT(<P>)$ 关于 $\supseteq$ 的极小集。

从原始的 Buchberger 算法到当今的 F_4，F_5，XLs 等各种消项算法，虽然求 G

的具体方法千差万别，但其指导思想在本质上却是完全一致的。即按照某种预定规则构造有可能蕴涵新极小首项的<P>成员 $p=\sum_i t_i \times p_i$，而后通过首项约化进行验证，从而不断找到新的极小首项成员，逐步逼近结果 G。

对于任意<P>成员 $p=\sum_i t_i \times p_i$，总是可以使用当前极小首项成员集 G'(初始值为 P)约化它，检验其是否蕴涵新的极小首项成员。具体过程为，首先检查是否存在 $p^* \in G'$满足 $lt(p^*) \subseteq lt(p)$，若存在则令 $p_1=p+(lt(p)/lt(p^*)) \times p^*$，易知 $p_1 \in <P>$且 $lt(p_1)<lt(p)$，而后继续检查是否存在 $p^{**} \in G'$满足 $lt(p^{**}) \subseteq lt(p_1)$，如此反复，直至得到多项式 p_k，满足 $LT(G')$中不存在 $lt(p_k)$的子集，或者 $p_k=\Phi$，即为约化结果 $g=p_k$(可表示为 $g=\sum_j (t_j \times g_{ij})$的形式，其中 $g_{ij} \in G'$)。通过不断以得到的非空约化结果 g 更新 G'，最终可达成 $G'=G$。定理 1 保证了 $lt(p)>lt(p_1)>...>lt(p_k)$，因此计算过程必然会在有限时间内结束。

事实上，对于任意时刻的 G'，总是只需尝试约化如下形式的<P>成员。

$$p=t_A \times g_A + t_B \times g_B，\text{其中 } g_A，g_B \in G' \text{且满足 } lt(t_A \times g_A)=lt(t_B \times g_B)$$

其中根据 $g_A=g_B$ 或 $g_A \neq g_B$ 分为两种情况。前者的一般形式为。

$$s_{x_A}(g_A)=\{x_A\} \times g_A + g_A，\text{其中 } x_A \in lt(g_A)$$

后者的一般形式为：

$$s(g_A,g_B)=\frac{lt(g_A) \times lt(g_B)}{lt(g_A)} \times g_A + \frac{lt(g_A) \times lt(g_B)}{lt(g_B)} \times g_B$$

定理 2. 若 G'中所有 $s_{x_A}(g_A)$和 $s(g_A,g_B)$形式均被 G'约化为 Φ，则有 $G'=G$。

§3 传统多项式表示及其规模膨胀问题

消项求解算法使用的传统多项式表示包括 BDD[46]和系数矩阵。BDD 是二叉树的变种，树中每个非叶子节点的两个分支分别记录多项式关于节点变元的系数式和常数式。例如，图 5-1 是布尔多项式 $x_1x_3+x_2x_3+x_3$ 的 BDD 表示，虚线指向的分支表示系数式，实线指向的分支表示常数式。系数矩阵则以列作为多项式中项的索引，以每个行表示一个布尔多项式。例如，图 5-2 的系数矩阵表示由布尔多项式 $x_1x_2+x_2x_3$，$x_1x_3+x_2$，x_1+x_3 组成的布尔多项式集。

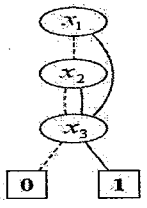

图 5-1 BDD 图

消项求解算法的空间开销主要来自 G'。虽然其起点 P 规模有限，但在计算过程中，不断引入到 G'中的非空约化结果 g 的项数却呈高速增长趋势。而 BDD 和系数矩阵的规模属性(分别为节点个数和列数)直接取决于多项式的项数，这使

得 G'的存储需求随之高速膨胀。

$$\begin{array}{c} \\ f_3 \\ f_2 \\ f_1 \end{array}\begin{array}{c} \begin{array}{cccccc} x_1x_2 & x_1x_3 & x_1 & x_2x_3 & x_2 & x_3 \end{array} \\ \begin{pmatrix} 1 & & & 1 & & \\ & 1 & & & 1 & \\ & & 1 & & & 1 \end{pmatrix} \end{array}$$

图 5-2　系数矩阵图

看一个简单的例子(n= 7)：

$$\begin{cases} g_1 = x_1x_2x_3 + x_1x_3 + x_3x_4 + x_5x_6 \\ g_2 = x_1x_3x_4 + x_3x_4 + x_4x_5 + x_6x_7 \\ g_3 = x_2x_3x_4 + x_3x_5 + x_3 + x_4x_6 \end{cases}$$

不妨设消项算法首先选择约化 $s(g_1, g_2)$和 $s(g_2, g_3)$，分别得到新极小首项成员 g_4 和 g_5。

$$\begin{cases} s(g_1,g_2): g_4 = x_2x_4x_5 + x_2x_6x_7 + x_3x_5 + x_3 + x_4x_5x_6 + x_4x_5 + x_4x_6 + x_6x_7 \\ s(g_2,g_3): g_5 = x_1x_3x_5 + x_1x_3 + x_1x_4x_6 + x_2x_3x_4 + x_2x_4x_5 + x_2x_6x_7 \end{cases}$$

注意 g_4 和 g_5 的项数明显多于 g_1，g_2 和 g_3。接下来， 消项算法又通过约化 $s(g_4, g_5)$得到新极小首项成员 g_6。

$$s(g_4,g_5): g_6 = x_1x_2x_4x_6 + x_1x_3x_4x_5x_6 + x_1x_3x_4x_5 + x_1x_3x_4x_6 + x_1x_3x_4 + x_1x_3x_5 + x_1x_3 + x_2x_3x_4 + x_2x_4x_5 + x_2x_4x_6x_7 + x_3x_4x_6x_7 + x_3x_4 + x_4x_5x_6 + x_5x_6x_7$$

不难看出，g_6 的项数相比 g_4 和 g_5 再次大大增加。

这种项数增长现象并非偶然。图 5-3 展示了消项算法工作过程中，非空约化结果项数规模的典型变化过程(n=20)。图中坐标系的横坐标为时间轴，纵坐标为该时间点 G' 中多项式的平均项数 l。可以看到，l 随时间推移呈超线性增长趋势。

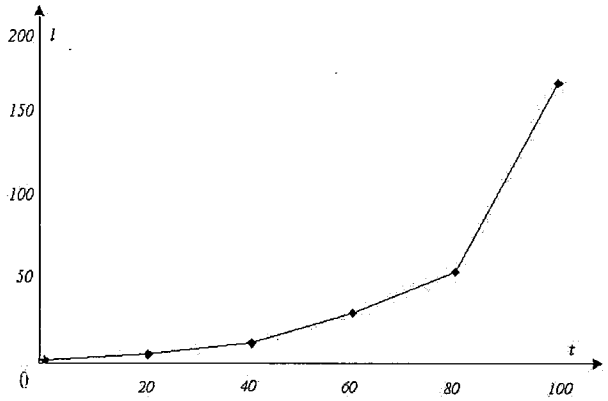

图 5-3 非空约化结果的规模增长

§4 设计思想

事实上，非空约化结果 $g=\Sigma_j(t_j \times g_{ij})$的项数增长现象并不难解释。因为在 G' 尚稀疏时，我们很难期待在各个 $t_i \times g_{ij}$ 之间存在除待消首项之外的大量重复成员，或者说，重复成员所占的比例不显著。设当前 G'中多项式的平均项数为 l，若 $l << 2^n$，则对 g 的项数有以下估计：

$$|g| = \left|\sum_{j=1}^{m} t_j \times g_{i_j}\right| \approx \sum_{j=1}^{m} \left|g_{i_j}\right| - m \approx m \times (l-1)$$

可见，$|g|$相对于 l 近似增长了 m 倍之多。这种增长在大量的 g 中不断累积，导致了 G'总规模的爆炸性增长。相较于其他空间需求低的求解算法(例如转化为 SAT)，消项算法在自身空间需求能够被满足时，往往具有优秀得多的速度表现。

但在问题规模增加后，空间需求无法被满足导致的程序崩溃使得其在计算速度上的优越性不再具有实际意义。

从算法设计的角度，$|g|$的增长难以控制。但若从算法实现的角度考虑，虽然$g=\Sigma_j(t_j\times g_{ij})$本身难以存储，但其生成信息，即$(t_j,\ i_j)_j$却相对简单得多，存储开销仅为 $O(m)$，远小于保存 g 本身所需的 $O(m\times l)$。这就意味着，如果不实际存储 G' 的成员（除 P 的成员外），而只记录它们是如何由其他 G'成员生成的，则可在很大程度上缓解 G'的存储负担。此时，G'不再是一组彼此独立的多项式，而成为一系列互相引用的$(t_j,\ i_j)_j$形式。例如对于之前的例子，可得到如下的新表示形式：

$$\begin{cases} g_4 = x_4\times g_1 + x_2\times g_2 + g_2 + g_3 \\ g_5 = x_2\times g_2 + x_1\times g_3 \\ g_6 = x_1x_3\times g_4 + x_2x_4\times g_5 + x_4\times g_1 + x_6x_7\times g_1 \end{cases}$$

可以看到，三个新多项式总计只占用了相当于 9 项的额外存储空间，远小于其本身的项数之和 26.随着计算的推进，这一差距会变得愈发巨大.因此，通过记录中间结果的生成信息而非其本身，可以避免算法实现陷入项数规模高速膨胀带来的巨大存储负担.这正是下一部分将介绍的布尔多项式表示 BanYan 的设计思想.不过，和保存 g 本身相比，要从其生成方式$(t_j,\ i_j)_j$中提取出 g 的各种信息则相对困难得多，而这可能严重影响算法实现的运行效率.但幸运的是，对于基于消项的布尔多项式求解算法而言，需要关心的多项式信息恰好是十分有限的。

§5 BanYan

正如上一部分介绍的，BanYan 将消项算法计算过程中生成的每个非空约化结果 $g=\Sigma_j(t_j\times g_{ij})$记录为$(t_j,\ i_j)_j$的形式。此时，$G'$中的成员除直接来自 P 的多项式外(称作**根多项式**)均非独立存在，而是引用了若干其他 G'成员。我们将这种引用了其他成员的多项式称为**簇多项式**，将被其引用的多项式称为其**分支**。这样，G'中全部多项式根据彼此的引用关系，构成以根多项式为起点的有向无环图。

图 5-4 是与之前的例子中$\{g_1,\ g_2,\ g_3,\ g_4,\ g_5\}$对应的 BanYan 表示。

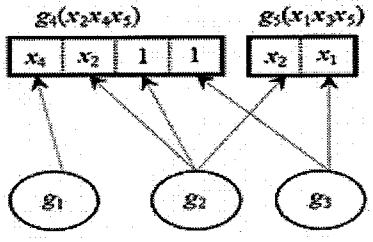

图 5-4 BanYan 示例

图中每个节点均代表一个多项式。其中 g_1，g_2，g_3 为根多项式，g_4，g_5 为簇多项式。g_4 引用了分支 g_1，g_2，g_3，而 g_5 则引用了分支 g_2，g_3。此外，每个簇多项式节点还记录了自己的首项信息，用 msg 表示。

$$\begin{cases} g_4 = x_4\times g_1 + x_2\times g_2 + g_2 + g_3 \\ msg(g_4) = x_2x_4x_5 \end{cases} \qquad \begin{cases} g_5 = x_2\times g_2 + x_1\times g_3 \\ msg(g_5) = x_1x_3x_5 \end{cases}$$

接下来，我们通过下个非空约化结果，即 g_6 的生成过程，展示基于消项的求解算法在 BanYan 下是如何工作的。首先，根据 g_4，g_5 的首项信息，生成对应

$s(g_4, g_5)$的新节点，并获得其首项，如图 5-5(A)之后，不断约化该节点对应的多项式，直至得到新的极小首项成员 g_6，如图 5-5(B)和图 5-5(C)。

下面以 $s(g_A, g_B)$的约化算法为例，给出消项求解算法的 BanYan 版本($s_{x_A}(g_A)$的约化算法类似)。

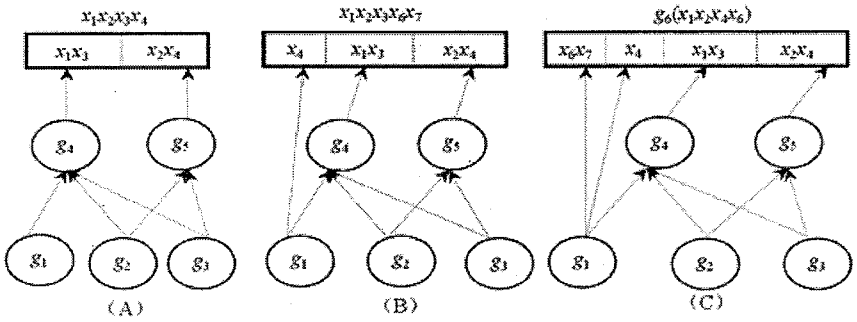

图 5-5　BanYan 的工作过程

算法 1. $gen(g_A, g_B)$，计算 $s(g_A, g_B)$关于 G'的约化结果。

(1)　$t_A=msg(g_A)$，　$t_B= msg(g_B)$

(2)　$g=\{(t_A, A), (t_B, B)\}$

(2)　$S=expand(t_A\times t_B/t_A, g_A)+expand(t_A\times t_B/t_B, g_B)$

(3)　**while** $\exists g_C\in G'$，　$msg(g_C)\subseteq lt(S)$

(4)　　$g=g\cup\{lt(S)/msg(g_C), C\}$

(5)　　$S=S+expand(lt(S)/msg(g_C), g_C)$

(6)　**return** $(g, lt(S))$

在约化结果 g 的生成过程中，算法维护了 g 表示的多项式本身 S。为此，需要使用节点的展开算法 *expand*。

算法 2. $expand(t, p)$，获得 $t\times p$ 的项集。

(1)　$U=\Phi$

(2)　**if** p 为根多项式

(3)　　return $t\times p$

(4)　**for** p 的每个分支(t_j, i_j)

(5)　　$U=U+expand(t\times t_j, g_{ij})$

(6)　**return** U

计算完成后，S 被释放，只有 $lt(S)$作为 $msg(g)$保存下来。

注意，*expand* 算法并非简单地直接自底向上递归计算，而是首先将生成信息 t_j 自顶向下传递。

现在比较 BanYan 与基于项的传统布尔多项式表示下，约化结果 g 的生成速度。由算法 2 可知，BanYan 下 $g=\Sigma_j(t_j\times g_{ij})$的生成最终归结为根多项式的运算。

追溯 g 与根多项式的关系，可得到 $g=\Sigma_k(\Pi_h t_{k,h}\times p_{ik})$的形式，其中 $t_{k,h}\in T$，$p_{ik}\in P$。故 BanYan 结构下生成 g 的总时间复杂度为 $O(\Sigma_k(|p_{ik}|\times\lg|p_{ik}|))$。另一方面，若采用传统的基于项的多项式表示，即直接由 g_{ij} 生成 g，总时间复杂度则为 $O(\Sigma_j(|g_{ij}|\times\lg|g_{ij}|))$。考虑到 g 稀疏时有$\sum_j|g_{ij}|\approx\sum_k|p_{ik}|\approx|g|$，故 BanYan 结构虽然并未保存 g_{ij} 本身，但 g 的生成速度并不低于传统表示。

事实上，这是由布尔多项式相较一般非线性多变元多项式的数学特性决定的，即对任意项 $t_A>t_B$和任意 $t_C\in T$，$t_A\times t_C>t_B\times t_C$不总是成立。考虑从 g 的某个分支 g_{ij} 到根多项式 p 的一条路径$\Pi_h t_h\times p$，传统多项式表示相当于要计算各个$\Pi_h t_h\times p$的严格有序形式，但除首项外，g_{ij} 中各个项的相对序信息对于 g 的计算并无用处，最终在项乘计算 $t_j\times g_{ij}$ 中完全丢失。而在 BanYan 结构下，只有路径信息$\Pi_h t_h\times p$和必要的首项信息 $msg(g_{ij})$被记录下来。在 $t_j\times g_{ij}$ 的计算中，BanYan 可以直接计算$(t_j\times\Pi_h t_h)\times p$，其时间复杂度并不高于计算 $t_j\times g_{ij}$。因此，保存分支 g_{ij} 本身，不仅带来了空间上的负担，在时间上也不能带来速度提升。

§6 实验结果

借助 GNoMoN 通用密码计算平台[89]，我们对比了 BanYan 与 BDD 在各种不同规模的布尔多项式求解问题下的实际工作表现，见下表：

表 5-1 BanYan 与 BDD 的空间需求对比

问题规模		中间结果规模(项数)			空间消耗(KB)		
n	d	BDD	BanYan	%	BDD	BanYan	%
25	3	983，522	4，637	**0.47**	16，452	35	**0.21**
	10	217，768	1，875	**0.86**	15，664	13	**0.08**
50	3	555，188	3，182	**0.57**	15，840	25	**0.16**
	25	55，416	622	**1.12**	15，656	3	**0.02**
100	3	524，398	5，263	**1.00**	15，628	38	**0.24**
	25	37，664	390	**1.04**	15，716	4	**0.03**
200	3	463，516	4，684	**1.05**	15，624	36	**0.22**
	25	27，694	280	**1.01**	15，640	3	**0.02**

实验对象为给定变元数(n)，次数(d)和项数(l，固定为 100)条件下的随机布尔方程组，方程组中的方程个数等于变元数。求解工作使用同一求解算法(优化的 Buchberger 算法)。表 1 展示了分别使用 BDD 和 BanYan 表示，计算工作到达统一状态点时(BDD 表示下节点数量到达 1，000，000，约 16MB 空间开销)，中间结果的项数规模和空间消耗(不包括问题本身的存储开销)。

可以看到，截止到统一状态点，BanYan 的中间结果规模只有 BDD 的约 1%($1/l$)。而且随着计算的深入(本实验中对应于变元数较小时)，这一比值还会继

续减小。

空间消耗方面，在 BDD 表示下，附加空间开销达到约 16MB 时，BanYan 的空间开销普遍不超过 50KB，而且差距在高次方程求解中会更为明显。

§7 小结

本章给出了布尔多项式的一种高效计算机表示，BanYan。作为 BDD 等基于项的传统多项式表示的完美替代，BanYan 可以显著提升各求解算法的计算能力极限。其设计思想与基于消项的布尔方程组求解理论高度一致，通过最大限度地避免求解过程中无意义的存储和计算，BanYan 可以在不影响计算速度的前提下，大大降低求解工作的空间需求。

第六章　基于变元猜测的对 Bivium 流密码的代数攻击

非线性方程组的求解是代数攻击的关键一环。对于一个具体的密码系统，在转化为方程组后，由于其计算上的复杂性，一般采用先猜测部分变元，再进行求解分析的方法。本章首先给出了对于猜测部分变元后子系统平均求解时间的估计模型，提出了基于动态权值以及静态权值的猜测变元选则方法和面向寄存器的猜测方法。在计算 Groebner 基的过程中，对变元序的定义采用了 AB，S，S-rev，SM，DM 等十种新的序。同时，提出了矛盾等式的概念，这对正确分析求解结果以及缩小猜测空间有重要作用。最后，我们对 Bivium 流密码算法的攻击时间进行了估计。结果表明，在最坏情况下，使用 DM-rev 序及 Evy3 的猜测位置，猜测 60 个变元有最优的攻击结果，约 $2^{39.16}$ 秒。

§1 引言

早在 1949 年，Shannon 便指出:破译一个密码系统所做的工作等价于对一个多变元的非线性方程组的求解。即把密码系统转化为多变元方程系统后，密码破译的难点在于有效转化与对该方程组的求解，然后从中得到原始密钥。这是最早出现的代数攻击的思想。

近年来，代数攻击逐渐成为密码分析中的热点，出现了大量的对于分组密码，流密码，公钥密码的研究成果[90-99]。代数攻击的发展也促进了解方程技术的进一步发展。如线性化算法，XL 算法，吴方法 Groebner Basis 算法等。其中，Groebner Basis 算法最令人瞩目，其实现版本包括 Buchberger 算法，slimgb 算法，F_4，F_5 算法等。2003 年的美密会上，Faugère 公布了他对 HFE 公钥密码系统的分析结果，即使用 F_5 算法可以在 2 天内破解 HFE Challenge 1。 然而，众所周知，求解多变元二次方程系统，也即 MQ 问题是一个 NP 难题。对于更大规模的方程系统，上述算法均很难直接求解。

先猜测部分变元，把原始的大的方程系统化为若干小的方程系统，逐个进行求解分析是一种变通的做法。如可以指定猜测 n 个变元，把整个方程系统化为 2^n 个子系统， 一旦求出某个子系统中其余变元的值，便可从中恢复密钥。否则，若方程无解，说明猜测错误，继续进行下一组猜测。在最坏情况下的猜测数为 2^n 个子系统。本章针对 Bivium 流密码所产生的方程系统，采用 Groebner 基算法进行了相关求解分析，如何选择最优的猜测变元，最优的 Groebner 基算法的变元序以及最佳的猜测变元个数是本章主要要解决的问题。其相关实验均是在最坏情况下。

本章的结构是这样安排的:在第 2 节，我们对 Bivium 流密码进行了简单介绍。当 n 比较大时，如何有效地估算 2^n 个子系统的平均求解时间是首先要解决的问题，本章在第 3 节给出了相应的估计模型;不同的猜测变元集合，对于平均求解时间有很大影响;本章在第 4 节给出了如何选择合适的变元进行猜测的策略;变元

序以及 Groebner 基算法的选择对于求解结果同样有较大的影响，在第 5 部分对该内容进行了相关讨论;在本章的第 6 部分，给出了矛盾等式的概念，并指出利用矛盾等式对正确分析求解结果以及缩小猜测空间有重要作用。最后，对 Bivium 流密码进行了相关实验。

§2 Bivium 流密码算法描述

Trivium[100-101]是由 C.De Canniere 和 B.Preneel 在 2005 年为欧洲流密码计划而设计的流密码算法 Bivium 是 Trivium 的一个简化版本。

Bivium 的主要参数包括：80 比特的密钥，80 比特的初始值，177 比特的内部状态。Bivium 工作分两步，第一步由 80 比特的密钥和 80 比特的初始值经过初始化，生成 177 比特的初始内部状态。第二步对内部状态进行非线性反馈移位，并由关于内部状态的线性函数输出密钥流比特。密钥流由以下步骤生成：

算法 1. Bivium(m)

1: **for** $i = 1$ to m **do**

2: $t_1 \rightarrow s_{66} + s_{93}$

3: $t_2 \rightarrow s_{162} + s_{177}$

4: $z_i \rightarrow t_1 + t_2$

5: $t_1 \rightarrow t_1 + s_{91}s_{92} + s_{171}$

6: $t_2 \rightarrow t_2 + s_{175}s_{176} + s_{69}$

7: $(s_1, s_2, \ldots, s_{93}) \rightarrow (t_2, s_1, \ldots, s_{93-1})$

8: $(s_{94}, s_{95}, \ldots, s_{177}) \rightarrow (t_1, s_{94}, \ldots, s_{176})$

9: **end for**

这里 m 表示需要生成的密钥流比特数，$m \leqslant 2^{64}$。初始状态的生成，是把 80 比特的密钥，80 比特的初始值和一些 0/1 比特装载入内部状态，然后经过 177×4 轮状态更新得到。并且这个内部状态更新与第二步中的内部状态更新完全相同，只是不输出密钥流比特。用变量 s_1，s_2，$\ldots s_{177}$ 表示 177 比特内部状态，用 z_i 表示每个时钟所产生的密钥流。内部状态的更新是可逆的，只要知道了初始内部状态，就可以恢复出 80 比特的密钥和 80 比特的初始值。目前对 Bivium 的攻击都是假设知道某一时刻及其以后的一段密钥流，试图恢复出这个时刻的内部状态。

由算法 1，可以写出变元 s_i 与 z_i 之间的等式。对于 Bivium 流密码系统，可以有两种方法写出其方程系统[96-97]，一种是不添加任何新的变元，即保持 s_1，s_2，$\ldots s_{177}$ 作为变元。它所产生的方程系统的次数随时钟而递增。另一种是每个时钟增加两个新的变元和三个方程，这样做的优点是保持整个系统的稀疏性，并且每个方程的最大次数不超过 2。对 Groebner 基算法来说，前一种方法有更快的求解时间[96]，其产生的方程系统如下:

$$
\begin{aligned}
&s_{66} + s_{93} + s_{162} + s_{177} + z_1 = 0; \quad \text{1st clock} \\
&s_{65} + s_{92} + s_{161} + s_{176} + z_2 = 0; \quad \text{2nd clock} \qquad (1) \\
&s_{64} + s_{91} + s_{160} + s_{175} + z_3 = 0; \quad \text{3rd clock} \\
&\cdots\cdots
\end{aligned}
$$

§3 变元猜测的攻击方法与统计模型

3.1 算法描述

由于在代数攻击中，对于大的方程系统，很难直接求解。所以，一般采用先猜测 n 个变元，然后再进行求解分析的方法。具体如算法 2 所描述:

算法 2. GDAlgorithm(*ES*， *n*)
输入:环上方程系统 *ES*;猜测的变元数，*n*.
输出:0/1，表示是否找到解.

```
1.  选择 n 个变元的所有猜测集合置为 GSet
2.  for vc in GSet do
3.      把 vc 中变元的值代入方程系统 ES，得到 ES(vc)
4.      if ES(vc)有解
5.          记录 vc 的值
6.          return 1
7.      end if
8.  end for
9.  return 0
```

在算法 2 中，对于猜测 n 个变元，有 2^n 种猜测组合，每种猜测值代入后，产生一个方程子系统。假如原方程系统有唯一解，则在 2^n 种猜测组合中，只有一个是正确的。设 $\bar{t}$ 表示求解一个错误猜测子系统的平均时间，t'表示求解一个正确猜测子系统的时间。在最坏情况下，找到正确解所要花费的时间是:

$$T = (2^n - 1)\bar{t} + t', \bar{t} = \frac{1}{2^n - 1}\sum_{i=1}^{2^n-1} t_i \qquad (2)$$

然而，在现实中，当猜测的变元数 n 很大时，如 n=32，则要分别计算 2^{32} 数量级的子系统方程。当每个子方程系统的求解时间较长时，(如超过 5 分钟)，则很难统计出每个子系统的求解时间 t_i，从而计算出所有子系统的平均求解时间 $\bar{t}$，选择合适的统计模型是我们下面要解决的问题。

3.2 统计模型

当猜测的变元数 n 很大时，一般在 2^n 个子系统方程中选择 k 个，用这个 k 个子系统方程的平均求解时间来估计 2^n 个子系统的平均求解时间 $\bar{t}$[54]:

$$\bar{t} \approx \frac{1}{k}\sum_{i=1}^{k} t_i \tag{3}$$

然而，猜测变元的汉明重量(猜测变元的非零值数目)对于求解时间有重要的影响。一般来说，汉明重量越小，则求解时间越快。这是因为大部分的变元都被零值取代，从而使整个方程系统更为稀疏，平均而言，每个方程的次数也更小，从而更容易求解。

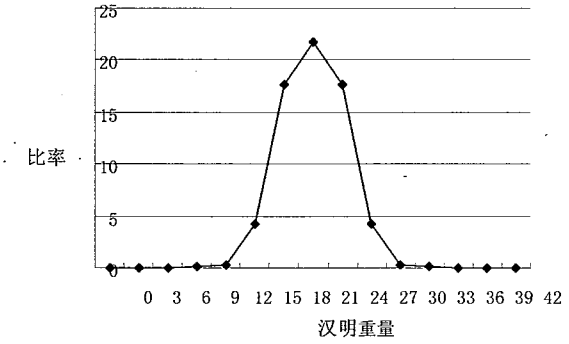

图 6-1 汉明重量比率分布图

所以，随机选取 k 个子系统方程很难准确的估计 2^n 个子系统方程的平均求解时间。我们采用根据不同汉明重量猜测值的比率，在整个猜测空间中选取。假如 n 个猜测变元的汉明重量为 $d(0\leqslant d\leqslant n)$，则其所占的比率为 $R_d=C_n^d/2^n$。图 6-1 给出了猜测变元 n=42 时的不同汉明重量猜测值的比率比率分布。

由图 1 可见，汉明重量接近 $n/2$ 的猜测占了整个猜测空间的绝大部分。因此，在实验中，我们按照如下步骤进行计算:

1. 对所选取的 k 个子系统方程，汉明重量为 $d(0\leqslant d\leqslant n)$ 的猜测应占到 $n_d=[kR_d]$ 个。

2. 对每个汉明重量 $d(d=0, 1, \ldots, n)$，计算求解平均值 $\bar{t}_d$:

$$\bar{t}_d=\frac{1}{n_d}\sum_{i=1}^{n_d} t_{di} \tag{4}$$

其中 t_{di} 是猜测汉明重量为 d 的一个方程子系统的求解时间。

3.整个 2^n 个子系统的平均求解时间 $\bar{t}$:

$$\bar{t} \approx \sum_{d=0}^{n} R_d \bar{t}_d \tag{5}$$

§4 对猜测变元的选取

在实验中，对于特定的猜测变元数 n，选取不同的猜测变元，对于平均求解

时间 t 有很大影响。如何选取使求解效率更高的变元去猜测，是我们下面要讨论的问题。这里，分两大类去讨论，其中基于权值的策略不仅对于 Bivium 流密码，对于任何布尔方程组均适合。

4.1 基于权值的选择策略

对于每个方程系统中的变元，我们可以以一定的策略统计所有变元的权值，然后从中选择权值最大的 n 个变元去猜测。根据权值计算方法的不同，分为静态权值与动态权值两种。

1.静态权值

静态权值即只是在变元替换前，对整个系统中的每个变元计算一次权值，然后选择权值最大的 n 个变元进行猜测。一种直接的方法是按照变元在整个方程系统中出现的次数作为该变元的权值，这样选出猜测的 n 个变元即为方程系统中出现变元次数最多的 n 个变元。

在实验中，我们采用了更为科学的权值计算方法。因为每个变元的重要性不仅与其出现次数有关，而且与其出现位置有关。从解方程得角度来说，出现在高次项中的变元更值得去猜测，因为这样可以更大限度的降低该多项式的次数，使整个方程系统更容易求解。鉴于此，我们把单项式的次数作为出现在单项式中每个变元的权值，然后再对整个方程系统中所有出现该变元的权值进行累加。最后选择权值最大的 n 个变元进行猜测。

举例来说，对于多项式 $s_1+s_1s_2+s_1s_2s_3$，由于 s_1 分别出现在三个单项式里，所以其权值为 1+2+3=6，同理，s_2，s_3 的权值为 5，3。

2.动态权值

静态权值计算简单，但却没有反映出变元在猜测过程中的重要性变化。实际上，对于单项式 $s_1s_2...s_m$ 来说，一旦其中某个变元 s_i 作为猜测变元，则 $s_j(j\neq i)$ 的重要性立刻降低了。举例来说，对于多项式 $s_1s_2s_3+s_4s_5s_6$，若 s_1 作为猜测变元，则 s_2, s_3 或者不会再出现(s_1=0)，或者变为二次的单项式 s_2s_3，则此时，选择 s_4, s_5, s_6 中的任一变元作为下一个猜测变元无疑会更大限度的降低整个多项式的次数。因此，我们设计了动态权值的计算方法:第一个猜测变元依据静态权值的计算方法进行选取，即选择初始权值最大的那个变元。一旦该变元确定后，我们重新计算剩余各个变元的权值，更新权值表，然后选择下一个权值最大的变元。

权值的更新策略是依据概率来进行的。如对于单项式 $s_1s_2s_3$， 若 s_1 作为猜测变元，则 s_1 为 0 或 1 各有 50%的概率，这样 s_2 和 s_3 的权值就等于 0×0.5+2×0.5，即为 1。我们在算法 3 对动态权值进行更细致的描述。

算法 3 DynamicWeightAlgorithm(*ES*， *n*)

输入:环上方程系统 *ES*;猜测的变元数 *n*.

输出:$list_2$ 中选定的 n 个猜测变元.

1: **while** 1 **do**
2: **if** | $list_2$| = n **then**// $list_2$中存放最终选择的n个变元
3: **return** $list_2$;
4: **end if**
5: 对所有的变元权值置0，结果存在$list_1$中
6: **for** ES 中的每个单项式$m_i = s_1 s_2 \ldots s_j$ **do**
7: **for** m_i 中的每个变元v **do**
8: **if** v在$list_1$中 **then**
9: v的权值增加$x \times 0.5^y$. 这里， x是m_i中属于$list_1$中的变元个数， y是m_i中属于$list_2$的变元
10: **end if**
11: **end for**
12: **end for**
13: 把当前$list_1$中权值最高的变元移到$list_2$
14: **end while**

这里，第一次循环由于 $list_2$ 为空，所以选出的是按静态权值策略计算出的最大权值的变元.随后，一旦最大权值的变元被选出，与该变元在同一单项式的变元的权值都会降低，然后，在动态变化的基础上，选择权值次大的变元，直到选择 n 个变元为止。

4.2 面向寄存器的选择策略

对于 Bivium 流密码来说，我们还可以从其生成密钥流的寄存器结构来选择具体猜测变元的位置。我们知道，Bivium 流密码的方程系统是随时钟产生的。由于加/解密效率的原因，在每一个时钟周期，并非所有的变元都参与了方程系统的生成。因此，原则上，我们选择最先参与生成方程系统的变元或对方程系统次数有重要影响的变元。

对于 Bivium 流密码，令两个寄存器中的从前到后的变元分别为 s_1，s_2...，s_{93} 和 s_{94}，s_{95}，...s_{177}。根据算法 1，可总结出 Bivium 流密码的以下特征：

1. 首先，存贮在两个寄存器后面部分的变元更早的参与了方程系统的生成，如 s_{93}，s_{66}，s_{177} 和 s_{162} 出现在第一个时钟周期所生成的等式;而变元 s_1，s_{94} 分别在第 66 和 69 个时钟周期才出现。
2. 非线性的单项式都来自于两个寄存器的倒数 2，3 个变元。
3. 所有的非线性的单项式，其变元的下标都是连续的，即形如 $s_i s_{i+1} s_{i+2}$.....

依此三条特征，表 6-1 给出了一些最优猜测变元的位置。由特征 1，2，我们可以选择每个寄存器的最后变元进行猜测，具体包括 End1，End2，End0;由特征 1，2，3，可以每间隔 2、3 个变元进行猜测，也即 Ery2，Ery3。

§5　Groebner 基算法中对变元序的选择

对一个多项式方程组 F 来说，其 Groebner 基与该方程系统有同样的解。但是，由于 Groebner 基算法的消去属性，其最终结果中会有一个多项式是单变元的。于是，该变元的解可以很方便得到;随后，我们把该变元的值代入到求出的 Groebner 基中，于是，我们又能获得只含一个变元的多项式，依次类推，直至求出所有变元的解。实际上，Groebner 基算法可以看作一般化的高斯消元算法。对于 Bivium 密码系统来说，由于其方程组只含有一个解，于是可以通过计算出既约 Groebner 基来求出方程系统的解。既约 Groebner 基形如 $G=\{x_1+s_1, \ldots, x_n+s_n\}$，则（$s_1, \ldots, s_n$）即为原方程系统的解。

Groebner 基算法的最快的两个实现是 Faugère 的 F_4 与 F_5 算法，当前比较流行的代数分析软件中基本都实现了 F_4 算法。在实验中，我们选择了当前最流行的三款软件 Singular，MAGMA 以及 PolyBoRi，其内嵌的 Groebner 基算法算法都是基于 F_4 算法或 F_4 算法的变形。

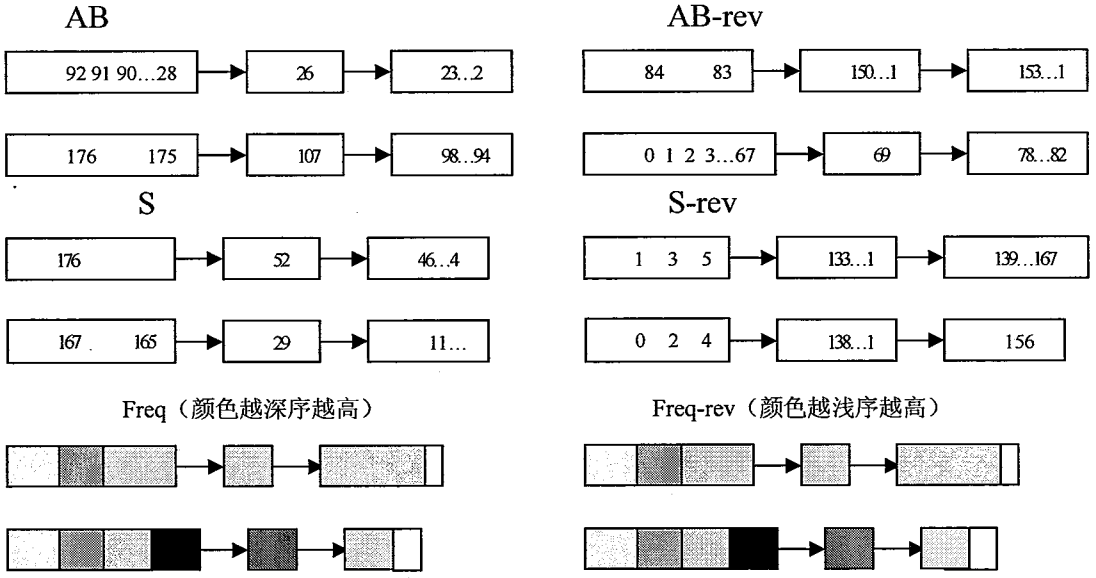

图 6-2 Bivium 密码系统变元序图示

表 6-1 猜测变元位置说明

对 Groebner 基算法而言，有两个序非常重要。一个是变元序，即规定哪个

Ery3	从两个寄存器的最后一位开始，交替的每隔 2 个变元选择下一个作为猜测变元
Ery2	与 Ery3 类似，每隔 1 个变元选择下一个作为猜测变元
End1	选择第一个寄存器后的若干变元作为猜测变元
End2	选择第二个寄存器后的若干变元作为猜测变元
End0	同时选择两个寄存器后的若干变元作为猜测变元
SW	静态权值策略
DW	动态权值策略

变元优先级高;一个是单项式序，规定了整个多项式的单项式排列顺序。变元序一般分字典序，反字典序;而单项式序一般有字典序，分次字典序，分次反字典序等。分次反字典序被公认为求解 Groebner 基的最快的序。对于 Bivium 密码系统，我们给出了 10 种变元序，如表 6-2 所示。前 6 种序的说明见图 6-2[22]。实验表明（见第 7 节），应该选择那种出现不频繁或不太重要的变元先处理，这样可以保持整个方程系统在计算过程中不会膨胀的太快。

表 6-2 Bivium 密码系统变元序说明

AB	在 A 寄存器中最后一个变元的序最高，接着从 A 到 B 依数字从小到大
S	对 AB 寄存器而言，最高序在后面，低序在前面
Freq	按变量出现的频率排序，这里颜色最深的部分序最高
SW	静态权值策略
DW	动态权值策略
*-rev	表示各种序的逆序

§6 矛盾等式

在选择 n 个猜测变量后，首先应该检查猜测后的方程系统中是否存在矛盾等式。利用这些矛盾等式，我们可以缩小变元的猜测空间;而忽略了矛盾等式，有可能得到一个不正确的时间统计。

对于方程 $f(s_1, s_2, \ldots, s_t) = 0$ 来说，如果其中的所有变元均被猜测，且所到的等式为"1=0"，则该等式称为矛盾等式。一旦方程系统中发现矛盾等式，可以立刻推断出，此次猜测是错误的，不必要再计算猜测后整个系统的 Groebner 基，考虑下一组猜测即可。

对于一个密码系统来说，如果猜测变元覆盖了该密码系统的部分线性方程，我们把该部分线性方程间记为 ***GLE***（Guessing Linear Equations），则首先分析该 *GLE* 中有可能产生的冲突等式的比率。由线性代数中对线性方程组的解空间理论可知，对于一个含有 n'个变元，秩为 m 的线性方程组来说，其解的个数或为 $2^{n'-m}$，或 0 个解。

假设总共猜测 n 个变元，其中覆盖到的线性方程中有 n'个变元。对于包含这 n'个变元的线性方程组 *GLE*，若其秩为 m，则共有 $2^{n'-m}$ 个解。而对于所有的 2^n 个猜测组合，理论上有 $2^{n-n'} \times 2^{n'-m} = 2^{n-m}$ 个猜测组合满足 *GLE*。

假设在平均情况下，需要时间 t'去生成一个满足 *GLE* 的猜测，然后设求解满足 *GLE* 的子系统的平均求解时间 $\bar{t}$。则猜测 n 个变元的求解时间如下式表示:

$$T = 2^{n-m}\bar{t} + 2^{n-m}t' \qquad (6)$$

由于 t'远小于 $\bar{t}$，所以 T 近似表示为 $2^{n-m}\bar{t}$。也就是说，对于猜测 n 个变元，

其中覆盖到的线性方程中有线性无关的方程 m 个，则通过预先对矛盾等式的判断，排除了 2^n-2^{n-m} 种导致矛盾等式的猜测组合，才能正确的对 $\bar{i}$ 进行估计，也即整体求解时间 T 的估计。

§7 实验结果

所有实验均是在一台双核（2.6GHz 每核）8G 内存的机器上完成的。时间统计模型采用的是 3.2 节的概率统计模型。对于每次猜测特定数目的变元，我们都进行了约 1000 次的猜测组合。

1. 实验工具的对比

首先在当前最流行的三款软件 Singular，MAGMA 以及 PolyBoRi 中，选择最快的 Groebner 基实现算法。这里，不失一般性，我们选择了 Ery3 位置进行猜测，猜测了 58 个变元，在排除了矛盾等式的情况下，共进行了 1000 次猜测组合，并对结果按概率模型进行了统计，结果如下表：

表 6-3 Bivium 密码系统猜测 58 变元在各种变元序及三种计算工具下的平均计算时间

算法	十种变元序下的平均计算时间									
	AB	AB-rev	S	S-rev	Freq	Freq-rev	SW	SW-rev	DW	DW-rev
slimgb	35.23	32.18	36.57	34.98	35.17	33.68	37.39	35.36	35.57	34.33
PolyBoRi	3.225	2.958	3.197	2.891	3.302	3.235	3.278	3.225	3.354	2.764
magama	9.4	8.1	11.20	10.31	11.12	10.09	12.51	11.57	10.2	9.96

由于 PolyBoRi 是专用的求解布尔方程组的工具，所以效率最高，其底层的多项式采用二元决策图表示，相对于另外两款求解软件，有明显的求解优势.所以，下面的实验均采用 PolyBoRi 计算工具。

2. 变元序与猜测位置的确定

我们使用 PolyBoRi 软件，在 10 种变元序下，选择了 7 种猜测策略进行分析。同样猜测了 58 个变元，在每种序与猜测位置下均生成了 1000 次猜测组合，并对结果按概率模型进行了统计，结果如下表：

表 6-4 Bivium 密码系统猜测 58 变元在 10 种变元序及 7 种猜测位置下的平均计算时间

猜测位置	十种变元序下的平均计算时间									
	AB	AB-rev	S	S-rev	Freq	Freq-rev	SW	SW-rev	DW	DW-rev
Ery2	0.041	0.019	0.040	0.027	0.022	0.027	0.018	0.018	0.022	0.021
Ery3	3.225	2.958	3.197	2.891	3.302	3.235	3.278	3.225	3.354	2.764
End0	0.729	0.428	0.540	0.416	0.606	0.596	0.584	0.574	0.714	0.491
End1	2.333	1.875	5.298	1.407	5.781	5.755	5.788	5.786	4.525	1.378
End2	1.595	0.685	1.456	0.619	1.886	1.813	1.863	1.821	1.374	0.535
SW	2.737	1.864	5.583	1.689	5.732	5.414	5.350	5.281	4.198	1.942
DW	0.298	0.175	0.198	0.220	0.245	0.222	0.230	0.189	0.184	0.148

由表 6-4 数据，我们可以推出以下结论：

1）对各种序而言，反序普遍优于正序，因为反序下，首先选择那种出现不频繁或不太重要的变元先处理，这样可以保持整个方程系统在计算过程中不至膨胀的太快

2）相对而言，最佳的序是 DW-rev，可以推断，对变元在整个方程系统中的重要性进行有效分析，并按照重要性逐渐降低得顺序去逐个消去，对求解效率有很大影响。

3）仅从时间来看，最佳猜测位置是 Ery-2。但是在 Ery-2 位置猜测的 58 个变元并没有覆盖一个线性方程，从而无法减少猜测空间。以表 6-5 来看，尽管 Ery3 位置的计算时间最长，但是其猜测位置覆盖了 20 个线性方程。从而大大减少了猜测空间，即 2^{58} 变为 2^{38}，对于其他（2^{58}-2^{38}）种猜测必然是不满足 20 个线性方程的。

表 6-5 Bivium 密码系统猜测 58 变元在 7 种猜测位置下的最快平均计算时间及总体求解时间估计

猜测位置	Ery3	End2	End1	End0	Ery2	SW	DW
最快时间（秒）	2.764	0.535	1.378	0.428	0.018	1.689	0.148
GLE 个数	20	0	0	2	0	4	3
求解时间估计（秒）	2^38*2.764	2^58*0.535	2^58*1.378	2^56*0.428	2^58*0.018	2^54*1.689	2^55*0.148

4）对于随机多变元方程组，只能采用 SW 或 DW 的猜测方式.而 DW 相对于 SW 更有优势，为随机多变元方程组求解中的变元猜测给出了确定的方法.然而，对于 SW 或 DW 的猜测方式中的权值的大小该如何定义，是我们后续要解决的问题。即要分析出对于特定的方程系统，变元的出现次数更重要还是其幂次越高越重要。

3. 变元猜测个数的确定

在确定了算法，序，猜测位置后，需要对不同的变元猜测个数分别做出时间复杂度的估算，我们选择猜测 62，60，58，56，54 个变元。在 Ery3 猜测位置以及 DW-rev 序下，依概率模型进行了 1000 次猜测组合，并对结果进行了统计。

表 6-6 Bivium 密码系统猜测不同个数变元的平均计算时间及总体求解时间估计

猜测变元数	62	60	58	56	54	52
GLE 个数	22	21	20	19	18	17
时间 $\bar{t}$	0.919	1.112	2.764	6.412	16.126	35.147
时间 T	239.80	239.16	239.38	239.68	240.01	240.14

由表 6-6 可知，最佳猜测变元数是 60 个变元，其计算时间约为 $2^{39.16}$ 秒。

§8 小结

在代数攻击中，对多变元方程组的求解是一个 NP 问题。通过先猜测部分变元，把整个方程系统进行分割，然后选择一个子集进行求解分析，可以有效的估

计整个方程系统的求解复杂度。本章使用了概率分布的方法去估计对子系统求解的时间平均值。对于猜测变元的选择，本章提出了静态权值与动态权值的方法，其中动态权值的方法比较有竞争力，为对于一个随机方程组系统的猜测变元选择提供了理论准则（确切的方法）。同时，提出了矛盾等式的概念，这对正确分析求解结果以及缩小猜测空间有一定的价值。最后，我们通过实验给出了新的对 Bviuim 攻击的结果约为 $2^{39.16}$ 秒，研究更优的猜测策略以及并行求解是我们下一步工作的重点。

第七章　对 CTC 分组密码的抽取初始密钥变元的攻击

本章给出了 CTC 分组密码[102]的字典序方程组描述，在此基础上，通过最简约化消除中间变元，获得只含有初始密钥变元的方程，简称 K_0方程。通过实验表明，对于一对明/密文产生的方程，在获得 K_0方程后再进行求解的方法要优于对 CTC 密码方程的直接求解。当 CTC 密码系统的轮数、S 盒数增大，使得一对明/密文产生的方程在有限资源下无法直接求解时，通过增加多个同一密钥下的明/密文对，以产生更多的 K_0方程，使得问题能够求解。同时，把差分方程引入 CTC 密码系统，在此基础上的获得的 K_0方程会产生多个低次多项式，使得求解速度进一步提高。尤其对于密码系统轮数较多而差分轮数较少的情况，获得 K_0方程后再求解的效率要远优于直接对密码系统方程并上差分方程的求解，有利于使用低轮差分的高概率去攻击更多轮的密码系统。

§1 引言

本章的研究目标是针对于 SP 型的分组密码系统的代数攻击。对于一个密码系统，我们总可以把它写成多变元二次方程系统。然而，众所周知，求解多变元二次方程系统，也即 MQ 问题是一个 NP 难题。对于密码参数（轮数/S 盒个数）较大的方程系统，由于会产生更多的变元与方程，很难直接求解。为了便于研究，我们选择了 CTC 分组密码系统，该密码系统是 Courtois 为了验证代数攻击方法的正确性和有效性而提出的一个实验性分组密码算法。

通过先消除非初始密钥 K_0 的其它所有变元（中间变元），然后再进行求解的方法是本章的主要研究内容，下文简称该方法为 K_0 方法。通过消除同一密钥下的多个明/密文对方程的中间变元，我们可以获得异常超定的仅含 K_0 变元的方程，下文简称 K_0 方程。实验表明，在一对明/密文所生成的 K_0 方程无法在有效时间成功求解时，增加明/密文对的方法效果显著。为了进一步提高求解效率，我们把 K_0 方法与差分攻击相结合，以期望获得更加稀疏，次数更低的 K_0 方程，从而能够攻击密码参数更大的方程系统。

本章的结构是这样安排的:在第 2 节，我们对获得 K_0 方程以及 Groebner Basis 计算的一些数学基础进行了简单介绍。在第 3 节给出了获得 CTC 密码系统的 K_0 方程的具体过程;本章在第 4 节给出了一对与多对明/密文所生成的 K_0 方程的求解与直接求解密码方程的时间对比，并分析了密码参数变化（对数/轮数/S 盒数）对 K_0 方程的影响，K_0 方程的变化直接影响到了求解效率。在第 5 节中研究了差分方程引入代数攻击的基本方法，同时对差分 K_0 方程的求解与对密码方程的直接差分代数攻击进行了对比。最后对本章的工作进行了总结，并指出了下一个分析目标:Present 分组密码系统[103]。

§2 数学基础

2.1 项序

定义 1 项序:设 $T(R)$ 表示环 R 上的项的集合，在 $T(R)$ 上的项序定义$\leqslant$如下:

1. 对所有的项 $t\in T(R)$，均有 $1\leqslant t$

2. 对所有的项 s，t_1，$t_2\in T(R)$，当 $t_1\leqslant t_2$时，均有 $st_1\leqslant st_2$

在 Groebner Basis 运算中最常用的两个序，即字典序与次数反字典序。另 $t=v_1^{e1}\ v_2^{e2}\cdots v_n^{en}\in T(R)$，其幂指数向量定义为$(e_1,\ e_2,\ \ldots e_n)\in N^n$;项的次数 $\deg(t)=\sum_{i=1}^{n}e_i$。

定义 2 字典序(简记为 lex):对于项 s，$t\in T(R)$，其指数向量分别为 $(a_1,\ a_2,\ a_3\ldots,\ a_n)$，$(b_1,\ b_2,\ b_3\ldots,\ b_n)\in K^n$，若 $s<_{lex}t$，则存在 $1\leq k\leq n$，使得 $a_j==b_j$，$1\leq j\leq k$，$a_{k+1}<b_{k+1}$.

由于字典序是一种消去序，所以，其最终结果中会有一个多项式是单变元的。于是，该变元的解可以很方便得到;随后，我们把该变元的值代入到求出的 Groebner Basis 中，于是，我们又能获得只含一个变元的多项式，依次类推，直至求出所有变元的解。然而在实际计算中，字典序的计算效率往往不如次数反字典序，次数反字典序在一般情况下是计算 Groebner Basis 的最快的序[104-105]。

定义 3 次数反字典序(简记为 degrevlex):对于项 s，$t\in T(R)$，其幂指数向量分别为 $(a_1,\ a_2,\ a_3\ldots,\ a_n)$，$(b_1,\ b_2,\ b_3\ldots,\ b_n)\in K^n$，若 $s<_{degrevlex}t$，则或者 $\deg(s)<\deg(t)$；或者 $\deg(s)==\deg(t)$且在幂指数向量中右数第一个不同的坐标位置 j，有 $a_j>b_j$.

定义 4 乘积序(简记为 product):对于变元集合 $x=(x_1,\ x_2,\ x_3\ldots,\ x_n)$，$y=(y_1,\ y_2,\ y_3\ldots,\ y_m)$，$<_1$是环 $R[x_1,\ x_2,\ x_3\ldots,\ x_n]$上的项序，$<_2$是环 $R[y_1,\ y_2,\ y_3\ldots,\ y_m]$上的项序。若在环 $R[x_1,\ x_2,\ x_3\ldots,\ x_n,\ y_1,\ y_2,\ y_3\ldots,\ y_m]$上有 $st<_{product}s't'$，则或者 $s<_1s'$；或者 $s==s'$且 $t<_2t'$

2.2 Groebner Basis

定义 5 Groebner 基:设 I 是环 R 中任意给定的非零理想，$F=\{f_1\ f_2\ f_3,\ldots,\ f_n\}$是 I 中非零多项式的集合，我们称 F 是 I 的 Groebner 基，当且仅当对 I 中任意多项式 f，存在 $1\leq i\leq n$，使得 $lt(f_i)|lt(f)$。其中 $lt(f)$为 f 在序$\leqslant$下的首项。

Buchberger 第一准则[7]中指出若两个多项式首项互素，则这两个多项式没有必要组成关键对去约化。因为其约化结果必然为 0。由该准则，我们立刻可以得到定理 1。

定理 1:设 G 为多项式集合，另 $H=\{lt(f):f\in G\}$，若 H 中的元素均两两互素，则 G 是一个 Groebner 基。

定理 1 给出了不需经过约化而直接构造 Groebner 基的方法。即如果在特定的项序下，所生成的密码系统的多项式的首项两两互素，则该方程系统本身就是一个 Groebner 基。

定义 6 字典序线性首项重写系统(linear lexicographical lead rewriting system):设 L 为一组布尔多项式，如果 L 中的所有元素在字典序下，其首项为线性且均两两互素。则称 L 为字典序线性首项重写系统。

由定理 1，可以推出字典序线性首项重写系统在字典序下构成了一个 Groebner 基。

定理 2（变元消去定理）:设 G 为多项式集合，其中的多项式在字典序下均有线性首项。另 L 为 G 中的最大字典序线性首项重写系统。令 $H=G-L$，用 H 中的元素去最简约化(reduced nomal form)L 会得到多项式结合 N，N 中的多项式不会出现 L 中的任何首项变元。

证明:对于 H 中的多项式，只要其首项依然与 L 中的某个多项式首项相同，就会继续被约化，直到结果中所有的多项式的首项都不出现在 L 的首项集合中为止。也就是得到的结果多项式集合 N。因为在 N 中消去了所有在 L 中的首项变元，所以称该定理为变元消去定理。

举例如下，令 $G=[x_1+1, x_1+x_2, x_2+x_3*x_4]$，则 $L=[x_1+1, x_2+x_3*x_4]$，$H=[x_1+x_2]$，用 H 中的多项式最简约化 L，得到 $N=[x_3*x_4+1]$，也即在 N 中消除了变元 x_1，x_2

§3 获得 CTC 密码系统的初始密钥 K_0 组成的方程

3.1 CTC 密码算法描述

CTC 算法是 Courtois 为了验证代数攻击方法的正确性和有效性而提出的一个试验分组密码算法。该算法采用 SPN 结构，分组长度 $Bs(1\leqslant B\leqslant 128, s=3)$位、密钥长度 Bs 与迭代轮数均可变，轮函数包括轮密钥加、S 盒混淆和扩散变换。

通过选择合适的变元序，我们可以获得以单变元为首项的 CTC 分组密码系统的方程。这是获得初始密钥 K_0 方程的先决条件。对于 r 轮的 CTC 分组密码，下面的过程重复 r 次。

1)轮密钥加

轮密钥加过程即每一轮的输出变元 $Z_{i-1,j}$ 加上轮密钥 $K_{i-1,j}$ 得到下一轮的输入变元 $X_{i,j}$ 的过程，这里 $1\leq i\leq r$ ， $0\leq j\leq Bs-1$。 所以，只要定义项序 $Z_{i-1,j}<X_{i,j}$ ，$K_{i-1,j}<X_{i,j}$ ，每轮均会产生 Bs 个首项两两互素的多项式。这些多项式的首项是 $X_{i,j}$。对于初始轮来说，变元 $Z_{0,j}$ 即为输入明文

2)S 盒混淆

混淆变换是由 B 个 3 比特 S 盒并置构成，每个 S 盒由非线性变换表{7， 6， 0， 4， 2， 5， 1， 3}来确定。设 S 盒的以 $X_{i,j}$ 作为输入变元，$Y_{i,j}$ 作为输出

变元，这里 $1\le i\le r$ ， $0\le j\le Bs-1$。 由于 S 盒的输入变元与输出变元的关系是非线性关系， 由待定系数法获得的 CTC 密码系统 S 盒方程见附录 2。所以，只定义项序 $X_{i,j}<Y_{i,j}$ 并不能保证 S 盒多项式的首项为两两互素的单变元。。

为了使 S 盒能够写成 $Y_{i,j}$=SBOX($X_{i,j}$)的形式，我们定义如下多式环:

$R<y_1, y_2, y_3, x_1, x_2, x_3>$=BooleanPolynomialRing(6, order='degrevlex(3), degrevlex(3)')

这里用 x_i， y_i，$1\le i\le 3$ 代表任一 S 盒的输入输出变元。在环 R 上的序是乘积序，且 $x_i<y_i$，在 x 或 y 变量内部，采用次数反字典序。计算既约 Groebner bases，所得到的结果如下:

$$y_3 + x_1*x_2 + x_1*x_3 + x_2*x_3 + x_2 + x_3 + 1,$$

$$y_2 + x_1*x_3 + x_2 + 1,$$

$$y_1 + x_1*x_2 + x_1 + x_2 + x_3 + 1$$

得到 S 盒的精简表示后，定义项序 $X_{i,j}<Y_{i,j}$，则每轮产生 $3B$ 个首项两两互素的多项式。这些多项式的首项是 $Y_{i,j}$。

3)扩散层

扩散变换是线性变换。具体定义如下:

$Z_{i,\ (257 \bmod Bs)} = Y_{i,\ 0}$ $\quad 1\le i\le r$

$Z_{i,\ (j\cdot 1987+257 \bmod Bs)} = Y_{i,\ j} \oplus Y_{i,\ (j+137 \bmod Bs)}$ $\quad 1\le i\le r$ ， $0\le j\le Bs-1$

扩散层以 $Y_{i,j}$ 作为输入变元，$Z_{i,j}$ 作为输出变元，这里由于输入变元与输出变元的关系是线性关系。所以，只要定义项序 $Y_{i,j}<Z_{i,j}$，每轮均会产生 Bs 个首项两两互素的多项式。这些多项式的首项是 $Z_{i,j}$。

4)子密钥生成

轮子密钥仅是通过初始密钥循环移动若干比特位得到。具体定义如下:$K_{i,j}=K_{0,\ (j+i \bmod Bs)}$。所有轮的子密钥 $K_{i,j}$ 仅与初始轮密钥 $K_{0,j}$ 有关，且仅存在线性关系，这里 $1\le i\le r$ ， $0\le j\le Bs-1$。 所以，只要定义项序 $K_{i+1,j}<K_{i,j}$，每轮均会产生 Bs 个首项两两互素的多项式。这些多项式的首项是 $K_{i,j}$。

3.2 生成 K0 组成的方程

对于 r 轮的 CTC 密码系统，其变元序如下:$K_{0,j}<\cdots<X_{i,j}<Y_{i,j}<Z_{i,j}<K_{i,j}<\cdots X_{r,j}<Y_{r,j}<Z_{r,j}<K_{r,j}$。也就是说变元序是按照变元生成的顺序依次升高的。我们给出一轮一个 S 盒所生成的方程，其变元序由高到低如下:K10， K11， K12， Z10， Z11， Z12， Y10， Y11， Y12， X10， X11， X12， K00， K01， K02，生成方程见附录 1。

这里需要注意的是最后一轮密钥加并没有生成新的变元。因为其输出结果即

为密文，为已知常量。这样与子密钥生成的首项变元 $K_{1,j}(0\leq j\leq 2)$重合，由定理 2，可利用最后一轮的重叠首项来获得仅含初始密钥的变元 K_0，即用轮密钥加多项式去最简约化其余各层多项式。因为在所有的多项式中，除了 $K_{0,j}$ $(0\leq j\leq 2)$变元外，其他所有变元均在多项式的首项出现过。

§4 直接求解与 K_0 系统求解对比

4.1 由一对明/密文所生成的方程求解对比

以下实验均是在一台内存 4G 的双核 2.6GHz 的 CPU 上进行，使用 PolyBoRi 作为方程求解工具。每次求解均取 100 次计算的平均值。计算结果中“NM”表示无内存，“-”表示不存在或不需考虑。这里分别定义 Attack1 与 Attack2。Attack1 代表直接求解一对明/密文的方程。Attack2 代表从一对明/密文的方程中仅仅抽取初始密钥变元 K_0 所生成的方程组。Attack1 与 Attack2 的求解对比如表 7-1: Attack2 的性能明显优于 Attack1。这是因为 Attack2 通过耗时极短的消去中间变元，使整个系统变成极其超定的只含密钥变元 K_0 的方程。然而当轮数与 S 盒个数增加到一定数目时，Attack1 与 Attack2 均难在短时间内求解。

表 7-1 CTC 在不同轮和 S 盒个数下对一对明/密文方程的的平均计算时间

轮数	S 盒个数							
	2		3		4		5	
	Attack1	Attack2	Attack1	Attack2	Attack1	Attack2	Attack1	Attack2
1	0.019	0.018	0.034	0.015	0.046	0.016	0.115	0.076
2	0.051	0.021	0.200	0.114	0.721	0.353	3.404	2.291
3	0.192	0.022	2.127	0.290	15.143	14.529	11947.675	5434.245
4	0.381	0.031	26.683	0.446	NM	32.867	NM	NM

4.2 由多对明/密文所生成的方程求解对比

当一对明/密文的方程无法在有效时间内求解时，通过增加由同一密钥生成的明/密文对的个数，以增加对密钥变元的关系描述。这里分别定义 AttackA 与 AttackB。AttackA 代表直接求解多对明/密文的方程。AttackB 代表从多对明/密文的方程中仅仅抽取初始密钥变元 K_0 所生成的方程组。AttackA 与 AttackB 的求解对比如表 7-2:

表 7-2　CTC 在不同轮和 S 盒个数下对多对明/密文对的方程的平均计算时间

轮数/S 盒个数	明/密文个数							
	2		3		4		5	
	AttackA	AttackB	AttackA	AttackB	AttackA	AttackB	AttackA	AttackB
3/4	14.827	9.091	15.671	9.245	26.485	8.369	76.391	9.434
4/3	44.609	0.286	164.105	0.395	366.308	0.462	NM	0.682
3/5	-	742.638	-	731.109	-	268.681	-	545.682
4/5	-	2503.257	-	2499.113	-	2521.636	-	2486.274

AttackA 在增加方程数的同时，也增加了变元数。所以，其求解时间随明/密文对的增多而快速增长。其求解效率远不如只求解一对的情况。所以对于 AttackA 的后两种算例我们不再考虑。而由于 AttackB 仅仅是把初始密钥 K_0 所生成的方程组抽取出来。在相同的初始密钥下，当明/密文增多时，其变元数并没改变；而每对明/密文都会增加对初始密钥 K_0 的关系，从而使 AttackB 去求解一个异常超定的系统。然而，对数增加到一定数值时，其求解效率一般趋于稳定，不再有显著提高。如 3 轮 4S 盒，当求解 4 对时，其求解效率最佳。但所有多对计算时间都优于一对的计算时间。

当 AttackB 在有效时间能求解一对明/密文时，增加更多的对一般会有两倍的加速。但一旦一对明/密文 AttackB 难以在短时间内求解时，增加对的个数时，其攻击效果明显。如 3 轮 5S 盒，增加到 4 对时，约有 20 倍的加速。

4.3 对初始密钥 K_0 组成的方程分析

1）轮数增加对 K_0 方程的影响

如表 7-3 所示，轮数增加时，方程的总体个数不变，但生成方程次数总体来说不断升高。直到达到最高次，即所有变元总数时，次数不能再升高。方程在最高次个数有较小波动。这也在某种程度上解释了随着轮数的增加，Attack2 的计算时间会迅速增加，但到了一定范围后，增加趋于平缓。因为此时轮数的增加已不再生成更多的高次方程。 表 7-3 CTC 在不同轮数下的 K_0 方程统计

对数/轮数/S 盒个数	多项式在不同 degree 个数					
	7	8	10	11	12	Total
5/3/4	48	12	-	-	-	60
5/4/4	-	-	-	52	8	60
5/5/4	-	-	-	30	30	60
5/6/4	-	-	-	26	34	60
5/7/4	-	-	-	31	29	60
5/8/4	-	-	-	31	29	60
5/9/4	-	-	-	26	34	60

2）对数增加对 K_0 方程的影响

如表 7-4 所示，对数增加时，方程个数按一定比率增加，最高次方程个数增

加，同时，低次方程个数也增加，但次数保持不变.即对数越多，低次方程越多，系统越超定。此时，有可能仅由低次的方程组就可以解出全部解。

表 7-4　CTC 在不同对数下的 K_0 方程统计

对数/轮数/ S 盒个数	多项式在不同 degree 个数		
	14	15	total
3/5/5/	22	23	45
4/5/5/	36	24	60
5/5/5	40	35	75
6/5/5	41	49	90
7/5/5	48	57	105
8/5/5	57	63	120
9/5/5	70	65	135

3）S 盒个数增加对 K_0 方程的影响

如表 7-5 所示，方程个数与次数均迅速增加，所以 K_0 方法的计算时间随 S 盒个数的增加而迅速增长。如何把 K_0 方程进行有效的降次，是我们在下一节要解决的问题，即引入差分方程。

表 7-5　　CTC 在不同 S 盒个数下的 K_0 方程统计

数/轮数/ S 盒个数	多项式在不同 degree 个数								
	8	9	11	12	14	15	17	18	total
5/5/3	23	22	-	-	-	-	-	-	45
5/5/4	-	-	23	37	-	-	-	-	60
5/5/5	-	-	-	-	34	41	-	-	75
5/5/6	-	-	-	-	-	-	47	43	90

§5 引入差分方程后获取 K_0 变元方程的求解分析

5.1 差分方程的引入

密码设计者会精心设计分组密码的 S 盒与混淆层，以使当差分轮数升高时，差分特征成立的概率大大降低。所以，对现代分组密码来说，很难找到高概率的高轮数差分。对差分分析来说，利用低轮差分的高概率结合代数攻击的方法是一种降低所需要明/密文对的有效方法[106-108]。

设 CTC 密码系统的两对明/密文的方程分别为 F'，　F''。对于 r 轮差分特征 Δ，若每一轮的差分概率为 P_i，则 r 轮差分特征的有效概率:$P=\prod_{i=1}^{r} P_i$，$(1\leqslant i\leqslant r)$。则由每轮的差分所引起的方程为:$X'_{i,j}+X''_{i,j}=\Delta x_{ij}$ 与 $Y'_{i,j}+Y''_{i,j}=\Delta y_{ij}$，$(1\leqslant i\leqslant r,\ 1\leqslant j\leqslant Bs-1)$。$\Delta x_{ij}$ 和 Δy_{ij} 是已知的被激活的 S 盒所决定的差分常量；而未被激活的 S 盒，其对应位的差分为:$X'_{i,j}+X''_{i,j}=0$ 与 $Y'_{i,j}+Y''_{i,j}=0$，$(1\leqslant i\leqslant r,\ 1\leqslant j\leqslant Bs-1)$。

为了使 r 轮差分特征 Δ 有最大的差分概率，应该选择一条每轮被激活的 S 盒个数最少的路径。实验中，对于 B=4，B=5，我们令输入差分的最低位为 1，其余位为零。表 7-6 给出了前五轮的差分路径，这里每一轮都只有一个活动 S 盒。

表 7-6 CTC 在 4/5 S 盒下的差分路径描述

轮	B=4	B=5	概率
In1	100000000000	100000000000000	1
Out1	010000000000	001000000000000	1/4
In2	110000000000	010000000000000	1/4
Out2	001000000000	010000000000000	1/16
In3	000000011000	000000000110000	1/16
Out3	000000010000	000000000001000	1/64
In4	000000110000	000011000000000	1/64
Out4	000000001000	000010000000000	1/256
In5	011000000000	110000000000000	1/256
Out5	010000000000	001000000000000	1/1024

令 $\bar{F}$= $F'\cup F''\cup$ 由差分特征 Δ 所决定的以概率 P 成立的差分方程。平均情况下，在 $1/P$ 对的明密文所生成的 $\bar{F}$ 方程中，应该有一个对有非空解，从而能够获得密钥。我们称这个明/密文是一个正确的对，对 $\bar{F}$ 计算 Groebner 基可以获得密钥变元的解。而一个错误的对，所得到的 Groebner 基为{1}。

由于差分方程的引入，求解 $\bar{F}$ 要比求解 $F'\cup F''$ 相对容易，因为差分方程增加了已有变元间的线性关系。实验中，应该选择一条最优的以概率 P 成立的差分路径，求解约 $1/P$ 的 $\bar{F}$ 方程所需要的时间总和最短。我们称对 $\bar{F}$ 的直接求解为 AttackC；

定理 3:设 F'，F'' 分别是形如 3.1 节所描述的两对明/密文生成的方程。$\bar{F}$= $F'\cup F''\cup$ 由差分特征 Δ 所决定的以概率 P 成立的差分方程。则由对 $\bar{F}$ 应用消去定理，会得到只含初始密钥 K_0 的方程。

证明:对于 $\bar{F}$ 中的多项式，其中的多项式在字典序下均有线性首项。令 L 为 $\bar{F}$ 中的最大字典序线性首项重写系统，$H=\bar{F}-L$，由于 H 中只有初始密钥 K_0 没有出现在 L 的首项，则由消去定理，用 H 中的元素去最简约化 L 会得到仅含初始密钥 K_0 的方程组。

我们称对 $\bar{F}$ 计算出仅含初始密钥 K_0 的方程组，然后进行求解的方法为 AttackD。同样，该方程组以概率 P 成立。

5.2　AttackC 与 AttackD 的求解时间对比

我们对 6~8 轮，S 盒个数分别为 4/5 的 CTC 密码系统，在分别引入 2~5 轮的差分方程下进行了相关实验.结果如表 7-7 所示。

表 7-7 CTC 在 4/5 S 盒下引入差分方程后的求解时间对比

轮数/S 盒个数	加入差分方程的轮数							
	2		3		4		5	
	AttackC	AttackD	AttackC	AttackD	AttackC	AttackD	AttackC	AttackD
6/4	646.811	2.644	3.386	2.583	0.459	2.570	0.379	2.624
7/4	≈3h	3.633	109.535	3.513	4.499	3.457	0.727	3.437
8/4	NM	4.820	215.070	4.576	14.927	4.572	2.266	4.630
6/5	>10h	79.142	3.368	76.656	0.817	79.203	0.629	76.453
7/5	>10h	111.161	1877.754	113.816	3.222	112.347	1.132	111.412
8/5	NM	151.575	NM	148.166	NM	146.964	3.143	141.158

表 7-7 可知，对于一个 Nr 轮的 CTC 分组密码，引入 r 轮差分方程后，其攻击复杂度取决于自由轮数 $R=Nr-r$。当 R 相对较小时，AttackC 优势很明显，因为其有较多的低次关系去判断一个明/密对是否是正确的对；而 AttackD 为了判断一个明/密对是否正确，先要进行中间变元消去，由 5.3 节可知，中间变元消去占了很大一部分时间。但在实际密码攻击中，当 R 太小时，也即差分轮数接近密码算法的轮数时，差分方程的成立概率大大降低，平均所需要的明/密文会大幅提高。

反之，当 R 相对较大时，AttackD 的优势就体现出来了。此时差分路径较短，差分方程的成立概率较大，平均所需要的明/密文相对较少。尤其是当 AttackC 在低差分轮数下无法在短时间内完成方程求解时，如表 7-7 中两轮差分下的 AttackC 攻击，AttackD 有很强的竞争性。

所以，当轮数较多且差分路径较短时，产生的方程必然是不利于 Groebner 基直接计算的，因为每增加一轮要增加很多变量，且较短的差分路径不会引入更多的线性方程，大大增加了求解难度。而 AttackD 不管增加多少轮，其通过变元消去都仅剩下 K_0 变元，所以轮数增加时，其效果明显。可以利用这个特性来利用低轮差分的高概率去攻击更多轮的密码系统。

5.3 差分方程的加入对 K_0 方程的影响

由表 7-8 可知:由于差分方程的引入，我们得到的仅含初始密钥 K_0 的方程组更加稀疏，更容易求解。而且差分的轮数越多，差分方程越多，从而会出现次数更

低的 K_0 方程组。从理论上说，其求解时间应该更快。然而，实验中，更多轮差分方程的引入并没有使 AttackD 的攻击时间显著减少。

表 7-8 7 轮 4S 盒的 CTC 密码系统在不同差分轮数下的多项式次数

轮数/S 盒个数/差分级别	多项式在不同 degree 个数						
	1	3	7	10	11	12	total
7/4/0					11	13	24
7/4/1	3				11	13	27
7/4/2	3	8			11	13	35
7/4/3	3	13	12		11	13	52
7/4/4	3	13	23	2	21	13	75
7/4/5	3	13	23	4	35	20	98

我们把 AttackD 的求解时间分解为获取 K_0 方程组的时间 t_1，求解 K_0 方程组的时间 t_2，由表 7-9 可知:当轮数升高时，AttackD 的攻击时间主要消耗在了获取 K_0 方程组的时间 t_1 上，这是因为中间变元增多，而 AttackD 要逐轮消去中间变元，最终获得 K_0 变元。另外，由于差分方程的引入使得 K_0 方程组存在低次方程，如 7 轮 4S 盒的 CTC 密码系统，即使只加入 1 轮差分方程，也会出现 3 个 1 次方程，使得求解时间 t_2 相对要小得多。

表 7-9 AttackD 对 CTC 在 3/5 轮差分下的求解时间

轮数/S 盒个数	AttackD 在 3/5 轮差分方程下的求解时间			
	3		5	
	t_1	t_2	t_1	t_2
6/4	2.531	0.107	2.427	0.329
7/4	3.601	0.146	3.684	0.264
8/4	4.454	0.122	4.483	0.227
6/5	70.532	3.471	72.309	5.956
7/5	101.653	4.069	104.622	6.101
8/5	138.240	4.015	140.967	5.853

§6 小结

本章给出了对 CTC 密码系统的字典序方程描述，在此基础上，通过消除中间变元，获得只含有初始密钥 K_0 变元的方程.通过实验分析，对于一对明/密文产生的方程，在获得初始密钥 K_0 变元的方程后再进行求解的方法要优于对方程的直接求解。当 CTC 密码系统的轮数、S 盒数增大，使得一对明/密文产生的方程无法直接求解时，通过增加多对同一密钥下的明/密文，以产生更多的 K_0 变元方程，使得问题能够得以解决。

同时，我们把差分方程引入 CTC 密码系统，在此基础上的获得得 K_0 变元方程

会产生多个低次多项式，使得求解效率大幅提高。尤其对于密码系统轮数较多而差分轮数较少的情况，K_0 方法的优势更加明显，适合用低轮差分的高概率去攻击更多轮的密码系统。

在本章的研究基础上，下一步的研究目标是 Present 分组密码系统，由于其设计简单，S 盒相对较小（4 位），轮数较多（31 轮）。适合于在高轮数密码系统下引入低轮差分方程的求解 K_0 方程攻击。

第八章　对 Present 分组密码的代数攻击

§1 引言

代数分析是一种针对密码算法的新型攻击方法，与传统的基于统计学思想的差分分析、线性分析等分析方法相比有许多优点：需要的明/密文对数量少，更加符合实际需求；能够针对密码算法中代数结构的弱点进行攻击，有很强的针对性。然而由于一般密码系统的方程过于庞大，所以直接使用解方程代数攻击的方法难于取得显著的成果。

但代数分析可以作为一种基本思想，与各种传统分析方法相结合寻求更好的攻击效果。本章探讨了代数攻击与错误攻击，积分攻击，及侧信道（冷冻）攻击的一些结合，对 Present 密码算法进行了分析。实践表明，使用代数攻击与错误攻击或侧信道攻击相结合，可以求解出全部的 Present 分组密码的初始密钥；而使用代数攻击与积分攻击相结合，可以减少传统积分攻击中所需要的明/密文对数量。

§2 Present 分组密码算法描述

在 CHES2007 上，Bodganov 等人提出了 Present 对称加密算法。该算法是一种超轻量级的 SPN 型分组密码，主要应用于 RFID 系统、嵌入式系统等计算资源受限的环境中。由于采用了更有利于硬件实现的小型 S 盒，Present 算法有良好的硬件实现性能。

其分组长度为 64 位、密钥长度为 80 位或 128 位。迭代 31 轮，在 31 轮迭代执行完毕后，输出再与最后一个轮密钥逐位模 2 加。轮函数由轮密钥加、代替变换和置换变换组成。其中，轮密钥加是将轮函数的输入与轮密钥逐比特模 2 加，代替变换由 16 个 4 进 4 出 S 盒并置构成，S 盒均为{12，5，6，11，9，0，10，13，3，14，15，8，4，7，1，2}。

轮密钥通过初始密钥通过以下方式得到。对于80位密钥的情形，设$K=k_{79}k_{78}k_{77}...k_0$ 为初始密钥，对于i从1~32，则选取$K_i=k_{79}k_{78}k_{77}...k_{16}$作为第$i$轮的轮密钥，然后利用下述步骤对$K$进行刷新:

1)执行$[k_{79}k_{78}k_{77}...k_0]=[k_{18}k_{17}...k_{20}k_{19}]$;

2)执行$[k_{79}k_{78}k_{77}k_{76}]=S[k_{79}k_{78}k_{77}k_{76}]$;

3)执行$[k_{19}k_{18}k_{17}k_{16}k_{15}]=[k_{19}k_{18}k_{17}k_{16}k_{15}]\oplus$轮常数;

§3 对 Present 分组密码的故障代数攻击

攻击思想[109-110] 是在第 29 轮注入 1 位错误信息，之所以选择第 29 轮是因为如果轮数太低则产生的方程太多，不易求解，且这样 31 轮的输入差分的任一 nibble（4 位 SBOX）的差分位会多于一位不为零，则很难由最终的密文倒推

出前面各轮 S 盒的输入、输出差分值。而若再更高轮注入，所涉及的密钥变元太少，不足以解出整个子轮密钥。该错误信息在经过后继 S 盒，P 盒的处理后会将错误放大并扩散到更多的数据位。最后联立后 3 轮的正确数据加密方程组及由一位错误引起的后 3 轮的错误数据方程组，并结合后 3 轮的正确位与错误位的差分关系，从而求出第 31 轮的轮子密钥。而由该轮子密钥可倒推出初始密钥。这里要解决的核心问题是如何确定后 3 轮的正确位与错误位的差分关系。

3.1 错误传输模式

图 8-1[109]给出了在第 29 轮的第 1 位注入一位错误，所引起的后继错误的放大与扩散情况。该图展示的是把错误放大到最大的情况，即 S 盒的输入差分为 $(1000)_2$，而输出差分为$(1111)_2$ 的情况。

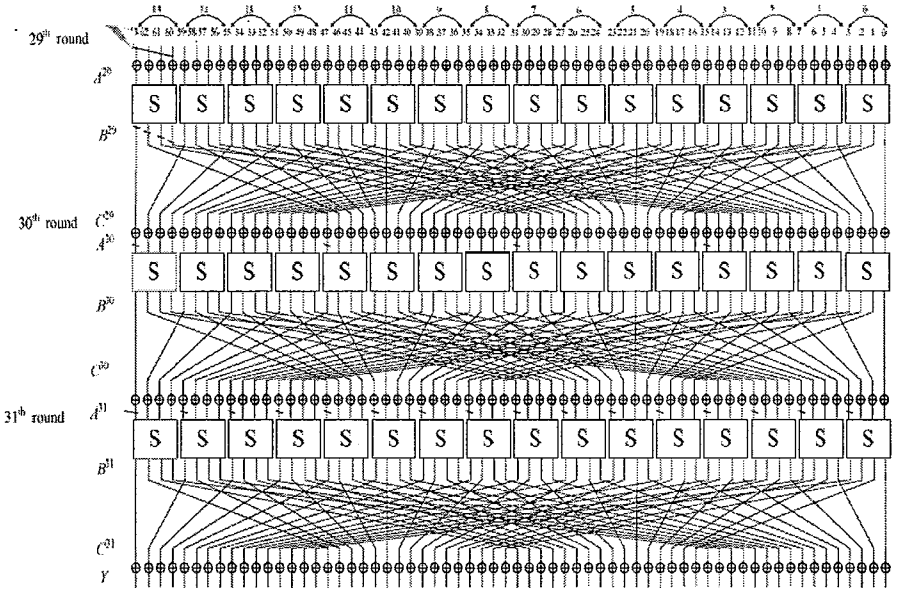

图 8-1 第 29 轮的第 1 位注入一位错误传播图

实际在第 29 轮的任一位置引入一位错误后，由 S 盒在仅一位输入差分为 1 的情况下，其输出差分表（见表 8-1）可知：29 轮的一位错误会引起 30 轮的输入数据（A^{30}）的至少两个 nibble，至多 4 个 nibble 的输入错误。进一步，由于 30 轮 S 盒、P 盒对错误的放大与扩散，会使 31 轮的输入数据（A^{31}）的至少 4 个 nibble，至多 16 个 nibble 的输入错误。

表 8-1 输入输出差分表

输入差分	输出差分
$(0001)_2$	$(0011)_2$, $(0111)_2$, $(1001)_2$, $(1101)_2$
$(0010)_2$	$(0011)_2$, $(0101)_2$, $(0110)_2$, $(1010)_2$, $(1100)_2$, $(1101)_2$, $(1110)_2$
$(0100)_2$	$(0101)_2$, $(0110)_2$, $(0111)_2$, $(1001)_2$, $(1010)_2$, $(1100)_2$, $(1110)_2$
$(1000)_2$	$(0011)_2$, $(0111)_2$, $(1001)_2$, $(1011)_2$, $(1101)_2$, $(1111)_2$

3.2　错误差分关系的分析

首先 $\Delta B^{31}=PL^{-1}(\Delta Y^{31})$，而根据 ΔB^{31} 结合表 8-1 可以推出 ΔA^{31} 的值，举例来说，若发现 ΔB^{31} 的某个 nibble 的输出差分为$(0011)_2$，则可断定该 nibble 的输入差分为$(0001)_2$或$(0010)_2$。通过产生新的正确/错误加密，再次收集 ΔB^{31} 的该 nibble 的输出差分，若为$(0110)_2$，则可断定该 nibble 的输入差分为$(0010)_2$。以此类推，整个 ΔA^{31} 便可获得。而 $\Delta B^{30}=PL^{-1}(\Delta A^{31})$，同理可得到 ΔA^{30}，进而得到 ΔB^{29}，ΔA^{29}。

3.3　错误差分方程的建立

首先生成同一明文在正确加密及 29 轮引入错误下加密的方程，称为 F_1，F_2。随后，由 3.2 节分析所

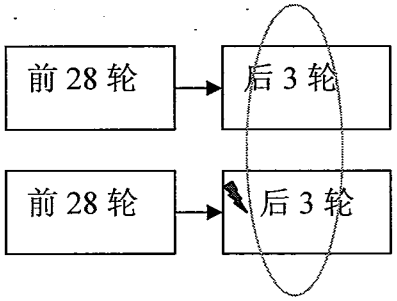

图 8-2　后 3 轮及相应的差分关系图

获得的差分关系，添加形如下差分关系方程，简记为 ΔX，ΔY。这里的 X 与 Y 变元分别代表每一轮 S 盒的输入与输出变元。所要求解的方程组是图 8-2 中用椭圆括起的后 3 轮及相应的差分关系方程组。最终要求解的方程组为 F= $F_1 \cup F_2 \cup \Delta X \cup \Delta Y$

$$\begin{array}{ll} X_{0\_29}[i]+X_{1\_29}[i]+\Delta A^{29} & Y_{0\_29}[i]+Y_{1\_29}[i]+\Delta B^{29} \\ \Delta X: X_{0\_30}[i]+X_{1\_30}[i]+\Delta A^{30} & \Delta Y: Y_{0\_30}[i]+Y_{1\_30}[i]+\Delta B^{30} \quad 0\leqslant i\leqslant 63 \\ X_{0\_31}[i]+X_{1\_31}[i]+\Delta A^{31} & Y_{0\_31}[i]+Y_{1\_31}[i]+\Delta B^{31} \end{array} \quad (4)$$

3.4　实验结果

所有实验均是在一台双核（2.6GHz 每核）4G 内存的笔记本上完成的。求解工具选择的是当前较流行的 CryptoMinisat 求解器。

1. 31 轮的密钥协商后的符号表示

对于 Present 分组密码，其初始密钥变元可用 K0000~K0079 表示。每一轮密钥协商会产生四个新的变元，如 K0100~ K0103，K0200~ K0203，直到第 31 轮的 K3100~ K3103。对任一初始密钥，首先分析出其 31 轮子密钥的符号表示。实际对任意初始密钥，第 31 轮的密钥协商结果的符号表示均是固定的。用符号表

示的密钥生成过程见附录，限于篇幅，仅给出第 1，2，30，31 轮的密钥协商结果。

2. 第 29 轮的任一位置引入一位错误

在第 29 轮的任一位置引入一位错误，通过求解方程组 F，在平均时间 12 秒左右可以获得 92 个密钥变元：K3100~ K3103，...K1200~ K1203， K1101~ K1203， ...K0801~ K0803，而这些变元涵盖了所有 31 轮的符号密钥值。所以可以完全获得第 31 轮的子密钥，由 31 轮的子密钥逆推可获得初始密钥。这里 29 轮的不同错误位置对求解结果影响不大。故在 29 轮的任一位置引起一位错误便可在短时间内攻破整个 Present 分组密码系统。

§4 对 Present 分组密码的积分代数攻击

积分代数攻击的思想是构造一个或多个结构（structure）的明文组，利用其前几轮（在平衡结构前）特定位的积分特性，列出相应的线性关系，并通过这些线性关系以加速方程组的求解。对于轮数较高时（大于 4 轮），尽管增加了额外地线性关系，也难以直接求出解。但可以解出 2 次 groebner 基，以求出在高轮的相应的输入变元的线性关系。这些线性关系在对更高轮系统攻击时，可以过滤掉更多的错误猜测密钥，以达到减少所需明、密文对的目的。

4.1 在一个 structure 下的积分位的分析[111]

输入位选择 51，55， 59， 63 作为激活（active）位，其他位作为常量位。在一个 structure 中，构造 16 个明文，其中激活位跑遍所有 4 位二进制位，常量位为任意常数。在 3.5 轮之前，所有位的积分均是平衡的，具体见图 8-3。图中的 c 表示该位为常量；a_i 表示该位为连续交错的 2^i 位相同的位。d_i 表示该位或为常量，或为 a_i。b_i 表示该位为连续但不交错的 2^i 位相同的位。特别的 b_0 表示该位不一定平衡，而 b_0^* 表示该位在积分下平衡。

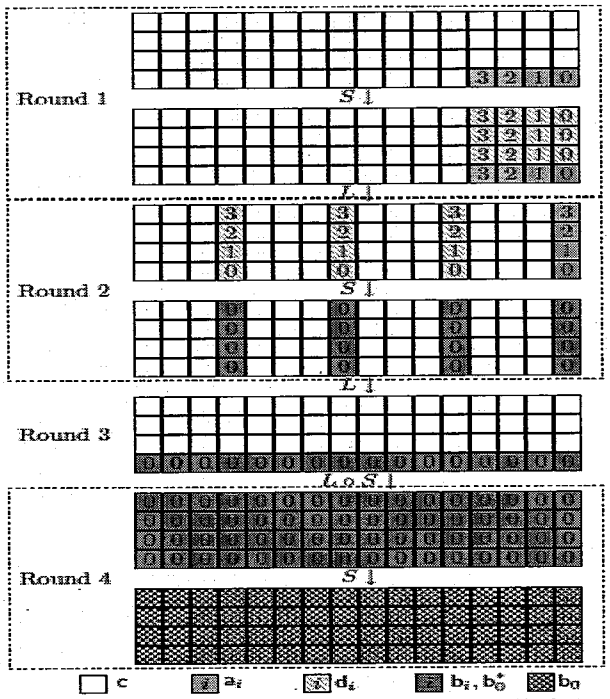

图 8-3　Present 分组密码的前 4 轮的积分关系图

4.2 积分攻击下的第一轮的线性关系

这里第一轮的在 S 盒前的 a_3，a_2，a_1，a_0 表示输入位 51，55，59，63 的积分特性。首先，对第一轮的输入变元，前面 48 个变元在一个 structure 中是常量，所以其前 12 个 SBOX 的输出也应该相同。所以可添加如下线性关系：

$X_{0,1,i}+X_{sample,1,i}$ ； $Y_{0,1,i}+Y_{sample,1,i}$　其中 $1\leqslant sample\leqslant 15$，$0\leqslant i\leqslant 47$

后面的 48，49，50，52，53，54，56，57，58，60，61，62 输入变元相同，但由于 SBOX 的作用，其输出变元定不相同。所以可添加如下线性关系：

$X_{0,1,i}+X_{sample,1,i}$　$i\in(48, 49, 50, 52, 53, 54, 56, 57, 58, 60, 61, 62)$

对于 63 位，其积分效果是 a_0，即表示该位的输入位是 0，1 交错出现的，所以可添加如下线性关系：

$X_{sample,1,63}+X_{sample+1,1,63}+1$； $Y_{sample,1,63}+Y_{sample+1,1,63}+1$　$sample\in(0, 2, 4, 6, 8, 10, 12, 14)$

而对于 62，61，60 位，其 S 盒的输出积分特性为 d_0，即表示该位是 0，1 交错出现或者是常量，所以可添加如下线性关系：

$Y_{sample,1,i}+Y_{sample+2,1,i}$　$0\leqslant sample\leqslant 13$ ， $i\in(60, 61, 62)$

对于 59 位，其积分效果是 a_1，即表示该位的输入位是相邻两位相等的，所以可添加如下线性关系：

$X_{sample,1,63}+X_{sample+1,1,63}$　$sample\in(0, 2, 4, 6, 8, 10, 12, 14)$

而对于 56，57，58，59 位，其 S 盒的输出积分特性为 a_1 或 d_1，表示这几位的相邻两位必然是相等的。所以可添加如下线性关系：

$Y_{sample,\ 1,\ i} + Y_{sample+1,\ 1,\ i}$ sample∈(0，2，4，6，8，10，12，14)，i∈（56，57，58，59)

对于 **55** 位，其积分效果是 a_2，即表示该位的输入位是四个相同位交错出现的，所以可添加如下线性关系：

$X_{sample,\ 1,\ 55} + X_{sample+1,\ 1,\ 55}$；$X_{sample,\ 1,\ 55} + X_{sample+2,\ 1,\ 55}$；$X_{sample,\ 1,\ 55} + X_{sample+3,\ 1,\ 55}$ sample∈(0，4，8，12)

而对于 52，53，54，55 位，其 S 盒的输出积分特性为 a_2 或 d_2，表示这几位的相邻四位必然是相等的。所以可添加如下线性关系：

$Y_{sample,\ 1,\ i} + Y_{sample+1,\ 1,\ i}$；$Y_{sample,\ 1,\ i} + Y_{sample+2,\ 1,\ i}$；$Y_{sample,\ 1,\ i} + Y_{sample+3,\ 1,\ i}$ sample∈(0，4，8，12)，i∈（52，53，54，55)

对于 **51** 位，其积分效果是 a_3，即表示该位的输入位是八个相同位交错出现的，所以可添加如下线性关系：

$X_{sample,\ 1,\ 51} + X_{sample+1,\ 1,\ 51}$；$X_{sample,\ 1,\ 51} + X_{sample+2,\ 1,\ 51}$；$X_{sample,\ 1,\ 51} + X_{sample+3,\ 1,\ 51}$;

$X_{sample,\ 1,\ 51} + X_{sample+4,\ 1,\ 51}$；$X_{sample,\ 1,\ 51} + X_{sample+5,\ 1,\ 51}$；$X_{sample,\ 1,\ 51} + X_{sample+6,\ 1,\ 51}$; $X_{sample,\ 1,\ 51} + X_{sample+7,\ 1,\ 51}$ sample∈(0，8)

而对于 48，49，50，51 位，其 S 盒的输出积分特性为 a_3 或 d_3，表示这几位的相邻八位必然是相等的。所以可添加如下线性关系：

$Y_{sample,\ 1,\ i} + Y_{sample+1,\ 1,\ i}$；$Y_{sample,\ 1,\ i} + Y_{sample+2,\ 1,\ i}$；$Y_{sample,\ 1,\ i} + Y_{sample+3,\ 1,\ i}$ $Y_{sample,\ 1,\ 51} + Y_{sample+4,\ 1,\ 51}$；$Y_{sample,\ 1,\ 51} + Y_{sample+5,\ 1,\ 51}$；$Y_{sample,\ 1,\ 51} + Y_{sample+6,\ 1,\ 51}$; $Y_{sample,\ 1,\ 51} + Y_{sample+7,\ 1,\ 51}$ sample∈(0，8)，i∈（48，49，50，51)

4.3 积分攻击下的第二轮的线性关系

第二轮的输入变元在 12，13，14，15， 28，29，30，31， 44，45，46，47， 60，61，62，63 位的积分特性是从第一轮 S 盒的输出变元经过 P 盒变换而来。而其输出变元在经过第二轮 S 盒后，所有上面的相关位置的积分特性都变成了 d_0^*。其它位置的积分特性是保持常量 c。所以可添加如下线性关系：

$X_{0,\ 2,\ i} + X_{sample,\ 2,\ i}$ 1≤sample≤15， i not in(12，13，14，15，28，29，30，31，44，45，46，47， 60，61，62，63)

$Y_{0,\ 2,\ i} + Y_{sample,\ 2,\ i}$ 1≤sample≤15， i not in(12，13，14，15，28，29，30，31，44，45，46，47， 60，61，62，63)

$Sum(X_{sample,\ 2,\ i})$ 1≤sample≤15， i in (12，13，14，15，28，29，30，31，44，45，46，47， 60，61，62，63)

$Sum(Y_{sample,\ 2,\ i})$ 1≤sample≤15， i in (12，13，14，15，28，29，30，31，44，45，46，47， 60，61，62，63)

实际对于第二轮的(12，13，14，15，28，29，30，31，44，45，46，47， 60，61，62，63)位的输入变元有更好的方程描述，即按照第一轮的 S 盒的输出变元即可，这里用了相对简单的描述方式。

4.4 积分攻击下的第三、四轮的线性关系

第三轮的输入变元在 3，7，11，...63 位的积分特性是从第二轮 S 盒的输出变元经过 P 盒变换而来。而其输出变元在经过第二轮 S 盒后，所有上面的相关位置的积分特性都变成了 d_0^*这是由于 P 盒并不改变积分属性，所以由第四轮输入变元的积分特性可反推出第三轮输出变元的积分属性。而其他位置的输入变元保持常量。所以可添加如下线性关系：

$X_{0,3,i}+X_{sample,3,i}$ $1\leqslant sample\leqslant 15$ ， $i\%4 != 3$

$Sum(X_{sample,3,i})$ $1\leqslant sample\leqslant 15$， $i\%4 = 3$

$Sum(Y_{sample,3,i})$ $1\leqslant sample\leqslant 15$， $0\leqslant i\leqslant 63$

在第四轮时，其输入变元的所有位置的积分特性都变成了 d_0^*。而其输出变元不再维持平衡。所以可添加如下线性关系：

$Sum(X_{sample,4,i})$ $1\leqslant sample\leqslant 15$， $0\leqslant i\leqslant 63$

4.5 积分攻击下线性关系的利用

所有实验均是在一台双核（2.6GHz 每核）4G 内存的笔记本上完成的。求解工具选择的是当前较流行的 CryptoMinisat 求解器以及 PolyBori 求解器。在列出所有 16 个选择明密文组成的方程后，尝试用 CryptoMinisat 求解器以及 PolyBori 求解器去直接求解，在有限时间（>3hour）内无法直接解出密钥值，哪怕仅生成 1 轮的方程系统。原因是 16 个选择明密文组成的方程过于庞大，其方程数量与变元数量相当于一个 16 轮的明密文生成的方程。

然而可利用列出的含有积分关系的方程组求出在次数为 2 下的低轮（4-6）groebner base。对于第四轮，可以找到 500 多个输入变元的线性关系。这些线性关系可用于通过猜测最后一轮的子密钥对密文解密后，看是否符合该线性关系，过滤掉错误的猜测子密钥。实际起到减少所需选择明/密文对的目的。对于第五轮，可找到 26 个输入变元的线性关系。同样，这些线性关系可减少所需的选择明/密文对。对于第六轮，无法再找到输入变元的线性关系，但可通过猜测第一轮的部分子密钥以达到对第六轮攻击的目的。

§5 对 Present 分组密码的冷冻攻击

最近，一个从内存中提取密钥材料的方法被提出了，这种技术被称为冷冻攻击。它的攻击原理主要是利用内存中的数据位在低温下不会立刻衰退，而是有一个较长得衰退期。一般而言，对于一个分组密码算法，在对数据进行加密前，出于对效率的考虑，其会预先根据初始密钥生成各轮子密钥存于内存中。所以在内存被冷冻的空隙间，如果我们能够获得物理内存，从中分析出一些加密算法密钥位所在的内存位置，并从中提取出密钥位，便达到了攻击密码算法的目的。但这里有两个问题要解决：一是密钥位的定位；二是如何从部分衰退的密钥位中恢复出正确的密钥位。

5.1 内存中密钥位的确定

假设已获得在冷冻下的正在衰减的内存，Present 密码算法的初始密钥及其子密钥存在于一块连续的内存中。这里其初始密钥或子密钥中的某些为可能由于位衰减而发生了错误。密钥位置的定位方法是这样的：循环访问内存里的每个字节，把它作为初始密钥。然后计算出相应的子密钥与实际在内存中获得的有可能错误的密钥去比较，记录相应的汉明距离。若汉明距离大于预先设定的数值，则很有可能初始密钥是错误的。最后记录所有可能正确的初始密钥和相应的子密钥作为候选。

5.2 从部分错位的密钥位中恢复出正确的密钥位

从内存中得到的密钥会由于内存位的衰退而发生部分位的错误，而密钥生成（key schedule）的过程一般会含有很多冗余信息，所以可以通过把整个密钥生成的过程写成概率方程组的形式，并转化为整数规划的问题，利用 mip 求解器去求解，以找到真正的密钥。

1)概率方程的生成

在概率方程组中分为正确方程和以一定概率正确方程的两个集合，定义为 H 和 S。其中 H 集合中存放密钥协商方程。由于密钥协商过程是固定的，所以一定正确。而集合 S 中的方程代表从实际内存中获得的密钥位，由于内存电压的衰减，这些密钥位以一定的概率正确。在软件模拟中，通过对随机产生的初始密钥值以及后继生成的每轮子密钥在错误概率 Δ_0 以及 Δ_1 的控制下，进行相应位的翻转。其中 μ_0 表示密钥位发生从“1”衰减到“0”的概率；Δ_1 表示密钥位发生从“0”衰减到“1”的概率。实际中 Δ_0 的概率一般远大于 Δ_1 的概率。下面以 python 代码的形式描述了上述过程：

```
p = PRESENT(Nr=n)
K = p.random_element(80)
hard = []
soft = []
Kvar = list(p.K0)

for i in range(n+1):
    kv = p.vars("K",i)
    Ki,K = p.keySchedule(K, i+1)
    Kii, Kvar, key = p.key_schedule_polynomials(Kvar,i)
    key += p.field_polynomials(p.vars("K",i))
    hard.extend(key)
    soft.extend(map(add,zip(Ki,kv)))

for i,f in enumerate(soft):
    if random() <= delta0 and soft[i].constant_coefficient() == 1:
        soft[i] += 1
    if random() <= delta1 and soft[i].constant_coefficient() == 0:
        soft[i] += 1

F = ProbabilisticMPolynomialSystem(p.ring(),hard,soft)
```

2)使用 mip 求解器求解概率方程组 F

Mip 的全称为 mixed integer programming，即混合整数规划。其过程为在等式或不等式条件的约束下，求目标函数的最大值或最小值。其中对于约束条件中的某些变元或全部变元有只能取整数的限制。使用 mip 求解器作为密码分析工具是在 2007 年对 Bivium 流密码的分析中首次应用。这里如何把概率方程组转化为混合整数规划问题已在 Martin 的论文中有所论述，下面仅分析使用 mip 求解器对 Present 分组密码的攻击结果。

表 8-2 mip 求解器对 Present 分组密码的攻击结果

轮数 n	Δ_0	Δ_1	密钥变元数	平均时间(s)	成功率
3	0.15	0.001	77	58	65%
4	0.15	0.001	85	94	70%
4	0.30	0.001	85	97	63%
5	0.15	0.001	89	103	73%
5	0.30	0.001	89	106	68%
5	0.30	0.15	89	127	36%
6	0.15	0.001	93	155	75%
6	0.30	0.001	93	166	71%
6	0.30	0.15	93	532	40%

由 8-2 表可知，试验的效果随着 Δ_0，Δ_1 的概率的增大而变坏，时间变长，成功率变低。使用的密钥协商方程轮数越高，则由于增加的密钥变元之间的冗余关系越多，成功率越高，但相应的用时越多。实际攻击中不用生成全部 31 轮的方程，但至少需要 4 轮，才能够解出大于 80 的密钥变元值，从而可反推出初始密钥。如轮数 n=4 时，所解出的 85 个密钥变元是：[K0403， K0402， K0401， K0400，K0303， K0302， K0301， K0300， K0203， K0202， K0201， K0200， K0103，

K0102, K0101, K0100, K0068, K0067, ..., K0001, K0000]

§6 小结

本章探讨了代数攻击与错误攻击，积分攻击，及侧信道（冷冻）攻击的一些结合，对 Present 密码算法进行了分析。实践表明，使用代数攻击与错误攻击或侧信道攻击相结合，可以求解出全部的 Present 分组密码的初始密钥；而使用代数攻击与积分攻击相结合，可以减少传统积分攻击中所需要的明/密文对数量。研究代数攻击与其他攻击更广泛的结合与应用是我们下一步工作的重点。

附录：

[K0000, K0001, K0002, K0003, K0004, K0005, K0006, K0007, K0008, K0009, K0010, K0011, K0012, K0013, K0014, K0015, K0016, K0017, K0018, K0019, K0020, K0021, K0022, K0023, K0024, K0025, K0026, K0027, K0028, K0029, K0030, K0031, K0032, K0033, K0034, K0035, K0036, K0037, K0038, K0039, K0040, K0041, K0042, K0043, K0044, K0045, K0046, K0047, K0048, K0049, K0050, K0051, K0052, K0053, K0054, K0055, K0056, K0057, K0058, K0059, K0060, K0061, K0062, K0063]

[K0100, K0101, K0102, K0103, K0065, K0066, K0067, K0068, K0069, K0070, K0071, K0072, K0073, K0074, K0075, K0076, K0077, K0078, K0079, K0000, K0001, K0002, K0003, K0004, K0005, K0006, K0007, K0008, K0009, K0010, K0011, K0012, K0013, K0014, K0015, K0016, K0017, K0018, K0019, K0020, K0021, K0022, K0023, K0024, K0025, K0026, K0027, K0028, K0029, K0030, K0031, K0032, K0033, K0034, K0035, K0036, K0037, K0038, K0039, K0040, K0041, K0042, K0043, K0044]

.....

[K2900, K2901, K2902, K2903, K1201, K1202, K1203, K1600, K1601, K1602, K1603 + 1, K2000, K2001, K2002, K2003 + 1, K2400, K2401, K2402, K2403 + 1, K2800, K2801, K2802, K2803, K1101, K1102, K1103, K1500, K1501, K1502, K1503 + 1, K1900, K1901, K1902, K1903 + 1, K2300, K2301, K2302, K2303 + 1, K2700, K2701, K2702, K2703, K1001, K1002, K1003, K1400, K1401, K1402, K1403 + 1, K1800, K1801, K1802, K1803 + 1, K2200, K2201, K2202, K2203 + 1, K2600, K2601, K2602, K2603 + 1, K0901 + 1, K0902 + 1, K0903]

[K3000, K3001, K3002, K3003, K1301, K1302, K1303 + 1, K1700, K1701, K1702, K1703 + 1, K2100, K2101, K2102, K2103 + 1, K2500, K2501, K2502, K2503 + 1, K2900, K2901, K2902, K2903, K1201, K1202, K1203, K1600, K1601, K1602, K1603 + 1, K2000, K2001, K2002, K2003 + 1, K2400, K2401, K2402, K2403 + 1, K2800, K2801, K2802, K2803, K1101, K1102, K1103, K1500, K1501, K1502, K1503 + 1, K1900, K1901, K1902, K1903 + 1, K2300, K2301, K2302, K2303 + 1, K2700, K2701, K2702, K2703 + 1, K1001 + 1, K1002 + 1, K1003 + 1]

第九章　基于遗传算法的对 Trivium 流密码的区分攻击

§1 引言

区分器及非随机性检测器对于对称密码系统来说是两个重要的检测指标。设计良好的密码系统所生成的密文流（密钥流）应具有良好的随机性，即对外部观察者而言，应无法判断该密文流（密钥流）是否由一个具体的密码系统生成。

区分器和非随机性检测器均是把密码系统作为一个黑盒模型来处理。通过选择不同的密钥位（Key）及 IV 位对密码系统的输出作分析。以流密码为例，对区分器而言，密钥是未知和固定的，而初始向量可以选择不同的值进行测试。这与现实中的应用相吻合。而非随机性检测器既可用到 Key 位，又可用到 IV 位。这与现实中应用不太吻合，但却可以更大限度的检测出流密码算法的非随机性。因为区分器只能检测出 IV 位是否混合充分，而随机性检测器同时考虑到 Key 及 IV 位对密码系统的影响。

近年来,出现了若干区分器及非随机性检测器的成果[113-115]，其中最好的结果使用了最大次数单项式测试（maximum degree monomial test）的思想,简称 MDMT[113].使用 MDMT 方法可以高效的检测出特定密码算法的非随机性，从而给出了流密码或分组密码的强度分析：如分组密码的加密轮数是否足够安全；流密码的初始向量（IV）是否混合的足够充分等。这对于对称密码的设计者而言是很重要的考虑因素。

MDMT 方法的核心是最优测试位集合的选择，Paul Stankovski 在 2010 年提出了贪心位集合选择算法[113]，对于组合优化问题而言，该算法是一种简单的确定性的算法。其实验结果为当前对 Trivium 流密码进行随机性分析的最好结果：即以 2^{45} 复杂度找到 1026 轮的非随机性检测器；以 2^{44} 复杂度找到 806 轮的区分器。另外 Itai Dinur 等人[114]使用 cube tester 方法分别以 2^{30},2^{27} 复杂度找到了 Trivium 流密码的 790 轮区分器及 885 轮非随机检测。然而贪心算法的天性上的弱点是从单个解开始，每一步的选择只根据当前情况，而未从全局考虑，容易陷入局部最优。

遗传算法对于最优测试位集合选择这类 NP 问题非常有效。它与贪心算法的很大区别在于：遗传算法从串集开始搜索，通过复制、交叉、突变等操作产生下一代的解，并逐步淘汰掉适应度函数值低的解，增加适应度函数值高的解。这样进化 N 代后就很有可能会进化出适应度函数值很高的个体。该算法覆盖面大，利于全局择优，易于实现并行化，从某种程度上克服了贪心优化算法容易误入局部最优解的弊端。

本章的结构是这样安排的:在第 2 节，我们对 Trivium 流密码来的最大次数单项式测试进行了简单介绍，在第 3 节给出了位集合选择的两种算法，即贪心算法和遗传算法。.第 4 节对以 MDMT 技术为基础的 Trivium 流密码的区分器进行了

相关实验。实验结果表明：遗产算法能够在更小的 IV 权重下，区分更多的初始轮，提高了 Trivium 流密码的区分器的攻击能力。

§2 研究背景

2.1 布尔函数的的代数正规式（ANF）描述

设 f 是从 F_2^n 映射到 F_2 上的函数，即 f 为一布尔函数。布尔函数有多种表示方法，比较常用的是代数正规式描述法。即以下多项式描述：

$$f(x_1,x_2,\ldots x_n)=a_0+a_1x_1+\ldots+a_nx_n+a_{n+1}x_1x_2+\ldots+a_{2^n-1}x_1x_2\ldots x_n \qquad (9\text{-}1)$$

其中 a_i 式为定义在 F_2 上的的多项式的系数。对于 Trivium 流密码的初始化来说，初始密钥与 IV 均为 80 位，初始轮为 1152 轮，即有 1152 位初始化密钥流。其中每一个输出位均可看做输入为特定 Key 位与 IV 位的布尔函数。设 $K=(k_0,\ldots,k_{79}),IV=(iv_0,\ldots,iv_{79}),Z=z_0,z_1,\ldots$ 为别位密钥变元，初始向量变元，输出位变元，则对每个 z_i 均可写成 $f_i(K,IV)$ 的形式，其中 $1\leqslant i\leqslant 1152$。

2.2 最大次数单项式测试（MDMT）

对 Trivium 流密码来说，最大次数单项式测试即对 Trivium 初始化的每个输出位 z_i 的布尔函数做的一种是否包含最大次数单项式的统计性测试。理论上在初始化轮数较低时，IV 变元位或 Key 变元位很难混合充分，相应的输出位 z_i 对应的多项式很难有最大次数单项式的出现。随着初始轮数的增加，IV 或 Key 变元混合的逐渐充分，对输出位 z_i 来说，其最大次数单项式会以 1/2 的概率出现。然而究竟初始化至少要多少轮，才能使得最大次数单项式会以 1/2 的概率出现？即至少要多少轮，才能使得 IV 或 Key 变元混合的较为充分，这便是区分器及伪随机检测器所要做的工作。

对于特定的布尔函数（9-1），可以通过穷举其部分输入变元的所有值，代入 f 后所得到的所有值相加即为最大次数单项式的系数 $a_{2^n-1}\in\{0,1\}$，从而判断出该最大项是否存在。举例来说，对于 $f(x_1,x_2,x_3,x_4)=x_1+x_1x_2x_3+x_1x_2x_4+x_3$，选择 x_3,x_4 作为自由变元，得到如下结果：

$f(x_1,x_2,0,0)+f(x_1,x_2,0,1)+f(x_1,x_2,1,0)+f(x_1,x_2,1,1)=0$,则表明函数 f 无最大项 $x_1x_2\,x_3x_4$。

对 Trivium 流密码的区分器而言，MDMT 主要是分析每个输出位 z_i 对应的 $f_i(IV)$ 形式的多项式中是否含有最大次数项 $iv_0iv_1\ldots iv_{79}$，即次数为所有 IV 位变元相乘的项，这里可以通过穷举 IV 的某个子集来测试。而对伪随机性检查器而言，MDMT 主要是分析每个输出位 z_i 对应的 $f_i(IV,K)$ 形式的多项式中是否含有最大次数项 $iv_0iv_1\ldots iv_{79}\ k_0\ k_1\ldots k_{79}$，即次数为所有 IV 位变元及 Key 变元相乘的项。同样可通过穷举 IV,K 的某个子集来测试。

Trivium 流密码的初始化为 1152 位，以区分器为例，当 IV={4,7, 10, 13, 19, 28,

34, 46, 49, 58, 64, 67, 73, 76, 79 }时，其余 IV 位及 Key 变元取 0 值，MDMT 的输出位为：***00000...0000***101...可以看到，有连续的 835 个 0（以斜体表示），随后是以一定概率交替出现的 0,1 序列。前 835 个零即表示前 835 个输出位对应的布尔函数无最高次项。即对 Trivium 流密码来说，初始化 835 轮依然不能使 IV 位充分混合。Trivium 流密码的 MDMT 测试即统计 1152 轮的初始化输出位函数中有多少个无连续最高次项，即其最高次项系数有多少个连续的 0。

从复杂度角度来说，穷举 n 位 IV 位或 IV 与 Key 变元，复杂度为 $O(2^n)$。由于仅考虑初始化的 1152 轮，其空间复杂度即 $O(l)$, l=1152。算法 1 给出了对于特定的待穷举的 IV 变元集合及密钥变元集合 Key 下的 MDMT 方法。显然，IV 或 Key 变元集合的大小及不同的选择方式对 MDMT 的测试结果有重要影响。如何在有限的复杂度下，即特定数目的 IV 或 Key 变元集合下，找到最好的变元组合是第三节要解决的主要问题。

算法 1.MDMT(*IV,Key*)

输入:待穷举的 IV 变元集合及密钥变元集合，其余的 IV 与 Key 变元设为某固定值

输出:*z*, *XorBuf* 即 1152 轮的初始化输出位函数中最高次项系数位数组及连续为 0 的个数

1. 生成 *n* 个 *IV* 及 *Key* 变元的所有猜测集合置为 *GSet*
2. 初始化 1152 位的全 0 缓冲区 *XorBuf*
3. **for** *vc* in *GSet* **do**
4 　　把该 *vc* 值代入密码系统，把所有 1152 轮的输出结果与 *XorBuf* 相异或
5. **end for**
6. *z*=*XorBuf* 位数组中连续为 0 的个数
7. **return** *z, XorBuf*

§3　最优位集合的选择策略

3.1　贪心算法

在最优位集合的选择中，贪心算法是简单有效的方法。这里设 n 为最终要选择的变元个数；集合 S 为最终选取的 n 个变元集合，集合 B 为可选的位集合。对 Trivium 流密码来说，集合 B 或为 80 位的 IV 变元(区分器)；或为 160 位的 IV 及 Key 变元空间。其工作流程见算法 2：

算法 2. Greedy(*B*, *n*)

输入:可选的变元集合 *B*,变元个数 *n*.

输出: 最终选取的 *n* 个变元集合 *S*.

1. 另 *S* 为空集
2. **repeat n times {**
3 　　*bestBit*=none; *max*=-1
4. 　　**for** *all b*∈ *B**S*
5. 　　*z*= MDMT（*S*∪{*b*}）
6. 　　**if** z>max

```
7.                max=z
8.                bestBit=b
9.            end if
10.       end for
11.       S=S∪{bestBit}
12.   }
13.   return 0
```

在算法 2 中，可选的变元集合 B 的长度为 m，则对贪心算法而言，第一位变元选择有 m 种，第二次选择有 m-1 种，两个变元又有 4 种取值方法。注意这里有一半是不用重复计算的。选择 n 个变元的复杂度为：

$$1+\sum_{0\le i\le n}(m-i)2^{i}<m2^{n} \qquad (9\text{-}2)$$

在测试中，一般随着 n 值的增大，MDMT 方法找到的连续为 0 的个数也越多。但复杂度也随之升高。对普通 PC 而言，当 n=30 时，对 Trivium 流密码的区分器（m=80）而言，所需的复杂度$\approx 2^{36.5}$，所需计算时间大于 4 小时。这里的复杂度实际为加密生成 1152 位密钥流的次数。

3.2 遗传算法

遗传算法（Genetic Algorithm）是一类借鉴生物界的进化规律（适者生存，优胜劣汰遗传机制）进化而来的随机化搜索方法。对于 Trivium 流密码的区分器或伪随机检测器来说，随着集合 S 中的变元数目的增大，搜索空间也急剧增大，以目前的计算能力用枚举法很难求出最优解。遗传算法则可以在有限的时间内给出“满意解”。

染色体：这里每条染色体实际为一种 IV 或/及 Key 变元的选择。对 Trivium 流密码的区分器而言，每条染色体 80 位；对伪随机检测器来说，每条染色体 160 位。相应位的取值 0/1 表明对应的变元选或不选。这里把染色体中 1 的位数称为染色体的权重。

种群：初始的染色体集合。初始种群的大小对求解结果有重要影响。初始种群不能太少，否则容易“早熟”，即局部最优。在实验中，初始种群中所有染色体都是一样的。即通过贪心算法选择的 15 个 IV 或/及 Key 变元

选择下一代染色体：下一代染色体的选择采用了“精英主义”的策略，即首先保留 20%的上一代的最优染色体。其余 80%的染色体的生成则来自于上一代染色体的交叉和变异。

交叉：随机选择两个父染色体，通过互换其部分“基因”来达到交叉目的。交叉是有一定概率的，称之为交叉率。高概率的随机交叉严重破坏了复制过程中

产生的高适配值。所以交叉率不应过大。交叉过程如下所示：

交叉前：00000|*011100000000*|10000　　11100|000001111110|00101

交叉后：00000|000001111110|10000　　11100|*011100000000*|00101

变异：在繁殖过程，新产生的染色体中的基因会以一定的概率出错，称为变异。变异发生的概率成为变异率。变异可以保证种群的多样性，从而防止算法收敛于某个局部解。如下所示：

变异前：00000111000000010000

变异后：00000111000010010000

适应度函数：用于评价某个染色体的适应度，显然这里用 MDMT 的测试结果作为适应度函数的输出值。即 1152 轮的初始化输出位函数中最高次项系数位数组的连续为 0 的个数。

终止过程：理论上当连续 10 代以上没有获得更优解或当染色体之间的相似度达到阈值时（如 90%），算法即停止运行。因为在进化的后期，大量个体集中在某一极值点上，后代造成了近亲繁殖，容易陷入局部最优，可采取小生境技术改进。

然而对于 Trivium 流密码的区分器或伪随机检测器来说，由于一般随着 IV 或/及 Key 变元的集合 S 中的变元数目 n 值的增大，MDMT 方法找到的连续为 0 的个数也越多，所以遗传算法会更倾向保存权重更大的染色体，但计算复杂度也随之升高。对普通 PC 而言，当染色体的权重接近 30 时，有限时间内很难有新一代的进化。

算法 3 给出了遗传算法的工作过程，其中染色体用结构体 *ch* 来存储，其中的 *items* 项为 80/160 位长的位数组，*f* 记录相应的该染色体的适应度值。

算法 3 Genetic(*gens*, *pop*,*crp*)
输入:进化的代数 *gens*，种群中的染色体个数 *pop*，交叉率 *crp*
输出:.统计出每一代的最优及平均适应度值
1: **for** *i*:=1 **to** *pop* **do**
2:　*ch*[*i*].*items*=通过贪心算法选择的15个IV或/及Key变元集合*S*
3:　*ch*[*i*].*f*=MDMT(*S*)
4: **end for**
5: **for** *j*:=1 **to** *gens* **do**
6:　对染色体数组按适应度值排序
7:　**for** *k*:=1 **to** *pop* **do**
8:　　**if**　*k*>*pop**0.2
9:　　**if** random( )>*crp*
10:　　*ch*[*k*] .*items* =随机选父代中的两条染色体进行交叉

```
11:              ch[k] .f=新染色体的适应度值
12:          else
13:              ch[k] .items=对ch[k]进行变异
14:              ch[k] .f=变异后染色体的适应度值
15:          end if
16:      end if
17:       avg+=ch[i].f
18:       best=每一代中最好的适应度值
19:  end for
20:  输出当前代的最优及平均适应度值
21: end for
```

在实验中,进化的代数 *gens* 可设置为一任意大数，因为算法在每一代都会记录当前的最优解。*pop**0.2 表明每次把上一代的最优的 20%染色体直接复制到下一代，其余 80%的染色体的生成则来自于上一代染色体的交叉和变异，其比率按交叉率 *crp* 和变异率(1-*crp*)来进行。

若设 *gens* 代中每一代最好的适应度值的染色体权重分别为 $i_1,i_2,...,i_{gens}$,则 *gens* 进化所需要的计算量约为：

$$pop \times (2^{i_1} + 2^{i_2} + ... + 2^{i_{gens}}) \qquad (9\text{-}3)$$

这里取每一代最好的适应度值的染色体权重是一种近似取法，因为每一代 *pop* 个染色体的权重一般不会完全相同，最好的适应度值的染色体权重一般较大，对于计算复杂度而言是较合理的近似。

§4 实验分析

所有实验均是在一台双核（2.6GHz 每核）8G 内存的机器上完成的。

4.1 贪心算法的实验结果

首先以 Trivium 流密码的区分器作为实验对象，在最优位集合的选择中从最初的 1 位直到贪心选择到 29 位（由于计算能力的限制，目前只分析到 29 位）。表 1 给出了每一步选择的 IV 变元及所获得的 1152 轮的初始化输出位函数中最高次项系数位数组的连续为 0 的个数，用 MDMT(IV)来表示。实验中除了所选择的 IV 变元作为变元外，其余 IV 及 Key 变元赋值为 0。

表 9-1 使用贪心算法对 Trivium 流密码的区分器的 IV 变元选择

IV 权重	MDMT(IV)	具体的 IV 变元
1	68	{ 79 }
2	472	{ 79, 4 }
…	…	…
18	865	{ 79, 4, 67, 64, 73, 58, 13, 49, 19, 28, 46, 34, 10, 76, 7, 22, 43, 52 }
19	862	{ 79, 4, 67, 64, 73, 58, 13, 49, 19, 28, 46, 34, 10, 76, 7, 22, 43, 52, 0 }
20	861	{ 79, 4, 67, 64, 73, 58, 13, 49, 19, 28, 46, 34, 10, 76, 7, 22, 43, 52, 0, 25 }

…	…	…
23	896	{ 79, 4, 67, 64, 73, 58, 13, 49, 19, 28, 46, 34, 10, 76, 7, 22, 43, 52, 0, 25, 40, 55, 37 }
24	913	{ 79, 4, 67, 64, 73, 58, 13, 49, 19, 28, 46, 34, 10, 76, 7, 22, 43, 52, 0, 25, 40, 55, 37, 61 }
25	921	{ 79, 4, 67, 64, 73, 58, 13, 49, 19, 28, 46, 34, 10, 76, 7, 22, 43, 52, 0, 25, 40, 55, 37, 61, 31 }
…	…	…
28	885	{ 79, 4, 67, 64, 73, 58, 13, 49, 19, 28, 46, 34, 10, 76, 7, 22, 43, 52, 0, 25, 40, 55, 37, 61, 31, 70, 16, 74 }
29	872	{ 79, 4, 67, 64, 73, 58, 13, 49, 19, 28, 46, 34, 10, 76, 7, 22, 43, 52, 0, 25, 40, 55, 37, 61, 31, 70, 16, 74, 72 }

由表 9-1 可以看出，IV 变元从 1 个贪心选择到 18 个时，所获得的 MDMT 值都是逐渐增长的。然而，选择第 19 个 IV 变元时，MDMT 值反而减少，直到选择第 23 个 IV 变元, MDMT 值又重新增长，最大的 MDMT 值出现在选择 25 个 IV 变元时，所获得的 MDMT 值为 921。这表明在密钥为全 0 状态下，Trivium 流密码的区分器在使用 25 个 IV 变元时，能够检测出初始化 921 轮依然是不够充分的。

4. 遗传算法的实验结果

同样以 Trivium 流密码的区分器作为实验对象，种群的大小，交叉率，变异率这三个参数对实验结果有一定的影响。初始种群中所有染色体都是一样的，即通过贪心算法选择的 15 个 IV 变元，以初始种群数为 50,交叉率为 65%,相应的变异率为 35%为例，实验结果见表 9-2：

表 9-2　使用遗传算法对 Trivium 流密码的区分器的 IV 变元选择

进化代数	最优 MDMT(IV)值	平均 MDMT(IV)值	IV 权重
0-4	835	811.78	15
5-9	836	812.74	16
…	…	…	…
16-25	862	838.62	18
26-29	865	843.38	20
…	…	…	…
36-41	904	879.62	22
42-43	913	880.10	22
…	…	…	…
44-68	922	894.46	23
…	…	…	…

由表 9-2 数据,我们可以推出以下结论：

1）随着进化代数的增加，MDMT(IV)值是逐渐增大的，这正是遗传算法每代选择优秀染色体的结果。相比于贪心算法，遗传算法在 IV 权重为 23 时，获得了最优为 922 的 MDMT(IV) 值，其对应的 IV 变元集合为：{4, 7, 10, 13, 19, 22, 25, 28, 31, 34, 37, 40, 46, 49, 52, 55, 58, 61, 64, 67, 73, 76, 79}，这比贪心算法寻找到 IV 权重为 29 时，获得的最优为 921 的 MDMT(IV)值有一定的改进。

2）在进化到 44 轮以后，出现了连续 24 轮的相同最优解，即没有获得更优解。其主要原因是：随着进化的深入，染色体之间的相似度增大，大量个体集中在某一极值点上，后代造成了近亲繁殖，这是由 Trivium 流密码的区分器的自身性质决定的，因为其 MDMT(IV)的值并非随着 IV 权重的增大而一直增大，中间可能会反而减小。这时，遗产算法会始终记录当前权重相对小而 MDMT(IV)的值却相对大的那些染色体。此时即陷入了局部最优，可采取灾变技术改进。

另外对于种群的大小，交叉率，变异率这三个参数对实验结果的的影响见图 1,2

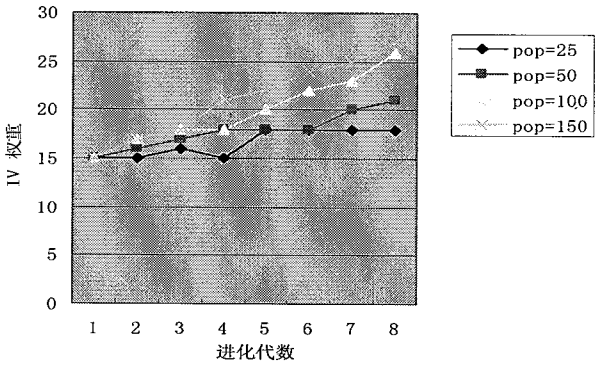

图 9-1 不同种群数的进化过程中的 IV 权重增长图

图 9-1 是种群为别为 25,50,100,150，交叉率为 65%时的进化过程 IV 权重增长图。由图 9-1 可看出种群越大，进化过程中 IV 权重增长的越快，相应的所获得的 MDMT(IV)的值相对越大。这是因为种群越大，相应的进化出的“优秀”染色体会更多，而一般而言（除了在特定的进化阶段，IV 变量增多，MDMT(IV)的值反而减少），IV 权重大的染色体更容易获得高的 MDMT(IV)值。但对于 Trivium 流密码的区分器来说，进化过快反而容易丢掉那些权重相对不大，但 MDMT(IV)值相对较大的染色体。因为这些真正“优秀”的染色体被权重更大的“伪优秀”的染色体取代了。所以在实验中种群不宜太大，因为这样不仅会影响计算速度，而且容易丢掉“真优解”。

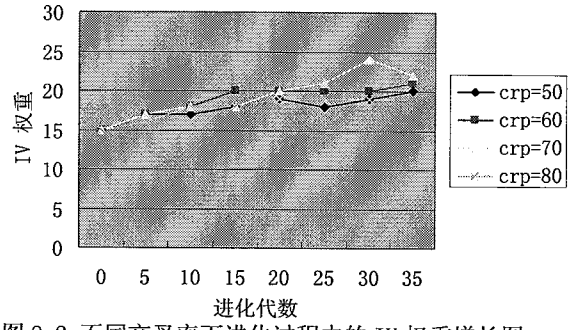

图 9-2 不同交叉率下进化过程中的 IV 权重增长图

图 9-2 是交叉率为 50%，60%，70%，80%下，种群数为 100 时的进化过程 IV 权重增长图。由图 9-2 可看出，随着交叉率的增大（由 50%增长至 70%），进

化过程中 IV 权重增长的越快，相应的所获得的 MDMT(IV)的值相对越大。但交叉率增长到 80%时，IV 权重增长的反而平缓。因为此时变异率相对较小（20%），大量的新一代的染色体来自于父代染色体的随即交叉，而大量的随机交叉有可能会破坏已经进化出的优秀染色体。实验中发现种群数和交叉率/变异率是相互影响的两个参数。

§5 结束语

在对 Trivium 流密码的区分器的寻找中，进行了贪心算法及遗传算法的实验，实验结果表明：遗传算法能够在更小的 IV 权重下，区分更多的初始轮。当前我们的最好结果是找到了一个权重为 24 的 IV 变元集{ 4, 7, 10, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46, 49, 52, 55, 58, 61, 64, 67, 70, 73, 76, 79 }，可区分 929/1152 的初始轮，这比贪心算法搜到 IV 权重为 29 时，只找到 921 轮的区分器来说有不少进步。还有更多的工作正在进行中：如在遗传算法中引入小生境及模拟退火技术；分层遗传算法；自适应遗传算法；遗传算法的并行处理；对 Trivium 流密码的伪随机检测器的分析；在不同密钥权重下的区分器的工作性质的分析等等。相信很快会有更好的结果，我们将另撰文描述。

第十章 结论

§1 总结

本文描述了Gnomon分布式计算平台的总体架构及Gnomon密码计算数学库的设计框架。在此基础上，重点探讨了 Groebner 基算法的实现、优化及在代数攻击中的应用。主要取得了以下成果：

1. 设计了一个通用的、可扩展的分布式密码计算框架——Gnomon 分布式计算平台。实现了常用的几个分布式密码计算算法，并测试了系统的性能。实验表明，随着计算节点数量的增加，其加速比的增长接近线性。

2. 设计了支持符号计算及密码分析工具的数学库——Gnomon 密码计算数学库。解决了在符号计算中不同数据对象互操作的方法。设计了根据不同计算需求的多种多项式表示。设计了密码学中常用的大素数域、二元域及在此基础上的任意次扩域，实现了任意域上的多项式和元素的运算。

3. 提出了对非线性超定方程组的求解算法 F5 的改进方案 F5D 算法。F5D 算法完全继承了 F5 和 MatrixF5 算法在方程组求解问题上的优点，同时分别避免了 F5 算法带来的计算冗余和 MatrixF5 算法过大的存储需求。实验结果表明，在超定环境下，F5D 算法可被用作 F5 算法和 MatrixF5 算法的替代。

4. 设计了一种全新的布尔多项式计算机表示，称之为 BanYan。 BanYan 适用于基于首项约化的求解算法，如 F4，F5，XLs 等算法。与 BDD 和系数矩阵等基于项的传统布尔多项式表示相比，BanYan 可以降低在计算过程中的空间需求，从而显著提升布尔方程组求解算法的现实求解能力。

5. 对 Bivium 流密码算法进行了猜测求解分析。给出了猜测部分变元后子系统平均求解时间的估计模型，提出了基于动态权值以及静态权值的猜测变元选则方法和面向寄存器的猜测方法。在计算 Groebner 基的过程中，对变元序的定义采用了 AB，S，S-rev，SM，DM 等十种新的序。同时，提出了矛盾等式的概念。最后，我们的攻击时间进行了估计。在最坏情况下，使用 DM-rev 序及 Evy3 的猜测位置，猜测 60 个变元有最优的攻击结果，约 $2^{39.16}$ 秒。

6. 给出了 CTC 分组密码的字典序方程组描述，在此基础上，通过最简约化消除中间变元，获得只含有初始密钥变元的方程，简称 K_0 方程。通过实验表明，对于一对明/密文产生的方程，在获得 K_0 方程后再进行求解的方法要优于对 CTC 密码方程的直接求解。同时，把差分方程引入 CTC 密码系统，在此基础上的获得的 K_0 方程会产生多个低次多项式，使得求解速度进一步提高。尤其对于密码系统轮数较多而差分轮数较少的情况，获得 K_0 方程后再求解的效率要远优于直接对密码系统方程并上差分方程的求解，有利于使用低轮差分的高概率去攻击更多轮的密码系统。

7. 探讨了代数攻击与错误攻击，积分攻击，及侧信道（冷冻）攻击的一些结合，对 Present 密码算法进行了分析。实践表明，使用代数攻击与错误攻击或

侧信道攻击相结合，可以求解出全部的 Present 分组密码的初始密钥；而使用代数攻击与积分攻击相结合，可以减少传统积分攻击中所需要的明/密文对数量。

8. 在对 Trivium 流密码的区分器的寻找中，进行了贪心算法及遗传算法的实验。实验结果表明：遗传算法能够在更小的 IV 权重下，区分更多的初始轮。当前我们的最好结果是找到了一个权重为 24 的 IV 变元集{ 4, 7, 10, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46, 49, 52, 55, 58, 61, 64, 67, 70, 73, 76, 79 }，可区分 929/1152 的初始轮，这比贪心算法搜到 IV 权重为 29 时，只找到 921 轮的区分器来说有不少进步。

§2 工作展望

未来的工作计划主要集中在以下几个方面：

1.根据 Groebner 基算法的复杂度理论，自动决策出最优猜测变元的数目；根据静态权值或动态权值法得到最优猜测变元的位置，首先尝试单机求解分析，若超过一定时间无法求解，则调用分布式求解引擎进行分布式求解。

2.方程组的求解工具除了 Groebner 基算法（$F_{4/5}$）外，还有 XL 系列算法，如 XL2，FXL，Mutant-XL 等；Agreeing-Gluing 算法；SAT 求解器；吴方法；穷举方法。每种方法都对特定的方程组（如不同稀疏度，超定性）有效，所以总结出一般性规律，设计出自动选择求解工具的分析工具是一项有价值的工作。

3.面对大的密码系统，单纯的使用解方程方法去攻击往往是很难有效的。所以代数攻击与线性攻击、差分攻击、侧信道攻击、错误攻击等技术的结合是未来进一步的研究内容。

4. 对 Trivium 流密码的区分器与伪随机检测器的更深入分析：如在遗传算法中引入小生境及模拟退火技术；分层遗传算法；自适应遗传算法；遗传算法的并行处理；在不同密钥权重下的区分器的工作性质的分析等等。