基于时空折中算法的密码分析研究与实现

郑亮

北京科技大学

密　　级：公开

论文题目：**基于时空折中算法的密码分析研究与实现**

学　　号：S20082566

作　　者：郑亮

专 业 名 称：计算机科学与技术

2011 年 11 月 20 日

基于时空折中算法的密码分析研究与实现

Research and Implementation On Cryptanalysis Based on Time-memory Trade-off Algorithm

研究生姓名：郑亮

指导教师姓名：王昭顺

北京科技大学计算机与通讯工程学院

北京 100083，中国

Master Degree Candidate： Liang Zheng

Supervisor： Zhao Shun Wang

School of Computer & Communication Engineering

University of Science and Technology Beijing

30 Xueyuan Road，Haidian District

Beijing 100083，P.R.CHINA

分类号：TP393.08　　密　　级：公开

ＵＤＣ：004.77　　单位代码：１０００８

北京科技大学硕士学位论文

论文题目：基于时空折中算法的密码分析研究与实现

作者：郑亮

指 导 教 师：王昭顺 教授　单位：北京科技大学

指导小组成员：　　单位：

　　单位：

论文提交日期：2011 年 11 月 20 日

学位授予单位：北 京 科 技 大 学

致 谢

本论文及毕业设计是在我的导师王昭顺教授的悉心指导下完成的，在此表示由衷的感谢。王老师在我研究生阶段的科研工作给予了大力的指导，他为人师表的专业知识技能、敬业精神和对科学不懈的探索和追求所给予我的影响也将使我在未来的学习和工作中受益。

论文的顺利完成同时得到了赵万里同学、汪翔同学的大力支持和无私帮助，在此表示诚挚的谢意。

感谢在我攻读研究生学位过程中所有给予我帮助的老师们、同学们，你们的帮助使我受益匪浅。

最后，谨向在百忙之中抽出宝贵时间评审本论文的专家、学者致以最诚挚的感谢！

摘 要

在密码分析学中有许多方法对密钥进行破解和分析，但在实际当中使用最多的，同时也是最有效的方法是时空折中法。在 1980 年 Hellman 第一次提出基于时空折中算法进行密码分析，在随后的 2003 年，Oechslin 在原有的算法上提出了现在著名的彩虹表算法。

本文基于以上理论基础，对彩虹表算法进行深入的理论分析和研究，给出了完整的实现，并从基于新型的 Fermi 构架的 CUDA 并行计算、存储结构和算法结构三方面进行优化设计。我们对 SHA1、MD5 和 NTLM 三种 Hash 算法进行实验分析，实验数据表明，SHA1 算法破解的速度提升了 5.88 倍，MD5 算法破解的速度提升了 6.3 倍，NTLM 算法破解的速度提升了 1.77 倍；在磁盘存储空间上，我们重新设计了表的存储结构，得到的新表比原来的节省了 56.25% 的磁盘存储空间，进一步提升了实际破解密钥的时间。

本文可以实现多种 Hash 密码算法的破解，其他研究人员可以在此基础上加入特定的 Hash 算法，如 Word、pdf 文档的 Hash 加密算法，数据库中的 Hash 加密算法等等。

关键词：彩虹表，密码分析，时空折中

Abstract

In cryptanalysis, there are many ways to crack the key and analysis, but in practice the most used, but also the most effective way is Time-Memory Tradeoff (TMTO). In 1980, Hellman first proposed the algorithm based on Time-Memory Trade-off cryptanalysis, in the subsequent 2003, Oechslin on the original algorithm proposed algorithm is now famous rainbow table.

Based on the above theory, based on rainbow tables algorithm in-depth theoretical analysis and research, shows the complete implementation, and from the Fermi architecture based on the new CUDA parallel computing, storage structure and algorithm structure to optimize the design in three areas. Our experimental analysis of the three Hash algorithms:SHA1,MD5 and NTLM. Experimental data show that, SHA1 algorithm to break the 5.88 times faster, MD5 algorithm cracked 6.3 times faster, NTLM algorithm improves the speed of crack 1.77 times; In the disk storage space, we redesigned the table storage structure to be the new table than the original 56.25% saving of disk storage space, further enhancing the actual time to crack the key.

We can achieve a variety of Hash algorithms to crack,other researchers can join on the basis of specific Hash algorithms, such as Word, pdf documents Hash encryption algorithm, encryption algorithm of Hash database, etc.

KeyWords: Rainbow Table, Cryptanalysis, Time-Memory Trade-off

目录

致 谢 I

摘 要 III

Abstract V

第一章 绪论 1

1.1 研究背景及意义 1

1.2 国内外研究现状与进展 1

1.3 本文研究现状 2

1.4 论文组织结构 3

第二章 相关知识背景 4

2.1 密码分析学概述 4

2.1.1 密码分析学的发展背景 4

2.1.2 密码分析学的定义 4

2.1.3 密码分析学研究的必要性 5

2.2 密码分析方法 5

2.2.1 密码攻击类型 6

2.2.2 暴力攻击法 7

2.3 单向散列函数 8

2.3.1 Hash 函数的简介 8

2.3.2 对 Hash 函数攻击的方法 9

2.4 CUDA 并行计算相关知识 11

2.4.1 GPU 并行计算概述 11

2.4.2 CUDA 体系结构 12

2.4.3 Fermi 构架 13

2.5 存储技术 14

2.5.1 直接附加存储 14
2.5.2 网络附加存储 15
2.5.3 SAN 存储方式 16
2.5.4 DAS、NAS 和 SAN 三种存储方式比较 17
2.6 本章小结 17

第三章 时空折中算法 18

3.1 Martin Hellman 最初的算法 18
3.1.1 预运算 18
3.1.2 在线分析阶段 20
3.1.3 TMTO 曲线 20
3.1.4 密钥分析的成功率 21
3.2 Ronald Rivest 差异点 DP 方法 23
3.3 本章小结 23

第四章 彩虹表算法的实现 24

4.1 彩虹表的算法分析 24
4.1.1 非完美彩虹表 24
4.1.2 完美彩虹表 27
4.2 彩虹表的构建与产生 28
4.2.1 定义 Hash 函数 28
4.2.2 定义 Reduce 函数 28
4.2.3 彩虹表参数 28
4.2.4 构建彩虹表 28
4.3 密钥破解 32
4.3.1 搜索算法 32
4.3.2 假警分析 33
4.4 实验结构与分析 33
4.5 本章小结 33

第五章 彩虹表算法的优化 **34**

5.1 基于 CUDA 并行计算优化 34

5.1.1 减小数据传输开销 35

5.1.2 共享内存访问冲突 36

5.1.3 核心配置优化 36

5.1.4 指令级别优化 36

5.1.5 存储合并访问 37

5.1.6 基于 CUDA 的云计算平台 37

5.2 存储优化 37

5.3 算法结构优化 40

5.4 实验结果 42

5.5 本章小结 44

第六章 本文总结与展望 **45**

6.1 总结 45

6.2 展望 45

参考文献 **47**

附录 A MATLAB 程序 **48**

A.1 破解成功率 48

A.2 磁盘空间与成功率 49

A.3 密钥空间 50

A.4 彩虹链长度 50

A.5 磁盘占用空间 51

A.6 最大破解时间 51

A.7 实际破解时间 51

A.8 最大磁盘读取时间 52

A.9 实际磁盘读取时间 52

附录 B 彩虹表部分实现代码 **53**

B.1 彩虹表生成 53

B.2 彩虹表破解 54

附录 C 密钥字符集对照表 **56**

插图

2.1 暴力攻击的几个方法比较 8
2.2 GPU 和 CPU 速度对比 12
2.3 GPU 并行运算体系结构 13
2.4 CUDA 的编译流程 13

3.1 碰撞示意图 22

4.1 彩虹表生成程序 31
4.2 ntlm 算法彩虹表 31
4.3 彩虹表查表过程 32

5.1 EWSA 基于 GPU 加速破解 WPA 速度对比 34
5.2 Pyrit 基于 CUDA 性能测试 34
5.3 Fermi 构架的存储层次结构 36
5.4 优化后彩虹表的空间大小 38
5.5 XFS 文件系统性能测试 39
5.6 EXT4 文件系统行性能测试 39
5.7 链表合并过程示意图 40
5.8 无冲突链表数－链表长度关系 40
5.9 5 张彩虹表的破解成功率 41
5.10 5 张彩虹表的破解成功率（放大） 41
5.11 总表空间大小－成功率关系图 41
5.12 总表空间大小－成功率关系图（放大） 41

表格

5.1 三代 GPU 间的一些规格比较 35
5.2 各种存储系统读速度对比数据 39
5.3 实验数据测试平台配置 42
5.4 SHA1 破解实验数据对比 42
5.5 MD5 破解实验数据对比 43
5.6 NTLM 破解实验数据对比 43

第一章　绪论

1.1　研究背景及意义

随着电子商务的发展，网上银行、网上合同、电子签名等的应用越来越广泛，网络已经成为我们生活中不可或缺的一部分。电子商务在给我们的工作生活带来便捷的同时，也存在着安全隐患。举个简单的例子，Hash 密码算法一直在这些领域中起着身份验证、口令加密、防篡改和重放攻击等作用，目前的用户口令认证机制中，系统将用户的口令进行 Hash 算法加密后存储，以便下次检验用户身份，如果攻击 Hash 算法得到了口令，可想而知对整个系统的安全造成了多大的威胁。

对密码进行分析主要是为了发现加密算法、密钥或密码系统的弱点，以完善加密过程，更有利于信息的安全。另一方面，是为了掌握密码分析者或破译者攻击密码的方法，找出其方法的漏洞，便于预防他们的攻击。同时也是为了更进一步提高广大计算机用户的安全意识和知识水平，减少针对系统的非法入侵和攻击带来的损失。我们知道在整个密码系统中，最有价值的信息就是密钥，绝大部分系统的密钥是用 Hash 函数来保护的，因此，针对密钥的攻击分析是密码分析领域的一个非常有价值的研究的课题。

1.2　国内外研究现状与进展

在 1980 年，Martin Hellman[1]提出了一个“时间空间折中”的密码分析算法，使用了预先计算好并保存在内存和磁盘里面的数据，减少了密码分析需要的时间。这个算法在 1982 年被 Ri vest 提出改进，减少了密码分析过程中所需要的存储空间。

2003 年 7 月瑞士洛桑联邦技术学院的 Philippe Oechslin 公布了一些实验数据，他及其所属的安全及密码学实验室（LASEC）采用了时间空间折中算法，使得密码破解的效率大大提高。他们开发的 O phcrack 项目可以将一个操作系统的用户登录密码破解速度由 1 分 41 秒，提升到 13.6 秒[2]。该

项目提供了一个破解视窗作业系统下的 LAN Manager 散列（比如 hash 文件）的程序，作者免费提供了一些 Rainbow table，可以在短至几秒内破解最多 14 个英文字母的密码，有 99.9%的成功率。从 2.3 版开始可以破解 NT 散列，这功能对已经关闭 LAN Manager 散列的系统（Windows Vista 的预订设定）或是长于 14 个字母的密码特别有用。

同年 project-rainbowcrack 项目开始立项，该项目基于 Philippe Oechslin 提出的彩虹表，用 C++ 基本实现了对 MD5、SHA-1 算法的低位数低密钥空间的破解[3]。接着出现了一个分布式彩虹表项目 F ree Rainbow Tables，这个项目的分布式系统是基于伯克利开放式网络计算平台（BOIN）。

在我国密码分析还处于初级阶段，由于软、硬件及技术等各种原因，大部分密码分析方法还处于理论阶段。目前，已经出现了各种各样的密码分析系统，都是针对某种加密方法进行分析的，功能和方法上还具有一定的局限性。2004 年 8 月，在美国加州圣芭芭拉召开的国际密码大会上，山东大学王小云教授在会议上首次宣布了她及她的研究小组近年来的研究成果——对 MD5、HAVAL － 128、MD4 和 RIPEMD 等四个著名密码算法的破译结果。2008 年国际密码学家 Lenstra 利用王小云提供的 MD5 碰撞，伪造了符合 X.509 标准的数字证书，说明了 MD5 的破译已经不是理论破译结果，而是可以导致实际的攻击，目前 SHA － 1 在理论上已经被破译，离实际应用也为期不远。目前国内已经有对基于时空折衷算法的 Word 文档破解研究[4]和对 DES 密码算法的彩虹攻击技术及其 GPU 实现[5]两篇与彩虹表算法相关的文献。

1.3 本文研究现状

本文研究的主要内容就是基于时间空间折中算法的 Hash 密钥分析。主要采用彩虹表进行 Hash 算法破解，并进一步对时空折中算法的研究和优化，开发出基于 CUDA[6]模型的彩虹表算法实现。主要研究成果有：

1. 优化彩虹表算法参数，减少破解时间；

2. 优化彩虹表的数据结构，减少表的存储空间；

3. 利用 GPU 高性能并行运算提高破解速度。

1.4 论文组织结构

本文共分六章，全文结构安排如下：

第一章　绪论。介绍了本课题的研究背景及意义、国内外研究现状与进展、研究现状以及本文组织结构。

第二章　相关知识背景。

第三章　时空折中算法。

第四章　彩虹表算法的实现。

第五章　彩虹表算法的优化。

第六章　本文总结与展望。

第二章　相关知识背景

2.1　密码分析学概述

2.1.1　密码分析学的发展背景

在古代，人类就已经开始使用密码用于保护外交和军事通信，而随着科技和信息技术的发展，在当今互联网性信息时代，大量的敏感和机密信息，如网上银行账户、私人信件、各个账户的密码等，都是常通过非可信的互联网来进行传输和交换的，然而对于这些信息的机密性、完整性和可用性是人们迫切需要的。因此，随之而来的信息安全问题日益突出，信息的安全威胁主要来自黑客攻击、计算机病毒、拒绝服务、中间人攻击、嗅探重放等。这就需要一个安全可靠的系统来保护信息的安全，目前人们对信息安全越来越重视，因此也有越来越多的人从事密码分析的研究。

2.1.2　密码分析学的定义

密码分析学是一门研究在不知道通常解密所需要的秘密信息的情况下对加密信息进行解密的学科，是密码学的一个分支，它的主要目的是研究信息的破解和信息的伪造。试图发现明文或密钥的过程就叫做密码分析。密码分析人员使用的策略取决于加密方案的特性和分析人员可用的信息。密码分析学是对密码算法进行分析或破译，在未知密钥的情况下，从密文推出明文或密钥的技术。密码编码学和密码分析学这两门学科尽管表面上看来是相互对立，但在整个密码学的发展过程中，却又是相辅相成、相互促进的。

在一个不可信的信息传输和处理系统中，除了合法的接收者外，还有非授权者，他们通过窃听、中间人攻击和重发攻击等手段来获取机密信息。他们虽然不知道系统所用的密钥，但通过分析可能从截获的密文分析出原来的明文甚至加密算法的密钥，这一过程称作密码分析[7]。从事这一工作的人称作密码分析员或密码分析者。一个密码是可破的，是指的通过密文能够在可容忍代价下分析出明文或密钥，或者通过明文一密文对能够确定密钥。

2.1.3 密码分析学研究的必要性

虽然密码分析的目标在密码学的历史上从古至今都一样，实际使用的方法和技巧则随着密码学变得越来越复杂而日新月异。密码学算法和协议从古代只利用纸笔等工具，发展到第二次世界大战时的恩尼格玛密码机，直到目前的基于电子计算机的方案。而密码分析也随之改变了。无限制地成功破解密码已经不再可能。事实上，只有很少的攻击是实际可行的。在上个世纪 70 年代中期，公钥密码学作为一个新兴的密码学分支发展起来了。而用来破解这些公钥系统的方法则和以住完全不同，通常需要解决精心构造出来的纯数学问题。其中最著名的就是大数的质因数分解。

对密码进行分析主要是为了发现加密算法、密钥或密码系统的弱点，以完善加密过程，更有利于信息的安全。另一方面，是为了掌握密码分析者或破译者攻击密码的方法，找出其方法的漏洞，便于预防他们的攻击。同时也是为了更进一步提高广大计算机用户的安全意识和知识水平，减少针对系统的非法入侵和攻击带来的损失。因此，进行密码分析是非常必要的。

2.2 密码分析方法

密码学在[8]中可以分为经典密码学和现代密码学，而我们现在研究分析的主要领域在现代密码学，现代密码学包括分组密码算法、消息摘要算法、非对称密钥算法、公/私钥签名算法等。密码分析可以从不同的角度进行分类，并且每种方法之间也没有严格的界限，在这里我们根据上述密码学中密码体制的类型来对密码分析方法进行大体分类。密码分析方法从大的方面可分为: 古典密码分析方法，对称密码分析方法，非对称密码分析方法。因为密码有序列密码和分组密码之分，所以对称密码分析方法又分为序列密码分析方法和分组密码分析方法。现有的大多数非对称密码都属于分组密码，所以对非对称密码分析方法不再从这方面分类。本文主要讨论的是针对密码算法中密钥分析的很实用的一些方法，如穷举法、查表法、时空折中法。

2.2.1 密码攻击类型

我们在进行密码分析时是在假设密码分析者知道目标体统所使用的加密体制和密码算法的前提下进行的，也就是密码分析者可以根据密码算法得到明文和密文等方面的信息。这样我们可分将密码攻击为以下几种主要类型:

1. 唯密文攻击: 密码分析者已知加密算法和待破译密文或部分密文，需要对信息加密的方法进行正确的猜测，对编码者的编码风格及密文的题材有一定的了解。

2. 已知明文攻击: 密码分析者已知加密算法，有一些明文及相应的密文。用这些信息推出用于产生密文的信息。

3. 选择明文攻击: 也称差分密码分析。密码分析者有机会使用密码机，且已知加密算法、待破译的密文、由密码分析者选择的明文信息。密码分析者用一个密钥对他所选择的明文加密以获得结果中的密文，但密钥本身不能被分析，密码分析者通过将整个密文与最初的明文作比较推出密钥。

4. 选择密文攻击: 密码分析者已知加密算法，待破译的密文和密码分析者选择的猜测性密文。密码分析者将自己猜测的密文发给信息的实际接收者，接收者解密后得到一些杂乱的数据，于是他可能将这些杂乱的数据寄回给信息发送者或者以不安全的方式存储，则密码分析者可通过某些手段可能得到这些杂乱数据，再与猜测的密文作比较可推出密钥。

5. 选择文本: 密码分析者已知加密算法，待破译的密文，密码分析者选择的明文信息及其对应的由密钥生成的密文，密码分析者选择的猜测性密文及其对应的由密钥生成的已破译的明文。密码分析者通过他所掌握的这些所有信息可推出密钥。

上述是密码攻击的主要五种类型。这五种攻击类型的强度按序递增，唯密文攻击是最弱的一种攻击，最容易防护，因为密码分析者拥有的可供利用信息量最少。选择密文和选择文本是最强的攻击，如果一个密码系统能够抵抗这两个攻击，那么它当然能够抵抗其余三种攻击，这两者很少被使用，但他们

也是可能的攻击途径。对一个密码系统采取截获密文进行分析的这类攻击称作被动攻击。

密码系统还可能遭受到的另一类攻击是主动攻击，非法入侵者主动向系统采用监听、删除、修改、增添、重放、伪造等手段向系统注入假消息。防止这种攻击的一种有效方法是使发送的消息具有可被验证的能力，使接收者或第三者能够识别和确认消息的真伪，实现这类功能的密码系统称作认证系统。消息的认证性和消息的保密性不同，保密性是使截获者在不知道密钥的条件下不能解读密文的内容，而认证性是使任何不知道密钥的人不能构造出一个密报，使意定的接收者解密成为一个可理解的消息 (合法的消息)。

进行密码分析时，我们还应考虑一种密码攻击的复杂度，当然复杂度越低越好。可将密码攻击复杂度分为两部分，数据复杂度和处理复杂度。数据复杂度是实施该攻击所需输入的数据量; 而处理复杂度是处理这些数据所需的计算量。这两部分的主要部分通常被用来刻划该攻击的复杂度。例如，在穷举密钥搜索攻击中，所需要的数据量与计算量相比是微不足到的，因此，穷尽密钥搜索攻击的复杂度实际是处理复杂度。在差分密码分析中，实施攻击所需的计算量相对于所需的明密文对的数量来说是比较小的，因此，差分密码分析的复杂度实际是数据复杂度。

2.2.2 暴力攻击法

暴力攻击法可用于任何分组密码算法和消息摘要算法，而且攻击的复杂度只依赖于分组长度和密钥长度，暴力攻击主要有: 穷举密钥攻击、字典攻击、查表攻击、时间 -存储攻击。

穷举密钥搜索攻击中，设 k 是密钥长度 (以比特为单位)，在唯密文攻击下，攻击者依次试用密钥空间中所有 2^k 个密钥解密一个或多个截获的密文，直至得到一个或多个有意义的明文块。在已知 (选择) 明文攻击下，攻击者试用密钥空间中的所有 2^k 个密钥对一个已知明文加密，将加密结果同该明文相对应的已知密文比较，直至二者相等，然后再用其他几个已知明密文对来验证该密钥的正确性。穷举密钥搜索的复杂度平均为 2^{k-1} 次加密，实际上这种攻击方法适用于任何密码体制。

字典攻击中，攻击者搜集明密文对，并把它们编排成一个“字典”。攻击者看见密文时，检查这个密文是否在字典里，如果在，他就获得了该密文相对应的明文。如果 n 是分组长度，那么字典攻击需要 2^n 个明密文对才能使攻击者在不知道密钥的情况下加解密任何消息。

查表攻击中，设 k 是密钥长度，查表法采用选择明文攻击，其基本观点是: 对一个给定的明文 x，用所有 2^k 个密钥 K(记其全体为 K)，欲计算密文 $y_k = E_k(x)$。构造一张有序对表 $(y_k\ K)_{k \in K}$，以 y_k 给出 K 的标号。因此，对于给定的密文，攻击者只需从存储空间中找出相对应的密钥 K 即可。

时间 -空间折中攻击法是一种选择明文攻击方法，它由穷尽密钥搜索攻击和查表攻击两种方法混合而成，它在选择明文攻击中以时间换取空间。它比穷尽密钥搜索攻击的时间复杂度小，比查表攻击的空间复杂度小。如图2.2所示，比较了这种暴力攻击方法的特点：

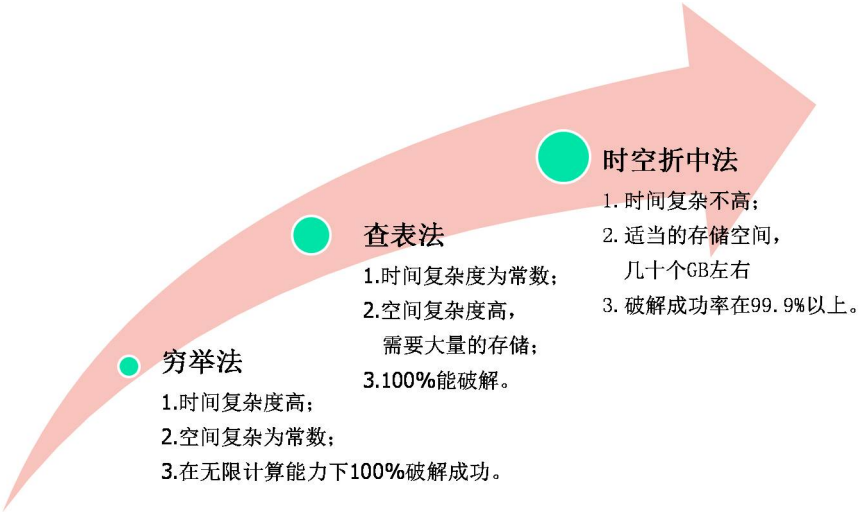

图 2.1 暴力攻击的几个方法比较

2.3 单向散列函数

2.3.1 Hash 函数的简介

单向散列函数，又称单向 Hash 函数、杂凑函数，就是把任意长的输入消息串变化成固定长度的输出串的一种不可逆函数。这个输出串称为消息的散列值。一般用于密钥加密，产生消息摘要等。单向散列函数是现代密码学的一个重要领域，它是数据完整性检测、数字签名和认证方案中必不可少的

一部分。单向 Hash 函数[9]是许多协议框架中的一个模块。目前由许多 Hash 函数的公开算法，一般一个安全的 Hash 函数应该至少满足以下几个条件：

1. 输入串值的长度是任意的；
2. 输出串 Hash 值长度是固定的；
3. 对每个给定的输入串的值，计算机得到输出 Hash 值是很容易的；
4. 给定 Hash 函数的描述，已知一个 Hash 值时，要找到输入串使它的的 Hash 值等于已知的这个 Hash 值在计算上时不可行的，或是找到两个不同的输入串，计算得到相同的输出 Hash 值在计算上是不可行的。

Hash 函数主要用于数据完整性校验和数字签名的有效性，常被用在身份认证上。例如在一个身份验证系统上，保存用户的密码时，需要把密钥用 Hash 算法进行加密，得到一个 Hash 值。由于 Hash 函数本身的特点，其他用户即使得到了这个 Hash 值也无法还原密码。当用户登录时，系统把用户输入的密码再用 Hash 算法进行计算，得到的 Hash 值与保存在系统中的 Hash 值进行比较，从而验证用户的合法性。MD5[10]（Message-Digest Algorithm 5）是目前应用最广泛的 Hash 函数之一。MD5 将任意长度的“字节串”变换成一个 128 比特的大整数，并且它是一个不可逆的字符串变换算法，换句话说就是，即使你看到源程序和算法描述，也无法将一个 MD5 的值变换回原始的字符串，从数学原理上说，是因为原始的字符串有无穷多个，这有点象不存在逆函数的数学函数。MD5 在经过一些初始处理后，将明文分成了 512 位的块，再将每一块分成 16 个 32 位的子块。算法的输出是 4 个 32 位的块，连接起来就是 128 位的输出的 Hash 值。

2.3.2 对 Hash 函数攻击的方法

1. 替换法

   这是一个十分实用的攻击方法，它并不对 Hash 算法本身作任何攻击，只是利用系统中的 Hash 函数重新生成一个 Hash 值，这个 Hash 值的输入串是攻击者已知的，如“password”，这样我们就可能把这串新生成的 Hash 值替换掉系统本身的 Hash 值，此时攻击者就能用“password”

能登陆系统，从而达到绕过系统的认证机制。这种攻击方法需要攻击者已知目标系统认证机制使用的 Hash 算法函数（一般的系统都使用 MD5 算法函数）和由替换的权限。

2. 字典查表法

还有一种在实际破解中使用较多得方法是字典查询法，攻击者需要预先对目标 Hash 算法构造相应得字典文件，然后把需要破解得 Hash 值跟这个字典文件里得 Hash 值进行检索比较，通常这种办法需要 TB 级甚至跟大得存储空间，并且预运算的时间代价也是很大的。

3. 碰撞法

所谓杂凑碰撞指两个完全不同的讯息经杂凑函数计算得出完全相同的杂凑值。根据鸽巢原理，以有长度限制的杂凑函数计算没有长度限制的讯息是必然会有冲撞情况出现的。可是，一直以来，电脑保安专家都认为要任意制造出冲撞需时太长，在实际情况上不可能发生。2004 年 8 月 17 日的美国加州圣巴巴拉的国际密码学会议（Crypto’2004）上，来自中国山东大学的王小云教授做了破译 MD5、HAVAL-128、MD4 和 RIPEMD 算法的报告，公布了 MD 系列算法的破解结果。在破解 MD5 之后，2005 年 2 月，王小云教授又破解了另一国际密码 SHA － 1，王小云的研究成果表明了从理论上讲电子签名可以伪造，必须及时添加限制条件，或者重新选用更为安全的密码标准，以保证电子商务的安全。2005 年 8 月，王小云、姚期智，以及姚期智妻子姚储枫 (即为 Knuth 起名高德纳的人) 联手于国际密码讨论年会尾声部份提出 SHA-1 杂凑函数杂凑冲撞演算法的改良版。此改良版使破解 SHA-1 时间缩短。

曾与王小云共同攻破完整 MD5 算法的中科院的冯登国与长沙国防科大的谢涛最近在 IACR 上发布了一篇题为《仅用一个消息块实现 MD5 碰撞》的文章。出于安全因素的考虑，文章中并未公开具体实现方法，但他们向全球密码学爱好者发出挑战：第一个在 2013 年 1 月 1 日之前找出新的单块碰撞的人将获得一万美元的奖励。他们公布的产生碰撞的消息块为：0x6165300e,0x87a79a55,0xf7c60bd0,0x34febd0b,0x6503cf04,0x854f709e,

0xfb0fc034,0x874c9c65,0x2f94cc40,0x15a12deb,0x5c15f4a3,0x490786bb,
0x6d658673,0xa4341f7d,0x8fd75920,0xefd18d5a
0x6165300e,0x87a79a55,0xf7c60bd0,0x34febd0b,0x6503cf04,0x854f749e,
0xfb0fc034,0x874c9c65,0x2f94cc40,0x15a12deb,0xdc15f4a3,0x490786bb,
0x6d658673,0xa4341f7d,0x8fd75920,0xefd18d5a
这两个消息块有两处不同，但 MD5 均为 0xf999c8c9 0xf7939ab6 0x84f3c481 0x1457cb23。

2.4 CUDA 并行计算相关知识

2.4.1 GPU 并行计算概述

GPU 即图形处理器，目前，主流计算机的处理器主要是中央处理器 CPU 和图形处理器 GPU。传统上，GPU 只负责图形渲染，而大部分的处理都交给了 CPU。然而 GPU 在处理能力和存储器带宽上相对 CPU 有明显优势，在成本和功耗上也不需要付出太大代价。2007 年 6 月，NVIDIA 公司推出了 CUDA，CUDA 不需要借助图形学 API，而是采用了类 C 语言进行开发。同时，CUDA 的 GPU 采用了统一处理架构，以及引入了片内共享存储器，这大大降低了基于 GPU 的并行计算程式的开发难度。CUDA 为开发人员有效利用 GPU 的强大性能提供了条件。自推出后，CUDA 被广泛应用于密码破解、石油勘测、天文计算、流体力学模拟、分子动力学仿真、生物计算、图像处理、音视频编解码的领域，在很多应用中获得了几倍、几十倍、甚至上百倍的加速比。

在现代计算机体系结构中属于显卡外设的一个专门图形处理器。随着人们对显卡并行性的研究和开发，我们对它的可编程性也提出了更高的需求。比如从 Nvidia Geforce 8 系列的显卡开始，英伟达公司就在其上面配置了大量的可独立运行计算的处理器核心。在 2010 年的 ISCA 大会上，英特尔公司向大会递交了一份关于 GPU 和 CPU 的测试技术文档，在文档中正式承认 GPU 的运行速度比 CPU 快 14 倍[11]。测试中英特尔采用的是 GeForce GTX 280，在当时是英伟达上一代产品，并且没有经过任何代码优化，而实际上新一代的 NVIDIA GPU 的运行速度是英特尔 CPU 的上百倍。

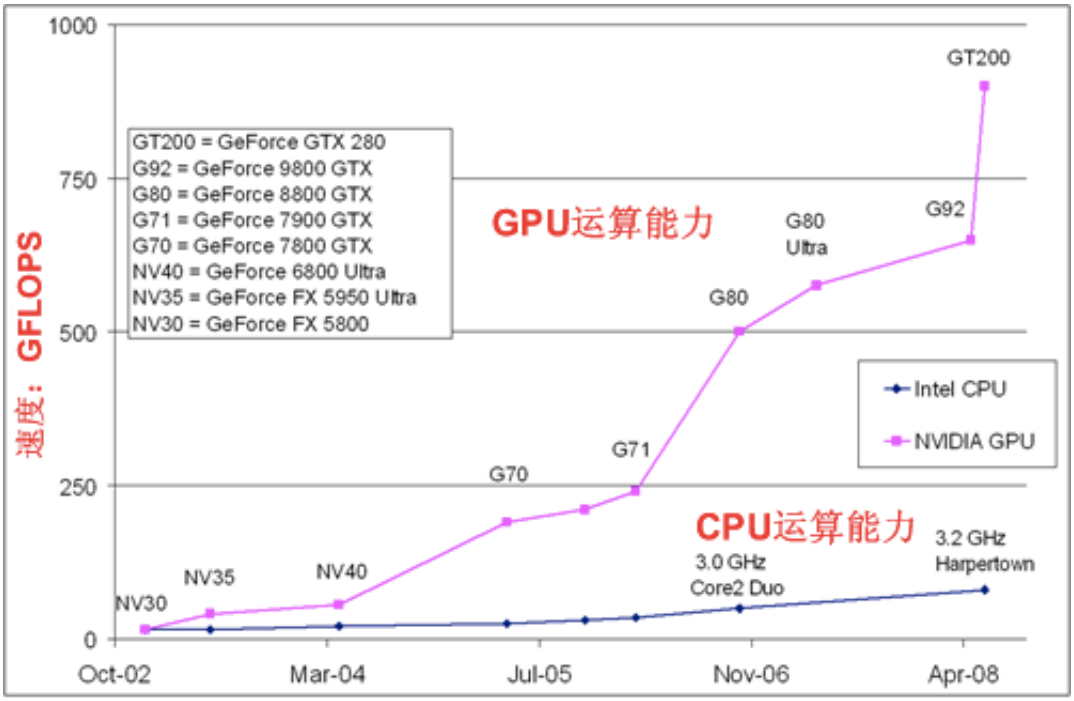

图 2.2 GPU 和 CPU 速度对比

英伟达公司通过对显卡增加一些硬件接口，使得对显卡的编程已经不再需要像操作硬件设备那样依赖特定的语言和程序接口。他们提出了 CUDA 的编程模型，这个编程模型是为了方面开发者能够便捷快速地在 Nvida 显卡上开发应用程序。

2.4.2 CUDA 体系结构

2007 年 6 月，NVIDIA 公司推出了 CUDA[6]，它是一个开发应用程序的平台，或者说是一种编程模型。该模型将用户应用程序代码分为两部分：宿主（Host）代码和设备（Device）代码。宿主代码由 CPU 负责串并行化的处理，而设备代码则是交给一个或多个 GPU 并发处理。同时每个 CUDA 处理器都支持单指令多数据（SIMD）方法,[12]可以使得所有核心处理器上并发运行的线程基于相同的代码。虽然由于数据的不同会导致产生分支，但是仍然支持全局的内存共享。下图2.3为 Nvidia GeForce 8800 显卡的工作示意图，我们可以看到程序从主存把数据拷贝到 GPU 的显存里，然后在 GPU 中每个核心处理器中进行并行计算，最后把结果拷贝回主存里。

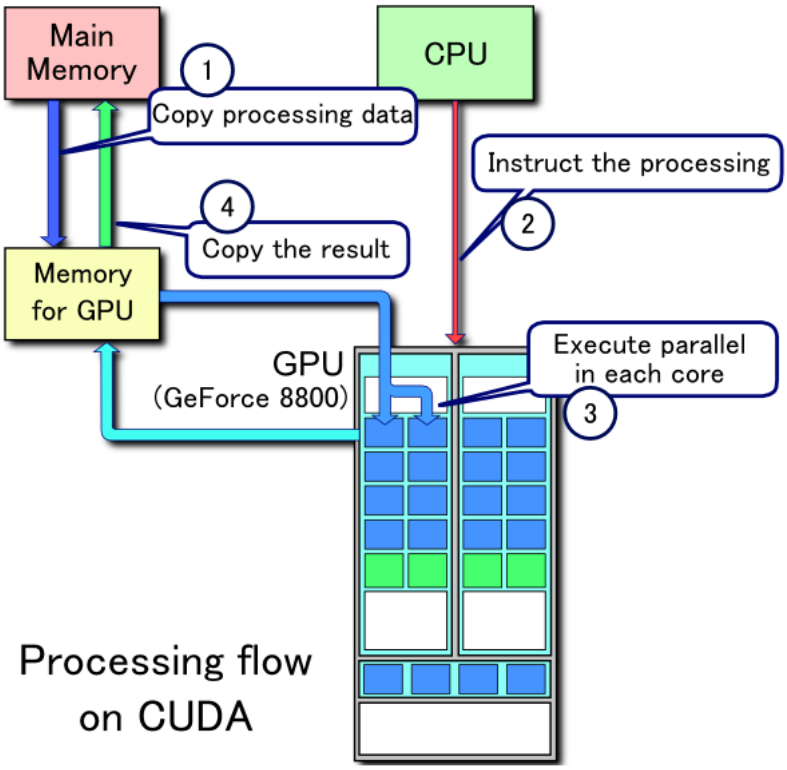

图 2.3　GPU 并行运算体系结构

CUDA 程序由 nvcc 编译器执行，编译流程如图2.4所示。代码运行期间，程序执行可能循环经历宿主态和设备态的转换，这种转换其实质是 CPU 和 GPU 之间的转换，我们应该尽量减少这样的状态转换，避免上下文切换带来的代价。因此在编程阶段应当尽量使 GPU 能运行足够长的时间。

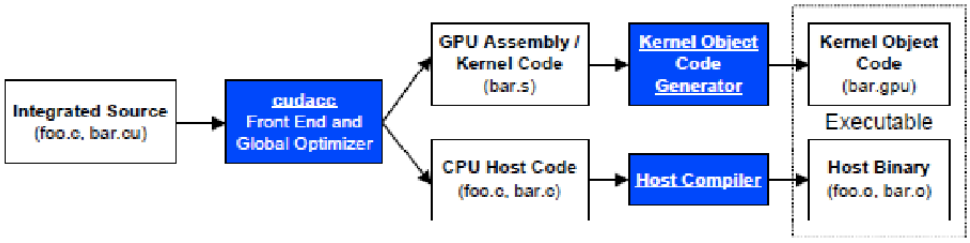

图 2.4　CUDA 的编译流程

2.4.3　Fermi 构架

Fermi 是 NVIDIA 公司最新一代的显卡构架代号，在 2010 年 3 月 27 日，NVIDIA 公司正式发布 Fermi 构架的桌面显卡 GeForce GTX480。Fermi 是新一代 CUDA 计算与图形构架，针对双精度运算、ECC 内存容错能力、Cache 体系结构、上下文切换和原子读写操作等方面进行了全面的设计，通

过新的构架设计不仅仅增加了计算能力，并且支持更好的可编程能力和计算效果。Fermi 构架有如下主要新的特性：

1. 第三代 Streaming Multiprocessor（SM）
   - 每个 SM 包含 32 个 CUDA Core，是 GST200 的 4 倍
   - 8 倍于 GT200 的双精度浮点性能
   - Dual Warp 调度策略，一个周期内可启动两个 warp 进行计算
   - 每个 SM 有 64KB 的 RAM，可灵活地配置共享内存和 L1 缓存
2. 第二代 PTX ISA 架构（PTX 2.0）
   - 统一寻址空间，完整支持 C++ 特性
   - 针对 OpenCL 和 DirectCompute 进行优化设计
   - 完整支持 IEEE 754-2008 32bit/64bit 精度
   - 通过 Predication（断言）来提高性能
3. 改进的内存操作子系统
   - 可配置的 L1 和统一的 L2 并行数据高数缓存
   - 内存支持 ECC 校验
   - 极大增强原子内存操作性能
4. 第三代千兆线程引擎（GigaThread 3.0）
   - 比上一代强 10 倍的上下文切换能力
   - 并发核心程序执行机制

2.5 存储技术

这里我们主要介绍三种存储技术：DAS、NAS 和 SAN。

2.5.1 直接附加存储

直接附加存储方式与我们普通 PC 的存储构架一样，外部的存储设备都是直接连接在服务器的内部总线上，是整个服务器结构的一部分。

DAS 存储方式主要使用与以下环境：

1. 小型网络

   因为网络规模较小，数据存储量小，且也不是很复杂，采用这种存储方式对服务器的影响不会很大。并且这种存储方式也十分经济，适合拥有小型网络的企业用户。

2. 地理位置分散的网络

   虽然企业总体网络规模较大，但在地理分布上很分散，通过 SAN 或 NAS 在它们之间进行互联非常困难，此时各分支机构的服务器也可采用 DAS 存储方式，这样可以降低成本。

3. 特殊应用服务器

   在一些特殊应用服务器上，如微软的集群服务器或某些数据库使用的原始分区，均要求存储设备直接连接到应用服务器。

2.5.2 网络附加存储

NAS(网络附加存储) 方式则全面改进了以前低效的 DAS 存储方式。它采用独立于服务器，单独为网络数据存储而开发的一种文件服务器来连接所存储设备，自形成一个网络。这样数据存储就不再是服务器的附属，而是作为独立网络节点而存在于网络之中，可由所有的网络用户共享。

NAS 的优点：

1. 真正的即插即用　NAS 是独立的存储节点存在于网络之中，与用户的操作系统平台无关，真正的即插即用。

2. 存储部署简单　NAS 不依赖通用的操作系统，而是采用一个面向用户设计的，专门用于数据存储的简化操作系统，内置了与网络连接所需要的协议，因此使整个系统的管理和设置较为简单。

3. 存储设备位置非常灵活

4. 管理容易且成本低　NAS 数据存储方式是基于现有的企业 Ethernet 而设计的，按照 TCP/IP 协议进行通信，以文件的 I/O 方式进行数据传

输。

NAS 的缺点：(1) 存储性能较低　(2) 可靠度不高

2.5.3 SAN 存储方式

1991 年，IBM 公司在 S/390 服务器中推出了 ESCON(Enterprise System Connection) 技术。它是基于光纤介质，最大传输速率达 17MB/s 的服务器访问存储器的一种连接方式。在此基础上，进一步推出了功能更强的 ESCON Director(FC SWitch)，构建了一套最原始的 SAN 系统。

SAN 存储方式创造了存储的网络化。存储网络化顺应了计算机服务器体系结构网络化的趋势。SAN 的支撑技术是光纤通道 (FC Fiber Channel) 技术。它是 ANSI 为网络和通道 I/O 接口建立的一个标准集成。FC 技术支持 HIPPI、IPI、SCSI、IP、ATM 等多种高级协议，其最大特性是将网络和设备的通信协议与传输物理介质隔离开，这样多种协议可在同一个物理连接上同时传送。

SAN 的硬件基础设施是光纤通道，用光纤通道构建的 SAN 由以下三个部分组成：

1. 存储和备份设备：包括磁带、磁盘和光盘库等。
2. 光纤通道网络连接部件：包括主机总线适配卡、驱动程序、光缆、集线器、交换机、光纤通道和 SCSI 间的桥接器
3. 应用和管理软件：包括备份软件、存储资源管理软件和存储设备管理软件。

SAN 的优势：

1. 网络部署容易；
2. 高速存储性能。因为 SAN 采用了光纤通道技术，所以它具有更高的存储带宽，存储性能明显提高。SAn 的光纤通道使用全双工串行通信原理传输数据，传输速率高达 1062.5Mb/s。

3. 良好的扩展能力。由于 SAN 采用了网络结构，扩展能力更强。光纤接口提供了 10 公里的连接距离，这使得实现物理上分离，不在本地机房的存储变得非常容易。

2.5.4 DAS、NAS 和 SAN 三种存储方式比较

存储应用最大的特点是没有标准的体系结构，这三种存储方式共存，互相补充，已经很好满足目前企业信息化应用。从连接方式上对比，DAS 采用了存储设备直接连接应用服务器，具有一定的灵活性和限制性；NAS 通过网络（TCP/IP,ATM,FDDI）技术连接存储设备和应用服务器，存储设备位置灵活，随着万兆网的出现，传输速率有了很大的提高；SAN 则是通过光纤通道（Fibre Channel）技术连接存储设备和应用服务器，具有很好的传输速率和扩展性能。三种存储方式各有优势，相互共存，占到了现在磁盘存储市场的 70% 以上。

2.6 本章小结

本章简要介绍了本文涉及到的相关知识背景，主要包括密码分析学的发展背景、定义和研究的意义，阐述了现代密码分析的方法手段和密码攻击的类型，比较了暴力攻击法的几种方法，穷举法的虽然可以 100% 破解成功，但这是建立在付出巨大的计算代价和时间代价；查表法则以巨大的存储代价来达到密码破解的目的；而时空折中法则是前两种办法的折中办法，以空间代价换取时间代价或者以时间代价换取空间代价，在两个中间找到一个平衡点，这样会损失一些破解成功率。最后以单向散列函数的破解为例，介绍了对 Hash 函数加密了的密钥的破解。介绍 CUDA 并行计算的相关知识，包括 GPU 并行计算体系结构以及 CUDA 编程模型，一个基本的 CUDA 程序由宿主程序和设备程序组成；最后介绍了三种存储方式技术：DAS、NAS 和 SAN。

第三章　时空折中算法

本章主要要介绍时空折中算法的相关理论基础。从最初 Martin Hellman 的时空折中算法、Ronald Rivest 的差异点 DP 方法到 Philippe Oechslin 在 2003 年基于前两种方法提出的彩虹表算法（RainBow Table），时空折中算法已经成为现代密码分析算法中一类极具现实意义的算法之一。这类算法一般都包括了以下两个主要步骤：1，预运算（pre-computation）；2，在线分析（online phase）。本章节3.1,3.2和4.1将分别介绍 Martin Hellman 表以及彩虹表的设计原理和构造思想。

我们将在本章中统一使用以下定义：N 表示目标密码算法的密钥空间大小；T 和 M 分别表示在线分析的时间代价以及预运算步骤所产生的密钥表空间大小；对于预运算的成功率 P，我们通常认为分析者具有足够长的时间，所以该代价一般不作为讨论的内容并且将其粗略等价于穷举所有密钥表空间的时间。

3.1　Martin Hellman 最初的算法

Martin Hellman 在 1980 年第一次提出基于时间空间折中算法的分组密码算法 DES 的密码分析[1]。攻击者使用的是选择明文攻击，也就是给定一个指定的明文加密后的密文，尝试从密文中分析恢复出这次加密的密钥。因此攻击者所要关心的问题是如何从 N 中找出对应的密钥，以 56 比特 DES 为例，其密钥空间为 $N = 2^{56}$。（若无特殊说明，下文都将以 56 比特的 DES 为目标密码算法）

3.1.1　预运算

在预运算阶段，算法将首先固定一个目标密文所对应的明文 P，一般大小为 8 个字符（64 比特），接着结合 P 构造如下非逆函数 f：

$$f(k) = R(S_k(P)) \tag{3.1}$$

其中 P 是所选的固定明文信息，S 表示伪随机函数，R 是一个从密文空间到密文空间的简约（Reduction）函数，并且在下文令 S=DES。对于 R 函数的选择，若无特别说明，我们将假定任一从 64 比特到 56 比特的映射函数均可适用。通常为了简便起见，令 R 函数为仅简单地地去掉 64 比特的高 8 位。从而 R、S 复合而成的 f 函数可以看作是一个 56 比特到 56 比特的伪随机函数。

预运算开始时，算法将选择 m 个来自密钥空间 N 的随机密钥作为开始节点（Start Point，简称 SP），令其为 $SP_1, SP_2, SP_3 \ldots SP_{m-1}, SP_m$。接着，将 SP_1 作为输入代入公式(3.1)，并迭代 t 次，得到如下两式(3.2)和(3.3)：

$$k_i = f(k_{i-1})(1 \leq i \leq t) \tag{3.2}$$

$$SP_1 = k_{10} \xrightarrow{f} k_{11} \xrightarrow{f} k_{12} \xrightarrow{f} \cdots \xrightarrow{f} k_{1t} = EP_1 \tag{3.3}$$

其中，令结束节点（End Point，简称 EP）$EP = f(K_{t-1})$。当每个 SP_j 都完成 t 次迭代后，我们就会得到一张有 m 对形式如 $(SP_j, EP_j)(1 \leq j \leq m)$ 的二元组构成的 Hellman 表。

$$\underbrace{\begin{bmatrix} SP_1 = k_{10} \xrightarrow{f} k_{11} \xrightarrow{f} k_{12} \xrightarrow{f} \cdots \xrightarrow{f} k_{1t} = EP_1 \\ SP_2 = k_{20} \xrightarrow{f} k_{21} \xrightarrow{f} k_{22} \xrightarrow{f} \cdots \xrightarrow{f} k_{2t} = EP_2 \\ \vdots \\ SP_m = k_{m0} \xrightarrow{f} k_{m1} \xrightarrow{f} k_{m2} \xrightarrow{f} \cdots \xrightarrow{f} k_{mt} = EP_m \end{bmatrix}}_{\text{迭代}t\text{次}} \tag{3.4}$$

$$\begin{bmatrix} SP_1 & EP_1 \\ SP_2 & EP_2 \\ \vdots & \vdots \\ SP_m & EP_m \end{bmatrix} \tag{3.5}$$

这里要注意的是，在 SP 和 EP 之间的节点都不会被保存，如(3.5)式。这些值可以依靠相应的二元组在需要使用的时候可以在线计算生成，这也就是把节省了空间，而关于 m，t 的选择，就是这个时间于空间折中的选择，一般

它们应当满足：

$$mt^2 = N \tag{3.6}$$

而且(3.6)式也被成为矩阵终止规则（Matrix stopping rule）。根据生日悖论思想，当上式成立时，将不会有太多的重复节点出现，而当 m 或 t 过大使得 $mt^2 > N$，则冲突数量将快速上升，并最后导致 Hellman 表的成功率下降。而事实上，在实际的密钥攻击过程中，可以允许 $mt^2 < N$，只不过相应的攻击成功率会有所下降。

3.1.2 在线分析阶段

在线分析阶段，给定已知明文 P 和对应的密文 C，代入 R（C）可以得到 y_1，然后将 y_1 与 $EP_i(i = 1, 2, \ldots, m)$ 比较，若存在某个 i，使得等式 $y_i = EP_i$ 成立，则会出现以下两种情况：1，加密 y_1 的密钥为 K $k_{i,t-1}$；2，所对应的密钥不在表中，这个现象叫做 False alarm。若等式 $y_1 = EP_i$ 不成立，则继续迭代下一步 $y_2 = f(y_1)$，并重复上一步相同的比较，直到出现以下三种情况：1，找到密钥；2，出现 False alarm，就也就假警；3，表搜索结束。简单地讲，在线分析的目的是在 Hellman 表中搜索出正确的密钥 K, 使得 $K = k_{ij} = y_{t-j}$。需要注意的是，在线分析过程中的 y_j 是可以反复利用与 y_{j+1} 计算的，之后介绍的彩虹表将无法重用。

3.1.3 TMTO 曲线

在介绍 TMTO 曲线前，我们将先讨论 Hellman 表的空间代价和时间代价。在忽略二元组 (SP_j, EP_j) 本身大小和一些其他较小的常量后，我们可以计算出存储 t 张维度为 $mimest$ 大小的 Hellman 表需要的空间 M = mt。同时，由于试图要覆盖整个密钥空间，故预运算的代价 P = N。在线分析阶段，每一张表的搜索，函数 f 最多会被调用 t 次，因此 t 张表的总时间代价 $T = t^2$，由于搜索的代价相对比较低，在这里可以忽略不计。基于上述时间和空间的代价分析就很容易得到 TMTO 曲线：

$$TM^2 = N^2 \text{并且} P = N \tag{3.7}$$

反之，当给定时间代价 T 和空间代价 M 时，则可以通过 3.5 式推出 m 和 t，也就是说可以在时间 T 和空间 M 得限制下，从以 m，t 为参数得 Hellman 表中找到正确得密钥 K。

结合(3.6)式和(3.7)式我们得到一个重要的等式：

$$T = M = N^{\frac{2}{3}} \tag{3.8}$$

由等式(3.8)可知，密码分析者在使用 Hellman 表来成功破解密钥 K 所需要的代价会比穷举攻击快 $N^{\frac{1}{3}}$ 倍，当然前提都是在基于选择明文的攻击方式下。由伪随机函数 S 的特性可以得到，只要稍加修改，比如将函数 R 的输入输出长度改变，我们就能对其他的分组密码算法进行密钥破解。所以，对任意的密钥空间为 N 的分组密码，利用 Hellman 算法都能获得以 $N^{\frac{2}{3}}$ 为代价的密钥破解成功。而针对其他体制的加密算法的密钥分析，例如以杂凑 Hash 函数为代表的 MD5 和 SHA-1 的加密算法，我们将在下面的彩虹表讨论分析。

3.1.4 密钥分析的成功率

在上一章2.2节密码分析方法的比较中，我们已经得出了相对穷举法百分之百的破解成功率来说，Hellman 的时空折中算法并非是 100% 能破解成功的，即使你付出了 $N^{\frac{2}{3}}$ 的代价。换句话说，也就是这类的时空折中算法是以牺牲少量的成功率为代价换取了在时间或空间上的代价，这也充分体现了折中的这个思想。因此，本节将会重点分析 Hellman 算法的成功率。其实造成功率损失的根本原因在与算法本身，由于简约函数 R 仅是从密文空间 V 到密钥空间 N 的这么一个映射，而对于两个不同的密文 C_1,C_2，会有一定概率映射到同一个密文，并造成 Hellman 表中的节点因发生了碰撞而不唯一，如图3.1所示，因此导致了最终的破解成功率降低。一般来说，当 m，t 值越大，节点间发生碰撞合并的概率也就越高。并且由于 Hellman 表的每个节点都使用相同的 R 函数，所以一旦表中任意两个节点发生碰撞时，这两个节点之后的所有节点也将会发生碰撞，最坏的情况会导致 50% 的效率下降。

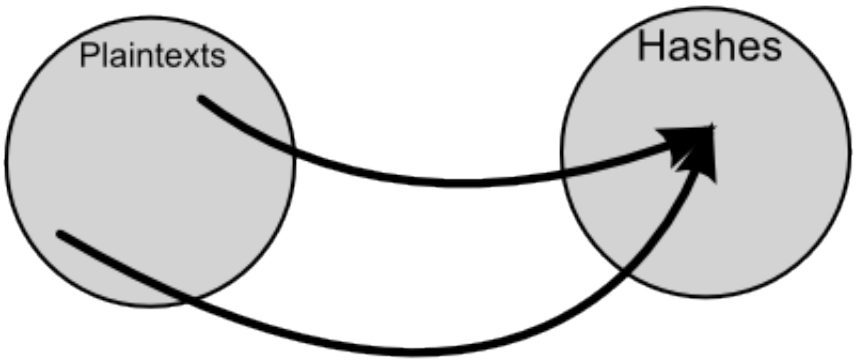

图 3.1 碰撞示意图

成功率是随着 m，t 的增大非线性地提升。在文献[1]中，我们可以得知一张 m 行，t 列的 Hellman 表能成功破解密钥 K 的概率公式为：

$$P_{Hellman} \geq \frac{1}{N}\sum_{i=1}^{m}\sum_{j=0}^{t-1}\left(1-\frac{it}{N}\right)^{j+1} \tag{3.9}$$

由于有节点的碰撞，单表的破解成功概率会随着表的大小增大而放缓增幅，所以为了得到更高的破解成功概率，我们一般都会使用多张表，例如(3.6)式中的 t 张表，并且为了避免表与表之间的链碰撞合并，不同的表当中应当使用不同的函数 $R_j(1 \leq j \leq t)$ 使得即使不同表间两个节点相同也难导致链表的合并。因此，我们也就很容易得到了 t 张 Hellman 表的总成功率公式：

$$\boxed{P_{Hellman}^{All} \geq 1-\left(1-\frac{1}{N}\sum_{i=1}^{m}\sum_{j=0}^{t-1}\left(1-\frac{it}{N}\right)^{j+1}\right)^{t}} \tag{3.10}$$

Hellman 称上述节点发生碰撞导致链合并的现象为 False Alarm，也就是出现了上文3.1.2提到的有 $y_k = EP_i$，但对应的 $k_{i,t-k}$ 不等于正确密钥。直观上分析，表的规模越大，越有概率发生假警，若假警率过高，则在线分析会花费大量的代价在剔除假警，为此 Hellman 给出了一个假警上界：

$$E(F) \leq \frac{mt(t+1)}{2N} \tag{3.11}$$

当假警发生时，最坏情况时导致 t 次函数 f 的迭代开销，也就是在一开始发生了假警，此时这个代价等同与在线分析时计算 $y_1, y_2 \ldots y_t$ 的代价。因此当

有 $mt^2 = N$，且 $N \gg 1$ 时，有 $E(F) \leq \frac{1}{2}$，则由于总的开销最多为 $\frac{t}{2}$ 次迭代，从而可以保持假警代价不超过 $\frac{t^2}{2}$，即 $\frac{T}{2}$。

3.2 Ronald Rivest 差异点 DP 方法

在随后的 1982 年，Ronald Rivest 提出了差异点（DP）的概念，通过使用差异点减少对磁盘的访问次数，有效地解决了碰撞的问题，从而降低破解的代价。差异点式满足一套标准的数据点。我们定义密钥的前 10 比特为一个特定的二进制，为了简便起见，这里设定为零。Rivest 提出只存储差异点作为终点，在破解密文时，简单地根据 Hellman 的方法生成链直到发现一个差异点，当且仅当发现一个差异点时，才在表中进行查找，这可以大大提升算法的性能。

差异点自提出后就被广泛地研究和使用，在 1982 年和 2003 年间这一领域的大多数研究都是基于差异点的。Koji Kusuda 和 Tsutomu Matsumoto[13]一文中具体讨论了如何提升破解的成功率，通过研究证明调整表中的参数可以降低内存的消耗，以获得更高的成功率，更快地破解密钥。另外，Johan Borst，Bart Preneel 和 Joos Vandewalle[14]共同发表的一篇文献中也对差异点进行了研究，他们介绍了一种分布式的密钥搜索算法，在基于 Hellman 的假设，认为存储访问代价时微不足道的，研究表明执行分布式密钥搜索时，这个假设不再合理，他们还提出一种可大大减少内存访问次数的折中算法，从而减少与分布式密钥搜索相关的问题。然而，他们的研究在 1998 年完成，这就意味着他们的工作与 2003 年 Oechslin 的工作毫无关联，分布式密钥搜索也与实际的表的生成无关，下面我们将重点介绍 Philippe Oechslin 在 2003 年改进后的算法，也就是目前著名的彩虹表算法。

3.3 本章小结

本章主要简要地介绍了 2 种典型的时空折中算法，Hellman 算法和差异点 DP 算法。时空折中算法主要包括预运算和在线分析两个步骤。Hellman 表和彩虹表具有不同的折中曲线，分别式 $TM^2 = N^2$ 和 $TM^2 = \frac{1}{2}N^2$。重点分析了算法的的成功率，公式的推导演算和证明。

第四章　彩虹表算法的实现

4.1　彩虹表的算法分析

4.1.1　非完美彩虹表

如3.1节所述，Hellman 表的一个主要不足是表大小的限制，当以增大表大小来获得更高得成功率时，表中的节点碰撞而导致的链合并概率也会随之增加，且增加的比例同 m 与 t 的平方成正比。为了解决这一不足，在 2003 年，Oechslin 在 Hellman 算法的基础上结合了 Rivest 的差异点（DP）的优势，提出了一种在当时比较先进的算法——彩虹表算法[2]。通过这种时空折中算法预运算所生成的表，我们称之为彩虹表（Rainbow Table），彩虹表中的行称为彩虹链。与 Hellman 算法在一张表中只使用一个 f 函数不同，彩虹表的每一列使用的 R 函数都不一样，以构造不同的 f 函数，如式(4.1)所示：

$$f_i(k) = R_i(S_k(P)) \qquad (1 \le i \le t) \tag{4.1}$$

从上式可以看出，彩虹表用 $R_1, R_2, \ldots, R_t$ 代替 Hellman 表中的 R 函数。因此，只有在当彩虹表中两个节点在同一列发生碰撞时，才会发生链合并。换句话说，如果碰撞发生在不同列的两个节点，由于不同列采用的不同的 R 函数，碰撞点之后的链不会被合并。我们假设一张彩虹表有 m 条彩虹链，每条彩虹链的长度为 t，即为 $m \times t$ 的矩阵，如式(4.2)：

$$\begin{bmatrix} k_{1,1} & k_{1,2} & \cdots & k_{1,t-1} & k_{1,t} \\ k_{2,1} & k_{2,2} & \cdots & k_{2,t-1} & k_{2,t} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ k_{m,1} & k_{m,2} & \cdots & k_{m,t-1} & k_{m,t} \end{bmatrix} \tag{4.2}$$

现在来计算这张彩虹表的破解成功率 P，这个问题实质上等价与整个矩阵每列的概率乘积。第一个元素 $k_{1,1}$ 命中的概率为 $\frac{1}{N}$, 那么这一列命中概率为 $P_1 = \frac{m_1}{N}$; 则第二列命中的概率为 $P_2 = 1 - (1 - \frac{1}{N})^{m_1}$，当 $N \gg m_1$ 时，

$P_2 = 1 - e^{-\frac{m_1}{N}}$，因此，可以得到第 i 列的命中概率公式：

$$P_i = 1 - e^{-\frac{m_{i-1}}{N}} = \frac{m_i}{N} \tag{4.3}$$

在概率定理可以推出整张彩虹表的成功率公式为：

$$P_{Rainbow} \geq 1 - \prod_{i=1}^{t}\left(1 - \frac{m_i}{N}\right)$$
$$\text{其中，} m_1 = m\text{，且} m_{n+1} = N\left(1 - e^{-\frac{m_n}{N}}\right) \tag{4.4}$$

比较(3.9)式和(4.4)式, 我们可以发现 t 张 $m \times t$Hellman 表与 1 张 $mt \times t$ 的彩虹表有大致相同的成功率。因为这两种算法所产生的表都覆盖了 mt^2 的密钥空间，同时也都包含了 t 个不同的 R 函数。在假警方面，两者也有类似的共同点，单张彩虹表的一列以及 t 张 Hellman 表，若包含 mt 个节点，则这些节点中的任一碰撞都将会导致链的合并。上述这样等规模的 Hellman 表和彩虹表的比较可以参考式(4.5)

$$\begin{bmatrix} k_{1,0}^1 \xrightarrow{f_1} k_{1,1}^1 \xrightarrow{f_1} \cdots \xrightarrow{f_1} k_{1,t}^1 \\ \vdots \\ k_{m,0}^1 \xrightarrow{f_1} k_{m,1}^1 \xrightarrow{f_1} \cdots \xrightarrow{f_1} k_{m,t}^1 \end{bmatrix} \tag{4.5}$$

$$\begin{bmatrix} k_{1,0}^t \xrightarrow{f_t} k_{1,1}^t \xrightarrow{f_t} \cdots \xrightarrow{f_t} k_{1,t}^t \\ \vdots \\ k_{m,0}^t \xrightarrow{f_t} k_{m,1}^t \xrightarrow{f_t} \cdots \xrightarrow{f_t} k_{m,t}^t \end{bmatrix} \tag{4.6}$$

$$\begin{bmatrix} k_{1,0}^1 \xrightarrow{f_1} k_{1,1}^1 \xrightarrow{f_2} \cdots \xrightarrow{f_t} k_{1,t}^1 \\ k_{2,0}^1 \xrightarrow{f_1} k_{1,1}^1 \xrightarrow{f_2} \cdots \xrightarrow{f_t} k_{2,t}^1 \\ \cdots \\ \\ \cdots \\ k_{mt-1,0}^1 \xrightarrow{f_1} k_{1,1}^1 \xrightarrow{f_2} \cdots \xrightarrow{f_t} k_{mt-1,t}^1 \\ k_{mt,0}^1 \xrightarrow{f_1} k_{1,1}^1 \xrightarrow{f_2} \cdots \xrightarrow{f_t} k_{mt,t}^1 \end{bmatrix} \tag{4.7}$$

在线分析过程中，彩虹表算法的搜索过程与 Hellman 算法有所不同，首先，将给定的密文 C 代入 R_t 的 y_1，并检索是否存在某个 $EP_i = y_1$，如果找到了匹配的 EP_i，这可以通过保存的对应 SP_i 来恢复 $k_{t,t-1}$。若找不到匹配的 EP，则将密文 C 依次代入 R_{t-1}，再次搜索判断是否存在正确的密钥在第 t-2 列中。以此类推，最坏的情况下将遍历表中所有的 t 列，以 f 函数迭代次数为标准，这个代价为 $1+2+\cdots+(t-1)=\frac{t(t-1)}{2}$ 次。而 Hellman 表需要搜索 t^2 次，因而单张的彩虹表相比 t 张 Hellman 表，搜索代价只有不到 Hellman 算法的一半。

在此小结一下彩虹表相对 Hellman 表的优点：

1），从搜索次数角度来看，彩虹表最多需要比较 $log(mt)*t$ 次，而 Hellman 表将最多需要 $log(t)*t^2$，即搜索 t 表 t 列。因此彩虹表的比较次数约为 Hellman 的 $\frac{1}{t}$，彩虹表相应的 TMTO 曲线为：

$$TM^2 = \frac{1}{2}N^2 \tag{4.8}$$

2），彩虹表中若发生彩虹链合并时，则会导致对应的 EP 点相同。当需要构造没有重复链的完美表（Perfect Table）时，可以通过检查 EPs 来完成。而 Hellman 表则无此特性。需要注意的是在完美表中并不是没有碰撞节点，而是碰撞节点不在同一列上出现。

3），由于彩虹链中的 t 个 R 函数均不相同，因此链中将几乎不会产生循环。以 Hellman 链为例，若同一链中出现了两个节点 k_i, k_j，使得 $k_i = k_j$，那么由于该链使用相同的 R 函数，故从这两个相同节点的后面节点都会相等，依次类推，最多会导致 j-i 个节点的重复，进而缩小了整个 Hellman 表的覆盖率。该现象不会产生假警，然而会降低成功率。而彩虹表则没有相应的问题。

4），与 Hellman 表的变形差异点（DP）算法相比，彩虹表的链条长度是固定的，这个特性对彩虹表的程序代码实现是十分有利的，同时也会使得假警率有所下降，从而提高了成功率。

4.1.2 完美彩虹表

当表中出现合并链时，彩虹表和差异点 DP 算法都可以通过检查是否具有相同的 EP 节点方式来检测出合并，而通常情况在预运算后生成的表需要排序的，这样一来就可以很容易地从表中剔除重复的链，而经过剔除重复链后的表我们称之为完美彩虹表。特别是在当内存空间有限时，我们总希望表能包含尽可能多的唯一节点，以提高成功率。生成相对应的 Hellman 完美表，则不得不搜索整张表，每个节点均要 O(mt) 次搜索，显然这是不现实的。在完美彩虹表中，有 $m_i = m_1 = m(2 \leq i \leq t)$，因此成功率可化简为：

$$P_{perfect} = 1 - \prod_{i=1}^{t} \left(1 - \frac{m}{N}\right) \tag{4.9}$$

从(4.9)式可知，成功率直接与完美彩虹表的初始参数 m，t 有关，也就是表越大，成功率 $P_{perfect}$ 应越大。同时与非完美彩虹表比较，不但减少了因存储重复节点而造成的空间浪费，而且节省了在线分析的时间代价。对于一个给定的 t，假设选择 $m_1 = N$ 时得到的 m_t为$m_{max}(t)$，表示为完美彩虹表最大的独立结束节点 EPs 的个数，其中 $m_{i+1} = N\left(1 - e^{-\frac{m_i}{N}}\right)(1 \leq i \leq t)$。

当 $t \gg 1$ 时，利用泰勒公式可以得到[15]：

$$m_{max}(t) \approx \frac{2N}{t+2} \tag{4.10}$$

将上式代入(4.9)式便可得出完美彩虹表成功率的最大期望值为：

$$P_{perfect}^{max} = 1 - \left(1 - \frac{m_{max}}{N}\right)^t \approx 1 - e^{-t\frac{m_{max}}{N}} \approx 1 - e^{-2} \approx 86\% \tag{4.11}$$

也就是说，对于 N 和较大的 t，完美彩虹表的成功率会随着 m，t 的增大而趋于一个常量。因此，单张的彩虹表的成功率要小于 80%，若要较高的成功率 P($P > 90\%$)，则可以通过多张彩虹表来实现。

4.2 彩虹表的构建与产生

4.2.1 定义 Hash 函数

本文的实现了对 SHA − 1、MD5 和 LM 三种 Hash 函数的破解，下面代码对 LM 算法的实现：

```
void HashLM(unsigned char* pPlain, int nPlainLen, unsigned char* pHash)
{
  int i;
  for (i = nPlainLen; i < 7; i++)
  pPlain[i] = 0;
  static unsigned char magic[] = {0x4B, 0x47, 0x53, 0x21, 0x40, 0x23,
                                                          0x24, 0x25};
  des_key_schedule ks;
  setup_des_key(pPlain, ks);
  des_ecb_encrypt((des_cblock*)magic,(des_cblock*)pHash,ks,DES_ENCRYPT);
}
```

SHA1 和 MD5 函数的实现则是调用了 OpenSSL 的 API 函数：

```
unsigned char *MD5(const unsigned char *d, size_t n, unsigned char *md);
```

```
unsigned char *SHA1(const unsigned char *d, size_t n, unsigned char *md);
```

4.2.2 定义 Reduce 函数

4.2.3 彩虹表参数

4.2.4 构建彩虹表

彩虹表算法实现的首要事情就是要产生符合自己需求的彩虹表文件，本次实现要求彩虹表的破解成功率在 99.9%以上。

彩虹表算法中有许多参数变量，在构造一张彩虹表前必须先弄明白每个参数的含义，在这里我们为了方便下文叙述，统一给出彩虹表算法中会出现的参数和变量：Hash 计算能力, 取决于 CPU 和 GPU 等硬件设备；硬盘读写速度, 取决于硬盘；密钥空间 N；

1. 每条彩虹链的长度 t；

2. 一张彩虹表中彩虹链的个数 m；

3. 彩虹表的个数 L；

4. 磁盘占用量 M　彩虹表所占用的存储空间；

5. 破解成功率 P　破解密钥的成功概率；

6. 破解时间；

7. 预运算时间；

由上述可知，一张彩虹表是由许多彩虹链组成的，每个彩虹链在程序中的数据结构如下：

```
struct RainbowChain
{
        unit64 nIndexS;         //链表的初始元素
        unit64 nIndexE;         //链表的末尾元素
}
```

函数主要输入参数：

```
        string sHashRoutineName  = argv[1];
        //需要破解的算法名称
        string sCharsetName      = argv[2];
        int nPlainLenMin         = atoi(argv[3]);
        //密码长度最小值
        int nPlainLenMax         = atoi(argv[4]);
        //密码长度最大值
        int nRainbowTableIndex   = atoi(argv[5]);
        int nRainbowChainLen     = atoi(argv[6]);
        int nRainbowChainCount   = atoi(argv[7]);
```

生成彩虹表的关键代码：

```
        int i;
        for (i = nDataLen / 16; i < nRainbowChainCount; i++)
        {
                cwc.GenerateRandomIndex();
                uint64 nIndex = cwc.GetIndex();
                int nPos;
```

```
7                 for (nPos = 0; nPos < nRainbowChainLen - 1; nPos++)
8                 {
9                         cwc.IndexToPlain();
10                        cwc.PlainToHash();
11                        cwc.HashToIndex(nPos);          //缩减函数
12                }
13                nIndex = cwc.GetIndex();
```

初始节点 nIndexS 由 cwc.GenerateRandomIndex() 函数随机产生，并把当前的 index 值也赋予 CChainWalkContext.m_nIndex 中，m_nindex 起到中间记录作用，当经过 nRainbowChainLen（参数 t）次循环计算得到 nIndexE。

彩虹表生成算法描述：

1. 将随机生成的 nIndexS 通过函数 cwc.IndexToPlain() 转化成明文 Plain，这一转化过程与二进制转 16 进制差不多，只是根据明文字符集长度（m_nPlainCharsetLen）来变化;

2. 对步骤 1 生成的 Plain 进行 Hash 函数计算，通过 PlainToHash() 函数实现；

3. 对步骤 2 生成的 Hash 值进行缩减运算，缩减函数为 HashToIndex(nPos)，最后得到的 nIndex 必须在预设的字符空间范围内。

将以上三步循环 nRainBowChainLen 次数后，我们将得到 nIndexE 和彩虹链的长度，当生成了 m 条彩虹链之后，这些彩虹链所组合起来的就是一张彩虹表文件。由于在循环过程中所产生的 index 只保存在内存里，并不写入磁盘文件，但是我们依然可以通过初始的 Index 计算出这过程中所有的 index 值。

$$\begin{bmatrix} Index_Start_1 & Index_End_1 \\ Index_Start_2 & Index_End_2 \\ \vdots & \vdots \\ Index_Start_m & Index_End_m \end{bmatrix} \tag{4.12}$$

从彩虹表的结构(4.12)我们可以很容易得知一个彩虹链的所占的磁盘空间为

$2*64=128$ 比特，也就是 16Byte。由此得到彩虹表磁盘空间占用公式：

$$M=16*m*l \tag{4.13}$$

其中 m 为彩虹链的条数，l 为彩虹表的个数。图4.1为彩虹表生成程序，hash_algorithm 为目标密码 hash 算法；charset 为密钥的字符集，决定这张彩虹表的密钥空间；还有链表长度和链表个数等参数。 从图4.2中我们可以

```
[/home/kaka/Downloads/rainbowcrack-1.5-linux32] [ 11 files 352Kb ]
[12:46:48 root] rainbowcrack-1.5-linux32 # ./rtgen
RainbowCrack 1.5
Copyright 2003-2010 RainbowCrack Project. All rights reserved.
Official Website: http://project-rainbowcrack.com/

usage: rtgen hash_algorithm charset plaintext_len_min plaintext_len_max table_index chain_len chain_num part_index
       rtgen hash_algorithm charset plaintext_len_min plaintext_len_max table_index -bench

hash algorithms implemented in alglib0.so:
    lm, plaintext_len limit: 0 - 7
    ntlm, plaintext_len limit: 0 - 15
    md5, plaintext_len limit: 0 - 15
    sha1, plaintext_len limit: 0 - 20
    mysqlsha1, plaintext_len limit: 0 - 20
    halflmchall, plaintext_len limit: 0 - 7
    ntlmchall, plaintext_len limit: 0 - 15
    oracle-SYSTEM, plaintext_len limit: 0 - 10
    md5-half, plaintext_len limit: 0 - 15

example: rtgen md5 loweralpha 1 7 0 1000 1000 0
         rtgen md5 loweralpha 1 7 0 -bench
```

图 4.1　彩虹表生成程序

```
[/media/kaka/ntlm_numeric#1-12] [ 20 files 8.8Gb ]
[12:38:02 root] ntlm_numeric#1-12 # ll
total 9175156
-rw-r--r--. 1 kaka kaka 469762080 May 26 10:51 ntlm_numeric#1-12_0_8300x67108864_0.rtc
-rw-r--r--. 1 kaka kaka 469762080 May 26 10:51 ntlm_numeric#1-12_0_8300x67108864_1.rtc
-rw-r--r--. 1 kaka kaka 469762080 May 26 10:52 ntlm_numeric#1-12_0_8300x67108864_2.rtc
-rw-r--r--. 1 kaka kaka 469762080 May 26 10:52 ntlm_numeric#1-12_0_8300x67108864_3.rtc
-rw-r--r--. 1 kaka kaka 469762080 May 26 17:44 ntlm_numeric#1-12_1_8300x67108864_0.rtc
-rw-r--r--. 1 kaka kaka 469762080 May 26 17:44 ntlm_numeric#1-12_1_8300x67108864_1.rtc
-rw-r--r--. 1 kaka kaka 469762080 May 26 17:45 ntlm_numeric#1-12_1_8300x67108864_2.rtc
-rw-r--r--. 1 kaka kaka 469762080 Jun  2 20:33 ntlm_numeric#1-12_1_8300x67108864_3.rtc
-rw-r--r--. 1 kaka kaka 469762080 Jun  2 20:33 ntlm_numeric#1-12_2_8300x67108864_0.rtc
-rw-r--r--. 1 kaka kaka 469762080 Jun  2 20:34 ntlm_numeric#1-12_2_8300x67108864_1.rtc
-rw-r--r--. 1 kaka kaka 469762080 May 25 21:25 ntlm_numeric#1-12_2_8300x67108864_2.rtc
-rw-r--r--. 1 kaka kaka 469762080 May 26 18:05 ntlm_numeric#1-12_2_8300x67108864_3.rtc
-rw-r--r--. 1 kaka kaka 469762080 May 26 18:05 ntlm_numeric#1-12_3_8300x67108864_0.rtc
-rw-r--r--. 1 kaka kaka 469762080 May 26 18:05 ntlm_numeric#1-12_3_8300x67108864_1.rtc
-rw-r--r--. 1 kaka kaka 469762080 May 26 18:06 ntlm_numeric#1-12_3_8300x67108864_2.rtc
-rw-r--r--. 1 kaka kaka 469762080 May 26 18:06 ntlm_numeric#1-12_3_8300x67108864_3.rtc
-rw-r--r--. 1 kaka kaka 469762080 May 26 18:07 ntlm_numeric#1-12_4_8300x67108864_0.rtc
-rw-r--r--. 1 kaka kaka 469762080 Jun  2 20:34 ntlm_numeric#1-12_4_8300x67108864_1.rtc
-rw-r--r--. 1 kaka kaka 469762080 Jun  2 20:34 ntlm_numeric#1-12_4_8300x67108864_2.rtc
-rw-r--r--. 1 kaka kaka 469762080 May 26 17:44 ntlm_numeric#1-12_4_8300x67108864_3.rtc
```

图 4.2　ntlm 算法彩虹表

看到 20 张彩虹表，hash_algorithm 为 ntlm 算法，ntlm 为 Windows NT 系统的用户登陆验证算法；密钥字符集为 numeric，也就是 0 ～ 9，最小长度为 1，最大长度为 12，因此这个密钥空间 $N=10^{12}+10^{11}+\cdots+10^{2}+10$，

其他的密钥并不在这 20 张表里，一定不会被搜索到，想要破解需要加大链的数目，或者链表长度，或者表的张数；链表长度为 8300，链表个数为 67108864，这样通过公式(4.13)我们很容易到这 20 张表的所占用的磁盘空间 $M = 20GB$，每张表的大小为 1GB。

4.3 密钥破解

4.3.1 搜索算法

用彩虹表进行破解的过程其实质就是对彩虹表文件进行整表搜索的过程，简单来讲就是查表。破解的成功率和破解的时间基本上由预运算所生成的彩虹表决定，生成优质的彩虹表对破解的效率影响很大。具体的优化方法和技巧我们将在下文详细介绍，这一节我们要讨论的是彩虹表如果进行查表破解。首先我们看图4.3:

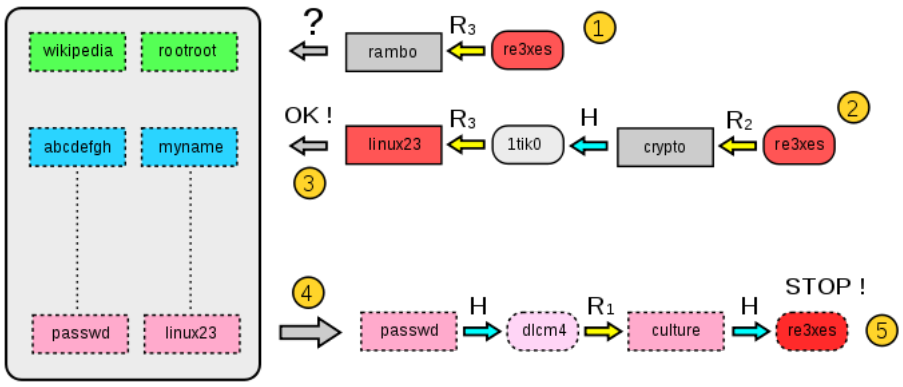

图 4.3 彩虹表查表过程

从图中我们看到需要破解的密钥为“re3xes”，第一步，将“re3xes”代入 R 函数，得到索引“rambo”，将此索引从每个彩虹链的末端开始对比；第二步，若不匹配，则通过算法中的 f 函数(??)进行遍历整条彩虹链，若在这条链表中没找到匹配的，则往后遍历表中的其他彩虹链；第三步，当在链中找到匹配的索引后，记录该链表的首部索引；第四步，通过第三步找到的索引，进行 f 函数计算得到产生目标密钥“re3xes”的明文“passwd”，并验证其正确性。

4.3.2 假警分析

4.4 实验结构与分析

4.5 本章小结

本章主要介绍了彩虹表算法的 C++ 实现，主要包括彩虹表的预运算，即彩虹表的生成和利用彩虹表进行密钥破解。关键的程序代码可以参考附件。

第五章　彩虹表算法的优化

5.1　基于 CUDA 并行计算优化

密码破解主要的瓶颈在于现有的计算能力上，假如我们目前已经拥有量子计算机的处理能力，那么现有的所有的现代加密算法都是可以短时间内被破解的，所以我们要提升破解时间，最首要的办法就是对系统的计算能力进行优化，本文将提出两个优化方案，并实现了基于 GPU 的优化方案。下面两张图为 EWSA（Elcomsoft Wireless Security Auditor）对无线加密算法 WPA/WPA2 PSK 密钥破解的数据对比和 Pyrit 软件在 GPU 上加速后的数据对比。从图中数据可以看出采用了 GPU 加速后的破解速度有很明显提升。

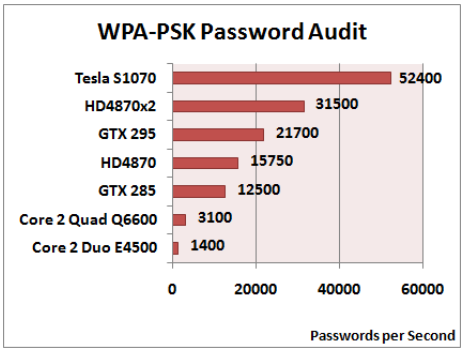

图 5.1　EWSA 基于 GPU 加速破解 WPA 速度对比

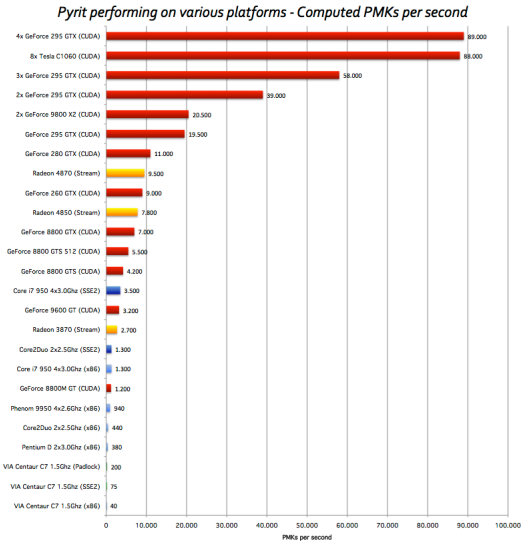

图 5.2　Pyrit 基于 CUDA 性能测试

基于 GPU 并行计算加快破解速度会受制与显卡本身的硬件条件限制，因为每个不同型号的显卡配备的显存、寄存器、线程调度数等等都是不一样的。如何在这些局限的硬件资源上合理分配，则是 GPU 程序设计的关键，也是性能提升的关键。例如如何控制在同一个线程组里的线程分支数、访问全局内存的顺序都是一些常用的优化代码的方法。在实际代码设计过程中，我们首先需要将计算任务并行化，把这些并行任务分配到线程上，再由这些

线程组成线程块，这些线程块将会被线程管理器动态地分配到各个流处理器进行独立的并行计算。下面我们将从 GPU 的硬件特性和软件特性两方面进行优化，例如减小数据传输的开销、解决共享内存的访问冲突和降低条件分支的影响等等[17]。Nvidia GPU 不同构架的一些规格比较，见表5.1

表 5.1 三代 GPU 间的一些规格比较

GPU	G80	GT200	Fermi
CUDA Cores	128	240	512
双浮点数计算能力	无	30FMA	256FMA
单浮点数计算能力	128MAD	240MAD	512FMA
特殊功能单元（SFUs）/SM	2	2	4
Warp 调度器/SM	1	1	2
共享内存/SM	16KB	16KB	可配置 48KB/16KB
L1 缓存/SM	无	无	可配置 16KB/48KB
L2 缓存	无	无	768KB
ECC 内存交验	不支持	不支持	支持
并发内核数	无	无	最多 16
地址位宽	32b-bit	32-bit	64-bit

5.1.1 减小数据传输开销

在目前的计算机体系结构中，像显卡这样的外设一般都是通过 PCI — Express 总线连接到北桥芯片，再通往 CPU，这和主机内存共享与 CPU 传输带宽。当 CUP 与 GPU 的数据频繁交换时，这个数据传输的开销将会是程序性能主要瓶颈。因此我们的程序必须尽量地减少 CPU 与 GPU 的数据交换，通常的办法有在 GPU 中动态生成程序所需的数据，也可以采用异步调用，使数据传输和计算能基本同时进行。

5.1.2 共享内存访问冲突

在表5.1中我们可以看到每个 SM 都会拥有 16KB 的以上的共享内存，这块共享内存的作用是存储全局内存以外的数据，也可以作为块与块之间线程交换数据的媒介。它的访问速度比寄存器慢，但要比局部内存和全局内要快一些。如果不出现访问冲突，一个线程访问共享内存的速度几乎接近访问寄存器的速度。图5.3Fermi 构架的内存体系结构，除了拥有可配置的共享内存外，还有 L1 和 L2 的高速缓存。

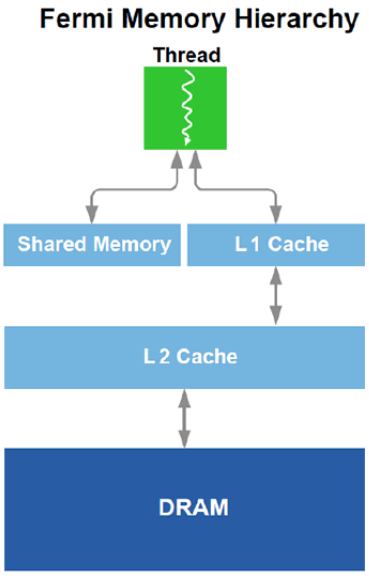

图 5.3 Fermi 构架的存储层次结构

5.1.3 核心配置优化

核心程序的配置如块数、块内线程数都依赖核心代码本身，但仍然需要手工评估性能，排除大部分低效的配置。配置好块数和块内线程后，计算 SM 利用率，便可得到 SM 上活动 Warp 和最大 Warp 的比，通常我们也无需达到 100%。

5.1.4 指令级别优化

这项优化工作一般是有 GPU 的编译器来作的，在 CUDA 中是有 nvcc 编译器来完成的。尽量使用低延迟的指令，对全局变量或寄存器以外的局部变量，尽量少用需要多次 Load 和 Store 的操作，需要减少循环操作的开销。

5.1.5 存储合并访问

从文献[18]可知，SM 会以时间片轮询的方式逐个调度每个 Warp。各个 SM 一次只需要足够的资源以运行运行 32 个线程（通常块内线程数要远大于 32），其中的 8 个 SP 将分 4 个及其周期的时间偏离所有的线程（我们假设此处的指令为大多数可在一个周期内完成简单指令）。当 SM 执行一条访存指令时，它首先会发出一条访存请求，并随机转向下一个就绪 Warp，而当前 Warp 将被挂起直到数据访问完成，才能再次进入就绪态并被等待属于它的时间片。理想的情况下，该 Warp 内的访存请求没有冲突，则该请求可在一个访存事务内完成。然而，事实上是否能在一个事务中结束访问，是严重依赖于 Warp 内每个线程所访问存储的地址是否可合并这个事实的（Coalesced MemoryAccesses）。简单地讲，如果每个线程的访存地址是连续的，则所有 Warp 内的请求可以合并为一个访存事务，否则将可能产生多个事务并导致该 Warp 需要更多的等待时间[19]。

5.1.6 基于 CUDA 的云计算平台

云计算是网格计算、分布式计算、并行计算、效用计算、网络存储、虚拟化、负载均衡等传统计算机和网络技术发展融合的产物。在[4]一文中使用了 Hadoop 分布式集群优化彩虹表算法，实际上 Hadoop 也可以称之为一种云平台。我们也可以设计一个基于 GPU 加速的云计算平台，通过云平台的计算能力提升彩虹表的预运算和破解的速度。亚马逊 EC2 就是这样的一个平台，它可以让用户按照自己的硬件需求租用计算机来获取计算能力。

5.2 存储优化

对于时空折中算法而言，除了要对时间进行优化，还有对空间进行优化，在不增加（或增加少量）时间代价的前提，如何减小空间代价，这就需要我们对系统进行存储优化。我们将从两方面进行改进。第一，针对存储的文件优化，也就是彩虹表文件，我们将设计一个新型的彩虹表存储结构体，减少彩虹表的磁盘存储空间，缩小系统读取文件的时间，从而提升破解的速度；第二，优化存储系统，如采用适合大文件的文件系统，采用快速的物理

存储设备等方面提升读取速度。

从上一章彩虹表磁盘空间占用公式(4.13)可知，想要覆盖更大的密钥空间要么增加彩虹链数或者彩虹表的张数，这都回产生庞大的彩虹表文件，可能会达到几百个 GB，甚至上 TB 的文件，这将会增加大量的文件读取时间和磁盘空间，因此对表的存储结构进行优化将十分有必要。我们将新设计的彩虹链，在以前的存储结构中我们是使用了一个 64 比特的非负整形变量定义开始节点，而在实际当中，我们并不需要这么大的节点，比如在第四章中的 ntlm 的彩虹表中，我们只定义了 26 比特的开始节点和 30 比特的末端节点，这样一条彩虹链所占用的字节数为 7Bytes。代入公式(4.13)可以得到一张优化后的表的大小为 448MB，优化了 56.25%。这样大大减小系统的存储空间和读取文件的时间。

```
[/media/kaka] [ 4 files 40Kb ]
[16:18:45 root] kaka # du -h ntlm_numeric#1-12/
8.8G    ntlm_numeric#1-12/
```

图 5.4 优化后彩虹表的空间大小

新的彩虹链的数据结构实现如下，在以后的测试实验中，我们将都采用这种优化过的彩虹表。

```
1 struct RTCFileHeader
2 {
3     unsigned int    uVersion;
4     unsigned short  uIndexSBits;
5     unsigned short  uIndexEBits;
6     uint64          uIndexSMin;
7     uint64          uIndexEMin;
8     uint64          uIndexEInterval;
9 };
```

在存储上，除了对表的大小进行优化，还可以对系统的储存架构进行优化。在文件系统上，我们采用先进的 XFS 文件系统，XFS 是由 Silico Graphics,Inc. 于 90 年代初开发的，它采用了优化算法，对查询分配存储空间非常快，可以支持上百万 T 字节的存储空间，特别是对大文件的支持表现相当出众；XFS 采用 B+ 树结构保证文件系统可以快速搜索于空间分配；XFS 几乎以接近裸设备 I/O 的性能存储数据，在单个文件系统测试种，其

吞吐量可高达 7GB 每秒，对单个文件的读写操作，其吞吐量可达 4GB 每秒。基于 XFS 以上特性，正符合彩虹表多个大文件读取的特点，下图为 XFS 与 EXT3、EXT4 的性能比较：

```
=====================================================================================================
Tue 01 Nov 2011 07:30:10 AM EDT

 KERNEL                                    xfs
 ----------                                ====
1 - 2.6.32-131.17.1.el6.x86_64
2 - 2.6.32-214.el6.x86_64

                      ALL  INIT  RE            RE   RANDOM RANDOM BACKWD  RECRE STRIDE  F     FRE     F     FRE
 TEST        KERNEL   IOS  WRITE WRITE  READ   READ  READ  WRITE  READ  WRITE  READ  WRITE WRITE  READ  READ
---------------------------------------------------------------------------------------------------------------
InCache        1     2099   964  1650  2870   2990  2783  1396  2592   3186  2073   905  1547  2850  3067
InCache        2     2082   968  1569  2955   3094  2830  1335  2638   2999  2928   886  1428  2860  3147
                       .     .     .     .      .     .     .     .   -5.9     .     .  -7.7     .     .


DirectIO       1      199   166   177   258    261   192   157   218    177   211
DirectIO       2      199   168   176   256    256   196   155   220    179   210
                       .     .     .     .      .     .     .     .      .     .


OutOfCache     1      452   564   565   700    712   109    63   342   4532   169   445   543   681   694
OutOfCache     2      450   567   567   691    686   108    66   339   4307   163   441   548   700   700
                       .     .     .     .      .     .     .     .      .     .     .     .     .     .


InCacheMMAP    1     1723   789  1562  2083   2344  2227  1260  2152   1503  2447
InCacheMMAP    2     1742   806  1593  2116   2350  2188  1283  2180   1513  2494
                       .     .     .     .      .     .     .     .      .     .


InCacheFsync   1      692   163   172  1162   2927  2636   132  1094    891  1092   154   175  2773  3128
InCacheFsync   2      694   166   176  1157   3025  2657   130  1150    834  1086   152   178  2746  3110
                       .     .     .     .      .     .     .  +5.1   -6.4     .     .     .     .     .
```

图 5.5 XFS 文件系统性能测试

```
=====================================================================================================
Tue 01 Nov 2011 07:28:31 AM EDT

 KERNEL                                    ext4
 ----------                                ====
1 - 2.6.32-131.17.1.el6.x86_64
2 - 2.6.32-214.el6.x86_64

                      ALL  INIT  RE            RE   RANDOM RANDOM BACKWD  RECRE STRIDE  F     FRE     F     FRE
 TEST        KERNEL   IOS  WRITE WRITE  READ   READ  READ  WRITE  READ  WRITE  READ  WRITE WRITE  READ  READ
---------------------------------------------------------------------------------------------------------------
InCache        1     1988   841  1429  3089   3293  2967  1199  2614   2554  2816   765  1319  2769  3322
InCache        2     1982   859  1438  3105   3294  2894  1184  2525   2568  2739   791  1333  2758  3266
                       .     .     .     .      .     .     .     .      .     .     .     .     .     .


DirectIO       1      204   170   182   255    260   198   167   221    188   213
DirectIO       2      199   161   175   254    258   196   161   220    181   210
                       .  -5.2     .     .      .     .     .     .      .     .


OutOfCache     1      453   540   566   706    706   111    75   348   3527   168   451   564   706   705
OutOfCache     2      456   549   570   707    709   111    74   351   3576   173   454   567   714   714
                       .     .     .     .      .     .     .     .      .     .     .     .     .     .


InCacheMMAP    1     1723   851  1508  2164   2341  2188  1155  2247   1443  2512
InCacheMMAP    2     1731   852  1514  2194   2362  2207  1160  2235   1451  2512
                       .     .     .     .      .     .     .     .      .     .


InCacheFsync   1      611   140   144   867   3679  2835   127   830    701   857   132   142  2713  3311
InCacheFsync   2      636   152   156   936   3180  2803   129   879    720   900   140   151  2716  3323
                       .  +8.5  +7.8  +7.9      .     .     .  +5.9      .  +5.1  +5.7  +6.4     .     .
```

图 5.6 EXT4 文件系统行性能测试

从这上面两组性能测试数据我们可以看出 XFS 在大多数的的选项上要优于 EXT4 文件系统，特别是在大文件的读取上。

接着是对存储硬件升级，这里我们主要升级的是硬盘，下表5.2是我们对不同配置硬盘的读速度进行数据对比，从表中可以明显看出使用 SSD 硬盘 RAID0 阵列后，读取速度将比一块 7200 转的硬盘提升了 10 倍，这将大大缩小破解的实际时间。最后的需要升级的系统内存，我们知道在计算机存储体系结构中，内存的读取速度要比硬盘快上许多倍，在 Linux 系统上内存以/dev/shm/设备呈现给用户，用户可以对次进行读写操作。

往后我们还可以采用 iSCSI 网络存储结构，iSCSI 技术一种优 IBM 公司研究开发的，是一种新存储技术，可是实现在 IP 网络上运行 SCSI 协议，使其能在高速千兆或万兆以太网上进行网络传输。再加上 FCoE 技术，可以达到几 Gb/s 的存储速度。

表 5.2 各种存储系统读速度对比数据

硬盘配置	读速度 (hdparm -t)	索引速度
1x Maxtor 6B250S0 7200 RPM	54MB/s	640h/s
6x Seagate 7200 RPM (RAID6)	390MB/s	4375h/s
2x APPLE SSD SM128C (RAID0)	522MB/s	6120h/s

5.3 算法结构优化

密钥明文字符集被预计算成 hash 密文并和明文成对地储存在彩虹链中，如果要破解的 hash 密文在我们之前生成好的彩虹链中，则破解成功，反之破解失败。彩虹链存储的“明文 -密文”对数量随着链表长度（t）的增加而增加，然而这些彩虹链会产生冲突，主要由于减约函数并不是一一对应的。图5.7示意了冲突的过程，最后这两条链表会合并成一条。

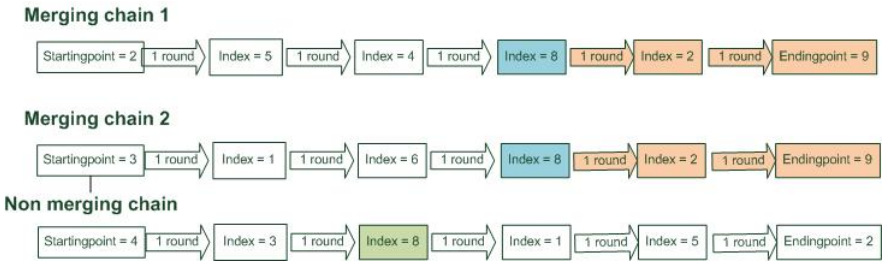

图 5.7 链表合并过程示意图

无冲突的彩虹链个数会随着彩虹表的长度增加而减少，根据公式(4.9)，利用 matlab 软件我们可以得到关系图5.8。

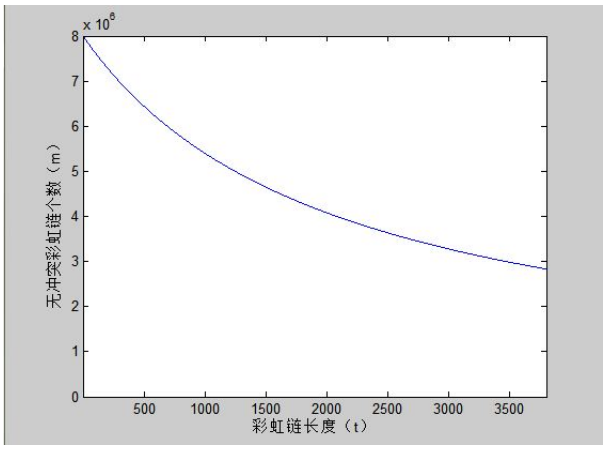

图 5.8 无冲突链表数－链表长度关系

接着我们来分析表的张数（参数 l）对破解成功概率的影响，我们把破解成功率公式(4.9)在 matlab 下的实现函数为：demo_advantage_of_multiple_table()，具体的函数实现可以参考附录A1.2。我们用不同的曲线表示当硬盘空间增大的情况下，破解的成功概率会随之增加，下图中有 5 条曲线，分别表示彩虹表张数为 15 时，破解的成功率和硬盘空间之间的关系，放大后，我们可以看出彩虹表的张数越

多，曲线越快接近 100%，也就是在其他参数不变的情况下，增加参数 l（彩虹表张数），可以得到越高破解成功概率；但也不是生成的彩虹表张数越多越好，当随着彩虹表的增加，我们需要在存储空间也就越大，这样会破解实际所消耗的时间就会增加，反而适得其反，所以时空折中算法的精髓就在于对时间和空间的代价进行不断地平衡，找出一个折中的代价。

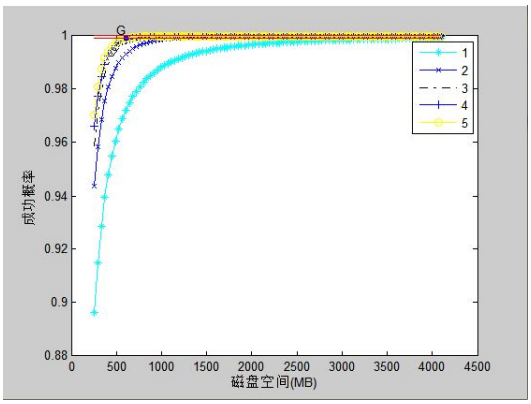

图 5.9 5 张彩虹表的破解成功率

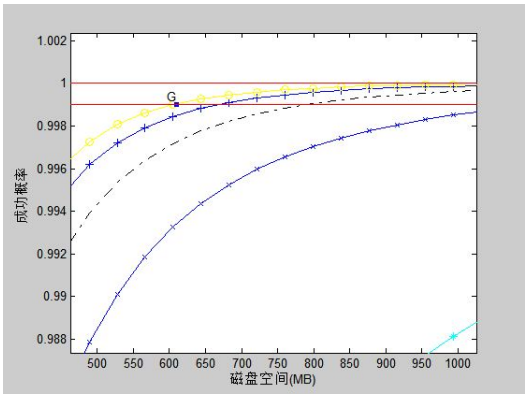

图 5.10 5 张彩虹表的破解成功率（放大）

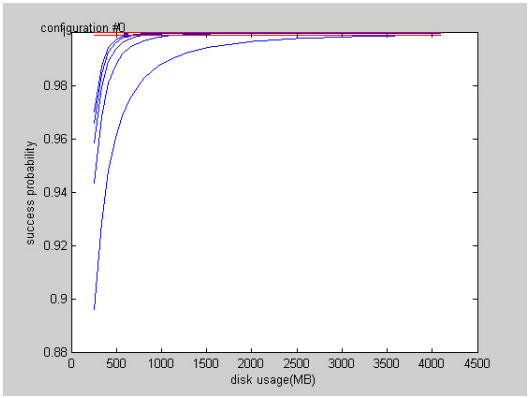

图 5.11 总表空间大小－成功率关系图

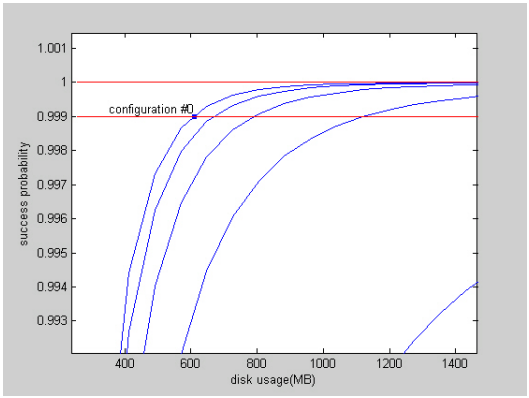

图 5.12 总表空间大小－成功率关系图（放大）

5.4 实验结果

实验所用的测试平台见下表5.3：

表 5.3 实验数据测试平台配置

CPU	Intel Core i7-920 OC 4.0GHz (EIST/C1E off)
主板	Asustek Rampage II GeneIntel X58 芯片
显卡	GeForce GTX 480 1536MB(Core/Shader/memory:700/1401/1848MHz)
内存	GSKILL F3-12800CL9T OC DDR3-1600 2G*3
硬盘	2x Intel 120G SSD SM128C (RAID0)
电源	CougarGX 900W

实验方法是通过采用上述优化方法和技术，对 SHA-1 、MD5 和 NTLM 三种 Hash 加密算法进行实验，我们规定密钥的字符集范围（字符集对照表参见附录C），密钥长度，限定成功率在 99.9% 以上，主要从破解时间方面针对优化前和优化后的实验数据进行对比。

在这里需要说明的是破解的时间主要分为对彩虹表文件的读取时间和密钥查表破解的时间。对彩虹表文件的读取时间主要优化方法和技术参见本章的存储优化一节，上文已经给出了基本的实验对比数据，就不在这里赘述，我们只对比最后的实际破解时间。所有实验只针对 1 条 Hash 密文进行破解，每组彩虹表破解 3 次。

SHA1 破解实验，破解速度提升了 5.88 倍，详细实验数据参考下表5.4：

表 5.4 SHA1 破解实验数据对比

字符集	密钥长度	成功率	优化前破解时间 (秒)	优化后破解时间 (秒)
numeric	1 ～ 12	99.9%	27.54	4.78
numeric	1 ～ 12	99.9%	23.32	4.04
numeric	1 ～ 12	99.9%	36.21	5.98

表 5.4 SHA1 破解实验数据对比（续）

接上页

字符集	密钥长度	成功率	优化前破解时间 (秒)	优化后破解时间 (秒)
loweralpha	1 ～ 9	99.9%	21.08	6.76
loweralpha	1 ～ 9	99.9%	19.65	5.88
loweralpha	1 ～ 9	99.9%	30.18	8.90

MD5 破解实验，破解速度提升了 6.3 倍，详细实验数据参考下表5.5：

表 5.5 MD5 破解实验数据对比

字符集	密钥长度	成功率	优化前破解时间 (秒)	优化后破解时间 (秒)
ascii-32-95	1 ～ 7	99.9%	372.33	69.47
ascii-32-95	1 ～ 7	99.9%	487.26	54.90
ascii-32-95	1 ～ 7	99.9%	432.98	96.22

NTLM 破解实验，破解速度提升了 1.77 倍，详细实验数据参考下表5.6：

表 5.6 NTLM 破解实验数据对比

字符集	密钥长度	成功率	优化前破解时间 (秒)	优化后破解时间 (秒)
numeric	1 ～ 12	99.9%	8.32	7.43
numeric	1 ～ 12	99.9%	10.76	6.90
numeric	1 ～ 12	99.9%	5.45	2.08

5.5 本章小结

本章对从计算能力到存储和彩虹表算法结构三方面进行优化和实现。在计算能力上我们提出了基于 CUDA 优化方法，并实现了 CUDA GPU 方案，从实际实验结果验证了这一设计方案；在存储系统上，我们提出了采用新型的彩虹表结构体，减少了彩虹表的磁盘存储空间，优化升级了物理的存储设备，在破解时间上得到了很好的提升；在彩虹表算法参数上，我们进行数据分析，优化了算法的参数。

第六章　本文总结与展望

6.1　总结

我们通过对经典的彩虹表算法进行详细的理论分析研究并给出了完整的实现，通过 GPU 并行计算加速、存储优化和对算法结构的调整三大方面对彩虹表算法进行优化，使得破解速度相对经典算法提升了 5.88 ～ 6.3 倍，在相同参数下表的磁盘储存上缩小了 56.25%。

6.2　展望

随着硬件设备的进一步发展，特别是 GPU 构架的更新换代、大容量高速磁盘和主存、128 位操作系统的实现，基于 GPU 的时空折中算法将可以对更大密钥空间的密码算法进行破解。

参考文献

[1] Martin Hellman. A cryptanalytic time-memory tradeoff. *IEEE Transactions on Information Theory*, vol.26(401-406), 1980. 1, 18, 22

[2] P.Oechslin. Making a faster crytanalytic time-memory trade-off. *Lecture Notes in Computer Science*, vol.2729, 2003. 1, 24

[3] ZhuShuangLei. The Time-Memory Tradeoff Hash Cracker, 2003. http://project-rainbowcrack.com. 2

[4] 方海英. 基于时空折中算法的 word 文档破解研究. Master's thesis, 杭州电子科技大学, 2009. 2, 37

[5] 金銓. DES 密码算法的彩虹攻击技术及其 GPU 实现. Master's thesis, 上海交通大学, 2010. 2

[6] Nvidia CUDA. http://developer.nvidia.com/object/cuda.html. 2, 12

[7] 冯登国. *密码分析学*. 清华大学出版社, 2000-08. 4

[8] 冯登国，裴定一. *密码学导引*. 科学出版社, 北京, 1999. 5

[9] 春天. 单向 hash 函数, 06 2005. http://blog.csdn.net/sunrisefe/archive/2005/10/12/500819.aspx. 9

[10] 舒畅. Md5 算法原理及其碰撞攻击. *软件导刊*, pages 103–104, 06 2007. 9

[11] Victor W. Lee. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *ACM SIGARCH Computer Architecture News*, 38, 2010. 11

[12] J.Nickolls and I. Buck. NVIDIA CUDA software and GPU parallel computing architecture. *Micorprocessor Forum*, May 2007. 12

[13] Koji Kusuda and Tsutomu Matsumoto. Achieving higher success probability in time–memory trade-off cryptanalysis without increasing memory

size. *TIEICE: IEICE Transactions on Communications/Electronics/ Information and Systems*, 1999. 23

[14] Johan Borst，Bart Preneel，Joos Vandewall. On the Time-Memory Tradeoff Between Exhaustive Key Search and table pre-computation. *Proceedings of the 19th symposium on Information Theory in the Benelux,Veldhoven(NL)*, pages 111–118, 1998. 23

[15] A. Biryukov and A. Shamir. Cryptanalysis time/memory/data tradeoffs for stream ciphers. Proceedings of Asiacrypt’ 00(T. Okamoto, ed.)no 1976 in Lecture Notes in Computer Science, pp. 1-13,Springer-Verlag,2000. 27

[16] Michael S. Distributed Pre-computation for a Cryptanalytic Time-Memory Trade-Off.
http://ritdml.rit.edu/dspace/bitstream/1850/7805/1/MTaberThesis10-2008.pdf.

[17] S. Che，M. Boyer，J. Meng，D. Tarjan，J. W. Sheaffer and K. Skadron. A Performance Study of General Purpose Applications on Graphics Processors using CUDA. *Journal of Parallel and Distributed Computing*, 2008. 35

[18] Hyesoon Kim Sunpyo Hong. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *Proceedings of the 36th annual international symposium on Computer architecture*, June 2009. 37

[19] NVIDIA Corporation. NVIDIA CUDA Programming Guide, 2011. 37

附录 A　MATLAB 程序

A.1　破解成功率

```
\% 1 - (1 - 1 / N)^(m(1) + m(2) + m(3) + ... + m(t - 1))
\% m(1) = m, m(i) = N * (1 - (1 - 1 / N) ^ m(i - 1))
function ret = calc_success_probability(n, min, max, t, m, table_count)
format long g;
arr = zeros(1, t - 1);
arr(1) = m;
format long g;
N = calc_N(n,min,max);
for i = 2 : t - 1
    format long g;
        arr(i) = N * (1 - (1 - 1 / N) ^ arr(i - 1));
end;
exp = 0;
for i = 1 : t - 1
    format long g;
        exp = exp + arr(i);
end;
format long g;
success_probability_1 = 1 - (1 - 1 / N) ^ exp;
success_probability_table_count=1-(1-success_probability_1)^table_count;
ret = success_probability_table_count;

function ret = calc_N( n, min, max)
arr = zeros(1, max - 1);
for i = 1 : max
        arr(i) = n^i;
end;
exp = 0;
for i = 1 : max
        exp = exp + arr(i);
end;
ret = exp;
```

A.2 磁盘空间与成功率

```
function advantage_of_multiple_table(N, t, disk_usage_min, disk_usage_max)
grid_count = 100;
disk_usage = linspace(disk_usage_min, disk_usage_max, grid_count);
m = disk_usage * 1024 * 1024 / 16;
success_probability = zeros(1, grid_count);

for table_count = 1 : 5
  for i = 1 : grid_count
    success_probability(i)=calc_success_probability(N,t,m(i)/table_count);
    success_probability(i)=1-(1-success_probability(i))^table_count;
  end;
  if table_count ==1
    plot(disk_usage, success_probability, '-c*');
  end
  if table_count ==2
    plot(disk_usage,success_probability, '-x')
   end
  if table_count == 3
    plot(disk_usage,success_probability, 'k-.')
  end
  if table_count == 4
    plot(disk_usage,success_probability, '-+')
  end
  if table_count == 5
    plot(disk_usage,success_probability, 'y-o')
  end
  if N == 8353082582 & t == 2100 & table_count == 5
    plot(8000000*16*5/1024/1024, 0.999, 's', 'MarkerFaceColor', 'b',
                              'MarkerEdgeColor', 'b', 'MarkerSize',3);
    text(8000000*16*5/1024/1024, 0.999, 'G', 'HorizontalAlignment', 'right',
                                          'VerticalAlignment', 'bottom');
  end;
  hold on;
end;
legend_int = zeros(5 - 1 + 1, 1);
for table_count = 1 : 5
        legend_int(table_count - 1 + 1, 1) = table_count;
```

```
end;
legend_str = int2str(legend_int);
legend(legend_str);
for i = 1 : grid_count
        success_probability(i) = 0.999;
end;
plot(disk_usage, success_probability, 'r');

for i = 1 : grid_count
        success_probability(i) = 1;
end;
plot(disk_usage, success_probability, 'r');

xlabel('磁盘空间(MB)');
ylabel('成功概率');
```

A.3 密钥空间

```
function ret = calc_N( n, min, max)
arr = zeros(1, max );
for i = 1 : max
        arr(i) = n^i;
end;

exp = 0;
for i = 1 : max
        exp = exp + arr(i);
end;
format long g;
ret = exp;
```

A.4 彩虹链长度

```
function ret = calc_t(N, success_probability, m, table_count)
success_probability_each=1-exp(log(1-success_probability)/table_count);
exp_min = log(1 - success_probability_each) / log(1 - 1 / N);

exp_all = 0;
t = 0;
```

```
next = m;

while exp_all < exp_min
        exp_all = exp_all + next;
        t = t + 1;
        next = N * (1 - (1 - 1 / N) ^ next);

        if t > 131072   % too large
                break;
        end;
end;

ret = t;
```

A.5 磁盘占用空间

```
function ret = calc_disk_usage(m, table_count)
ret = ceil(m*16*table_count / 1024 / 1024 );
```

A.6 最大破解时间

```
function ret = calc_max_cryptanalysis_time(t, table_count, step_speed)
ret = t*t/2 / step_speed * table_count;
```

A.7 实际破解时间

```
                                rate_of_each_table * 1
(1 - rate_of_each_table) ^ 1 * rate_of_each_table * 2
(1 - rate_of_each_table) ^ 2 * rate_of_each_table * 3
 ...
(1 - rate_of_each_table) ^ (table_count - 1)*rate_of_each_table*table_count
(1 - rate_of_each_table) ^ table_count * table_count

function ret = calc_mean_cryptanalysis_time(N,t,m,table_count,step_speed)
rate_of_each_table = calc_success_probability(N, t, m);
temp = rate_of_each_table;
all  = 0;
for i = 1 : table_count
    all = all + temp * i;
```

```
    temp = temp * (1 - rate_of_each_table);
end;
all = all + (1 - rate_of_each_table) ^ table_count * table_count;

ret = t*t/2 / step_speed * all;
```

A.8 最大磁盘读取时间

```
function ret = calc_max_disk_access_time(m, table_count, disk_speed)
ret = m*16/1024/1024 * disk_speed * table_count;
```

A.9 实际磁盘读取时间

```
                                rate_of_each_table * 1
(1 - rate_of_each_table) ^ 1 * rate_of_each_table * 2
(1 - rate_of_each_table) ^ 2 * rate_of_each_table * 3
 ...
(1-rate_of_each_table)^(table_count-1)*rate_of_each_table*table_count
(1 - rate_of_each_table) ^ table_count * table_count

function ret = calc_mean_disk_access_time(N,t,m,table_count,disk_speed)
rate_of_each_table = calc_success_probability(N, t, m);
temp = rate_of_each_table;
all  = 0;
for i = 1 : table_count
    all = all + temp * i;
    temp = temp * (1 - rate_of_each_table);
end;
all = all + (1 - rate_of_each_table) ^ table_count * table_count;

ret = m*16/1024/1024 * disk_speed * all;
```

附录 B 彩虹表部分实现代码

B.1 彩虹表生成

```
int main(int argc, char* argv[])
{
  string sHashRoutineName  = argv[1];
  string sCharsetName      = argv[2];
  int nPlainLenMin         = atoi(argv[3]);
  int nPlainLenMax         = atoi(argv[4]);
  int nRainbowTableIndex   = atoi(argv[5]);
  int nRainbowChainLen     = atoi(argv[6]);
  int nRainbowChainCount   = atoi(argv[7]);
  string sFileTitleSuffix  = argv[8];

  CChainWalkContext::Dump();

// Generate rainbow table
printf("generating...\n");
CChainWalkContext cwc;
clock_t t1 = clock();
int i;
for (i = nDataLen / 16; i < nRainbowChainCount; i++)
{
        cwc.GenerateRandomIndex();
        uint64 nIndex = cwc.GetIndex();
        if (fwrite(&nIndex, 1, 8, file) != 8)
        {
                printf("disk write fail\n");
                break;
        }

        int nPos;
        for (nPos = 0; nPos < nRainbowChainLen - 1; nPos++)
        {
                cwc.IndexToPlain();
                cwc.PlainToHash();
```

```
34                  cwc.HashToIndex(nPos);
35          }
36
37          nIndex = cwc.GetIndex();
38          if (fwrite(&nIndex, 1, 8, file) != 8)
39          {
40                  printf("disk write fail\n");
41                  break;
42          }
43
44          if ((i + 1) % 100000 == 0 || i + 1 == nRainbowChainCount)
45          {
46                  clock_t t2 = clock();
47                  int nSecond = (t2 - t1) / CLOCKS_PER_SEC;
48                  printf("%d of %d chains generated (%d m %d s)\n",i+1,
49                          nRainbowChainCount,
50                          nSecond / 60,
51                          nSecond % 60);
52                          t1 = clock();
53          }
54  }
55  }
```

B.2 彩虹表破解

```
1  int main(int argc, char* argv[])
2  {
3
4  // hs
5  CHashSet hs;
6  if (fCrackerType)
7  {
8    int i;
9    for (i = 0; i < vHash.size(); i++)
10   hs.AddHash(vHash[i]);
11 }
12 else
13 {
14   int i;
```

```
15    for (i = 0; i < vLMHash.size(); i++)
16    {
17    hs.AddHash(vLMHash[i].substr(0, 16));
18    hs.AddHash(vLMHash[i].substr(16, 16));
19    }
20  }
21
22  // Run
23  CCrackEngine ce;
24  ce.Run(vPathName, hs);
25  }
```

附录 C　密钥字符集对照表

numeric = [0123456789]

alpha = [ABCDEFGHIJKLMNOPQRSTUVWXYZ]

alpha-numeric = [ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789]

loweralpha = [abcdefghijklmnopqrstuvwxyz]

loweralpha-numeric = [abcdefghijklmnopqrstuvwxyz0123456789]

mixalpha = [abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU-VWXYZ]

mixalpha-numeric = [abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU-VWXYZ0123456789]