上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

学士学位论文

BACHELOR'S THESIS

论文题目：SHA512 逆向优化技术的研究

学生姓名：伯佳澳

学生学号：515021910429

专　　业：网络空间安全

指导教师：邱卫东

学院(系)：电子信息与电气工程学院

SHA512 逆向优化技术的研究

摘要

随着信息化的迅速发展，信息安全成为信息系统的重要要求。其中，保密以及安全认证是信息安全系统要解决的两个主要问题。哈希函数是目前世界上使用范围最广且最有效的一种安全消息认证方法，近年来受到人们越来越多的关注。目前常见的哈希函数之一：安全哈希算法（英语：Secure Hash Algorithm，缩写：SHA）是由美国国家安全局（NSA）美国国家标准和技术协会（NIST）提出的一个密码散列函数家族，作为联邦信息处理标准于 1993 年公布。SHA512 作为 SHA 系列的第二个修订版本之一在 2002 年正式发布。本文主要研究 SHA512 破解过程中的逆向优化部分，结合其他优化完成寻找哈希值的明文的程序。研究过程参考了其他 SHA 系列哈希函数以及 MD5 函数的破解优化。OpenCL(全称 Open Computing Language,开放编程语言)是一个面向异构系统通用目的的并行编程的开放式标准。在此次研究过程中利用 OpenCL 进行基于 GPU 上的并行编程从而实现暴力破解过程并在其中应用逆向优化技术。最终借助并行运算以及针对算法和 GPU 两个方面分别进行的优化实现 SHA512 暴力破解给定哈希值的时间缩短，最终将结果与主流哈希函数破解软件进行对比来分析优化的效果，并对未来可进一步深入研究的方向进行了展望。

关键词：安全哈希算法，SHA512，OpenCL, GPU, 逆向优化

RESEARCH ON REVERSE OPOTIMIZATION OF SHA512

ABSTRACT

With the rapid development of information technology, information security has become an important requirement of information systems. Among them, confidentiality and security certification are two major issues to be solved by information security systems. As the most widely used message security authentication method, hash function has received more and more attention in recent years. One of the most common hash functions: Secure Hash Algorithm (SHA) is a family of cryptographic hash functions proposed by the National Security Agency (NSA) National Institute of Standards and Technology (NIST), published as a federal information processing standard in 1993. SHA512 was officially released in 2002 as one of the second revisions of the SHA series. This paper mainly studies the reverse optimization part of the SHA512 cracking process, and combines other optimizations to complete the procedure of finding the plaintext of the hash value. The research process refers to other SHA series hash functions and the crack optimization of MD5 functions. OpenCL (full name Open Computing Language) is an open standard for parallel programming for general purpose heterogeneous systems. In this research process, OpenCL is used to perform parallel programming on GPU to implement the brute force cracking process and apply reverse optimization technology. Finally, the parallel computing and the optimization of the algorithm and GPU are used to realize the SHA512 brute force to shorten the given hash time. Finally, the result is compared with the mainstream hash function cracking software to analyze the optimization effect and the future. The direction of further research can be prospected.

Key words: Secure Hash Algorithm, SHA512, OpenCL, GPU, Reverse Optimization

目录

第一章 绪论

随着主流杂凑算法MD5与SHA-1的缺陷被陆续发现，人们的注意力逐渐转移到对SHA-2系列杂凑算法的研究上来。SHA-512作为SHA-2家族最复杂最有代表性的一个，对其研究的成果有着对第二代SHA系列其它算法的适用性。本文以SHA-512作为主要研究对象，以逆向优化为主结合其他优化对SHA-512的破解算法进行优化。

1.1 研究背景和意义

杂凑算法作为一种数字签名以及口令管理的手段最早出现于上世纪九十年代，也因此在32位机时代诞生了MD5, SHA-1这种应用广泛的以32位字长为处理对象的杂凑算法。而出于安全性的需要对更加复杂，摘要更长的算法开始有了需求。SHA-512就是在这样的背景下诞生的。

1.1.1 杂凑算法概述

杂凑算法，又名哈希函数（Hash Function）。它是一种可以将任意长度的消息压缩为一个固定长度摘要的算法[1]。目前可追溯到的最早公开的哈希函数是1989年由RSA Laboratories的B. Kaliski所设计的MD2算法[2]（Message-Digest Algorithm）,它是针对8位计算机设计的。随后在MD2的基础上在接下来的两年内美国密码学家罗纳德·李维斯特（Ronald Linn Rivest）陆续发表了MD4与MD5杂凑算法[3][4]。其中，MD5算法在成熟性与安全性上更加完善，在接下来的数年时间成为使用最广泛的杂凑算法。1993年，基于MD5所开发的SHA系列杂凑算法由美国国家标准与技术研究院作为美国的政府标准所发布。第一代SHA系列杂凑算法成为SHA-0代，发布不久便因安全性问题所撤回。1995年所发布的修订版本SHA-1系列杂凑函数，在这之后直至现在MD5和SHA-1成为使用最广泛的杂凑算法。2002年，SHA-2系列算法公布，SHA-2家族共有三个成员：SHA-256,SHA-384和SHA-512。

随着时间的推移，MD5与SHA-1杂凑算法的安全性受到了广泛的挑战。在2000年以前，差分攻击就被证明对MD5一次循环是有效的[1]。从1998年，针对SHA-0的有效攻击研究成果被陆续公布[5]。鉴于对SHA-0的攻击成果进展迅速，与SHA-0相似的SHA-1也被建议谨慎使用。在2004年，美国国家标准与技术研究院正式宣布将用SHA-2逐步替代SHA-1，在这之后的2005年，王小云等人提出了对SHA-1算法的有效差分攻击方案[6]。

1.1.2 杂凑算法的攻击方式

当前对哈希函数的攻击主要有以下几种思路：

- 直接攻击

直接攻击又被称作野蛮攻击。攻击方法是直接搜索一个和消息m相同摘要的消息m’，为此，攻击者需要搜索的空间大小与哈希函数输出的摘要长度相关。例如MD5的摘要长度为128比特，则搜索空间为2^{128}数量级，也就是要进行同数量级次数的哈希计算。SHA-1的消息长度为160比特，搜索长度为2^{160}数量级。直接攻击的方式最为直接，对其优化的方式也可应用到其他攻击，故本文对杂凑算法的破解优化主要基于直接攻击方式。

- 字典法

字典法的思路与直接攻击完全相反，字典法主要过程是提前生成可能密码与对应哈希串

的对照表，进行密码攻击时直接根据哈希串从对照表中查询对应的密码。与直接攻击需要大量的计算相反，这种方法需要海量的磁盘空间完成对照表的存储。举个例子：假设密码的长度为 14 位，则生成 32 位哈希串的对照表将占用 5.7*10^14TB 的存储空间，相比于直接攻击。字典法的成本更加高昂，海量的存储空间在当前无法实现。

- 彩虹表法。

彩虹表（rainbow table）法是一种基于字典法改进的算法[7]，它结合了时间成本和空间成本，在字典法基础上以时间换空间。它在储存哈希值时使用了彩虹表这一结构，其运算过程如图 1-1

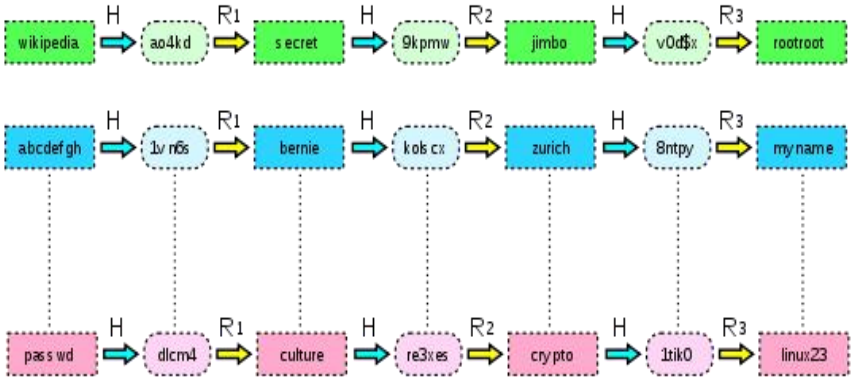


图 1-1 彩虹表的计算过程

在图 1-1 中，H 代表要破解的哈希函数，R_1,R_2,R_3 代表三个不同的约简函数。约简函数（reduction function）R 函数是构建这条链的时候定义的一个函数：它的值域和定义域与 H 函数相反。通过该函数可以将哈希值约简为一个与原文相同格式的值。

以第一条链为例，这条链共计算了三个明文：wikipedia,secret,jimbo 的对应的哈希值。但在最终的储存过程中只需要储存链两端的值，在第一条链中储存 wikipedia 与 rootroot.因此其所需空间较字典法大大减少。在破解时，只需由待破解的杂凑值通过 R 计算与 H 计算的不断交替计算，由结果来确定其在哪条链中，在根据储存的链结构的首端的值来计算出破解结果。

- 生日攻击

直接寻找与消息 m 相同摘要的消息 m’是困难的，但制造哈希碰撞则难度小了很多。生日攻击的目的也是找到相同摘要的另一个消息，只不过并未指定原消息 m，只要能找到任意两个消息有相同摘要即可。生日攻击是通用的寻找碰撞的方法[8]，其名字来源于著名的生日问题：在一个教室中最少应有多少学生才使得每个同学的生日都不一样。而一个 50 人的班级就有 97%的概率至少两人生日相同。事实上，50%碰撞概率所需的计算次数与哈希取值空间的平方根是同一个数量级。这大大降低了碰撞难度。这种利用哈希空间不足够大，从而制造碰撞的攻击方法称为生日攻击。生日攻击也可借助本文使用的优化方法来快速判断任意消息的摘要是否与 m 相同。在[9]这篇文章中，作者分析了对 SHA-512 进行生日攻击的难度，得出其难度相比于对 MD5,SHA-1 进行生日攻击要困难许多。

- 差分攻击

差分攻击是通过比较分析有特定区别的明文在通过加密后的变化传播情况来攻击密码算法的。差分攻击最早是作为针对分组加密算法所提出的攻击算法，在 2000 年之前也被证

明对 MD5 的一次循环有效，但似乎对全部四轮循环不奏效。2005 年，王小云、来学嘉等使用差分攻击的思路[10]，提出了针对 MD5 差分的有效算法，并在之后进一步提出消息修改算法其寻找相同摘要的两个明文的复杂度为 2^{37}，意味着任何人都可以相对容易的找到 MD5 的碰撞。同时，差分攻击也可对 SHA-1 生效。不过截止目前，本文的主要研究对象 SHA-2 还未有任何公开的算法缺陷，在[11]中，作者给出了对简化后仅有 18 轮的 SHA-512 进行了差分攻击研究，寻找碰撞的时间复杂度为 2^{14}。

1.1.3 SHA512 介绍及优化技术概述

SHA-512 作为第二代 SHA 系列函数中的一员，安全性强于 SHA-0 与 SHA-1 系列。在 SHA-2 系列之中其是生成摘要长度最高的一个，达到 512 比特，同时拥有最长的运算步骤数，达到了 80 轮运算，每轮运算相比 MD5 和 SHA-1 也复杂许多。可接受的消息长度也为最长，为 2^{128}。由此可见 SHA-512 作为 SHA 系列的一员其复杂度是主流杂凑算法中最强的，对其算法的优化破解研究是十分有意义的。

目前，优化主要有两个方向。一，针对计算过程和结果比较的算法的优化。比如进行预计算减少一组哈希函数的总的计算量，减少变量的使用保证在计算过程中使用更少的寄存器来提高速度，在进行完完整的计算过程之前就利用提前得到的部分结果来与目标哈希串进行比较以减少计算步骤等等。二，针对 GPU 的优化。本文的研究过程使用 OpenCL 作为编程语言来进行基于 GPU 的编程，在这过程中可以借助优化指令，合理利用 GPU 上的内存和寄存器，设计流水线结构，多线程等技术提高 GPU 的占用率，优化破解速度[12]。

1.2 本文的研究内容

当前哈希函数的使用十分广泛，且各类算法各异的哈希函数层出不穷。本文选取使用最广泛的 SHA 系列哈希函数中的摘要长度最大的 SHA-512 作为研究对象，主要研究在直接攻击过程中对杂凑算法的计算过程所做的优化。研究内容分为以下几个部分：

- 主流杂凑算法的运算过程探究

针对当前主流的杂凑算法，对其具体的运算过程进行研究与解析，并分析不同杂凑算法的异同点。

- 对当前广泛使用的杂凑算法优化过程进行分析

由算法计算过程的特征分析可行的优化方式，主要分为对算法进行的优化以及对 GPU 进行的优化两部分

- 对 SHA512 进行逆向为主的破解优化

根据 MD5 和 SHA 系列杂凑算法的算法实现和优化过程分析，给出对 SHA-512 进行破解逆向优化和其他优化。并借助 OpenCL 实现程序完成给定的杂凑值的明文查找。在 GPU 上进行并行计算最终完成对测试结果的对比分析。

本文针对 SHA-512 主要进行两种优化，首先是改进了 SHA-512 的逆向优化，使得最终可以在减少完整的三步运算以及四步部分运算的前提下完成计算结果与给定哈希串的对比。另一种是针对短口令的优化。由于以往的预计算对 SHA-512 的结构来讲效率并不高，在缩减运算程度不高的同时大幅度增加了寄存器的使用量，造成总的效率低下，针对于此我设计了面向短口令的优化，并将短口令分为 7 字节以下，15 字节以下和 23 字节以下三类。使得短口令在破解过程中可以实现在不增加寄存器使用的前提下减少计算量，最终实现运行速度提升。

1.3 本文的组织结构

本文分为绪论，MD5 与 SHA 系列杂凑算法介绍、杂凑算法优化技术介绍、SHA512 优

化实现，测试结果与分析以及结论六个部分。

第一章绪论主要介绍杂凑算法的研究背景和意义，杂凑算法的主要攻击方式以及本文的主要研究内容。

第二章介绍主流的杂凑算法的计算步骤及背景。主要分为 MD5，SHA-1 系列和 SHA-2 系列三个部分分别介绍。

第三章介绍杂凑算法的优化技术，分为通用优化技术中的提前比较，预计算等技术以及利用 GPU 进行优化时涉及到的口令生成技术，指令优化技术等。

第四章介绍对 SHA-512 优化所使用的方法，主要有针对短口令的优化和改进的逆向优化。并结合其他优化方法给出具体的 OpenCL 代码实现。

第五章进行 SHA512 优化过程的结果测试与分析。

第六章总结和展望了本文的工作内容以及成果，分析了研究过程的不足以及今后未来的改进方向。

第二章 MD5 与 SHA 系列杂凑算法介绍

2.1 MD5 杂凑算法介绍

MD5 算法全称 Message-Digest Algorithm，它可将任意长度的口令作为处理对象，将其进行多轮次压缩运算以产生一个 128 比特的摘要作为输出。它的 64 轮处理算法是以每 512 比特为对象进行处理的，因此在算法最开始需要对消息口令进行分组。图 2-1 描述了 MD5 算法的整体过程[13]。MD5 算法包含以下几个步骤：

（1）附加填充比特

首先需要对消息进行填充，直到消息的长度与 448 模 512 同余。填充的首位为 1，其余都是 0.

（2）附加长度值

用 64 比特的数字表示消息填充前的长度，将其填充到口令末尾。由于过程 1 中经过填充后消息长度模 512 余 448，因此附加长度值之后口令的最终长度可被 512 整除。

（3）缓存寄存器的初始化过程

在算法计算过程中，需要四个寄存器保存中间变量。每 512 比特处理后的结果保存在其中作为下一个 512 比特处理过程的输入值。这四个 32 位寄存器被初始化为以下十六进制的值：

$$A=67452301 \quad B=EFCDAB89$$
$$C=98BADCFE \quad D=10325476 \qquad (2\text{-}1)$$

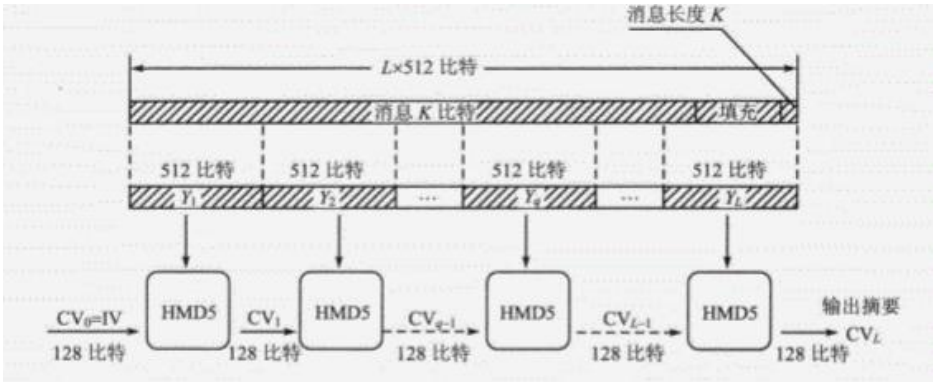


图 2-1 MD5 算法的整体过程

（4）顺序处理每个 512 比特

HMD5 结构是 MD5 中最主要的结构，它依次处理经过分组后的每一个 512 比特口令，每次处理完得到的输出作为下一个分组的链接变量输入，最后一组的输出作为最终的杂凑值。

单个 512 比特的 HMD5 处理过程如图 2-2 所示，HMD5 处理过程由四轮循环函数所组

成，每轮循环有十六步操作。每次循环都以当前正在处理的 512 比特分组 Y_q 和 32 比特链接变量 A，B，C，D 为输入，然后更新链接变量寄存器中的内容。

每个循环使用的逻辑函数各不相同，分别表示为 g_1,g_2,g_3,g_4。每步操作的具体运算流程如图 2-3 所示。

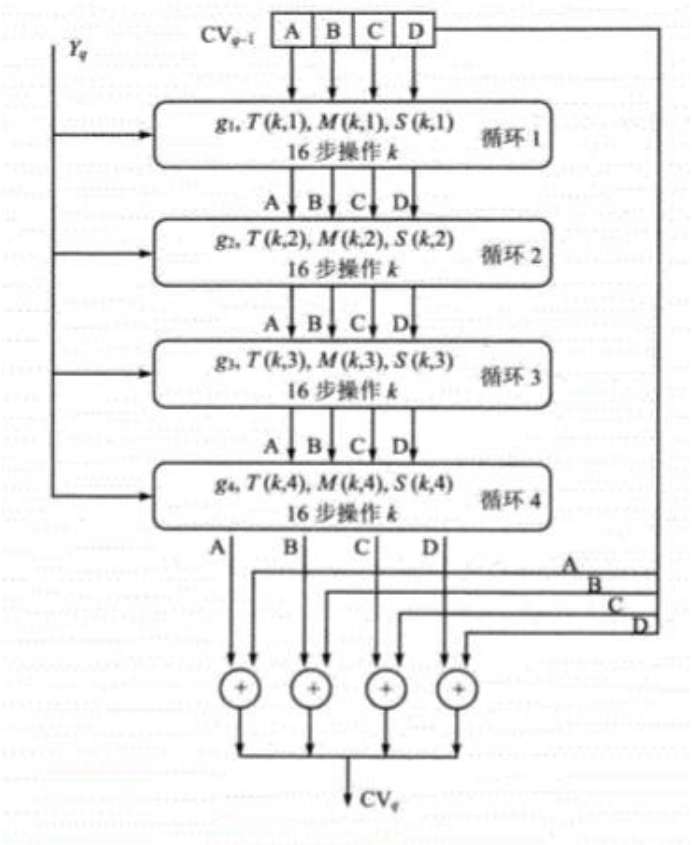


图 2-2 HMD5 处理过程

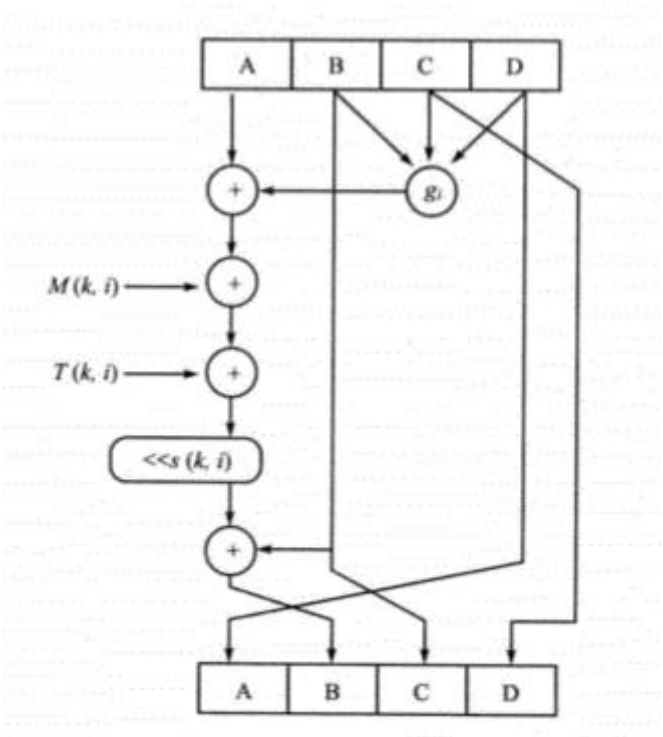


图 2-3 HMD5 单步操作

其中 T(K,i)是为了消除输入数据规律性而提供的 32 位随机数生成函数，它的计算公式如下：

$$T(K,i)= 2^{32}\times abs(sin(16(i-1)+k)) \qquad (2\text{-}2)$$

M(k,i)代表当前正在处理的 512 比特分组 Y_q 中的某个 32 位比特字。它在第 i 次循环中的第 k 次操作中所使用。我们以数组 X[0]-X[15]来代表当前 512 比特分组中的 16 个 32 为比特，则 M(k,i)的计算公式如下：

$$\begin{aligned} M(k,1) &= X[k-1] \\ M(k,2) &= X[(1+5(k-1))\bmod 16] \\ M(k,3) &= X[(5+3(k-1))\bmod 16] \\ M(k,4) &= X[7(k-1)\bmod 16] \end{aligned} \qquad (2\text{-}3)$$

<<s(k,i)表示 32 位比特字循环左移 s(k,i)位，s(k,i)可由表 2.1 所得。

表 2-1 循环左移的位数

	i=1	i=2	i=3	i=4
k=1	7	5	4	6
k=2	12	9	11	10
k=3	17	14	16	15
k=4	22	20	23	21

k=5	7	5	4	6
k=6	12	9	11	10
k=7	17	14	16	15
k=8	22	20	23	21
k=9	7	5	4	6
k=10	12	9	11	10
k=11	17	14	16	15
k=12	22	20	23	21
k=13	7	5	4	6
k=14	12	9	11	10
k=15	17	14	16	15
k=16	22	20	23	21

逻辑函数 g_i 的意义如表 2-2.

表 2-2 逻辑函数 g_i

g_1	$(b \wedge c) \vee (\bar{b} \wedge d)$
g_2	$(b \wedge c) \vee (\bar{d} \wedge c)$
g_3	$(b \oplus c \oplus d)$
g_4	$c \oplus (\bar{d} \wedge b)$

2.2 SHA-1 系列杂凑算法介绍

SHA-1 是 1995 年发布的 SHA 算法的第一个修订版，其算法的设计实现在很大程度上是模仿 MD5 算法的。它以 512 比特进行分组处理，产生一个 160 比特的摘要作为输出[14]。与 MD5 相同，它的算法也包括了附加填充比特，附加长度值，初始化链接变量缓冲区等一系列步骤。不同之处在于使用不同的初始化链接缓冲区以及单个 512 比特的压缩算法。它的初始化链接缓冲区由五个 32 位寄存器 A,B,C,D,E 表示，其初始值如下：

A=67452301　B=EFCDAB89　C=98BADCFE
D=10325476　E=C3D2E1F0　　(2-4)

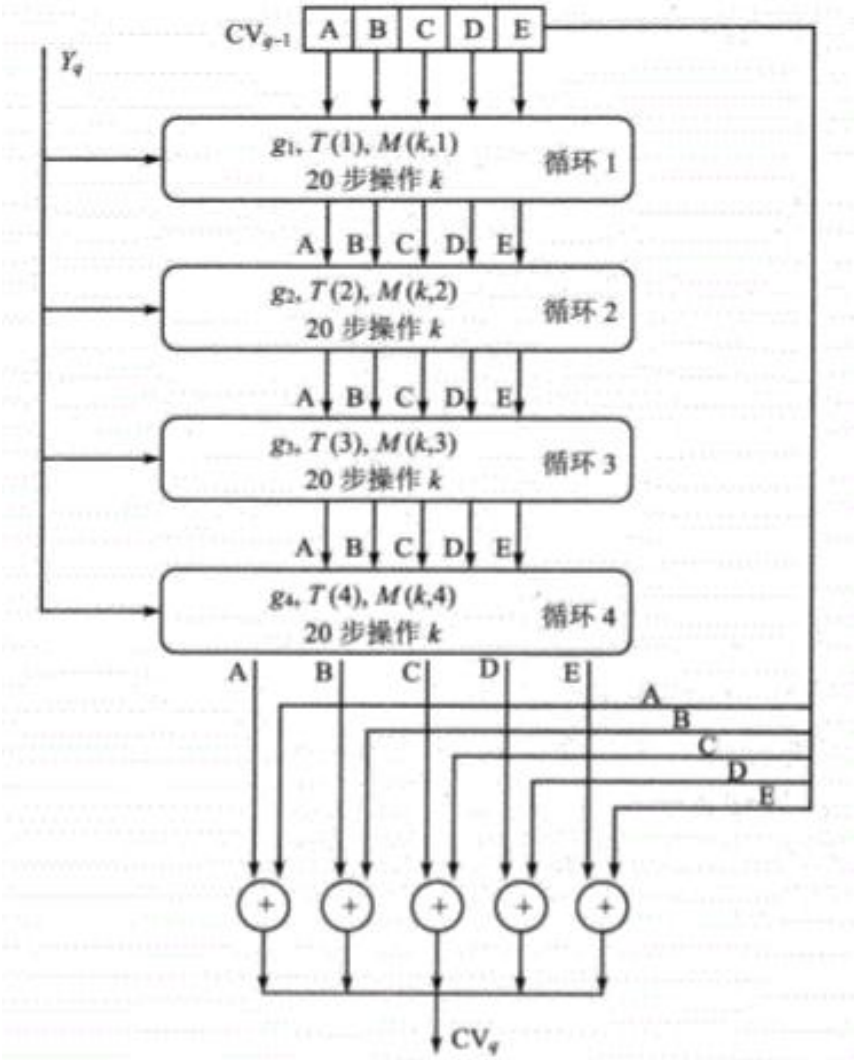


图 2-4　SHA-1 算法单个分组的处理过程

每个 512 比特由四轮循环组成。具体过程如图 2-4，每轮循环由对链接变量 ABCDE 的 20 步操作组成，每步操作如图 2-5 所示。

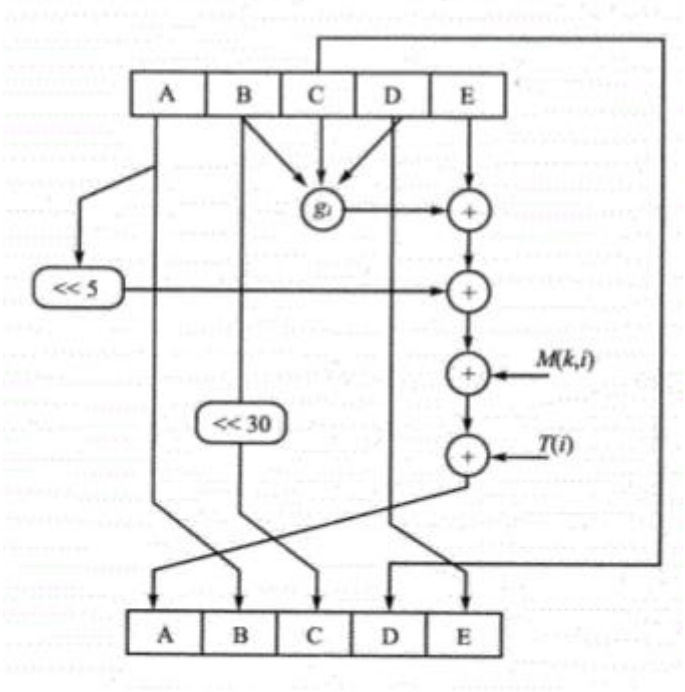


图 2-5　SHA-1 算法单步操作流程

其中，g_i 为四轮循环分别使用的逻辑函数。T(i)为额外的随机常数。它们的取值如下：

$$\begin{aligned} T(1) &= 5A827999 \\ T(2) &= 6ED9EBA1 \\ T(3) &= 8F1BBCDC \\ T(4) &= CA62C1D6 \end{aligned} \qquad (2\text{-}5)$$

逻辑函数 g_i 的意义如表 2-3.

表 2-3 SHA-1 的逻辑函数 g_i

g_1	$(b \wedge c) \vee (\bar{b} \wedge d)$
g_2	$(b \oplus c \oplus d)$
g_3	$(b \wedge c) \vee (b \wedge d) \vee (c \wedge d)$
g_4	$(b \oplus c \oplus d)$

M(k,i)代表当前正在处理的 512 比特分组 Y_q 中的某个 32 位比特字。它在第 i 次循环中的第 k 次操作中所使用。我们以数组 X[0]-X[15]来代表当前 512 比特分组中的 16 个 32 为比特，则 M(k,i)的计算公式如下：

令 j=20(i-1)+k-1 M(k,i)=W(j) 则：

$$\begin{aligned} &W(j)=X[j] \quad j<16 \\ &W(j)=(W_{j-16} \oplus W_{j-14} \oplus W_{j-8} \oplus W_{j-3}) <<1 \quad 16 \leqq j \leqq 80 \end{aligned} \qquad (2\text{-}6)$$

SHA-1 与 MD5 相比在算法上十分相似，不同之处在于 SHA-1 的 160 位摘要使得在面对强力攻击时更加安全，同时相比 MD5 更不容易受到密码分析的攻击。但也由于复杂性和步骤使得总的运算时间比 MD5 慢。

2.3 SHA-2 系列杂凑算法介绍

SHA-2 是 2002 年由 NIST 正式发布的 SHA 算法第二个修订版，它包含四个算法，分别是 SHA-224,SHA256,SHA384 以及 SHA512。其命名以 SHA 加摘要的比特长度结合而成。SHA-256 和 SHA-512 是很新的杂凑函数，前者以定义一个 word 为 32 位元，后者则定义一个 word 为 64 位元。它们分别使用了不同的偏移量，或用不同的常数，然而，实际上二者结构是相同的，只在循环执行的次数上有所差异。SHA-224 以及 SHA-384 则是前述二种杂凑函数的截短版，利用不同的初始值做计算。因此本节以 SHA-2 中最具代表性的 SHA-512 为例介绍 SHA-2 系列杂凑算法的特点[15]。

SHA-512 的分组流程与 MD5,SHA-1 保持一致。前一组的输出作为后一组缓冲区寄存器的初始化，最后一组的杂凑结果作为总的结果。不同之处在于 SHA-512 分组时以 1024 比特为一组，填充消息时同样填充一个 1 和若干个 0，然后使得长度模 1024 余 896.再以一个 128 位长度的值保存消息长度填充在末尾，最后使得填充完毕的值是 1024 的倍数。

SHA-512 的缓冲区由 8 个 64 位寄存器组成，64 位的字长使得其更适合在 64 位处理器上进行运算。寄存器初始化为以下 8 个常量：

$$\begin{aligned} A &= 6A09E667F3BCC908 & E &= 510E537FADE682D1 \\ B &= BB67AE8584CAA73B & F &= 9B05688C2B3E6C1F \\ C &= 3C6EF372FE94F82B & G &= 1F83D9ABFB41BD6B \\ D &= A54FF53A5F1D36F1 & H &= 5BE0CD19137E2179 \end{aligned} \quad (2\text{-}7)$$

每个寄存器内的内容是自然数前 8 个素数的平方根，取小数部分的前 64 位。SHA-512 对每个分组的运算过程具有 80 轮操作，比 MD5 的 64 轮多，与 SHA-1 相等。每个分组的运算流程如图 2-6

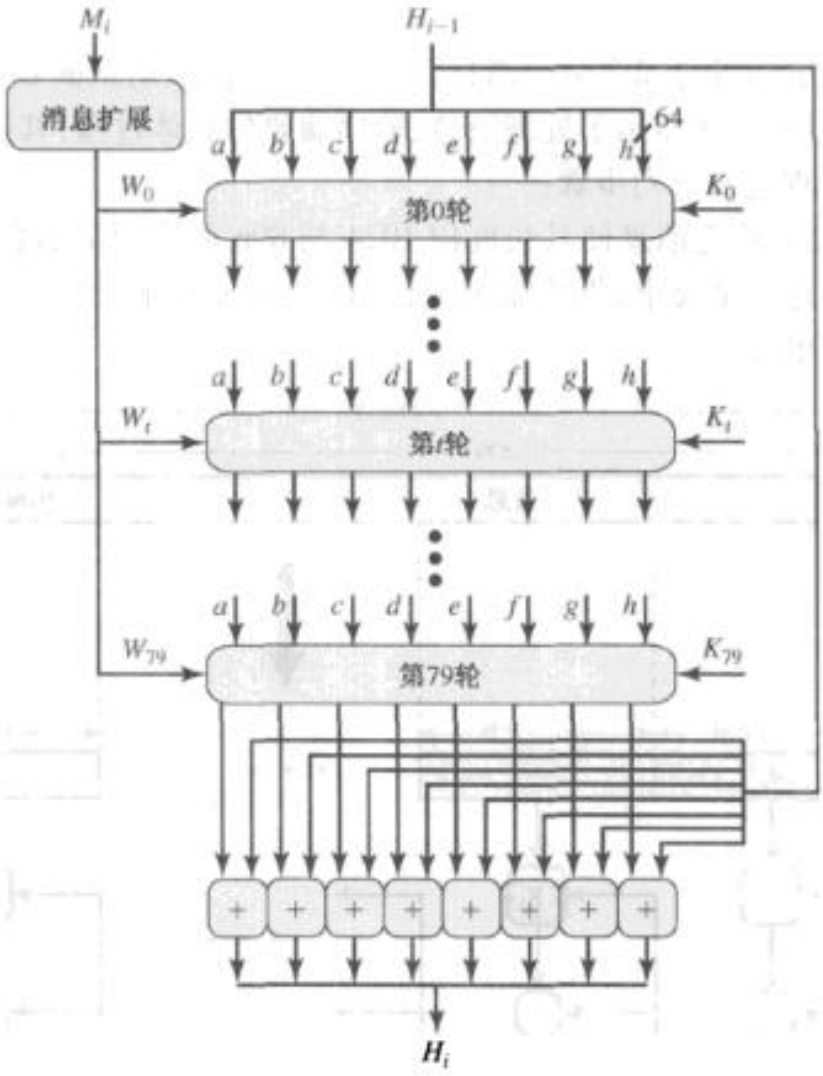


图 2-6　SHA-512 对单个分组的处理流程

图中，当前分组的 1024 比特 M_i 被扩展为 80 个 64 位消息扩展 W_t,扩展公式如下：假定当前的 M_i 用数组 S[0]-S[15]来表示自身的 16 个 64 位比特，则

$$W_t = S[t] \qquad t<16$$
$$W_t = \delta_1(W_{t-2}) + W_{t-7} + \delta_0(W_{t-15}) + W_{t-16} \qquad 16 \leqq t \leqq 79 \qquad (2\text{-}8)$$

其中：　$\delta 1(x) = ROTR1(x) \oplus ROTR8(x) \oplus SHR7(x)$

$\delta 0(x) = ROTR19(x) \oplus ROTR61(x) \oplus SHR6(x)$

ROTRn(x)为 x 循环右移 n 位，SHRn（x）为 x 右移 n 位，左侧填充 0.每轮运算的具体轮函数如图 2-7.

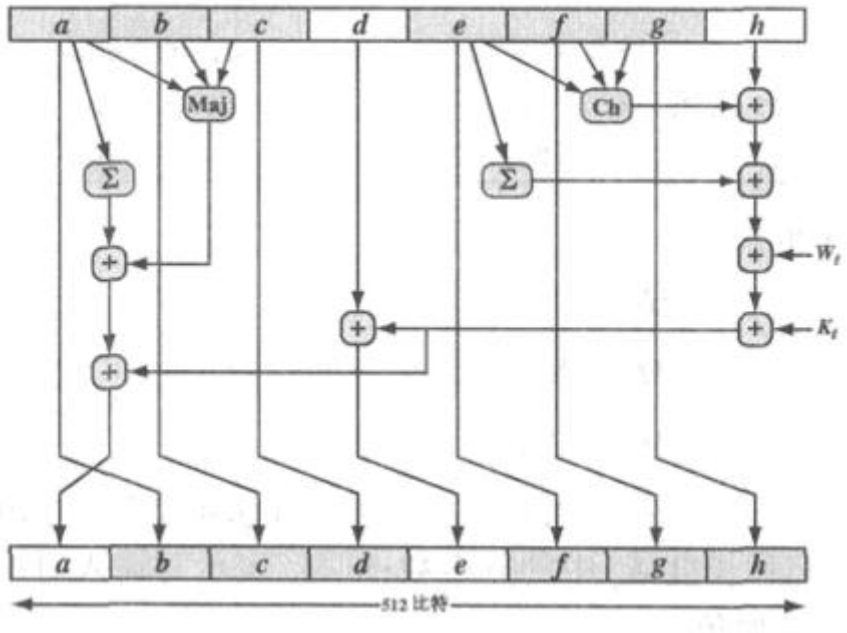


图 2-7　SHA-512 的轮函数运算流程

图中 K_t 是轮常数，80 个轮常数各不相同，他们是对自然数前 80 个素数开立方根取小数部分的前 64 位。逻辑运算 Maj(),Ch,$\sum_0$,$\sum_1$ 分别为：

$$\begin{aligned}
&Ch(e,f,g) = (e\ AND\ f) \oplus (NOT\ e\ AND\ g)\\
&Maj(a,b,c) = (a\ AND\ b) \oplus (a\ AND\ c) \oplus (b\ AND\ c)\\
&\sum\nolimits_0 a = ROTR28(a) \oplus ROTR34(a) \oplus ROTR39(a)\\
&\sum\nolimits_1 e = ROTR14(e) \oplus ROTR18(e) \oplus ROTR41(e) \qquad (2\text{-}9)
\end{aligned}$$

SHA-512 与 MD5 和 SHA-1 相比，安全性更高，其到目前为止还没有公开的缺陷发现。在运算过程中每轮都引用了无规律的常量，同时大量采用了移位运算和复杂的逻辑运算，显著的混淆了明文。更长的摘要也是的碰撞难度大大提高。找到两个具有相同摘要的消息的复杂度需要 2^{256} 次操作，给定摘要寻找消息的复杂度需要 2^{512} 次操作。

第三章 杂凑算法优化技术介绍

3.1 通用优化技术

对杂凑算法的优化目的是为了更快速的判断口令的哈希值是否与给定的哈希值相等。优化技术可分为两部分：只与杂凑算法相关的通用优化部分和与硬件体系结构相关的部分。本节介绍通用优化技术部分。为了简化问题，我们假定要计算的口令填充后长度都是一个分组，即 1024 比特，后文不在赘述。

3.1.1 正向优化技术

通用优化技术可分为正向优化技术和逆向优化技术两部分。正向优化技术指的是对杂凑算法正向的计算过程中通过减少计算量但不影响杂凑结果的生成的这类方法。预计算是目前主要使用的有效的正向优化技术。我们以 SHA-1 为例观察 SHA-1 中每一轮 Wj 的计算，由公式 2-6：

当 j < 16 时，Wj = Xj，Xj 对应于原始输入分组，32 位；

当 16 <= j < 80 时，Wj = (W[j-3] ^ W[j-8] ^ W[j-14] ^ W[j-16]) << 1。

对于 17-80 轮的每一轮，需计算 3 次异或和 1 次移位操作，在一个 SHA-1 计算中，总共为 64*3 次异或和 64 次移位操作。当我们在破解特定杂凑值而需要同时计算一批口令时，假设这批口令的字符集为 a-z，其除了前两个字符外余下字符均相同。则这组字符串共 26*26 个，一共需要计算 26*26 次，计算 W[16…75]总共需要 26*26*64*3=129792 次异或和 26*26*64=43264 次移位操作。

考虑所计算的一批 SHA-1 值中，这批 SHA-1 的 X[0…15]只有 X[0]不同，X[1…15]都相同。如果能够计算出某些中间结果，而且这些中间结果跟 X[0]无关，那么这批 SHA-1 的中间结果都是一样的，因此可以把计算中间结果的步骤从这批 sha1 的串行计算中分离出来，以减少计算步骤。可以预先计算如下中间值：

```
PW[16] = (W[13] ^ W[ 8] ^ W[ 2]) << 1
PW[17] = (W[14] ^ W[ 9] ^ W[ 3] ^ W[ 1]) << 1
PW[18] = (W[15] ^ W[10] ^ W[ 4] ^ W[ 2]) << 1
PW[19] = (PW[16] ^ W[11] ^ W[ 5] ^ W[ 3]) << 1
PW[20] = (PW[17] ^ W[12] ^ W[ 6] ^ W[ 4]) << 1
…
PW[79] = (PW[76] ^ PW[71] ^ PW[65] ^ PW[63]) << 1        (3-1)
```

注意这些中间值与 X[0]无关，计算这些中间值需要 64*3-1 次异或和 64 次移位。然后，对于这组口令的每次计算，计算如下值（20 次移位）：

```
W0_1 = W[0] << 1
W0_2 = W[0] << 2
…
W0_20 = W[0] << 20                                      (3-2)
```

最后，可以计算得：

```
W[16] = PW[16] ^ W0_1
W[17] = PW[17]
W[18] = PW[18]
W[19] = PW[19] ^ W0_2
W[20] = PW[20]
…
W[75] = PW[75] ^ W0_6 ^ W0_12 ^ W0_14
W[76] = PW[76] ^ W0_7 ^ W0_8 ^ W0_12 ^ W0_16 ^ W0_21
W[77] = PW[77]
W[78] = PW[78] ^ W0_7 ^ W0_8 ^ W0_15 ^ W0_18 ^ W0_20
W[79] = PW[79] ^ W0_8 ^ W0_22                          (3-3)
```

如上计算最终 W[16…79]需要 108 次异或。所以这时这组口令需要计算 64*3-1+108*26*26=73199 次异或和 64+20*26*26=13584 次移位，性能对比如表 3-1。

表 3-1 对 SHA-1 的预计算优化前后的性能对比

	优化前	优化后
异或	129792	73199
移位	43264	13584

对于 SHA-1 而言，预计算的思想局限性在于在计算这组口令时需要一定量的寄存器保存这些预计算的值，而这些寄存器在 GPU 内作为紧缺资源会影响线程的数量。故预计算是一个有利有弊的过程，需要平衡预计算强度和寄存器的使用两方面。

同时计算的这组口令必须只有前 32 位，也就是四个字节是不同的。如果增加到 64 位 8 个字节，也就是 W[0]和 W[1]不同会造成预计算的效果不再明显。

3.1.2 逆向优化技术

逆向优化技术主要指提前比较技术：在一次杂凑算法完整的 80 步计算结束之前就可以根据寄存器内变量的值来确定当前计算的口令是否是我们要寻找的口令。还是以 SHA-1 为例，SHA-1 对于 1 个分组的数据进行 4 个循环每个循环 16 步运算共 80 步运算，设最后一个循环的轮函数为 II。观察最后 5 步，即

```
II(b, c, d, e, a, W74, 0xca62c1d6)
II(a, b, c, d, e, W75, 0xca62c1d6)
II(e, a, b, c, d, W76, 0xca62c1d6)
II(d, e, a, b, c, W77, 0xca62c1d6)
II(c, d, e, a, b, W78, 0xca62c1d6)
II(b, c, d, e, a, W79, 0xca62c1d6)                     (3-4)
```

在第 76 步中，由图 2-2 可得 e = (a << 5) + I(b, c, d) + e + W75 + t4，并且在第 78 步中，e <<= 30，之后 e 不再改变。因此可以在运算完 76 步后，判断 e << 30 与目标是否一致，如

果不一致接下来的运算就没有必要了，从而减少了 4 步不必要的运算，可以提高效率 5%。逆向优化技术目的在于尽可能早的结束算法，提前进行判断，从而减少计算量[16]。

3.2 GPU 优化方式

凡是涉及到 GPU 体系结构以及并行编程的优化方式可将其归类为针对 GPU 的优化。对哈希函数的破解过程需要进行大量的异或运算以及移位运算并且在运算过程中几乎没有逻辑判断、条件分支等结构。基于这样的特殊性，如果选对了硬件平台则破解运算过程会事半功倍。在[17][18][19]这三篇文章中，作者设计了相应的硬件计算平台。除此之外，这样的程序也十分适合在 GPU 上进行并行计算。因为 GPU 相比于 CPU 有着数量十分庞大的核心和寄存器，更加适合进行大规模数据运算。而 CPU 的核心和寄存器虽然数量上不及 GPU,但其具有更加复杂的结构包括流水线，分支预测等等，这使得 CPU 擅长进行逻辑运算。因此，主流 GPU 平台比主流 CPU 进行哈希函数破解的速度要快上非常多[20]。目前，主流的 GPU 厂商有英伟达（NVIDIA）以及 AMD 两家。基于 GPU 优化的主要目标是提高 GPU 的占空比，即我们的程序运行时尽量不让 GPU 处于闲置的状态；另一方面是尽可能的提高并发数，做到在同一时刻有尽可能多的程序并行执行。

3.2.1 Opencl 介绍

OpenCL，全称 Open Computing Language，开放运算语言，在当前的 GPU 开发中，主要使用的语言有 OpenCL 和 CUDA 两类，不同于仅支持 NVIDIA GPU 的 CUDA, OpenCL 是第一个面向异构系统通用目的并行编程的开放式、免费标准，也是一个统一的编程环境，便于软件开发人员为高性能计算服务器、桌面计算系统、手持设备编写高效轻便的代码，而且广泛适用于多核心处理器(CPU)、图形处理器(GPU)、Cell 类型架构以及数字信号处理器(DSP)等其他并行处理器，在游戏、娱乐、科研、医疗等各种领域都有广阔的发展前景[21]。

当前 NVIDIA 和 AMD 的 GPU 均提供 OpenCl 的运行环境，本文采用 NVIDIA 的 Maxwell 架构 GPU：GTX970 来作为测试环境。

OpenCl 中有三个重要的概念，分别是 kernel,work_item 以及 work_group. kernel 是指一个用 OpenCl c 语言编写的、代表一个单一执行实例的代码单元。opencl c 语言看起来跟 c 语言函数非常相像，都有一个参数列表“局部”变量定义和标准控制流结构。opencl 术语中把这种 kernel 实例称为 work-item(工作项)。但 opencl kernel 与 c 语方函数的区别在于其并行语义。

work_item 是定义在一个很大的并行执行空间中的一小部分。是并行操作中每一部分的实例化。通俗来说，可以理解为 kernel 里定义的执行函数。当 kernel 启动后会创建大量 work_item 来同时执行，以完成并行任务。work_item 根所其数据结构大小可分为一维、二维和三维数据。work_item 之是的运行是相互独立的，不同步的。

opencl 将全局执行空间划分为大量大小相等的，一维、二维、三维的 work_item 集合，这个集合就是 work_group。在 work_group 内部，各个 work_item 之间允许一定程度的通信。而有 work_group 保证并发执行来允许其内部的 work_item 之间的本地同步[22]。

3.2.2 口令生成方式优化

通常口令生成有两种方法，一是放在 CPU 上进行，然后通过总线将口令传输到 GPU 中；二是在 GPU 中生成。两种方法各有利弊。CPU 上生成口令灵活度较大，可以容易地添加前缀、后缀、salt 等各种不同需求；然而受限于 PCI-E 总线数据传输率，对于快速的算法，例如 MD5、SHA-1、qq 等口令传输会成为整体性能的瓶颈，导致 GPU 占空比较低。在 GPU 上生成口令能够大大减少数据的传输量，适应高速算法；然而该方法灵活度较低，字符集大小、前后缀等都会对性能带来不小的影响。针对二者的优缺点可以使用流水线结构结合 CPU 与 GPU 共同产生口令。

在论文[12]中提到的方法如下：将口令拆分为前缀和后缀两部分。CPU 负责在初始状态生成一个前缀数组，数组中的前缀长度通常为四字节以下，将其保存到 GPU 的全局内存中；后缀也由 CPU 生成，在每次 GPU 完成当前口令组的计算后传到 GPU.GPU 负责口令的拼接，将每次 CPU 传入的后缀与保存着全局内存的前缀数组进行拼接，得到口令数组。在 GPU 进行运算的同时 CPU 可以生成下一个口令后缀，形成了一个流水线结构。这种口令生成方式保留了 CPU 生成口令的灵活性，同时极大的减少了带宽，而且口令的拼接仅仅是一条简单的赋值语句。口令的拼接如图 3-1.

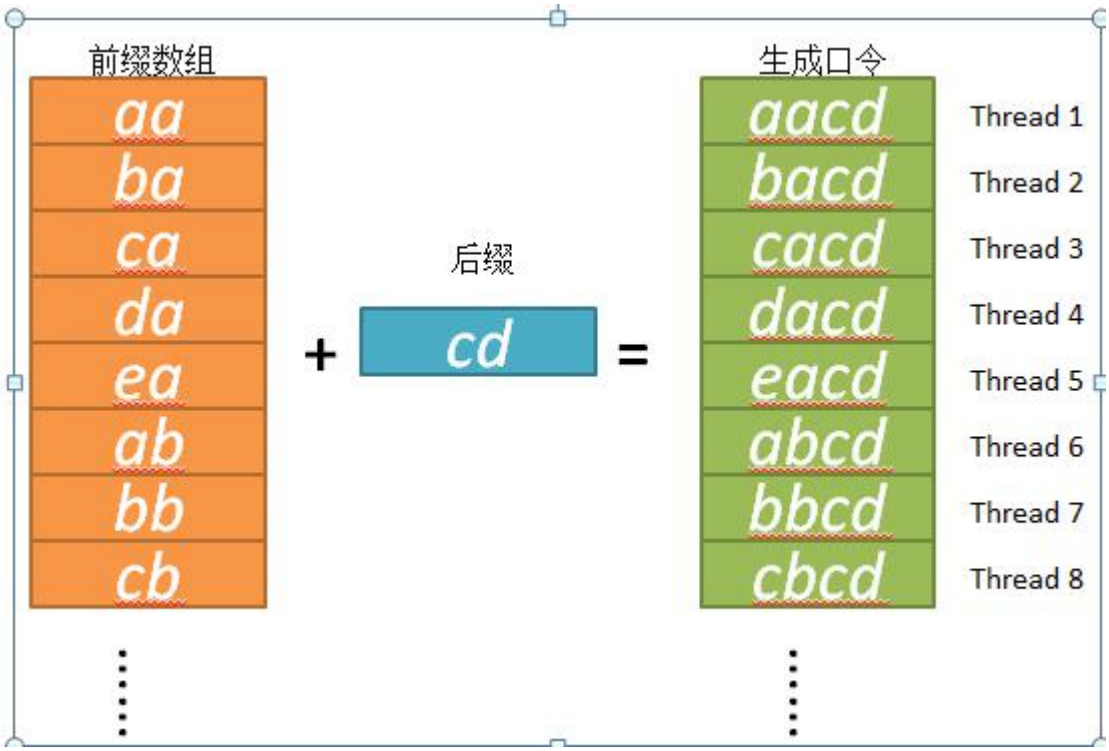


图 3-1 GPU 上的口令拼接过程

3.2.3 指令优化

将杂凑算法的计算过程落实到具体的程序指令的过程，可以改写运算表达式以达到减少指令的目的。例如函数：

$$F(X,Y,Z) =(X\&Y)|((\sim X)\&Z) \qquad (3\text{-}5)$$

可以改写为：

$$F(X,Y,Z) = (z \wedge (x \,\&\, (y \wedge z)) \qquad (3\text{-}6)$$

以这种方式运算可以将指令从四条简化到三条。进一步，在 NVIDIA 和 AMD 的新版驱动上均可以将 F 函数直接优化为一条 BFI_INT 指令：

$$bitselect(Z, Y, X) \qquad (3\text{-}7)$$

3.2.4 内存及线程优化

GPU 原先为并行计算所设计，作为多线程处理器，线程调度和 GPU 上的内存使用是影响性能的重要指标。

AMD 和 NVIDIA 的 DX11 GPU 都有一定的线程粒度，NVIDIA 的线程粒度被称作 Warp，一个 Warp 是 32 "线程";AMD 的线程粒度称作 wavefront，粒度是 64 "线程"[23][24]。在 GTX970 上最少要 32 个线程同时调度，在 OpenCl 环境下有两个重要的参数概念，分别是 local_work_size 以及 global_work_size。他们是 OpenCl 的 API: clEnqueueNDRangeKernel() 的参数。这是一个 OpenCl 中重要的函数。官方文档对其功能的解释如下：Enqueues a command to execute a kernel on a device.即它把编译好的在 GPU 上运行的内核程序放到命令队列中。它的完整参数列表如下：

```
cl_int clEnqueueNDRangeKernel (    cl_command_queue   command_queue,
                                   cl_kernel           kernel,
                                   cl_uint             work_dim,
                                   const size_t       *global_work_offset,
                                   const size_t       *global_work_size,
                                   const size_t       *local_work_size,
                                   cl_uint             num_events_in_wait_list,
                                   const              cl_event *event_wait_list,
                                   cl_event           *event)          (3-8)
```

global_work_size 可以简单理解为给每一个 kernel 函数编号的总数，local_work_size 可以理解为每个 work_group 中 work_item 的总数。

我们以 GTX970 为例，对于每个线程粒度有 32 个线程的 GTX970,Opencl 的程序需要将 local_work_size 设为 32 或者 32 的倍数是效率达到最高。同时由于 global_work_size/local_work_size 必须为整数，而且为了 global_work_size 能容纳完整的前缀数组，其必须达到一定的大小。最后，global_work_size 的数量最好是 GPU 中算数运算单元（ALU）的整数倍。我们假设当前使用的前缀数组中每个字符串长度为 4，字符集的数量为 charsetSize，GTX970 的 ALU 数量为 1664 个。因此，有：

$$\begin{aligned} &\text{global\_work\_size} > \text{charsetSize}^4 \\ &\text{global\_work\_size} \,\%\, 32 == 0 \\ &\text{global\_work\_size} \,\%\, 1664 == 0 \end{aligned} \qquad (3\text{-}9)$$

第四章 SHA512 优化实现

4.1 SHA512 的逆向优化技术

SHA-512 在计算过程中共有 8 个变量寄存器，每轮运算都会对这 8 个寄存器进行更新。与 SHA-1 和 MD5 一样的是，每一轮运算中并非每个寄存器内的值都是经过计算而来。由图 2-7，每一轮的计算只有 d 和 h 这两个寄存器内的值是经过运算而来，其余留个寄存器内的值都是由上一轮其他寄存器内的值经过简单移动而得到的。这使得 SHA-512 的破解程序也可使用提前退出来进行逆向优化。

4.1.1 现有逆向优化方案分析

在 Can Ge, Lingzhi Xu, Weidong Qiu 等人的[25]这篇论文中提到了关于 SHA-512 的提前退出算法。作者指出在第 77 轮运算中计算出了寄存器 H 的值，然后在第 78，79 和 80 三轮中 H 寄存器的值并未改变，因此可以将判断提前到第 78 轮，减少了两轮的运算。

4.1.2 改进的逆向优化方案

首先对 SHA-512 最后几轮循环做一个简要的分析。 由图 2-7 以及公式 2-9，可将每轮的运算定义为函数 P(a-f),它包含以下几个过程：

```
tmp1 = Ch(e,f,g) + SIGMA1(e) + h + K +W
tmp2 = Maj(a,b,c) + SIGMA0(a)
d += tmp1
h = tmp1 + tmp2                                    (4-1)
```

因此，在 P(a-f)中，每次只有 d,h 两个变量得到了更新，将更新之后的值记作 d',h'。h'在计算过程中所有原寄存器的值以及消息扩展和轮常数都参与到运算，而 d'只由 d,e,f,g,h 以及消息扩展 W 和轮常数 K 计算而来。现在我们将第 73 轮到第 80 轮的运算由函数 P 表示：

```
P(A, B, C, D, E, F, G, H, W(72), 0x28db77f523047d84);
P(H, A, B, C, D, E, F, G, W(73), 0x32caab7b40c72493);
P(G, H, A, B, C, D, E, F, W(74), 0x3c9ebe0a15c9bebc);
P(F, G, H, A, B, C, D, E, W(75), 0x431d67c49c100d4c);
P(E, F, G, H, A, B, C, D, W(76), 0x4cc5d4becb3e42b6);
P(D, E, F, G, H, A, B, C, W(77), 0x597f299cfc657e2a);
P(C, D, E, F, G, H, A, B, W(78), 0x5fcb6fab3ad6faec);
P(B, C, D, E, F, G, H, A, W(79), 0x6c44198c4a475817);        (4-2)
```

在公式 4-2 中，标注红色与蓝色的寄存器是在这一轮计算中值被更新的寄存器。其中，标红色的寄存器在本轮的更新过程中与 P 函数的前三个参数无关，标绿色的寄存器在本轮的计算中与所有寄存器都相关。在现有的提前退出方案中，用来做判断的寄存器是第 77 步运算中 P 函数的第四个参数 H 寄存器，它在第 78-80 轮次中均未得到更新。

更进一步的观察可以注意到如下事实：第 77 轮的 H 变量在更新过程中只依赖于 A，B,C，D 这四个寄存器的值和自己原本的值。而与 P 函数的前三个参数,E,F,G 这三个寄存器无关。同时，第 77 轮运算中更新的 D 寄存器也与 78-80 轮计算一样丧失了意义。因此，我们可以进一步放弃掉 77 轮中 D 寄存器的计算以及 74-76 轮中 G,F,E 这三个变量的更新。

我们定义函数 K(d-f)：

```
tmp1 = Ch(e,f,g) +    SIGMA1(e) + h + K +W
d += tmp1                                          (4-3)
```

因此，从第 73 轮计算开始可以简化为如下：

```
P(A, B, C, D, E, F, G, H, W(72), 0x28db77f523047d84);
  K( C, D, E, F, G, W(73), 0x32caab7b40c72493);
  K(B, C, D, E, F, W(74), 0x3c9ebe0a15c9bebc);
  K(A, B, C, D, E, W(75), 0x431d67c49c100d4c);
  K(H, A, B, C, D, W(76), 0x4cc5d4becb3e42b6);      (4-4)
```

首先，第 74 轮放弃了 G 寄存器的更新，其并未影响 75-77 轮中 B,A,H 的更新。然后第 75 轮放弃了对 F 寄存器的更新，未对 76,77 两轮中的 A,H 寄存器有影响。在这之后 76 轮，77 轮放弃了对 E,D 的更新，也未产生影响。综上，优化后的 SHA-512 提前退出可以由原来的从 80 轮优化到 77 轮到现在的 73 轮完整的 P 运算加上 4 轮不完整的 K 运算。

4.2 针对短口令的破解优化

针对短口令的破解优化是一种正向的优化方法。SHA512 的单个分组共 1024 个比特，排除掉 128 位的长度信息，单个分组下比特容量最高为 896，ASCII 编码下也就是 112 个字节。而当明文的长度只有几个字节的状态下是这 1024 个比特会有大量位被 0 填充，可以基于这一特点设计针对短口令的破解优化。

4.2.1 现有 SHA512 正向优化分析

在论文[15]中实现了对 SHA-512 的预计算算法，其预计算的目标是消息扩展 W_t,参考公式 2-8：

$$W_t = S[t] \quad t<16$$
$$W_t = \delta_1(W_{t-2})+ W_{t-7}+\delta_0(W_{t-15})+W_{t-16} \quad 16 \leqq t \leqq 79 \qquad (2\text{-}8)$$

当我们计算一组仅前缀不同的口令时，如果前缀小于 8 个字节，也就是 64 位。那么这一组口令仅 S[0]部分不同，S[1]至 S[15]均相同。进而这一组口令的消息扩展 W_0 是不同的，而 W_1 至 W_{15} 均是相同的。我们可以进行如下预计算：

$$P_{16}= \delta_1(W_{14})+ W_9+\delta_0(W_1)$$
$$P_{17}= \delta_1(W_{15})+ W_{10}+\delta_0(W_2)+W_1$$
$$P_{18}= W_{11}+\delta_0(W_3)+W_2 \qquad (4\text{-}5)$$

由公式 2-8 可得：

$$W_{16}= P_{16} + W_0$$
$$W_{17}= P_{17}$$
$$W_{18}= P_{18} + \delta_1(W_{16}) \quad (4\text{-}6)$$

因此，在这组口令的计算中每个口令的消息扩展的计算均可借助预计算的 P 的值来进行 W_{16}, W_{17} 与 W_{18} 的计算。

实际上，预计算的过程可以在 P_{18} 之后继续尝试。

$$P_{19}= W_{12}+\delta_0(W_4)+W_3$$
$$W_{19}= P_{19} + \delta_1(W_{17}) \quad (4\text{-}7)$$

当 t 进一步增大超过 33 的时候，由公式 2-8，W_t 的计算并未涉及到 W_1 至 W_{16}。同时，由于 W_t 的计算与 SHA-1 中 W_t 的计算不同，其无法像 SHA-1 中的 W_t 一样由于仅由异或和移位运算组成，使得每一个 W_t 都可以将与 W_0 无关的定值部分剥离开来。这使得对 SHA-512 的预计算有两个缺点：一，当 t 超过 33 时无法进行预计算。二，预计算产生的值在占用一个寄存器的情况下被使用的次数极少，使得预计算提高的性能远不及寄存器占用过多带来的性能损失。这也是在[25]这篇文章中仅仅进行三次预计算的原因。再往下进行预计算得不偿失。

在[26]这篇文章中，作者也利用到预计算的思想，同时尝试将每个口令的计算过程并行化，以此实现对 SHA-512 的优化，不过缺点依然存在，在计算过程中大量使用寄存器导致优化效果得不偿失，同时，由于作者测试时使用 FPGA, 其可以通过并行化来提高性能，而在 GPU 环境下已然是高度的并行化。对单个线程来并行运算会造成同时可以进行运算的线程数量减少，因此这种优化在 GPU 环境下并非十分合适。

4.2.2 短口令优化的实现

针对 W_t 的计算结构更加复杂带来的预计算无法深入进行，我们可以改变正向优化的思路。当待计算的字符串较短时，其转化为一个分组的内容会有大量填充比特。针对这一特点可以针对短口令的消息扩充算法进行简化。例如当口令长度最大为 7 字节时，其 ASCII 码比特长度为 56 比特。经过比特填充以及消息长度填充后整个分组的 1024 位仅有前 64 位和后 64 位是有内容的。即：

$$W_t = 0 \qquad (1 \leqq t \leqq 14) \quad (4\text{-}8)$$

因此，在后续的 Wt 计算过程中可以将 0 带入到算法公式中，使得计算过程得以简化。类似的，当口令长度为 8 至 15 个字节时，经过比特填充以及消息长度填充后整个分组的 1024 位仅有前 128 位和后 64 位是有内容的。即：

$$W_t = 0 \qquad (2 \leqq t \leqq 14) \quad (4\text{-}9)$$

当口令长度为 16 至 23 个字节时，经过比特填充以及消息长度填充后整个分组的 1024 位仅有前 192 位和后 64 位是有内容的。即：

$$W_t = 0 \qquad (3 \leqq t \leqq 14) \qquad (4\text{-}10)$$

当口令长度小于等于 7 字节时，将公式（4-8）带入到 W 的计算过程（2-8）中，可得当 t 不超过 30 时：

```
w[16] = w[0]
w[17] = sigma1(w[15])
w[18] = sigma1(w[16])
w[19] = sigma1(w[17])
w[20] = sigma1(w[18])
w[21] = sigma1(w[19])
w[22] = sigma1(w[20]) + w[15]
w[23] = sigma1(w[21]) + w[16]
w[24] = sigma1(w[22]) + w[17]
w[25] = sigma1(w[23]) + w[18]
w[26] = sigma1(w[24]) + w[19]
w[27] = sigma1(w[25]) + w[20]
w[28] = sigma1(w[26]) + w[21]
w[29] = sigma1(w[27]) + w[22]
w[30] = sigma1(w[28]) + w[23] +sigma0(w[15])        (4-11)
```

当口令长度小于等于 15 字节时，将公式（4-9）带入到 W 的计算过程（2-8）中，可得当 t 不超过 30 时：

```
w[16] = sigma0(w[1])+w[0]
w[17] = sigma1(w[15])+w[1]
w[18] = sigma1(w[16])
w[19] = sigma1(w[17])
w[20] = sigma1(w[18])
w[21] = sigma1(w[19])
w[22] = sigma1(w[20]) + w[15]
w[23] = sigma1(w[21]) + w[16]
w[24] = sigma1(w[22]) + w[17]
w[25] = sigma1(w[23]) + w[18]
w[26] = sigma1(w[24]) + w[19]
w[27] = sigma1(w[25]) + w[20]
w[28] = sigma1(w[26]) + w[21]
w[29] = sigma1(w[27]) + w[22]
w[30] = sigma1(w[28]) + w[23] +sigma0(w[15])        (4-12)
```

当口令长度小于等于 23 字节时，将公式（4-10）带入到 W 的计算过程（2-8）中，可得当 t 不超过 30 时：

```
w[16] = sigma0(w[1])+w[0]
w[17] = sigma1(w[15])+sigma0(w[2])+w[1]
w[18] = sigma1(w[16])+w[2]
w[19] = sigma1(w[17])
w[20] = sigma1(w[18])
w[21] = sigma1(w[19])
w[22] = sigma1(w[20]) + w[15]
w[23] = sigma1(w[21]) + w[16]
w[24] = sigma1(w[22]) + w[17]
w[25] = sigma1(w[23]) + w[18]
w[26] = sigma1(w[24]) + w[19]
w[27] = sigma1(w[25]) + w[20]
w[28] = sigma1(w[26]) + w[21]
w[29] = sigma1(w[27]) + w[22]
w[30] = sigma1(w[28]) + w[23] +sigma0(w[15])        (4-13)
```

更进一步，消息扩展 W 在计算的过程中使用了庞大数量的内存储存，每个线程都需要保存 80 个 W 的值提供给 80 轮运算中使用。在[2]中使用了寄存器复用优化，观察 W 的计算公式（2-8）：

$$W_t = S[t] \quad t<16$$
$$W_t = \delta_1(W_{t-2}) + W_{t-7} + \delta_0(W_{t-15}) + W_{t-16} \quad 16 \leqq t \leqq 79 \qquad (2\text{-}8)$$

当 t 从超过 16 开始，每个 W_t 在计算过程中只依据它自身之前 16 个 W 值之中的某四个来进行计算，因此，我们可以在后续的 W_t 的计算过程中放弃 t’小于 t 减 16 的 Wt’的值的保存，也就是说，在任何状态下只需保存 16 个 W 值就可以依据这些 W 来计算后续的 W.因此，可以在计算过程中只使用 16 个变量来完成所有 W 的计算。我们以 W[0]至 W[15]为这 16 个变量，初始状态下这十六个变量的值是当前分组 1024 位内容的 16 个 64 位比特值，在后续的计算过程中使用如下公式：

```
w[t & 0x0F] =sigma1(w[(t -2)& 0x0F])
                + w[(t - 7)&0x0F]
            +sigma0(w[(t-15)& 0x0F])
                + w[(t - 16) & 0x0F]          (4-14)
```

这样使得只有 16 个状态得以保存。我们将（4-8）（4-9）（4-10）带入到（4-14）中，可以得到从 W_{16} 到 W_{30} 的更新过程如下，当口令长度小于等于 7 字节时：

```
w[0] =   w[0]
w[1] =   sigma1(w[15])
```

w[2] = sigma1(w[0])
w[3] = sigma1(w[1])
w[4] = sigma1(w[2])
w[5] = sigma1(w[3])
w[6] = sigma1(w[4]) + w[15]
w[7] = sigma1(w[5]) + w[0]
w[8] = sigma1(w[6]) + w[1]
w[9] = sigma1(w[7]) + w[2]
w[10] = sigma1(w[8]) + w[3]
w[11] = sigma1(w[9]) + w[4]
w[12] = sigma1(w[10]) + w[5]
w[13] = sigma1(w[11]) + w[6]
w[14] = sigma1(w[12]) + w[7] +sigma0(w[15]) (4-15)

可将第一步 w[0] =w[0]这个过程省略掉，最终的计算过程如下：

w[1] = sigma1(w[15])
w[2] = sigma1(w[0])
w[3] = sigma1(w[1])
w[4] = sigma1(w[2])
w[5] = sigma1(w[3])
w[6] = sigma1(w[4]) + w[15]
w[7] = sigma1(w[5]) + w[0]
w[8] = sigma1(w[6]) + w[1]
w[9] = sigma1(w[7]) + w[2]
w[10] = sigma1(w[8]) + w[3]
w[11] = sigma1(w[9]) + w[4]
w[12] = sigma1(w[10]) + w[5]
w[13] = sigma1(w[11]) + w[6]
w[14] = sigma1(w[12]) + w[7] +sigma0(w[15]) (4-16)

当口令长度小于等于 15 字节时，可得：

w[0] = sigma0(w[1])+w[0]
w[1] = sigma1(w[15])+w[1]
w[2] = sigma1(w[0])
w[3] = sigma1(w[1])
w[4] = sigma1(w[2])
w[5] = sigma1(w[3])
w[6] = sigma1(w[4]) + w[15]
w[7] = sigma1(w[5]) + w[0]
w[8] = sigma1(w[6]) + w[1]
w[9] = sigma1(w[7]) + w[2]

w[10] = sigma1(w[8]) + w[3]
w[11] = sigma1(w[9]) + w[4]
w[12] = sigma1(w[10]) + w[5]
w[13] = sigma1(w[11]) + w[6]
w[14] = sigma1(w[12]) + w[7] +sigma0(w[15]) (4-17)

当口令长度小于等于 23 字节时，可得：

w[0] = sigma0(w[1])+w[0]
w[1] = sigma1(w[15])+sigma0(w[2])+w[1]
w[2] = sigma1(w[0]) +w[2]
w[3] = sigma1(w[1])
w[4] = sigma1(w[2])
w[5] = sigma1(w[3])
w[6] = sigma1(w[4]) + w[15]
w[7] = sigma1(w[5]) + w[0]
w[8] = sigma1(w[6]) + w[1]
w[9] = sigma1(w[7]) + w[2]
w[10] = sigma1(w[8]) + w[3]
w[11] = sigma1(w[9]) + w[4]
w[12] = sigma1(w[10]) + w[5]
w[13] = sigma1(w[11]) + w[6]
w[14] = sigma1(w[12]) + w[7] +sigma0(w[15]) (4-18)

4.3 其它优化技术实现

4.3.1 口令生成方式优化

在口令的生成过程中仿照[15]使用了口令拼接的方式生成口令。设置口令前缀长度为两个字符，在程序初始化过程中加载到 GPU 的全局内存中。GPU 每次运行当前口令集的破解时 CPU 会同时生成口令后缀。当 GPU 完成这一组计算后若没有找到正确的口令 CPU 会将口令后缀传递到 GPU 的全局内存中，等待 GPU 开始运行时 CPU 将再次生成下一个口令后缀。

使用这种流水线的口令拼接方式可以灵活的设计口令后缀使得破解更灵活。同时信息传输量也不高，尽可能地使 GPU 保持运行状态，提高占空比。

4.3.2 指令优化优化

指令优化主要使用 OpenCL 所支持的内置指令（Built-In Functions）来代替原有的多次指令的计算。其中，bitselect 指令提供了和 SHA-512 运算过程中的 Ch(x, y, z)函数一样的功能。Ch(x, y, z)的逻辑如公式 2-9：

$$Ch(x,y,z) = (x \text{ AND } y) \oplus (\text{NOT } x \text{ AND } z) \quad (2\text{-}9)$$

它的逻辑是 X 为 1 时 Ch(x, y, z)与 y 相等，X 为 0 时 Ch(x, y, z)与 Z 相等。这样它的逻

辑正和 bitselect（z,y,x）指令的逻辑一致，使用 bitselect 指令可以将指令从四条减少到一条。

4.3.3 内存及线程优化

在程序运行前需要依据具体硬件的配置合理设置 OpenCL 环境的参数。我们使用 GPU-Z 来分析本次优化的具体 GPU 平台：NVIDIA 在 2014 年发布的 GM204 架构 GPU,Geforce GTX 970。其具体配置如图 4-1.

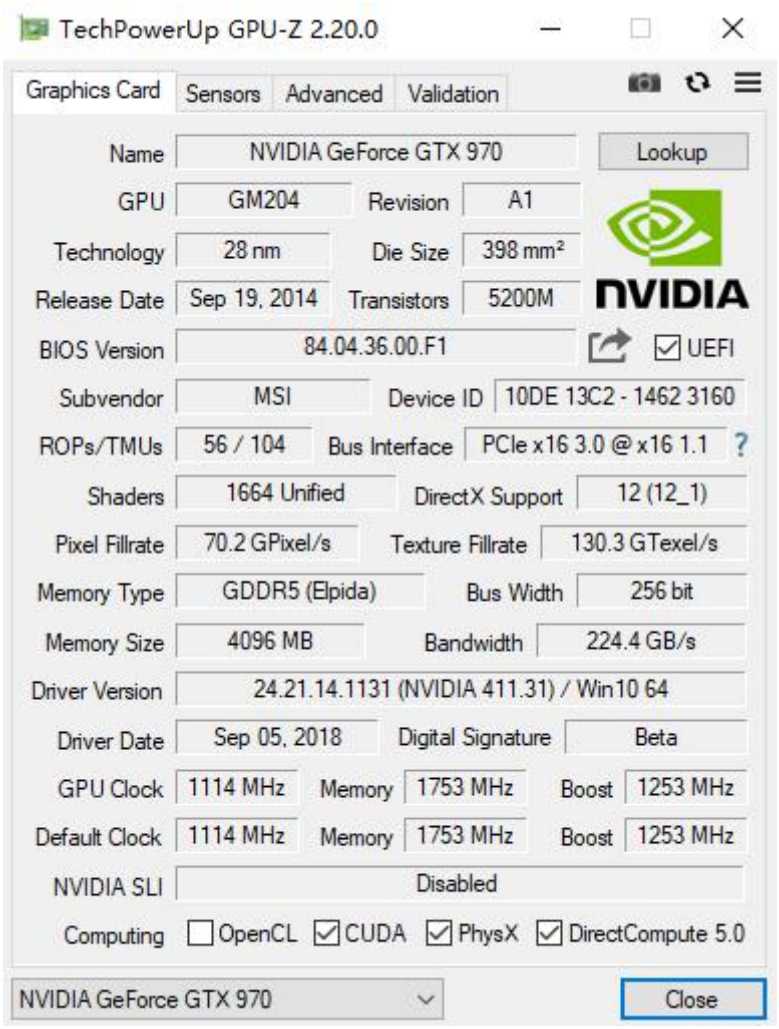


图 4-1　GTX970 的配置信息

GPU-Z 是一款专业的 GPU 配置分析性能检测软件，如图可见，GTX970 共有 1664 个计算单元，同时我们使用的口令前缀长度为 2 字节。因此，由公式可得。

$$\begin{aligned} &local\_work\_size = 32 * N \quad (N \text{ 为正整数}) \\ &global\_work\_size > charsetSize^2 \\ &global\_work\_size \,\%\, 32 == 0 \\ &global\_work\_size \,\%\, 1664 == 0 \end{aligned} \tag{4-19}$$

第五章 测试结果与分析

测试过程选用的平台为 NVIDIA 公司的 GPU，GTX970，驱动版本为 24.21.14.1131，使用驱动提供的 OpenCL 运行时环境。操作系统使用 windows 10，版本号 build 17763.1。

5.1 逆向优化的测试与分析

SHA-512 共有 80 轮运算，提前退出的过程减少了 78-80 这三轮完整的运算，使用四轮 K 运算取代了 74-77 这四轮的 P 运算。由公式 3-11

$$d \mathrel{+}= Ch(e,f,g) + SIGMA1(e) + h + K + W \tag{4-3}$$

K 运算的过程使用五次加法运算，Ch(e,f,g) 运算，SIGMA1(e)运算以及 W 的生成运算。其中，经过指令优化的 Ch(e,f,g)运算只需一条指令，而 SIGMA1()运算的展开如下：

$$SIGMA1(x)=(ROTR(x, 14) \wedge ROTR(x, 18) \wedge ROTR(x, 41)) \tag{5-1}$$

它需要两次异或运算和三次循环移位运算。W 运算如公式 3-21，寄存器优化后需要 6 次移位运算，4 次异或运算，7 次加减法运算以及 5 次按位与运算。

P 运算的计算过程如公式 3-9

$$\begin{aligned} tmp1 &= Ch(e,f,g) + SIGMA1(e) + h + K + W \\ tmp2 &= Maj(a,b,c) + SIGMA0(a) \\ d &\mathrel{+}= tmp1 \\ h &= tmp1 + tmp2 \end{aligned} \tag{3-9}$$

它需要两个临时变量寄存器，七次加法运算，Ch（运算），两次 SIGMA 运算，W 的生成运算以及一次 Maj（）运算。其他运算上面以及分析过，下面分析 Maj()运算。由公式 2-9：

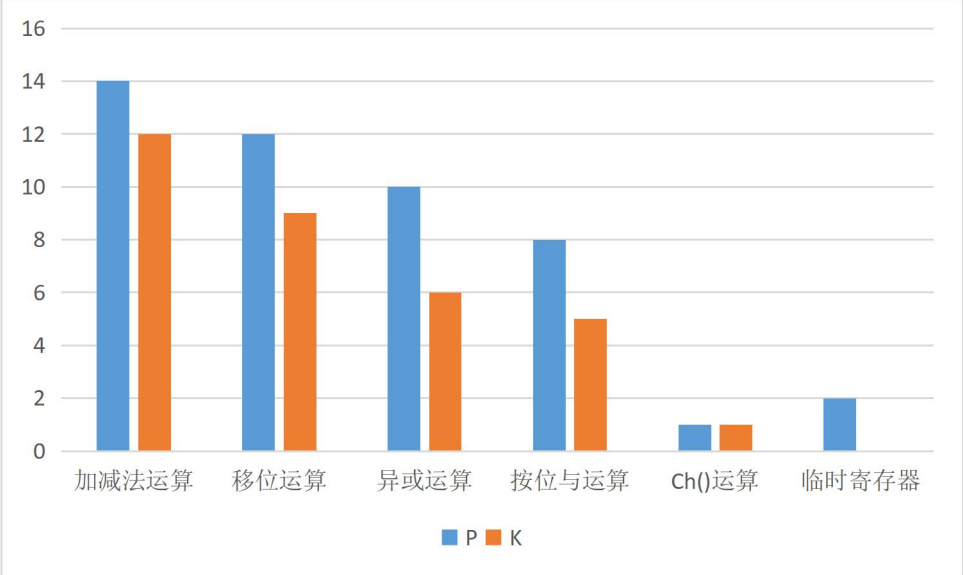
$$Maj(a,b,c) = (a \text{ AND } b) \oplus (a \text{ AND } c) \oplus (b \text{ AND } c) \tag{2-9}$$

Maj()运算需要两次加法和三次按位与运算。综上，可将 P 函数和 K 函数进行的运算进行统计，如表 5-1：

表 5-1 P 函数和 K 函数进行的运算数量对比

	加减法运算	移位运算	异或运算	按位与运算	Ch() 运算	临时寄存器
P	14	12	10	8	1	2
K	12	9	6	5	1	0

对比结果如图 5-1



5-1 函数 P,K 运算数量对比图

将 80 轮一共进行的运算数量与提前退出优化过的结果进行统计，可得表 5-2

表 5-2 提前退出优化前后运算数量对比

	加减法运算	移位运算	异或运算	按位与运算	Ch()运算	临时寄存器
80 轮 P 运算	1072	960	800	512	80	160
73 轮 P 运算 +4 轮 K 运算	1022	912	754	476	77	146
77 轮 P 运算	1030	924	770	488	77	154

对比结果如图 5-2，

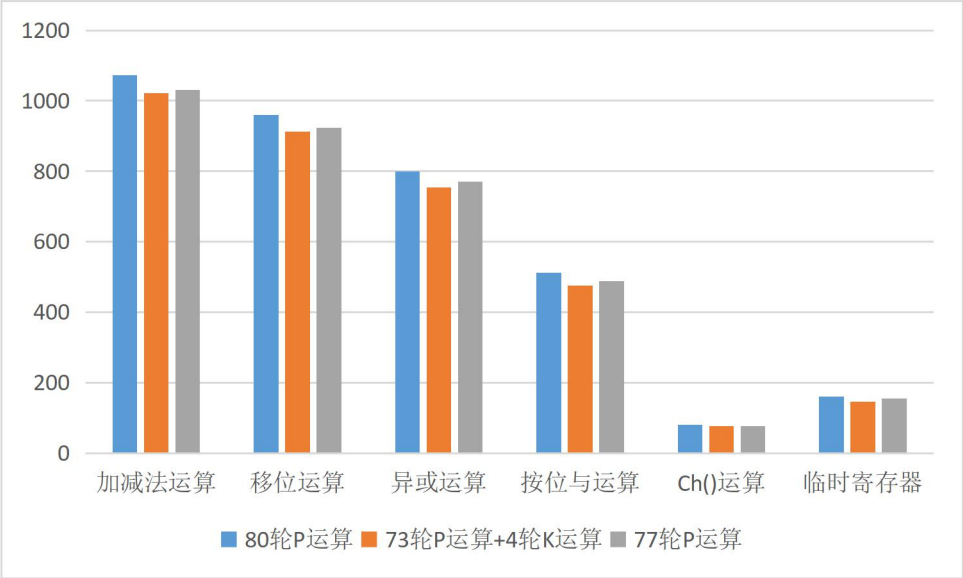


图 5-2，优化前后运算总量对比图

在 GTX970 上，对进行提前退出优化前后的性能进行了测试。当破解一百亿条长度为十字节的口令时，三者花费的平均时间如下表 5-3：

表 5-3 提前退出优化前后性能对比

	80 轮 P 运算	73 轮 P 运算+4 轮 K 运算	77 轮 P 运算
时间（s）	36.217293	**34.608374**	35.222591

优化后的提前退出算法相比于 80 轮完整算法性能提高了约 4.44%，相比于优化前的提前退出算法性能提高了约 1.74%。

5.2 短口令优化的测试与分析

短口令分为 7 字节及以下，8-15 字节以及 16-23 字节三类。目标是减少 t 介于 16 至 30 这一区间时 W_t 的计算量。根据上一节的分析，16 至 79 每轮的 W_t 的计算量为 6 次移位运算，4 次异或运算，7 次加减法运算以及 5 次按位与运算。经过优化后总的计算量统计如下。

表 5-4 短口令优化前后运算数量统计

	移位运算	异或运算	加减法运算	按位与运算
完整的 W 生成	384	256	448	320
7 字节以下优化后	339	226	353	245
8-15 字节优化后	342	228	355	245
16-23 字节优化后	345	230	357	245

对比结果如图 5-3，

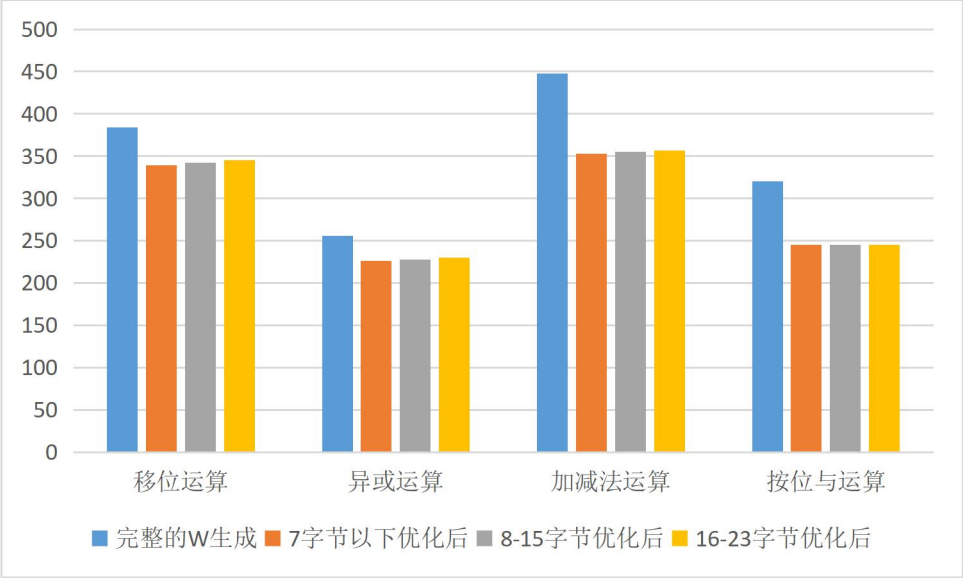


图 5-3 短口令优化前后运算数量对比

性能测试过程分别选用 100 亿个 7 字节，15 字节以及 23 字节长度的口令进行杂凑计算，优化前后时间消耗如表 5-5

表 5-5 短口令优化前后性能对比

	7 字节口令	15 字节口令	23 字节口令
优化前时间（s）	37.014784	37.012487	37.009279
优化后时间（s）	**35.613655**	**35.610773**	**35.615472**

7 字节，15 字节以及 23 字节长度的口令在优化前后性能提高分别约为 3.785%，3.787%和 3.766%。

第六章 结论

6.1 论文工作成果

本文分析了目前主流的杂凑算法的计算过程以及优化方式，基于此探究了针对当前主流杂凑算法中最安全最复杂的 SHA-512 的优化算法，主要工作成果如下：

- 优化了提前退出算法

逆向优化是杂凑算法优化的一种重要思路。目前，对于 SHA-512 提前退出的研究成果为提前三轮运算。即 77 轮运算后就可对当前口令是否正确进行判断。在本文中，我设计了轮函数 K 代替 73 至 77 轮运算使用的原轮函数 P，其中函数 K 仅完成了函数 P 中的部分运算，大大减少了运算量。因此，最终只需 73 轮完整运算加上 4 轮部分运算就可判断当前明文是否正确。

- 提出了针对短口令的正向优化

针对杂凑算法的正向优化主要使用预计算来进行。预计算就是当前进行计算的一组仅有前缀不同的口令在计算过程中可以共用部分计算，使得总的计算量下降的一种优化方式。不过，它在带来算法上优化的同时使得硬件平台在计算过程中使用了更多的寄存器，当更多的寄存器带来的性能损失超过预计算带来的性能提升时就会使得预计算得不偿失。

预计算的共用计算部分针对的是每个口令的口令扩展算法。这使得预计算同时也依赖口令扩展算法的具体运算过程。SHA-512 由于其算法相对更加复杂，使得其口令扩展算法的可重用度不高，因此预计算带来的性能提高非常有限，使得最终与寄存器性能损失相比性能下降。

针对普遍使用的预计算性能不佳，此本文设计了针对短口令的正向优化。短口令优化利用的是当口令明文转化为比特后远远不足一个分组的长度，导致需要大量使用扩展位这一特点。由于扩展位除了第一个比特外其他比特均是 0，这使得在口令扩展过程中前 16 个 W 会包含大量的 0，导致后续的 16 个 W 的计算可以大大简化，最终带来整个算法层面上的优化。

6.2 论文工作展望

对 SHA-512 破解过程中的优化可继续从逆向与正向两个方向深入进行，在这里本人提供两个未来工作的角度。

一，提前退出过程目前以一个链接变量的值确定为准，以这个变量的值与目标哈希值进行对比判断是否口令正确。SHA-512 的一个链接变量有 64 位，而排除一个错误的口令需要的位数可以远远小于 64 位。在接下来的工作中可以尝试对 512 位的结果中的连续某几位作为目标来进行提前优化，尝试提前更多的轮数。

二，对短口令的优化目前仅仅局限于短口令，也就是 23 个字节及以下的口令。如果期望获得对单个分组下的任意长度字节口令都适用的优化可以考虑优化目前的预计算的实现。比如可以重新设计口令生成的流水线结构，将口令生成分为前缀，中缀以及后缀三部分，以更多口令为一组进行预计算，提高预计算的利用率。

三，目前算法仅在单个 GPU 的平台上进行运算，当 GPU 资源更加充足时可以结合分布式应用来进行哈希函数在 GPU 集群上的破解研究[27]。

参考文献

[1] 郑东,李详学,黄征,郁昱. 密码学—密码算法与协议[M]. 北京:电子工业出版社,2014:
[2] Kaliski B. The MD2 message-digest algorithm[R]. 1992.
[3] Rivest R. The MD4 message-digest algorithm[R]. 1992.
[4] Rivest R. The MD5 message-digest algorithm[R]. 1992.
[5] Chabaud F, Joux A. Differential collisions in SHA-0[C]//Annual International Cryptology Conference. Springer, Berlin, Heidelberg, 1998: 56-71.
[6] Wang X, Yin Y L, Yu H. Finding collisions in the full SHA-1[C]//Annual international cryptology conference. Springer, Berlin, Heidelberg, 2005: 17-36.
[7] 荣凯, 邱卫东, 李萍. 基于彩虹表的 Hash 攻击研究[D]. , 2011.
[8] 王张宜, 李波, 张焕国. Hash 函数的安全性研究[D]. , 2005.
[9] 刘美, 王玉柱, 何定养, et al. SHA-512 算法及其基于生日攻击的安全性分析[J]. 后勤工程学院学报, 2010, 26(3):92-96.
[10] Wang X, Yu H. How to break MD5 and other hash functions[C]//Annual international conference on the theory and applications of cryptographic techniques. Springer, Berlin, Heidelberg, 2005: 19-35.
[11] 武金梅. 对缩短步数的 HASH 函数算法 SHA-256, SHA-512 的分析[D]. 山东大学, 2008.
[12] Qiu W, Gong Z, Guo Y, et al. GPU-Based High Performance Password Recovery Technique for Hash Functions[J]. J. Inf. Sci. Eng., 2016, 32(1): 97-112.
[13] 桑海, 李建宝. 加密算法 MD5 的研究与应用[D]. , 2006.
[14] Eastlake 3rd D, Jones P. US secure hash algorithm 1 (SHA1)[R]. 2001.
[15] Gilbert H, Handschuh H. Security analysis of SHA-256 and sisters[C]//International workshop on selected areas in cryptography. Springer, Berlin, Heidelberg, 2003: 175-193.
[16] Adinetz A V , Grechnikov E A . Building a collision for 75-round reduced SHA-1 using GPU clusters[C]// International Conference on Parallel Processing. Springer-Verlag, 2012.
[17] 席胜鑫, 周清雷, 斯雪明, et al. 可重构计算平台上 SHA 系列函数的优化实现[J]. 计算机应用研究, 2018, v.35；No.321(07):258-261.
[18] 席胜鑫, 张文宁, 周清雷, et al. 基于拟态计算机的 SHA512 算法高吞吐量实现[J]. 计算机工程与科学, 2018, v.40；No.284(08):12-18.
[19] Grembowski T, Lien R, Gaj K, et al. Comparative Analysis of the Hardware Implementations of Hash Functions SHA-1 and SHA-512[M]// Origins of the humanistic tradition :. 2002.
[20] 张健, 陈瑞, 芮雄丽. 安全散列算法 SHA-1 在图形处理器上的实现[J]. 网络安全技术与应用, 2009(1):46-48.
[21] Stone J E, Gohara D, Shi G. OpenCL: A parallel programming standard for heterogeneous computing systems[J]. Computing in science & engineering, 2010, 12(3): 66.
[22] Munshi A. The opencl specification[C]//2009 IEEE Hot Chips 21 Symposium (HCS). IEEE, 2009: 1-314.

[23] Lindholm E, Nickolls J, Oberman S, et al. NVIDIA Tesla: A unified graphics and computing architecture[J]. IEEE micro, 2008, 28(2): 39-55.

[24] Taylor R, Li X. A micro-benchmark suite for AMD GPUs[C]//2010 39th International Conference on Parallel Processing Workshops. IEEE, 2010: 387-396.

[25] Can Ge, Lingzhi Xu, Weidong Qiu 等. Optiized Password Recovery for SHA-512 on GPUs [C]. 2017

[26] 李鸿强, 苗长云, 石博雅, 等. 单向散列函数 SHA-512 的优化设计[D]. , 2007.

[27] 分布式：姜峰. 基于分布式 GPU 密码破译平台的研究与实现[D]. 北京邮电大学, 2013.

[28] Ahmad I, Das A S. Hardware implementation analysis of SHA-256 and SHA-512 algorithms on FPGAs[J]. Computers & Electrical Engineering, 2005, 31(6):345-360.

[29] Ahmad I, Das A S. Analysis and Detection Of Errors In Implementation Of SHA-512 [27] Algorithms On FPGAs[M]// Analysis and detection of errors in implementation of SHA-512 algorithms on FPGAs. 2007.

[30] 光焱, 祝跃飞, 吴树华,等. 一种基于 FPGA 的 SHA-512 算法高速实现[J]. 信息工程大学学报, 2008, 9(1):94-96.

谢辞

经过一学期的努力，这篇论文终于顺利完成。在这六个月的时间内我学习了 OpenCL 的相关编程技术以及 GPU 架构相关的知识以及学习了主流杂凑算法的计算过程。

在这期间首先要感谢邱卫东老师与王杨德学长，邱卫东老师的现代密码学这门课让我对密码学相关协议与算法产生了兴趣，并最终选择了这个方向的毕业设计。王杨德学长在我学习的过程中提供了巨大的指导与帮助，让我能顺利的完成论文。

其次要感谢我的同学倪星，我的同事 Lydia、陈文广以及林玮胜等人，是你们的照顾与支持让我能够心无旁骛的进行研究与论文撰写。

最后，对这四年来所有相处过的老师表示感谢，没有你们教给我的计算机科学基础知识我是没有办法完成这篇文章的，在此对你们表达感谢。

RESEARCH ON REVERSE OPOTIMIZATION OF SHA512

With the rapid development of information technology, information security has become an important requirement of information systems. Among them, confidentiality and security certification are two major issues to be solved by information security systems. As the most widely used message security authentication method, hash function has received more and more attention in recent years. One of the most common hash functions: Secure Hash Algorithm (SHA) is a family of cryptographic hash functions proposed by the National Security Agency (NSA) National Institute of Standards and Technology (NIST), published as a federal information processing standard in 1993. SHA512 was officially released in 2002 as one of the second revisions of the SHA series. This paper mainly studies the reverse optimization part of the SHA512 cracking process, and combines other optimizations to complete the procedure of finding the plaintext of the hash value. The research process refers to other SHA series hash functions and the crack optimization of MD5 functions. OpenCL (full name Open Computing Language) is an open standard for parallel programming for general purpose heterogeneous systems. In this research process, OpenCL is used to perform parallel programming on GPU to implement the brute force cracking process and apply reverse optimization technology. Finally, the parallel computing and the optimization of the algorithm and GPU are used to realize the SHA512 brute force to shorten the given hash time. Finally, the result is compared with the mainstream hash function cracking software to analyze the optimization effect and the future. The direction of further research can be prospected.

The hash algorithm, also known as the hash function, is an algorithm that can compress messages of any length into a fixed-length digest. The earliest publicly available hash function that is currently traceable is the MD2 algorithm published in 1989 by American cryptographer Ronald Linn Rivest, which was designed for 8-bit computers. Then on the basis of MD2, in the next two years, Rivest published MD4 and MD5 hash algorithms. Among them, MD5 algorithm is more perfect in maturity and security, and becomes the most widely used hash algorithm in the next few years. In 1993, the SHA series hash algorithm developed based on MD5 was released by the National Institute of Standards and Technology as the US government standard. The first-generation SHA series hash algorithm became the SHA-0 generation, and was released shortly after the release due to security issues. The revised SHA-1 series of hash functions released in 1995, after which MD5 and SHA-1 became the most widely used hash algorithm. In 2002, the SHA-2 family of algorithms announced that the SHA-2 family had three members: SHA-256, SHA-384, and SHA-512.

Over time, the security of the MD5 and SHA-1 hash algorithms has been widely challenged. Prior to 2000, differential attacks proved to be effective for a single cycle of MD5. In 2004, the results of effective attack research on SHA-0 were announced. In view of the attack on SHA-0, SHA-1 similar to SHA-0 is also recommended to be used with caution. In 2004, the National Institute of Standards and Technology officially announced that it would gradually replace SHA-1

with SHA-2.

The current hash function is widely used, and various hash functions of various algorithms come out in an endless stream. In this paper, the SHA-512 with the largest abstract length in the most widely used SHA series hash function is selected as the research object. According to the algorithm implementation and optimization process analysis of MD5 and SHA series hash algorithm, the reverse optimization of SHA-512 is given. Other optimizations. And use the OpenCL implementation to complete the plaintext lookup for the given hash value. Parallel computing on the GPU completes the comparative analysis of the test results.

This paper analyzes the calculation process and optimization method of the current mainstream hash algorithm, and optimizes the most secure and complex SHA-512 optimization algorithm for the current mainstream hash algorithm. Optimization implementation is mainly divided into improvements to existing reverse optimization and design optimization for short passwords. Reverse optimization is an important idea of hash algorithm optimization. At present, the research result of the early exit of SHA-512 is three rounds of calculation in advance. That is, after 77 rounds of calculation, it is possible to judge whether the current password is correct or not. In this paper, I designed the round function K instead of the original round function P used in the 73 to 77 round operation, in which the function K only completes part of the operation in the function P, which greatly reduces the amount of calculation. Therefore, in the end, only 73 rounds of complete operations plus 4 rounds of partial operations can be used to determine whether the current plaintext is correct.

The forward optimization for the hash algorithm is mainly performed using precomputation. The pre-calculation is an optimization method in which a set of passwords with only different prefixes can be shared in the calculation process, so that the total calculation amount is reduced. However, it brings algorithmic optimization while making the hardware platform use more registers in the calculation process. When the performance loss caused by more registers exceeds the performance increase brought by the pre-computation, the pre-calculation will not be worth the loss.

The pre-computed shared computing portion is for the password extension algorithm for each password. This allows precomputation to also rely on the specific computational process of the password extension algorithm. Due to its relatively more complex algorithm, SHA-512 makes its password extension algorithm less reusable, so the performance improvement caused by pre-computation is very limited, which ultimately results in performance degradation compared with register performance loss.

In view of the poor precomputation performance commonly used, this paper designs a forward optimization for short passwords. Short password optimization utilizes the length of a packet that is far less than one packet after the plaintext is converted into bits, resulting in the need to use a large number of extension bits. Since the extension bits are 0 except for the first bit, this makes the first 16 Ws contain a large number of 0s during the password extension process, which results in the subsequent 16 W calculations being greatly simplified, resulting in the entire algorithm level. Optimization on the top.

From the final result, the performance of early exit is 4.44% higher than that without early exit, which is 1.74% higher than the original early exit optimization. The optimization for short passwords improves performance by about 3.7% in the case of short passwords.

The optimization of the SHA-512 cracking process can continue in depth from the reverse direction and the forward direction. Here I provide two perspectives for future work. First, the early exit process is currently based on the value of a link variable, and the value of this variable is compared with the target hash value to determine whether the password is correct. A link variable for SHA-512 has 64 bits, and the number of bits required to exclude a wrong password can be much smaller than 64 bits. In the next work, you can try to optimize the number of consecutive 512-bit results as a target, and try to advance more rounds. Second, the optimization of short passwords is currently limited to short passwords, that is, passwords of 23 bytes or less. Optimizations of current precomputation implementations can be considered if it is desired to obtain an optimization that is applicable to any length byte password under a single packet. For example, the pipeline structure generated by the password can be redesigned, and the password generation is divided into three parts: a prefix, an infix and a suffix. And a pre-computation is performed with more passwords as a group to improve the utilization of the pre-calculation.