



吉林大学

JILIN UNIVERSITY

本科生毕业论文(设计)

中文题目 RSA 与 DSA 加密算法的研究与实现

英文题目 Research and Implementation of

RSA and DSA Encryption Algorithms

学生姓名 王逸飞

学 号 55200831

学 院 软件学院

专 业 软件工程

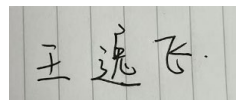
指导教师 刘亚波

2024 年 06 月

吉林大学学士学位论文（设计）承诺书

本人郑重承诺：所呈交的学士学位毕业论文（设计），是本人在指导教师的指导下，独立进行实验、设计、调研等工作基础上取得的成果。除文中已经注明引用的内容外，本论文（设计）不包含任何其他个人或集体已经发表或撰写的作品成果。对本人实验或设计中做出重要贡献的个人或集体，均已在文中以明确的方式注明。本人完全意识到本承诺书的法律结果由本人承担。

学士学位论文（设计）作者签名：

A rectangular box containing a handwritten signature in black ink. The signature is written in a cursive style and appears to read '王逸飞' (Wang Yifei).

2024 年 6 月 1 日

RSA 与 DSA 加密算法的研究与实现

摘要

RSA 算法是成为公钥加密领域的基石，它依赖于大数分解的难度来保证安全。相比之下，DSA 算法则基于离散对数问题，设计重点在于高效生成数字签名，虽在签名时速度快，但验证速度相对较慢。这两种算法在现代密码学中应用广泛，但面临量子计算的潜在威胁，传统加密方法正面临革新需求。

论文使用 C++ 语言实现 RSA 算法和 DSA 算法。从两个算法的数学基础展开分析研究，本论文分析了近些年国内外对于 RSA 和 DSA 的研究现状并给出了自己的相关算法实现。论文的主要内容包括：

首先，通过理论分析与实践编程，本论文剖析了两种不同的算法之间的理论基础，包括密钥生成、加密解密过程及其安全性基础，并讨论其在确保信息传输机密性与完整性方面的优势。

其次，在使用 Crypto++、Botan 和 OpenSSL 这三种加密库分别模拟实现了 RSA 和 DSA 算法后，重点分析了 OpenSSL 在 RSA 算法实现方面的源码以及架构，研究实施了 RSA 的 C++ 代码实现与 DSA 算法的 C++ 伪代码实现。在实际编码过程中，我进行了密钥生成、加密、解密和数字签名的实现和测试。

最后，在分析了量子计算机和量子算法对现有的加密算法带来的威胁之后，并利用 Python 代码的形式模拟实现了如何分别使用 Shor 算法和 Grover 算法实现 RSA 和 DSA 的量子破解算法，即整数分解的过程和离散对数问题求解，证明在新的计算机体系下，传统的加密算法安全性不再。

这些算法均在 ubuntu 20.04 上使用 gcc13 开发实现。通过本课题的研究与实践，不仅加深了对 RSA 和 DSA 两种经典非对称加密技术的理解，更证明了开发新型量子加密算法的重要性。

关键词：

RSA 算法，数字签名，加密算法

Research and Implementation of RSA and DSA Encryption Algorithm

Author: Wang Yifei

Supervisor: Liu Yabo

Abstract

The RSA algorithm is the cornerstone of public key encryption, relying on the difficulty of large number decomposition to ensure security. In contrast, the DSA algorithm is based on the discrete logarithm problem, with a focus on efficiently generating digital signatures. Although it has a fast signing speed, the verification speed is relatively slow. These two algorithms are widely used in modern cryptography, but they face potential threats from quantum computing, and traditional encryption methods are facing a need for innovation.

The paper uses C++ language to implement RSA algorithm and DSA algorithm. Starting from the mathematical foundations of the two algorithms, this paper analyzes the current research status of RSA and DSA both domestically and internationally in recent years, and provides its own implementation of relevant algorithms. The main content of the paper includes:

Firstly, through theoretical analysis and practical programming, this paper analyzes the theoretical basis between two different algorithms, including key generation, encryption and decryption processes, and their security foundations, and discusses their advantages in ensuring the confidentiality and integrity of information transmission.

Secondly, after simulating the implementation of RSA and DSA algorithms using three encryption libraries: Crypto++, Botan, and OpenSSL, the source code and architecture of OpenSSL in RSA algorithm implementation were analyzed, and the C++ code implementation of RSA and the C++ pseudocode implementation of DSA algorithm were studied. In the actual coding process, I implemented and tested key generation, encryption, decryption, and digital signature.

Finally, after analyzing the threats posed by quantum computers and quantum

algorithms to existing encryption algorithms, and using Python code to simulate and implement quantum cracking algorithms for RSA and DSA using the Shor algorithm and Grover algorithm respectively, namely the process of integer division and the solution of discrete logarithm problems, it is proved that under the new computer system, traditional encryption algorithms are no longer secure.

These algorithms were developed and implemented using gcc13 on Ubuntu 20.04. Through the research and practice of this project, we have not only deepened our understanding of RSA and DSA, two classic asymmetric encryption technologies, but also demonstrated the importance of developing new quantum encryption algorithms.

Keyword:

RSA algorithm, Digital signature, Encryption algorithm

目 录

第 1 章 绪论	1
1.1 研究背景	1
1.2 国内外研究现状	2
1.2.1 RSA 研究现状.....	2
1.2.2 DSA 研究现状.....	2
1.3 论文的主要工作	3
1.4 论文的组织架构	4
第 2 章 相关数学基础、RSA 算法与 DSA 算法.....	5
2.1 相关数学基础	5
2.1.1 大数因子分解问题	5
2.1.2 离散对数难题	5
2.2 密码学简述	6
2.2.1 密码学基础	6
2.2.2 加密算法描述	6
2.3 RSA 算法.....	7
2.3.1 RSA 算法内容.....	7
2.3.2 RSA 算法的安全性.....	9
2.3.3 RSA 算法的应用.....	10
2.4 DSA 算法.....	11
2.4.1 DSA 算法内容.....	11
2.4.2 DSA 算法的安全性.....	12

2.4.2 DSA 算法的应用	13
第 3 章 OpenSSL、Botan、Crypto++ 算法与自实现算法	16
3.1 OpenSSL 库与 RSA、DSA 实现	16
3.1.1 OpenSSL 库	16
3.1.2 OpenSSL 中 RSA 实现源码分析	16
3.1.3 OpenSSL RSA 算法实现	17
3.1.4 OpenSSL DSA 算法实现	22
3.2 Botan 库与 RSA、DSA 实现	26
3.2.1 Botan 库	26
3.2.2 Botan RSA 算法实现	26
3.2.3 Botan DSA 算法实现	29
3.3 Crypto++ 库与 RSA、DSA 实现	32
3.3.1 Crypto++ 库	32
3.3.2 Crypto++ 库 RSA 算法实现	32
3.3.3 Crypto++ 库 DSA 算法实现	35
3.4 RSA、DSA 自实现	37
3.4.1 RSA 算法的 C++ 实现	37
3.4.2 DSA 算法的 C++ 实现	40
第 4 章 量子计算对加密算法的影响	42
4.1 量子计算背景	42
4.2 量子计算基础	42
4.3 量子叠加和量子纠缠	42

4.4 Shor 算法.....	43
4.4.1 Shor 算法的步骤.....	43
4.4.2 使用 Python 模拟实现 Shor 算法	44
4.5 Grover 算法.....	47
4.5.1 Grover 算法的工作原理.....	47
4.5.2 Grover 算法的步骤.....	47
4.5.3 使用 Python 模拟实现 Grover 算法	48
第 5 章 总结与展望	50
5.1 RSA 算法的发展前景.....	50
5.2 DSA 算法的发展前景.....	51
5.3 量子计算	52
参考文献	54
致 谢	55

第1章 绪论

1.1 研究背景

随着全球信息化的加速发展，网络空间已成为现代社会不可或缺的一部分，人们在享受信息技术带来的便利的同时，也面临着日益严峻的信息安全挑战。从个人隐私泄露到企业数据被盗，再到国家关键基础设施的安全威胁，信息安全问题已经成为制约数字时代健康发展的一大瓶颈。在此背景下，加密技术，尤其是非对称加密算法，作为保护信息安全的核心手段，其重要性愈发凸显。

RSA 算法与 DSA 算法作为非对称加密技术的两大支柱，自诞生以来，便在全球范围内得到了广泛应用。RSA 算法，作为一种公钥加密标准，以其强大的加密能力和数学难题保证的安全性，在数据加密、安全通信协议、数字签名等领域扮演着关键角色。而 DSA 算法，作为美国国家标准与技术研究院（NIST）推荐的数字签名算法，以其高效性和安全性，在电子交易、身份认证、软件验证等方面展现了独特优势。

然而，随着计算技术的进步，尤其是量子计算的崛起，RSA 和 DSA 等基于传统密码学原理的算法正面临前所未有的挑战。量子计算机在理论上能够高效破解当前广泛使用的非对称加密系统，这迫使信息安全领域迫切需要对现有算法进行重新评估，并探索后量子密码学解决方案。

此外，随着云计算、物联网、大数据等新兴技术的应用普及，数据量爆炸式增长，对加密算法的效率和可扩展性提出了更高要求。如何在保证安全性的同时，提高算法的处理速度和降低资源消耗，成为亟待解决的问题，相关学者就对 RSA 算法在私有云情况下做出了研究的特化^[1]。

因此，本研究在这样的时代背景下展开，旨在深入剖析 RSA 与 DSA 算法的基本原理、安全机制及其在现代信息技术环境下的适用性与局限性。通过比较分析两种算法在不同应用场景中的性能表现，评估其面对未来安全威胁的韧性，本研究期望能为信息安全领域提供重要的理论参考与实践指导，同时应用代码的形式，尝试立足未来，畅想身处于后量子时代的我们，应该如何应对量子计算机和量子力学对于经典计算机加密算法的挑战。

1.2 国内外研究现状

1.2.1 RSA 研究现状

RSA 算法(Rivest - Shamir - Adleman)作为一种经典的非对称加密算法,一直受到国内外研究者的广泛关注和深入研究。在国内,RSA 算法的研究现状主要体现在以下几个方面:

首先,国内学术界对 RSA 算法的理论基础和数学原理进行了深入研究。研究者对 RSA 算法的安全性、复杂度以及密钥长度等进行了理论分析和探讨,以提高 RSA 算法在实际应用中的安全性和效率,更精确的来说:提高 RSA 算法中大数模乘运算速率^[2]。

其次,国内在 RSA 算法的应用领域也进行了一系列研究。随着互联网的普及和电子商务的发展,RSA 算法在网络通信、信息安全和数字签名等领域得到了广泛应用。研究者致力于优化 RSA 算法在这些领域的应用效果,提高系统的安全性和性能。

此外,国内还开展了一些针对 RSA 算法的改进和优化研究。例如,针对 RSA 算法的加密速度较慢和密钥长度较长的缺点,研究者提出了一些改进的算法和优化方案,以提高 RSA 算法的性能和效率^{[3][4][5]}。

在国外,RSA 算法的研究同样活跃。国外学术界在 RSA 算法的理论研究、应用探索以及改进优化等方面取得了许多成果。与国内类似,国外的研究者也致力于提高 RSA 算法的安全性、性能和适用性,以满足不断增长的信息安全需求。

总的来说,国内外对 RSA 算法的研究主要集中在理论探索、应用拓展和算法优化等方面,以不断提升 RSA 算法在信息安全领域的地位和作用。

1.2.2 DSA 研究现状

DSA (Digital Signature Algorithm) 作为专为数字签名设计的算法,其安全性基于离散对数问题,相比 RSA,在某些特定应用场景下展现出更高的效率。

在国内,DSA 算法的研究主要集中在以下几个方面:

首先,国内学术界对 DSA 算法的理论基础进行了深入研究。研究者关注 DSA 算法的数学原理、安全性、效率等方面,不断探索其在实际应用中的优势和局限性,并提出改进方案。

其次，国内在 DSA 算法的应用领域进行了一系列研究。DSA 算法在数字签名、认证、电子支付等方面具有重要应用价值，国内研究者积极探索 DSA 算法在这些领域的应用方法和技术，以提高系统的安全性和效率。

此外，国内还开展了一些针对 DSA 算法的改进和优化研究。例如，针对 DSA 算法的性能问题，研究者提出了一些对于素数选取的优化设计^[6]，以提高 DSA 算法在实际应用中的性能表现。

在国外，DSA 算法的研究同样活跃^[7]。国外学术界在 DSA 算法的理论研究、应用探索以及改进优化等方面取得了许多成果。与国内类似，国外的研究者也致力于提高 DSA 算法的安全性、性能和适用性，以满足不断增长的信息安全需求。

总的来说，国内外对 DSA 算法的研究主要集中在理论探索、应用拓展和算法优化等方面，以不断提升 DSA 算法在信息安全领域的地位和作用。

综上所述，RSA 与 DSA 的研究现状表明，虽然这两项技术已相对成熟，但面对技术进步和新兴安全挑战，国内外学者仍在不断深化对它们的理解，探索算法的优化路径。同时，如何在新环境中更好地应用这些算法，同时积极准备向后量子密码学过渡。

1.3 论文的主要工作

本文主要工作是通过研究 RSA 和 DSA 算法的基本数学原理和非对称加密的原理基础，借鉴多种开源库的算法实现，给出 RSA 和 DSA 的算法实现。同时讨论量子计算对加密算法安全性的威胁。本文主要研究内容如下：

第一，通过理论分析与实践编程，本研究剖析了两种不同的算法之间的理论基础，包括密钥生成、加密解密过程及其安全性基础，并讨论其在确保信息传输机密性与完整性方面的优势。

第二，在使用 Crypto++、Botan 和 OpenSSL 这三种加密库分别模拟实现了 RSA 和 DSA 算法后，研究实施了 RSA 与 DSA 算法的 C++ 代码实现。在实际编码过程中，我进行了密钥生成、加密、解密和数字签名的实现和测试。

第三，在分析了量子计算机和量子算法对现有的加密算法带来的威胁之后，并利用 Python 代码的形式模拟实现了如何分别使用 Shor 算法和 Grover 算法实现 RSA 和 DSA 的量子破解算法，即整数分解的过程和离散对数问题求解，证明在新的计算机体系下，传统的加密算法安全性不再。

1.4 论文的组织架构

第1章 介绍了本文研究背景、国内外研究现状、主要工作。

第2章 介绍 RSA 密码算法和 DSA 签名算法涉及到的相关数学基础、密码学基础。介绍 RSA 密码算法和 DSA 签名算法具体流程，并且分析他们的安全性和目前他们各自的流行领域。

第3章 分析三种加密开源库 OpenSSL、Botan 和 Crypto++，对三种加密库下如何分别实现 RSA 加密字符串以及如何实现 DSA 签名做出示例。重点分析 OpenSSL 开源库中涉及 RSA 加解密部分以及相关的架构设计。使用 C++ 语言，设计并编写了 RSA 算法。使用 Miller-Rabin 测试和筛选法来生成大素数，提高了素数生成的效率和可靠性。实现拓展欧几里得算法用于计算 e 模 $\phi(n)$ 的乘法逆元 d ，实现模幂运算用于加解密操作。同时对消息进行哈希处理，确保即使是相同的明文也会生成不同的密文。同时设计了 DSA 算法伪代码，在代码中清晰地分离出 DSA 的关键步骤，包括参数生成、密钥生成、签名生成和签名验证，并对每一步骤进行了详细的说明，使得整体流程非常明确。

第4章 在分析了量子计算机和量子算法对现有的加密算法带来的威胁之后，并利用 Python 代码的形式模拟实现了如何分别使用 Shor 算法和 Grover 算法实现 RSA 和 DSA 的量子破解算法，即整数分解的过程和离散对数问题求解，证明在新的计算机体系下，传统的加密算法安全性不再。

第5章 展望对 RSA 和 DSA 算法在后量子时代的发展。

第2章 相关数学基础、RSA 算法与 DSA 算法

2.1 相关数学基础

2.1.1 大数因子分解问题

因子分解问题是指将一个合数 n 分解成质因子的乘积。例如，给定 $n = 91$ ，其质因子分解为 $91 = 7 \times 13$ 。对于较小的数，这个过程可以通过简单的试除法完成，但对于大数，这变得极为复杂。

试除法是一种最基本的因数分解方法，即逐个试验所有小于 \sqrt{n} 的质数，看看它们是否是 n 的因子。这种方法的时间复杂度为 $O(\sqrt{n})$ ，对于大数来说，这个计算量是不可行的。例如，对于一个 200 位的数，其平方根仍然是一个 100 位的数，这意味着需要试验的质数数量极大。

尽管有一些比试除法更高效的算法，比如 Fermat 质数分解法，Pollard's rho 算法，椭圆曲线因子分解法（ECM）等，但这些算法的时间复杂度仍然很高。

大数因子分解的困难性主要源于试除法和其他现有算法在面对极大数时的高时间复杂度。尽管不断有新的算法被提出，但随着数的大小增加，因数分解问题依然是一个极其艰难的数学挑战。离散对数难题的困难性在于它没有已知的高效算法来解决，这使得它成为许多密码系统的基础。

2.1.2 离散对数难题

离散对数难题是指在给定一个有限域 Z_p 和一个生成元 g 的情况下，找到一个整数 x ，使得 $g^x \equiv h(\text{mod } p)$ 成立，其中 h 是 Z_p 中的一个元素。例如，给定 $g = 2, h = 3$ ，在 Z_7 中，找到 x ，使得 $2^x \equiv 3(\text{mod } 7)$ 。在这种情况下， $x = 3$ 因为 $2^3 \equiv 8 \equiv 1(\text{mod } 7)$ 。

离散对数难题的困难性在于它缺乏像连续对数那样的高效计算方法。在连续对数中，我们可以利用对数表或计算器快速找到对数值，但离散对数没有类似的简单算法。

现有的一些算法可以用于计算离散对数，但它们在大多数情况下都是不切实际的。例如，蛮力算法直接尝试所有可能的 x ，直到找到 $g^x \equiv h(\text{mod } p)$ ，这种方法的时间

间复杂度是 $O(p)$ ，对于大 p 来说是不可行的。

总的来说，离散对数难题的困难性在于它没有已知的高效算法来解决，这使得它成为许多密码系统的基础。

2.2 密码学简述

2.2.1 密码学基础

尽管密码学有着悠久的历史，但一般人对它仍然相当陌生，因为它主要在军事、情报、外交等敏感部门的小范围内使用。密码学则是研究计算机信息加密、解密及其变换的科学，涵盖了数学和计算机领域，是一门新兴的交叉学科。密码技术的主要目标在于保障电子数据的保密性、完整性和真实性。保密性指的是对数据进行加密，使得非法用户无法理解数据信息，而合法用户则可以通过密钥读取信息。完整性则是对数据完整性的验证，以确认数据是否被非法篡改，从而保证合法用户能够获取正确、完整的信息。真实性则是对数据来源及其内容的真实性进行验证，以确保合法用户不会受到欺骗。

2.2.2 加密算法描述

在加密算法中，存在着两种主要类型：对称加密和非对称加密。对称加密采用同一密钥进行加密和解密，通信的双方共享这个密钥。无论是信息的发送方还是接收方，都使用同一个密钥来进行加密和解密。然而，如果这个密钥在传输过程中被第三方获取，那么对信息加密所使用的密钥就会失去意义。此外，对称密钥的传输方式也是这种算法的不足之处。

相比之下，非对称加密算法如 RSA 则需要两个密钥：一个是公开的密钥^[8]，另一个是私密的密钥。这两个密钥是成对出现的，如果使用公开密钥对数据进行加密，则只有使用相对应的私密密钥才能进行解密；反之亦然。由于加密和解密使用的是两个不同的密钥，因此这种算法被称为非对称加密算法。经典描述公钥加密和密钥交换的过程如图 2-1 所示。

在简单的流程中，甲方生成一对密钥，并将其中一把作为公开密钥向其他方公开；乙方得到该公开密钥后使用它对机密信息进行加密，然后发送给甲方；甲方再使用自己保存的另一把专用密钥对加密后的信息进行解密。甲方只能使用其专用密钥来解密由其公开密钥加密后的任何信息。

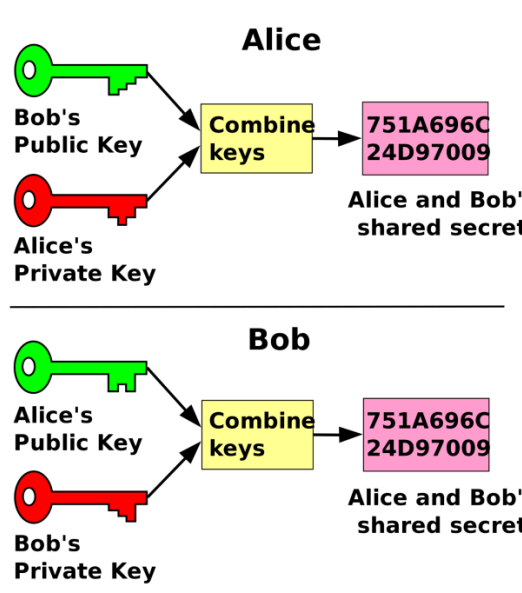


图 2- 1 Alice & Bob 问题

2.3 RSA 算法

2.3.1 RSA 算法内容

RSA 加密算法是一种非对称加密技术。RSA 算法基于数论中的大数因子分解难题，即寻找两个大质数的乘积相对容易，但分解该乘积回原质数则极其困难。这一特性确保了加密的安全性。以下是 RSA 加密算法的基本算法：

(1) 密钥生成：

- ① **选择两个大质数：**随机选取两个足够大的质数 p 和 q 。
- ② **计算模数：**计算这两个质数的乘积 $n = p * q$ ， n 将作为公钥和私钥的模数使用。
- ③ **计算欧拉函数 $\phi(n)$ ：** $\phi(n) = (p - 1) * (q - 1)$ 。
- ④ **选择公钥指数：**选取一个与 $\phi(n)$ 互质的整数 e ，满足 $1 < e < \phi(n)$ 通常 e 取值为 65537（因为它满足互质且加密效率较高）。
- ⑤ **计算私钥指数：**找到一个整数 d 作为 e 的逆模元，使得 $e \times d \equiv 1 \pmod{\phi(n)}$ 。

$1 \bmod \varphi(n)$ 。这一步可以通过扩展欧几里得算法实现， d 成为私钥的一部分。

⑥ **公钥:** (n, e) 作为公钥对外公布, 任何人都可以用这对公钥对消息进行加密。

⑦ **私钥:** (n, d) 作为私钥保密存储, 只有持有者可以用来解密信息。

(2) **加密过程:**

假设用户 A 要发送一条消息 M (M 需要转换成整数形式, 通常通过编码如 ASCII 或 UTF-8 然后映射为数字) 给用户 B。

用户 B 使用其公钥 (n, e) 对消息 M 进行加密, 计算密文 $C = M^e \bmod n$ 。

(3) **解密过程:**

用户 A 接收到密文 C 后, 使用其私钥 (n, d) 进行解密, 计算原文 $M = C^d \bmod n$ 。

由于 $e \times d \equiv 1 \bmod \varphi(n)$, 解密过程实际上恢复了原始消息。

如图 2-2 所示, RSA 算法的目标就是生成公钥和私钥, 隐藏在“初始化密钥对”这一步的背后。

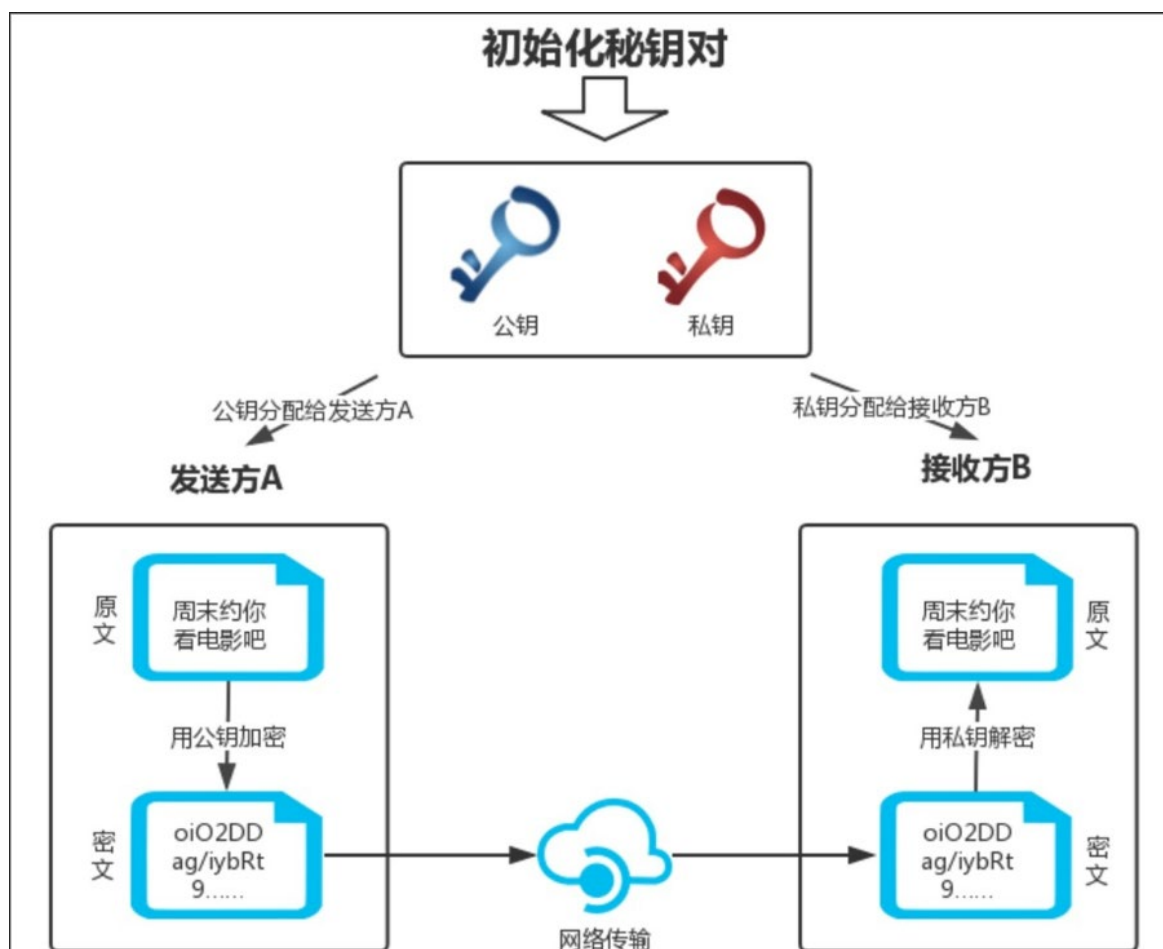


图 2-2 RSA 应用图示

2.3.2 RSA 算法的安全性

在传统的对称加密体制中，数据通信双方需要共享一个密钥进行加密和解密，这种对称加密方式存在诸多问题。首先，发送方和接收方必须共享同一个密钥，而这一密钥在某些情况下并不容易安全地传递。其次，为了保守秘密，通常需要频繁更换密钥，这使得密钥管理变得复杂和麻烦。特别是在网络通信的情况下，如果有 n 个用户，每两个用户都要进行加密通信，则需要 $n(n-1)/2$ 种不同的密钥，当 n 较大时，密钥数量和管理难度都会急剧增加。这些问题使得对称密钥在现代通信中的应用存在一定的局限性。为了克服这些问题，公开密钥密码体系被提出，其中最著名的便是基于数论的 RSA 密码体制^[9]。

RSA 算法的基本思想是利用一对密钥进行数据加密和解密，其中一把密钥是公开的，称为公钥，另一把密钥是私密的，称为私钥。公钥和私钥是一对一对应的关系，即只有用公钥加密的数据才能用对应的私钥解密，反之亦然。RSA 算法的安全性基于大数因子分解的困难性，即使知道公钥，也难以推算出私钥，因为这涉及到将一个大的数分解为两个大素数的乘积，而这一过程在现有技术下是极其困难的。

RSA 算法的核心在于两个大素数的选取。也因此，RSA 算法的安全性依赖于大数因子分解问题的困难性。分解一个大数为两个素数的乘积是一项非常困难的数学问题，尤其当这些素数非常大时。例如，1994 年，为破解 RSA-129 的公钥，需要 600 多名志愿者，使用 1600 多台计算机，耗时 8 个月才成功分解出密钥。而要破解一个 1024 位的 RSA 密钥，即使使用现代计算机技术，也需要数万亿年的时间。这种巨大的时间成本使得 RSA 算法的安全性得以保障。

然而，随着技术的发展，更快的因数分解算法不断被提出，使得 RSA 算法面临新的挑战。近年来，较低位数的大数已经被成功分解，例如 512 位的 RSA 密钥。这表明，攻击者可以更有效地破解 RSA 算法。因此，为了保证 RSA 的安全性，必须选择足够大的素数 p 和 q ，并使用更长的密钥。一般来说，选择 100 位以上的十进制数字作为 p 和 q 是比较安全的。这样一来，攻击者将无法在可接受的时间内分解 n ，从而保证 RSA 算法的安全性。

此外，为了应对不断变化的攻击技术，密钥的长度需要不断地更新和增强。当前，许多应用已经采用 2048 位甚至更长的 RSA 密钥，以提高安全性。例如，在电子商务系统中，RSA 被广泛应用于安全的交易过程中，如在线支付和银行卡转账等。在这些

过程中,发送方通常采用对称密钥加密技术对信息进行加密,然后使用接收方的公钥对对称密钥进行加密,形成数字信封。接收方收到信息后,先用私钥解密得到对称密钥,再用对称密钥解密信息。这种方式结合了对称加密和公钥加密的优点,既解决了密钥分发管理困难的问题,又克服了公钥加密速度慢的问题。

尽管 RSA 算法在保密方面运用得很好,但在实际应用中仍然存在一些问题。这些问题主要来自于密码体制的误用、参数选择错误和实现缺陷等。例如,如果在实际应用中选择了不够大的素数,或者在密钥管理过程中存在漏洞,都会降低 RSA 的安全性。因此,在使用 RSA 算法时,必须确保素数的选择足够大,密钥管理严格,并且实现过程中没有漏洞。

总的来说, RSA 算法的安全性虽然依赖于大数因子分解的困难性,但随着技术的发展,仍然面临着不断变化的挑战。为了确保 RSA 算法的安全性,必须不断地更新密钥的长度,并采取其他安全措施来保护数据和通信的安全。未来,随着分布式计算和量子计算技术的成熟, RSA 算法可能面临更大的挑战。因此,研究和开发新的、更安全的加密算法也是密码学领域的重要课题。

2.3.3 RSA 算法的应用

RSA 算法作为一种重要的非对称加密算法,在信息安全领域有着广泛的应用。以下是 RSA 算法在各个方面的应用:

(1) HTTPS/TLS 协议: RSA 算法是最常用于 HTTPS/TLS 协议中的密钥交换和身份验证的算法之一。在 HTTPS 连接中,服务器和客户端使用 RSA 算法进行密钥交换,以确保通信的安全性。通过使用 RSA 算法,网站可以保护用户与服务器之间的数据传输,防止数据被窃听或篡改。

(2) 数字签名: RSA 算法常用于生成和验证数字签名,以确保数据的完整性和来源的真实性。数字签名在软件发布、文档认证、电子邮件安全等方面十分常见。通过使用 RSA 算法生成的数字签名,接收方可以验证数据的完整性和真实性,确保数据未被篡改或伪造。

(3) SSH 密钥认证: RSA 密钥对被广泛应用于 SSH (Secure Shell) 连接中,用于用户与远程服务器之间的身份认证和安全通信。通过使用 RSA 密钥对,用户可以在不需要输入密码的情况下登录远程服务器,提高了系统的安全性和便利性。

(4) 加密邮件：一些电子邮件客户端和协议（如 PGP 和 S/MIME）利用 RSA 算法来加密邮件内容和验证发件人的身份。通过使用 RSA 算法，发送方可以将邮件内容加密，并使用私钥生成数字签名，接收方可以使用公钥解密邮件内容和验证签名，确保邮件的安全性和可信度。

(5) 云存储和备份服务：RSA 算法也被广泛应用于云存储和备份服务中，用于加密存储在云端的数据以及管理访问控制和权限验证。通过使用 RSA 算法，用户可以对存储在云端的数据进行加密，保护数据的隐私和安全性，并控制数据的访问权限，确保只有授权用户可以访问和操作数据。

综上所述，RSA 算法在 HTTPS/TLS 协议、数字签名、SSH 密钥认证、加密邮件和云存储等方面都有着重要的应用。通过使用 RSA 算法，可以保护通信的安全性、数据的完整性和来源的真实性，促进了信息安全领域的发展和应用。

2.4 DSA 算法

2.4.1 DSA 算法内容

DSA 是一种基于非对称加密技术的数字签名算法，由美国国家标准与技术研究所（NIST）在 1994 年发布，作为数字签名标准（DSS）的一部分。其核心理论基于数论中的离散对数问题，尤其是在有限域上的离散对数难题，该问题认为在给定某些参数的情况下，找到一个特定离散对数是非常困难的^[10]。下面是 DSA 算法的基本理论框架^[11]：

(1) 密钥生成：

① **选择两个大素数**：随机选取一个大素数 p 和另一个较小的素数 q ，其中 q 应该是 $p - 1$ 的因子。

② **计算生成元**：选择一个整数 g （通常 $1 < g < p$ ），满足 $g^q \bmod p = 1$ ， g 是一个生成元。

③ **私钥**：随机选择一个小于 q 的整数 x 作为私钥。

④ **公钥**：计算 $y = g^x \bmod p$ ， y 作为公钥发布。

(2) 签名生成：

发送方用私钥 x 和一个随机数 k ($0 < k < q$ ，且 k 不能重复使用) 来签署消息 m 。

计算 $r = (g^k \bmod p) \bmod q$, 要求 $r \neq 0$ 。

计算 $s = (k^{-1} \cdot H(m) + x \cdot r) \bmod q$, 其中 $H(m)$ 是消息 m 的哈希值, 且 k^{-1} 是 k 在模 q 下的乘法逆元。

签名对为 (r, s) 。

(3) 签名验证:

接收方使用发送方的公钥 (p, q, g, y) 和消息 m 来验证签名 (r, s) 的有效性。

计算 $w = s^{-1} \bmod q$, 即 s 在模 q 下的乘法逆元。

计算 $u_1 = (H(m) \cdot w) \bmod q$ 和 $u_2 = (r \cdot w) \bmod q$ 。

计算 $v = ((g^{u_1} \cdot y^{u_2}) \bmod p) \bmod q$ 。

如果 $v = r$, 则签名有效; 否则, 签名无效。

DSA 签名算法流程相关具体流程可见图 2- 3。

2.4.2 DSA 算法的安全性

DSA (数字签名算法) 是一种基于离散对数问题的数字签名算法, 在信息安全领域中扮演着重要的角色。

首先, DSA 算法的安全性建立在离散对数问题的困难性基础上。离散对数问题指的是在给定一个有限域上的素数 p 和一个生成元 g 的情况下, 找到一个整数 x , 使得 $g^x \bmod p = y$, 其中 y 是已知的。目前, 没有已知的高效算法可以在多项式时间内解决离散对数问题, 因此 DSA 算法的安全性取决于这一数学难题的困难性。离散对数问题的复杂性使得在现有的计算机技术下, 破解 DSA 算法成为一项非常困难的任务。

其次, 素数 p 的选择在 DSA 算法中至关重要。如果选择的素数 p 太小, 可能会容易受到攻击, 导致私钥被破解。因此, 为了确保算法的安全性, 需要选择足够大的素数 p , 以增加攻击者破解离散对数问题的难度。通常, 建议选择至少 1024 位的素数, 以确保算法的安全性。

密钥长度也是影响 DSA 算法安全性的重要因素之一。密钥的长度越长, 安全性就越高, 因为这增加了攻击者破解密钥的难度。随着计算能力的不断提升, 使用较长的密钥长度 (例如 2048 位或更长) 已经成为一种常见的做法, 以进一步增强 DSA 算法的安全性。

在 DSA 算法的签名生成过程中，需要生成一个随机数 k 。这个随机数的生成必须足够随机且不可预测，否则可能会导致私钥的泄露，从而破坏算法的安全性。因此，在 DSA 算法的实现中，需要使用安全的随机数生成器来生成随机数 k ，以确保签名的安全性。随机数生成器的质量直接影响到整个 DSA 签名过程的安全性。

除了素数 p 和随机数 k 之外，DSA 算法还涉及到其他参数的选择，如生成元 g 和子群 q 。这些参数的选择也会影响到算法的安全性。参数的选择应当是随机的、安全的，并且避免使用固定的、可预测的参数。通过选择合适的参数，可以有效提高 DSA 算法的安全性。

此外，DSA 算法还容易受到侧信道攻击的影响。侧信道攻击利用了实现上的缺陷或者物理特性来获取密钥信息。因此，在 DSA 算法的实现中，需要采取相应的措施来防范侧信道攻击，提高算法的安全性。防范侧信道攻击的措施包括物理保护、时序保护和错误注入防护等，通过这些措施可以有效减少侧信道攻击的风险。

具体到 DSA 签名的生成过程，DSA 签名的生成过程涉及到对随机数 k 和消息 m 的运算，这些运算基于离散对数问题的困难性。首先，随机选择一个私钥 k ，并基于私钥 k 和消息 m 计算出一个公钥 r ，然后计算出一个签名 s 。签名 s 的计算涉及到对公钥 r 的一系列运算，这些运算都是基于离散对数问题的困难性。在签名验证过程中，使用公钥来验证签名 s 是否有效。验证过程同样基于离散对数问题的困难性，涉及到对公钥 r 和签名 s 的一系列运算。

总的来说，DSA 算法的安全性取决于离散对数问题的困难性、素数的选择、密钥长度、随机数生成、参数选择以及对侧信道攻击的防范。通过合理选择参数、增加密钥长度、使用安全的随机数生成器等措施，可以提高 DSA 算法的安全性，确保其在实际应用中的可靠性和安全性。尽管 DSA 算法在信息安全领域具有重要地位，但随着计算技术的发展和攻击手段的不断进步，仍需不断改进和优化，以应对新的安全挑战。在实际应用中，严格遵循 DSA 算法的安全性原则，选择合适的参数，并采取必要的防护措施，是确保数字签名安全性和可靠性的关键。

2.4.2 DSA 算法的应用

DSA 算法作为一种重要的数字签名算法，在信息安全领域也有着一系列的应用。以下是 DSA 算法在不同领域的应用情况：

(1) 数字签名：虽然 DSA 在普及程度上不如 RSA，但它主要被设计用于数字签名。特别是在某些合规要求下或特定的系统标准中，例如美国的数字签名标准(DSS)，DSA 算法被广泛使用。数字签名在各种领域中都有应用，包括电子合同、法律文件、金融交易等，通过 DSA 算法生成的数字签名可以确保数据的完整性和来源的真实性。

(2) 安全协议和标准：在某些特定的安全协议和标准中，如某些版本的 SSL/TLS 协议、某些电子文档签名应用场景，可能会指定使用 DSA 算法。在这些标准和协议中，DSA 算法被用于实现数字签名和身份验证功能，以确保通信的安全性和可信度。

(3) 嵌入式系统和物联网（IoT）：由于 DSA 的计算复杂度相比 RSA 较低，它有时会被选择用于资源受限的设备中实现数字签名功能。在嵌入式系统和物联网设备中，资源有限是一个普遍的问题，而 DSA 算法由于其相对较低的计算需求，可以更适合这些环境下的数字签名需求。因此，在一些嵌入式系统和物联网设备中，DSA 算法被选用来实现数字签名功能，以确保设备间通信的安全性和数据的可信度。

综上所述，DSA 算法在数字签名、安全协议和标准以及嵌入式系统和物联网等领域都有着重要的应用。通过 DSA 算法，可以实现数据的完整性和来源的真实性验证，确保通信的安全性和可信度。在不同的应用场景中，DSA 算法都发挥着重要的作用，为信息安全领域的发展和应用提供了重要支持。

需要注意的是，随着时间推移和技术发展，加密标准和实践会不断演进，RSA 和 DSA 的使用可能会因新算法的引入和安全标准的更新而有所变化。例如，随着量子计算的潜在威胁，行业正在向抗量子的密码算法过渡。此外，ECC 算法因为其更高的效率和安全性，在现代应用中逐渐取代 RSA 和 DSA 的部分用途。

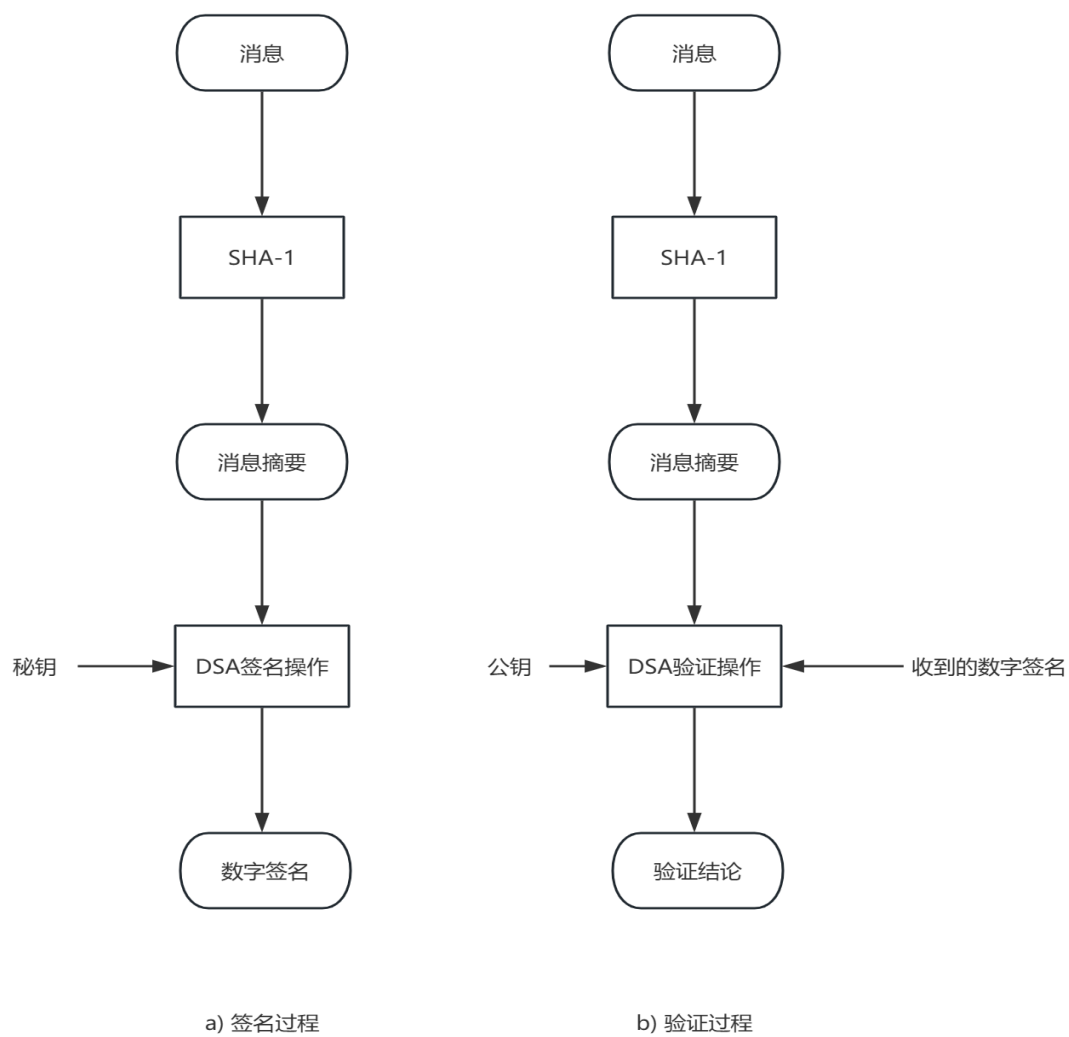


图 2- 3 DSA 签名算法流程

第3章 OpenSSL、Botan、Crypto++算法与自实现算法

在 C++ 中, RSA 和 DSA 这两种非对称加密算法的实现通常依赖于专门的加密库, 因为它们涉及到的大数运算超出了基本数据类型的处理能力, 甚至连 windows api 中的 128 位 int 都略显紧张。

3.1 OpenSSL 库与 RSA、DSA 实现

3.1.1 OpenSSL 库

OpenSSL 是一个广泛应用的开源加密库, 提供了丰富的加密算法和工具, 用于实现安全通信和数据保护。自 1998 年发布以来, OpenSSL 已经成为网络安全领域的基石, 支持各种协议和标准, 确保了互联网上数据传输的保密性和完整性。虽然主要用 C 编写, 但 OpenSSL 也提供了 C++ 接口, 支持 RSA 算法的全面实现。它广泛应用于 HTTPS、SSL/TLS 协议中, 同时也可用于生成 RSA 密钥对、进行加密解密和签名验证。

3.1.2 OpenSSL 中 RSA 实现源码分析

OpenSSL 的 RSA 相关数据结构主要定义在 openssl/rsa.h 中, 其中有相关的几个比较重要的数据结构和算法:

struct rsa_st, 其中由于版本变更, openssl 源码中加入了大量的宏去进行版本控制, 使其变得晦涩难懂。删去部分成员变量和无关紧要的宏之后, struct rsa_st 会变成如下这样:

```
1. struct rsa_st {
2.     const RSA_METHOD *meth;
3.     BIGNUM *n;    // n=p*q
4.     BIGNUM *e;    // 加密指数
5.     BIGNUM *d;    // 私钥
6.     BIGNUM *p;    // 大素数 p
7.     BIGNUM *q;    // 大素数 p
8.     BIGNUM *dmp1; // d mod (p-1)
9.     BIGNUM *dmq1; // d mod (q-1)
10.    BIGNUM *iqmp; // (inverse of q) mod p
11.    /* others */ };
```

这个结构体包含了 RSA 运算时需要的数据, 值得注意的是, 其中并没有使用平台规定的 int 类型, 而是自定义了一个 BIGNUM 大数类, 用来满足 RSA 这种对于数据

大小要求较高的算法的需求。

rsa_st 其中还包括了一根 RSA_METHOD 结构体的指针, 这个结构体中包含了很多的函数指针, 比如: 公钥加密函数、公钥解密函数、私钥加密函数、私钥解密函数、RSA 密钥对生成函数等等。用户在使用的时候, 既可以使用 OpenSSL 规定好的函数指针, 也可以自己实现一个函数, 修改结构体内的函数指针指向, 使其指向用户自定义函数, 从而实现了数据和处理函数的解耦。

3.1.3 OpenSSL RSA 算法实现

下面是一段 C++ 程序, 用来演示如何使用 openssl 的 RSA 算法对文本进行加解密:

```
1. #include <openssl/rsa.h>
2. #include <openssl/pem.h>
3. #include <openssl/err.h>
4. #include <iostream>
5. #include <string>
6.
7. // 该函数用于处理 OpenSSL 错误, 将错误信息打印到标准错误输出, 并终止程序。
8. void handleErrors() {
9.     ERR_print_errors_fp(stderr);
10.    abort();
11. }
12.
13. // 生成键值对
14. RSA* generateKeyPair() {
15.     int bits = 2048; // 2048 位的 RSA 密钥对
16.     unsigned long e = RSA_F4;
17.     BIGNUM* bne = BN_new(); // 创建一个新的 BIGNUM 对象
18.     if (!BN_set_word(bne, e)) {
19.         handleErrors();
20.     }
21.
22.     // 使用 RSA_new 创建一个新的 RSA 对象, 并使用 RSA_generate_key_ex 生成密钥对
23.     RSA* rsa = RSA_new();
24.     if (!RSA_generate_key_ex(rsa, bits, bne, NULL)) {
25.         handleErrors();
26.     }
27.     // 释放 BIGNUM 对象, RSA 对象 ownership 转移
28.     BN_free(bne);
29.     return rsa;
```

```

30. }
31.
32. std::string rsaEncrypt(RSA* rsa, const std::string& plaintext) {
33.     std::string ciphertext(RSA_size(rsa), '\0'); // 生成加密后的字符串存储位置
34.     // 使用公钥加密, 选择不同的 padding 模式不仅可以对齐待加密的字符串, 还可以用作混淆
35.     int len = RSA_public_encrypt(plaintext.size(), (const unsigned char*)plaintext.c_str(),
36.                                  (unsigned char*)ciphertext.c_str(), rsa, RSA_PK
37.                                  CS1_OAEP_PADDING);
38.     if (len == -1) {
39.         handleErrors();
40.     }
41.     return ciphertext;
42. }
43. std::string rsaDecrypt(RSA* rsa, const std::string& ciphertext) {
44.     std::string plaintext(RSA_size(rsa), '\0');
45.     // 公钥解密
46.     int len = RSA_private_decrypt(ciphertext.size(), (const unsigned char*)ciphertext.c_str(),
47.                                   (unsigned char*)plaintext.c_str(), rsa, RSA_PK
48.                                   CS1_OAEP_PADDING);
49.     if (len == -1) {
50.         handleErrors();
51.     }
52.     plaintext.resize(len);
53.     return plaintext;
54. }
55. int main() {
56.     // Initialize OpenSSL
57.     OpenSSL_add_all_algorithms();
58.     ERR_load_BIO_strings();
59.     ERR_load_crypto_strings();
60.
61.     // Generate RSA key pair
62.     RSA* rsa = generateKeyPair();
63.     if (!rsa) {
64.         std::cerr << "Key pair generation failed" << std::endl;
65.         return 1;
66.     }
67.
68.     // 创建了两个流 (Basic I/O) 用于存放公钥和私钥的 PEM 编码形式
69.     BIO* bp_public = BIO_new(BIO_s_mem());

```

```

70.     BIO* bp_private = BIO_new(BIO_s_mem());
71.
72.     // 将 RSA 结构体中的公钥和私钥写入到之前创建的流中
73.     PEM_write_bio_RSAPublicKey(bp_public, rsa);
74.     PEM_write_bio_RSAPrivateKey(bp_private, rsa, NULL, NULL, 0, NULL, NULL);
75.
76.     // 获取生物流中当前待读取数据的长度
77.     size_t pub_len = BIO_pending(bp_public);
78.     size_t priv_len = BIO_pending(bp_private);
79.
80.     // 分配足够的内存来存储公钥和私钥, 并使用 BIO_read 读取流中的数据到分配的内存中
81.     char* pub_key = (char*)malloc(pub_len + 1);
82.     char* priv_key = (char*)malloc(priv_len + 1);
83.
84.     BIO_read(bp_public, pub_key, pub_len);
85.     BIO_read(bp_private, priv_key, priv_len);
86.
87.     pub_key[pub_len] = '\0';
88.     priv_key[priv_len] = '\0';
89.
90.     std::cout << "Public Key: " << std::endl << pub_key << std::endl;
91.     std::cout << "Private Key: " << std::endl << priv_key << std::endl;
92.
93.     // Clean up
94.     BIO_free_all(bp_public);
95.     BIO_free_all(bp_private);
96.     free(pub_key);
97.     free(priv_key);
98.
99.     // Encrypt message
100.    std::string plaintext = "Hello, OpenSSL RSA!";
101.    std::string ciphertext = rsaEncrypt(rsa, plaintext);
102.    std::cout << "Encrypted: " << ciphertext << std::endl;
103.
104.    // Decrypt message
105.    std::string decryptedtext = rsaDecrypt(rsa, ciphertext);
106.    std::cout << "Decrypted: " << decryptedtext << std::endl;
107.
108.    // Free RSA key
109.    RSA_free(rsa);
110.
111.    // Clean up OpenSSL
112.    CRYPTO_cleanup_all_ex_data();
113.    ERR_free_strings();

```

```
114.  
115.     return 0; }
```

其中对于 `generateKeyPair` 来说，这个函数的主要作用是生成一个 2048 位的 RSA 密钥对。首先，使用 `BN_new` 创建一个新的 `BIGNUM` 对象，并使用 `BN_set_word` 将其初始化为公共指数 `e`（通常为 `RSA_F4`，即 65537）。接下来，使用 `RSA_new` 创建一个新的 `RSA` 对象，并使用 `RSA_generate_key_ex` 生成密钥对。最后，释放 `BIGNUM` 对象，并返回生成的 `RSA` 对象。

`rsaEncrypt` 函数使用公钥对明文进行加密。首先，创建一个字符串 `ciphertext`，其大小为 `RSA` 密钥的大小。然后，使用 `RSA_public_encrypt` 函数进行加密，使用 `RSA_PKCS1_OAEP_PADDING` 填充模式。如果加密失败，则调用 `handleErrors` 处理错误。

`rsaDecrypt` 函数使用私钥对密文进行解密。首先，创建一个字符串 `plaintext`，其大小为 `RSA` 密钥的大小。然后，使用 `RSA_private_decrypt` 函数进行解密，使用 `RSA_PKCS1_OAEP_PADDING` 填充模式。如果解密失败，则调用 `handleErrors` 处理错误。最后，调整明文字符串的大小以匹配实际解密的数据长度。

在主函数中，首先初始化 `OpenSSL` 库，并加载所有算法和错误信息。然后调用 `generateKeyPair` 函数生成 `RSA` 密钥对，并检查生成是否成功。接下来，使用 `PEM_write_bio_RSAPublicKey` 和 `PEM_write_bio_RSAPrivateKey` 函数将公钥和私钥写入内存 `BIO` 对象，并将其内容输出到控制台。然后，使用 `rsaEncrypt` 函数对一条消息进行加密，并输出加密后的密文。接着，使用 `rsaDecrypt` 函数对密文进行解密，并输出解密后的明文。最后，释放所有分配的资源，并清理 `OpenSSL` 库。

上述的代码在 Ubuntu 20.04 LTS 上的运行结果如图 3- 1 `rsa_openssl` 运行时的图所示。

```

lewis@lewis:/mnt/e/projects/CLionProjects/RSA_DSA/cmake-build-debug$
Public Key:
-----BEGIN RSA PUBLIC KEY-----
MIIBCgKCAQEA7+epa0uu0rVmFM/ApbRWuUId5bP0swqUblrtux0ibJuQK5p4884V
LIvyygSM5ldauCfrzia06qPMGYHUiEqBLADWB8c2qL86TD0vMnX9nu0pd8AQymMd
nNIsK1d1s0bYMZ46L/LpCArPq6Rov/16Kfc4nFcD/Qh/EI4i/uMRLgWpBNZA6Uo1
Eua2ria7V+kUnKC69TYwEfJ+Q2uKhPDQ13YbZ2o4u6Tz1qQKTjxi30n4yjcI5wC
tqPxqcLT3JZg8cY/nKZJRT1Vx96DjTwzUK+69MFHbD1jLqbdznIi/awkYco8QiuD
5D/o5lwSqu8Vh+r5ZVwsuSwPGPWP7lh5YwIDAQAB
-----END RSA PUBLIC KEY-----

Private Key:
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEA7+epa0uu0rVmFM/ApbRWuUId5bP0swqUblrtux0ibJuQK5p4
884VLIvyygSM5ldauCfrzia06qPMGYHUiEqBLADWB8c2qL86TD0vMnX9nu0pd8AQ
ymMdnNIsK1d1s0bYMZ46L/LpCArPq6Rov/16Kfc4nFcD/Qh/EI4i/uMRLgWpBNZA
6Uo1Eua2ria7V+kUnKC69TYwEfJ+Q2uKhPDQ13YbZ2o4u6Tz1qQKTjxi30n4yjc
I5wCtqPxqcLT3JZg8cY/nKZJRT1Vx96DjTwzUK+69MFHbD1jLqbdznIi/awkYco8
QiuD5D/o5lwSqu8Vh+r5ZVwsuSwPGPWP7lh5YwIDAQABAoIBACREddDCJqs6bB2+
UIXcbvXgu/oTRi677TeHqIP0gjtAAzoMRre40mujFoeiy3gxAYZwho3qPRRxb0j
dHFMlh3sXG7dL26HKeW6NLigrMA/YzL2xrMJ69qezU2aEbGSBrHtLTc0Yls0Yz00
DZ/klpxWgujjoLc6dxcWHcVBp4G4z2fSW+tLTpfCw7Buu4/Urc8D9vFaUdAHKHwF
vpxPAHJWronLtaDsDaNLjD8Eu5j92qHXxerUTTscfoSCfZwHjhMq1eZgePxqQUF3
21/UedaqbVr7XBEs6pB8Yk0x141z2sCNk0uC5rzdHvQ6ULT/BWM90jX7r+Mwy7ZM
CZ3bHeECgYEA+KYwFFHWF7tVwoE5VSu0AvTGQsyd+x5P1YfQK8dCHGwHocLXSduB
mcvp23fu2fNywyQyvS0AaZuBCb+wXfKT9jmjx1L4WU26ztuiUuE3Qw38AI91RUqr
W7DdaDu+V5+idV2MSSzCkbrv30Wzh/J7He733fHYCa0agi+PEUGXhAMCgYEA9v9L
2eDu3PZgYLUJTi31SCHYYKMJ8iYc1quA18hMowl5ZLC/Qs8oTg8JhES9gxQRsJqs
R4YNM5mFpXglsjcid6v6ziU2AlddyT+ZCzAI3+aa3tS/d/thfAcfZHPiti9FiNDk
M6CS45693vv0EViYihJ0YY9Y7/rAbm6LTHw8JyECgYEAuUj9pByl80FEMsCgvHro
lRtZcgWP48TesLuXwEenWA6YUvL7nKnZlYWCKmzgq8Dhz9B1js0ZuRiLu3wALMHK
yDyDctC9B68IeYTPHOF+kJUnKD0+q3fM4XKWSknau0jX4qVn6FAMXpN0LWh/M9rZ
K6EbS2QgHHXFvYU5cx5xjYkCgYEA32CGapB1PbBG4A8GwUMp6j7soUk80scHPN
C4p7vFNvInSgwwU4gVN0EUFY8HCs5jmo50nd27L08IMfb/7HgwDQRgAk7PpLADcc
2G8n1xQMQU0U7l+03reWp/In7med0Zd/jGSEn94VSUFLahiUJWGRfwZ8FRSXdcGg9
RVYcZCECgYBnizEZ2UtuB443pGNFLpm0pHBCKKAiz1jwCYX5pvvxbqvVgGeht9s
ciz+7ugHoo5aEKXBuR3/LXyDsX+sY4zy0ALgmisYS5mxt2JYaaF7ELetSm7CK5F4
ovL5Yb47SSG68DtKYuicY0045Aj4KGJKy/Hs9rEBZcHTbhzaU5cfUw==
-----END RSA PRIVATE KEY-----

$ python3 ^#####H#####"h>3#####1dE$e#####L2[W2[#####Z#####}T(Q6#####h#####
Decrypted: Hello, OpenSSL RSA!

```

图 3- 1 rsa_openssl 运行时图

3.1.4 OpenSSL DSA 算法实现

OpenSSL 支持 DSA 算法的实现, 包括密钥生成、签名生成与验证。OpenSSL 对 DSA 的支持遵循 FIPS 186-4 标准, 广泛应用于需要数字签名的场景。下面是使用 OpenSSL 进行数字签名的方法。

```
#include <openssl/dsa.h>
1. #include <openssl/pem.h>
2. #include <openssl/err.h>
3. #include <openssl/sha.h>
4. #include <iostream>
5. #include <string>
6. #include <vector>
7.
8. void handleErrors() {
9.     ERR_print_errors_fp(stderr);
10.    abort();
11. }
12.
13. DSA* generateDSAKeyPair() {
14.    DSA* dsa = DSA_new();
15.    if (!DSA_generate_parameters_ex(dsa, 2048, NULL, 0, NULL, NULL, NULL))
16.    {
17.        handleErrors();
18.    }
19.    if (!DSA_generate_key(dsa)) {
20.        handleErrors();
21.    }
22.    return dsa;
23. }
24. std::vector<unsigned char> dsaSign(DSA* dsa, const std::string& message) {
25.    unsigned char hash[SHA256_DIGEST_LENGTH];
26.    SHA256((unsigned char*)message.c_str(), message.size(), hash);
27.
28.    std::vector<unsigned char> sig(DSA_size(dsa));
29.    unsigned int sig_len;
30.
31.    if (DSA_sign(0, hash, SHA256_DIGEST_LENGTH, sig.data(), &sig_len, dsa)
32.        == 0) {
33.        handleErrors();
34.    }
```

```

34.
35.     sig.resize(sig_len);
36.     return sig;
37. }
38.
39. bool dsaVerify(DSA* dsa, const std::string& message, const std::vector<uns
    igned char>& sig) {
40.     unsigned char hash[SHA256_DIGEST_LENGTH];
41.     SHA256((unsigned char*)message.c_str(), message.size(), hash);
42.
43.     return DSA_verify(0, hash, SHA256_DIGEST_LENGTH, sig.data(), sig.size(
        ), dsa) == 1;
44. }
45.
46. int main() {
47.     // Initialize OpenSSL
48.     OpenSSL_add_all_algorithms();
49.     ERR_load_BIO_strings();
50.     ERR_load_crypto_strings();
51.
52.     // Generate DSA key pair
53.     DSA* dsa = generateDSAKeyPair();
54.     if (!dsa) {
55.         std::cerr << "Key pair generation failed" << std::endl;
56.         return 1;
57.     }
58.
59.     // Display keys
60.     BIO* bp_public = BIO_new(BIO_s_mem());
61.     BIO* bp_private = BIO_new(BIO_s_mem());
62.     PEM_write_bio_DSA_PUBKEY(bp_public, dsa);
63.     PEM_write_bio_DSAPrivateKey(bp_private, dsa, NULL, NULL, 0, NULL, NULL
        );
64.
65.     size_t pub_len = BIO_pending(bp_public);
66.     size_t priv_len = BIO_pending(bp_private);
67.
68.     char* pub_key = (char*)malloc(pub_len + 1);
69.     char* priv_key = (char*)malloc(priv_len + 1);
70.
71.     BIO_read(bp_public, pub_key, pub_len);
72.     BIO_read(bp_private, priv_key, priv_len);
73.
74.     pub_key[pub_len] = '\0';

```

```

75.     priv_key[priv_len] = '\0';
76.     std::cout << "Public Key: " << std::endl << pub_key << std::endl;
77.     std::cout << "Private Key: " << std::endl << priv_key << std::endl;
78.     // Clean up
79.     BIO_free_all(bp_public);
80.     BIO_free_all(bp_private);
81.     free(pub_key);
82.     free(priv_key);
83.     // Sign message
84.     std::string message = "Hello, OpenSSL DSA!";
85.     std::vector<unsigned char> signature = dsaSign(dsa, message);
86.     std::cout << "Signature: ";
87.     for (unsigned char c : signature) {
88.         printf("%02x", c);
89.     }
90.     std::cout << std::endl;
91.     // Verify signature
92.     bool is_valid = dsaVerify(dsa, message, signature);
93.     std::cout << "Signature is " << (is_valid ? "valid" : "invalid") << std::endl;
94.     // Free DSA key
95.     DSA_free(dsa);
96.     // Clean up OpenSSL
97.     CRYPTO_cleanup_all_ex_data();
98.     ERR_free_strings();
99.     return 0; }
    
```

在 DSA 算法中，最重要的就是对函数进行签名。在 dsaSign 函数中，首先使用 SHA-256 算法计算消息的哈希值，将消息转换为固定长度的哈希值，以确保签名的唯一性和安全性。之后使用 DSA_size 获取 DSA 密钥的大小，并创建一个适当大小的向量存储签名。继续调用 DSA_sign 函数对哈希值进行签名，并将签名结果存储在 sig 向量中。如果签名失败，调用 handleErrors 处理错误。调整签名向量的大小以匹配实际签名长度后，返回签名。

对签名进行验证自然也很重要。在 dsaVerify 函数中，计算需要验证的消息的 SHA-256 哈希值，这一步与签名过程中的哈希计算相同。之后调用 DSA_verify 函数，验证签名是否有效。如果签名有效，返回 true；否则，返回 false。

若是验证成功，则会显示出 Signature is valid. 字样，图 3- 2 dsa_openssl 运行时图是相关代码在 Ubuntu 20.04 上运行的结果。


```

Ubuntu: /mnt/e/projects/CLionProjects/RSA_DSA/cmake-build-debug/dsa_openssl
Public Key:
-----BEGIN PUBLIC KEY-----
MIIDRZCCAJoGByqGSM44BAEwggItAoIBAQDAjHMzxjNiJo4xriyN+DUt44PtRiBP
lNJ8Ly8TY8i61ryaQXYCYLXehnk9ZjsshJHyv6BLl0mDqwXb09a2l0wRBQVC1pa5
UIxY9w4cNTd1L0QFk73yRRX0K24gHmJ/ILyURW3/1LRjA5Py00+0XnCgClDQ8a9e
iCNebBPj6fR6moDEANXvVav1VW17WFrTSxwat/fJHUzsiYRPu35CVcqDe/8SaBi
XhFerYBBR8S64Pz1EpoLdQd28SZ+Ffk7ra6Slaecs13yB0VWmJ1j6A29wi9gv8kL
xFYRh/ZLJ32LnfW08UDjW3H03IJ56TIG6BcUEQdZRAbK0UWaUW6DjZTRAiEAtM0L
SeL2N45qj03E6RLNT6QAu9CeQvQWi3QVS0EBIkCggEBAJXhuCQP08DT6v0Rf5H4
VrQPDjuERqTeITiClQ11fogrTsFbScc7UfB0V3V9qHcwj+KDqs/Zuf1Mi06oflpq
vk0Q152XMybHsDjKDVc05BfUTLvZry0N46bnAT3Erkkj0W2twr61suMrCTV/no7p
Sk+nNFTjTzLkTanYlZV70ckohRF2LNXD0+0w+3+KpyHeoTlChwJ0UCA+VhSEF0i9
5d1sDTCiy0Jie5UHl39nWBki70qnkTXFw4KhCeQovcLPf6WtyA1IDlu0DdX73KGM
xMRqYzx+idbpy+VKHRE/APrHERTNKxxSKvyzi+otH6lddmk6W6EpMTVaz7W+0G0p
uV8DggEFAAKCAQA4q6b1cbAGDIdWbAC10gFL/N+2baEkL9Tfr9UNK+eChRm620c1
HwxgBU6DkjlZYIbL8Q2H2ZxmCvWW7UyFew0yQs9772fH//2iGdtA/FiHMMg01S
FVjof+ZRYjysgJwm7YjjousdggIXY3AqsQCeIvQSPG6UT0ZjgVT0wSneOpgdmmY0
Q14amUYNcpLRC7f5fWPod7UpGsKnJ3n8/HFJe5aDI0zbQD//FkN3FWNjr6+/XD8U
S7qxyTzhMvtSzKvn6rg0AJKoQhT0z/UrgYLH/DzH4QSWfb/xqZltiMoYKaVZVr+
gx4IHTgppYtDvugpskyT89VaWohXwt2UGjH2
-----END PUBLIC KEY-----

Private Key:
-----BEGIN DSA PRIVATE KEY-----
MIIDVwIBAAKCAQEAwIxzM8YzYia0Ma4sJfg1Le0D7UYgT5TSfC8vE2PIuta8mkF2
AmC13oZyvWY7LISR8r+gS5dJg6sF2zvWtpTsEQUFQtaWuVCMWPc0Hdu3dS9EBZ09
8kUV9CtuIB5ifcY8LEvt/9S0Yw0T8jjvjL5woApXUPGvXogjXmwT46un0RpqAxAD
V71Wr9VVte1ha00scGrf3yR1M7ImET7t+QLXKg3v/EmgYL4RXq2AQa/EuuD89Rka
C3UHdvEmfhX5062hki6nnLNd8gTLVpidY+gNvcIvYL/JJcRWEYf2Syd9pTX1tPFA
41txztyCeekyBhgXFBHwUQ6ytfFmLFug42U0QIhALTnJeXi9je0ao9Nx0kSzU+k
ALo/QnkL0Fot0FutBASJAoIBAQCv4bgkDzvA0+r9EX+R+Fa0Dw47hEak3iE4gpUN
dX6IK07B0WnH01HwdFd1fah3MI/ig6rP2bn9TIt0qH5aar5DkNedLzMmx7A4yg1X
N0QX1Ey72a8tDe0m5wE9xK5JIZltrcK+tbLjKwk1f5606UpPpZRU408yyrWp2Jc1
e9HJKIURdpTcQ9PjsPt/iqch3qE5Qh8CdFAgPLYUhBToveXdbA0wosjiYnuVB5d/
Z1gZC09Kp5E1xc0CoQnkKL3Cz3+LrcgNSA5btA3V+9yhjMTEamM8fonW6WPLSoUR
PwD6xxEUzSscUir8s4vqLR+pXXZp0LuhKTE1Ws+1vjhtKbLfAoIBADioZvVxsAYM
h1ZsALU6AWX837ZtsSQv1N+v1Q0r54KF6YbY5zUfDGAFT0S0VlghuVHXDYfZnGY
K9ZbtTIV7A7JCz3vvZ8f//aIZ20D8WJ8cyaA7VIVW0h/5LHInKyAnCbt100i6x2C
MjFjcCqxAJ4i9BI8bpRPRm0BVPTBKd46mB2aZjRDxhqZRg1yktELt/L9Y+h3tSka
wqcnefz8cUL7LoMjTNTAP/8WQ3cVY20sb79cPxRLurFhP0ExW1LMq+fqUDQAKqhC
FM7P9SuBgsf8PMfhBJZ9v/GpmW0iIyhgppVLWv6DHggd0Cmli00+6CmyTJPz1Vpa
iFfC3ZQaMfYCIQCKZer7I9zDAL9ptu/w5xbrUsA7272mUy2Sp3104s908w==
-----END DSA PRIVATE KEY-----

Signature: 3044022056a1653bcfdf4157d7c3bcd0a4ffbe9da78d56a1febc9e1b300c7ed57baf690022052becf4df
Signature is valid

```

图 3- 2 dsa_openssl 运行时图

3.2 Botan 库与 RSA、DSA 实现

3.2.1 Botan 库

Botan 是一款开源的 C++ 密码学库，旨在为开发人员提供一套强大且灵活的加密功能。自 2000 年发布以来，Botan 在密码学研究和应用领域中逐渐获得广泛认可。它以模块化设计和高效实现著称，支持多种加密算法和协议，适用于各种安全应用。

Botan 采用模块化设计，开发人员可以根据需要选择和组合不同的加密模块，简化了库的集成和扩展。

3.2.2 Botan RSA 算法实现

下面是一段使用 Botan 库实现 RSA 算法的 C++ 代码示例：

```
1. #include <botan/hex.h>
2. #include <botan/pem.h>
3. #include <botan/pkcs8.h>
4. #include <botan/pubkey.h>
5. #include <botan/rsa.h>
6. #include <botan/x509_key.h>
7. #include <botan/auto_rng.h>
8. #include <iostream>
9. #include <vector>
10.
11. int main() {
12.     try {
13.         // 初始化随机数生成器
14.         Botan::AutoSeeded_RNG rng;
15.
16.         // 生成 RSA 密钥对
17.         Botan::RSA_PrivateKey private_key(rng, 2048);
18.         Botan::RSA_PublicKey public_key = private_key;
19.
20.         // 获取私钥和公钥的 PEM 格式字符串
21.         std::string private_key_pem = Botan::PKCS8::PEM_encode(private_key
22.             );
23.         std::string public_key_pem = Botan::X509::PEM_encode(public_key);
24.
25.         // 打印私钥和公钥
26.         std::cout << "Private Key:\n"
```

```

26.         << private_key_pem << std::endl;
27.         std::cout << "Public Key:\n"
28.         << public_key_pem << std::endl;
29.
30.         // 要加密的消息
31.         std::string message = "Hello, Botan!";
32.
33.         // 加密消息
34.         Botan::PK_Ecryptor_EME encryptor(public_key, rng, "EME1(SHA-
256)");
35.         std::vector<uint8_t> ciphertext = encryptor.encrypt(
36.             reinterpret_cast<const uint8_t *>(message.data()), message
37.             .size(), rng);
38.         std::cout << "Encrypted message: " << Botan::hex_encode(ciphertext
39.             ) << std::endl;
40.         // 解密消息
41.         Botan::PK_Decryptor_EME decryptor(private_key, rng, "EME1(SHA-
256)");
42.         Botan::secure_vector<uint8_t> decrypted = decryptor.decrypt(cipher
43.             text.data(), ciphertext.size());
44.         std::string decrypted_message(decrypted.begin(), decrypted.end());
45.         std::cout << "Decrypted message: " << decrypted_message << std::en
46.             dl;
47.     } catch (std::exception &e) {
48.         std::cerr << "Exception: " << e.what() << std::endl;
49.         return 1;
50.     }
51.
52.     return 0; }
    
```

首先使用 Botan 的随机数生成器来生成密钥对和加密数据，之后生成 2048 位的 RSA 私钥，并从私钥中导出公钥。通过创建加密器和解密器对象，进行加密和解密操作，并输出结果。通过这种方式，可以生成 RSA 密钥对并打印它们，同时展示了加密和解密的完整过程。使用 Botan 库的接口，可以方便地处理各种加密任务。同时我们也更加清楚，通过加入了对象的概念后，Botan 的接口确实相对于 OpenSSL 的接口而言更为的 modern C++。

上面的代码在 Ubuntu 20.04 下的运行结果如图 3- 3 rsa_botan 运行时图所示。

```
Ubuntu: /mnt/e/projects/CLionProjects/RSA_DSA/cmake-build-debug/rsa_botan
```

Private Key:

-----BEGIN PRIVATE KEY-----

```
MIIEvgIBADANBgkqhkiG9w0BAQEFAASCBKggggSkAgEAAoIBAQDg5Y+LzHsbGXfI
3P0BHI02Nc/8/4345+4ai0VzWdyE0suptnLb3nCRJZiFDzR6hzJdtEobbS5gXYng
0XPYgH+xRh9cVZK4dAnE8c8yfU7eSRZqKM04MywBQNmra7rBC8vVpGF2NfuQuF38
eLXVwocwL2PJLEjBdmsQjV05glQsR7Y443gWhT8qpnRdJFh/+jL36U0NwuaVs6hb
4+bXFW+l1ZxS0uc95b+64vAH/udg6m9DNHAoW79BITdq8JCdzDndjbeu6FWfs/8L
lq2tUtQ6ZeA6fuepNmQHYr63z10RfXgpiAjL2ZPg0I13/TqpMz/cl7yzAJJZWbsF
DHoZVM0NAGmBAECggEADQNg1/791+1rKngVLLey594I7m1FJElwqXWJ675KjcDZ
3F9JGc2EZ28EynI89E1oDdcAUMJKZDDAE8rkMxweUhn/fg1vjh/Hbu9vsKQN07yD
6673KuyUbRr0XZEVadU1ErL5GuJNvoEv0G6DDV7ntAPr4LigxHJbM0Z84+eVgkhP
gn+R5tzq5hEgiFWmzMvriAuhCkBNdG5Wme0Nu7/tDIou5FmAxlhFQGRqLN+INsw6
vMj0SyuMWy62tydVSxUjSh/onn72Sovu6zSslelpsMViD2R0WLE0W4MhBoCuKN8S
f3w+oxvv4y9Zdq6pH9X+n1y7JM4LXiQ6ECamfWqDuwKBgQD+4R4DF7zgAN8wQKZ0
fzLyS18tH+tij4ii6+Njan9aRfgWW06KLL7kbi4meRzmbtPe8EFSr2tPrw3Umb2+
zZ01h6K+p2/ceXcf3ELld6V05P6mjAn5ETZFj5DE03LqiwmfLJIA5eq5YDrp0dvl
a51PW5rJ8fvKz02zSSWqyJLWzWKBgQDh4rJQaSFuKYSStxgdJtZ9drJoCAR0neBg
0PGWAISz35LxerpJAPnPsyKFA0wMx9hdG3pfX2jVdS8hz+a9+6qiyUwLDCHJmGaD
3DG1VuckYzXeu+90n+ga/9CumS8pTGoC9nPj9dN32hY0Q4bpQy5IzVw8HBnAWd+6
zlycv2k4wKBgQDzj+ZhKDb6jHq7npwyaTcjKpMlopwKNIVqwyYpY6TNfF2F570X
WutMWSnK5C06i6r5uPeDb1SJ8guyZTRYeBU1/KZ9sH2J350Bmj2EW5Jh4Ryjx0Dm
3RbSxE2f5/t5vGH2dN4U86Epgbpx/xtsTfDLrNG9fIarZ1fvX+8Qz0dKwKBGGeL
NDpvMraweDM1revaYTqqVwwi+aBkviR9P9LnmPP2nzW3d6VIMD5P82gqBy8hlfmKi
dBEE3z5qiZjTI0XzDwWopbauDwbJ9ef1Bh315RMtcp2b0h/ygXa6AI/Mm/YwKMh5
0w3TQxfQsqR62ZF4k0k4yWSPZEV9N4oonBgUP+0PAoGBAJ800kB5CGuWFdai+FrZ
LzBhogY1+d532vEoi9qmPW601JQ24WycSATzRtbjRaYoCX4pLJ9g/LF08n6QZxGF
ic3hDnPXU0bWRgDrJbBlaDY4p1HEbCMbMUKXiW1MvX8L2wv+GRETtlqqLKK7LDIt
0XOMRZEwTHM+r44XBauRaLEo
```

-----END PRIVATE KEY-----

Public Key:

-----BEGIN PUBLIC KEY-----

```
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA40WPpcx7GxL3yNzzgRyD
tjXP/P+N+0fuGotFc1nchDrLqbZy295wksWYhQ80eocyXbRK620uYF2J4DlZ8oB/
sUYfXfWSuHQJxPHPMn103kkWaijDuDMsAUDZq2u6wQvL1aRhdjX7kLhd/Hi11cKH
MJdjySxIwXZrEI1TuYJULEe200N4FoU/KqZ0XSRYf/o5d+LDjcLmLb0oW+Pm1xVv
pdWcUjrnPeW/uuLwB/7nYBpvQzRwKFu/QLXavCQncw53Y23rhhVn7P/C5atrVLU
Bs3g0n7nqTZKB2K+t89dEX14KYgIy9mT4DiNd/06qTM/3Je8swCSWVm7BQx6GvTD
jQIDAQAB
```

-----END PUBLIC KEY-----

Encrypted message: 2889BA350FA8A187D97662D25B2FD5714C032DB5133307DE74F3A16A9A

Decrypted message: Hello, Botan!

图 3- 3 rsa_botan 运行时图

3.2.3 Botan DSA 算法实现

以下是一个使用 Botan 3.3 库实现 DSA 算法的 C++ 示例程序，这个示例展示了如何使用 Botan 库生成 DSA 密钥对、签名和验证消息。

```

1. #include <botan/dl_group.h>
2. #include <botan/dsa.h>
3. #include <botan/hash.h>
4. #include <botan/hex.h>
5. #include <botan/pem.h>
6. #include <botan/pkcs8.h>
7. #include <botan/pubkey.h>
8. #include <botan/x509_key.h>
9. #include <iostream>
10. #include <vector>
11. #include <botan/auti_rng.h>
12.
13. void handle_errors(const std::string &msg) {
14.     std::cerr << "Error: " << msg << std::endl;
15.     exit(1);
16. }
17.
18. int main() {
19.     try {
20.         // 初始化随机数生成器
21.         Botan::AutoSeeded_RNG rng;
22.
23.         // 生成 DSA 参数
24.         Botan::DL_Group group("dsa/botan/2048");
25.
26.         // 生成 DSA 密钥对
27.         Botan::DSA_PrivateKey private_key(rng, group);
28.         Botan::DSA_PublicKey public_key = private_key;
29.
30.         // 获取私钥和公钥的 PEM 格式字符串
31.         std::string private_key_pem = Botan::PKCS8::PEM_encode(private_key
        );
32.         std::string public_key_pem = Botan::X509::PEM_encode(public_key);
33.
34.         // 打印私钥和公钥
35.         std::cout << "Private Key:\n" << private_key_pem << std::endl;
36.         std::cout << "Public Key:\n" << public_key_pem << std::endl;
37.

```

```

38.      // 要签名的消息
39.      std::string message = "Hello, Botan DSA!";
40.
41.      // 计算消息的哈希值
42.      const auto hash = Botan::HashFunction::create_or_throw("SHA-
256");
43.      auto message_hash = hash->process(message);
44.
45.      // 签名消息
46.      Botan::PK_Signer signer(private_key, rng, "EMSA1(SHA-256)");
47.      std::vector<uint8_t> signature = signer.sign_message(message_hash,
rng);
48.
49.      std::cout << "Signature: " << Botan::hex_encode(signature) << std:
:endl;
50.
51.      // 验证签名
52.      Botan::PK_Verifier verifier(public_key, "EMSA1(SHA-256)");
53.      bool valid = verifier.verify_message(message_hash, signature);
54.
55.      std::cout << "Signature is " << (valid ? "valid" : "invalid") << s
td::endl;
56.
57.  } catch (std::exception &e) {
58.      handle_errors(e.what());
59.  }
60.
61.  return 0; }

```

和 OpenSSL 不同的是, Botan 作为一个社区活跃、更新频繁的第三方加密库, 采取了更为现代的库设计, 使得用户在进行调用时, 只需要使用私钥创建签名器对象, 并对消息哈希进行签名, 即可生成签名。在对签名进行验证之时, 使用公钥创建验证器对象, 即可验证签名的有效性。

上面的代码在 Ubuntu 20.04 下的运行结果如图 3-4 dsa_botan 运行时图所示。


```

Ubuntu: /mnt/e/projects/CLionProjects/RSA_DSA/cmake-build-debug/dsa_botan
Private Key:
-----BEGIN PRIVATE KEY-----
MIICZAIBADCCAjkGBYqGSM44BAEwggIsAoIBAQCRIpP37z3wCrpXn2hJhIrXdKG
T1Wbh+jnSihTUVwb0d5o39ZF00AMYMCAAXiRmAN07rWUpTK/1nuaCerEuGY6B5E0
aPOUZftwQNjd8Tky66xDR6Uw7LphyFT5uIDTwMNmAIByfEVWba3ia9WjLL4J00wP
JLWv/vjsbFs+V/uJA1qbwWdpkyEx4W08L0/KsY0N8GEgPMU+YQ08ctVZS/1AymU4
D0SpqFHcsHVJX8AzqKWAcA69eP4FL2ZVVkjrs3GdKv6LSID42tbxWB16F4+JJ0yH
C+m5brCMRsQAQMwu/h37Gxho3TGd48NKMqY6tusSJCCaQZaAZHkC0XKNTfnhAiEA
jNfUUPhVctL07kzkaah1bR69EFgkGUPq/7CzVFhekk0CggEADZ9eB2G029GDPWqx
qWGgmWxfIjA/cthMFA9nxDHZSrVxW+qBoMmN0c5Lz3jWuevILdNP6J2UCR1YS6Fe
8V9ehvEdlVbJaeID3fpYNWQgpJy0RLWVuQGpM8/gdntZTxige3+R3s26RGUImQ94
8v+R8v581D/S5G0Y6tofe7ZgLGf/bv0ksoTy/ZuhCjYELej6h6LKNLl/7IEVehSF
5EBB3wKDARHLiAu+bTSUGUiG+WXNwxNfXM8Tg3KL9LuAb5aSwLENbEwJx1pso7QB
PLFqssEF9r4jrqaADqsheJhflyyYBX4chuR0chhojqSuDzY23Mp0XJ3NTmr/tnzL
wT1hMQQiAiB1LTP0Bc8N8LVqs040jxxKf5kMVre2sALivz144XkpQQ=
-----END PRIVATE KEY-----

Public Key:
-----BEGIN PUBLIC KEY-----
MIIDRjCCAjkGBYqGSM44BAEwggIsAoIBAQCRIpP37z3wCrpXn2hJhIrXdKGt1Wb
h+jnSihTUVwb0d5o39ZF00AMYMCAAXiRmAN07rWUpTK/1nuaCerEuGY6B5E0aPOU
ZftwQNjd8Tky66xDR6Uw7LphyFT5uIDTwMNmAIByfEVWba3ia9WjLL4J00wPjLWv
/vjsbFs+V/uJA1qbwWdpkyEx4W08L0/KsY0N8GEgPMU+YQ08ctVZS/1AymU4D0Sp
qFHcsHVJX8AzqKWAcA69eP4FL2ZVVkjrs3GdKv6LSID42tbxWB16F4+JJ0yHC+m5
brCMRsQAQMwu/h37Gxho3TGd48NKMqY6tusSJCCaQZaAZHkC0XKNTfnhAiEAjNfU
UPhVctL07kzkaah1bR69EFgkGUPq/7CzVFhekk0CggEADZ9eB2G029GDPWqxqWGg
mWxfIjA/cthMFA9nxDHZSrVxW+qBoMmN0c5Lz3jWuevILdNP6J2UCR1YS6Fe8V9e
hvEdlVbJaeID3fpYNWQgpJy0RLWVuQGpM8/gdntZTxige3+R3s26RGUImQ948v+R
8v581D/S5G0Y6tofe7ZgLGf/bv0ksoTy/ZuhCjYELej6h6LKNLl/7IEVehSF5EBB
3wKDARHLiAu+bTSUGUiG+WXNwxNfXM8Tg3KL9LuAb5aSwLENbEwJx1pso7QBPLFq
ssEF9r4jrqaADqsheJhflyyYBX4chuR0chhojqSuDzY23Mp0XJ3NTmr/tnzLwT1h
MQ0CAQUAAoIBAC0TznMBJ5Wq8VlsfWMMhy4uVCuG/QG21QCerwIC39vLpuxhVSzAN
pFUSOpGI6SAN4PSmQVj/Nt9upgU6CV5Zwjtd9N0PHy+WZUcWWEuPZ/DAFMNPJi5t
zZdliVw2h+tSod3Zz0EIjWhfNyrzEe0GE3MLk8D5WMLyIi98MciMrjws+MGe105
C2o9LK1uPshK88TrZDhAJLn0VF6BhzdwF4vq6EsH7/7hHh5QPHGomYQy0fxcF5Zk
khg1PNfiB8J2iKhArio0Y0ZvcrZS/ASUcWWzC00gNRI/dAyIZAfQs8dG/Q13Srgb
AzfQfj73qiRhprH31naDKu63P0jgEhFJcJ8=
-----END PUBLIC KEY-----

Signature: 35BF455DFDCD9D5B261D771D5B2119B498DCD805CBE98321F73A5433A230A60I
Signature is valid

```

图 3- 4 dsa_botan 运行时图

3.3 Crypto++库与 RSA、DSA 实现

3.3.1 Crypto++库

Crypto++库（也称 CryptoPP）是一个开源的 C++ 加密库，提供了一套全面的加密算法和工具。由 Wei Dai 于 1995 年创建，Crypto++ 在密码学研究和应用领域中享有盛誉，广泛用于开发安全软件和系统。

3.3.2 Crypto++库 RSA 算法实现

下面是一段 C++ 程序，用来演示如何使用 crypto++ 对文本进行加解密：

```
1. #include <iostream>
2. #include <cryptopp/rsa.h>
3. #include <cryptopp/osrng.h>
4. #include <cryptopp/base64.h>
5. using namespace std;
6. using namespace CryptoPP;
```

首先，通过 `#include` 指令引入了必要的库文件，包括 `iostream` 用于标准输入输出，`cryptopp/rsa.h`、`osrng.h` 和 `base64.h` 分别用于 RSA 加密算法、随机数生成和 Base64 编码。

```
1. void GenerateRSAKeyPair(RSA::PrivateKey& privateKey, RSA::PublicKey& public
   cKey)
2. {
3.     AutoSeededRandomPool rng;
4.     InvertibleRSAFunction parameters;
5.     parameters.GenerateRandomWithKeySize(rng, 2048);
6.     privateKey = RSA::PrivateKey(parameters);
7.     publicKey = RSA::PublicKey(parameters); }
```

`GenerateRSAKeyPair` 函数利用 `AutoSeededRandomPool` 生成安全随机数，通过 `InvertibleRSAFunction` 类生成一个 2048 位长度的 RSA 密钥对，并分别存储在 `RSA::PrivateKey` 和 `RSA::PublicKey` 对象中。

```
1. std::string RSAEncrypt(const RSA::PublicKey& publicKey, const std::string&
   plainText)
2. {
3.     AutoSeededRandomPool rng;
4.     RSAES_OAEP_SHA_Encryptor encryptor(publicKey);
5.
6.     std::string cipherText;
```



```

7.
8. StringSource(plainText, true,
9. new PK_EcryptorFilter(rng, encryptor,
10. new StringSink(cipherText)
11. )
12. );
13.
14. return cipherText; }

```

RSAEncrypt 函数接收公钥和明文字符串作为参数，使用 RSAES_OAEP_SHA_Encryptor 进行加密，这是一种基于 PKCS#1 v2.1 的加密方案，提供了更高级别的安全性。

利用 StringSource 和 PK_EcryptorFilter 将明文数据流式处理成加密后的密文字符串。

```

1. std::string RSADecrypt(const RSA::PrivateKey& privateKey, const std::string& cipherText)
2. {
3.     AutoSeededRandomPool rng;
4.     RSAES_OAEP_SHA_Decryptor decryptor(privateKey);
5.
6.     std::string recoveredText;
7.
8.     StringSource(cipherText, true,
9. new PK_DecryptorFilter(rng, decryptor,
10. new StringSink(recoveredText)
11. )
12. );
13.
14. return recoveredText; }

```

RSADecrypt 函数接受私钥和密文字符串，通过 RSAES_OAEP_SHA_Decryptor 进行解密，与加密过程类似，但使用的是解密器。

StringSource 和 PK_DecryptorFilter 同样用于数据流处理，将密文解码回原始的明文字符串。

```

1. int main()
2. {
3.     try
4.     {
5.         RSA::PrivateKey privateKey;
6.         RSA::PublicKey publicKey;
7.

```

```
8. // 生成 RSA 密钥对
9. GenerateRSAKeyPair(privateKey, publicKey);
10.
11. // 待加密的文本
12. std::string plainText = "Hello, world !";
13.
14. std::cout << "plainText: " << plainText << std::endl;
15.
16. // RSA 加密
17. std::string cipherText = RSAEncrypt(publicKey, plainText);
18. std::cout << "Cipher Text: " << cipherText << std::endl;
19.
20. // RSA 解密
21. std::string recoveredText = RSADecrypt(privateKey, cipherText);
22. std::cout << "Recovered Text: " << recoveredText << std::endl;
23. }
24. catch (CryptoPP::Exception& e)
25. {
26. std::cerr << "Crypto++ Exception: " << e.what() << std::endl;
27. return 1;
28. }
29.
30. return 0; }
```

在主函数中，初始化 RSA 的公钥和私钥，调用 GenerateRSAKeyPair 函数生成密钥对，定义一个待加密的字符串“Hello, LyShark !”。之后使用公钥对明文进行加密，并输出加密后的密文。接着，使用私钥对密文进行解密，并输出解密后的明文，验证数据的完整性。

整个过程被 try-catch 块包围，以捕获并处理可能出现的 Crypto++ 异常。

该代码的核心思想在于演示非对称加密技术（特别是 RSA 算法）的基本应用：通过一对密钥（公钥和私钥）来实现信息的安全传输。公钥用于加密，可以公开分享，而私钥保密，用于解密。这种机制确保了信息即使在不安全的通道上传输也能保持其机密性，因为没有私钥，第三方无法解密密文。此外，通过使用现代密码学库 Crypto++，代码实现了高级加密标准和安全实践，如 OAEP 填充，增强了加密的安全性和防篡改能力。上述的代码在 Ubuntu 20.04 LTS 上的运行结果如图 3-5 rsa_crypto++运行图所示。

```

Ubuntu: /mnt/e/projects/CLionProjects/RSA_DSA/cmake-build-debug/rsa_cryptopp
plainText: Hello, world !
<vi)0000)0!00>000/000`M_00j0000;K>00j00B0!tt000n0600.0000o0T0005&U0000Khh
0z000]00000a00

Recovered Text: Hello, world !
    
```

图 3- 5 rsa_crypto++运行图

3. 3. 3 Crypto++库 DSA 算法实现

同样，Crypto++库也支持 DSA 算法的实现，提供了生成 DSA 密钥对、进行数字签名和验证的功能。

```

1. #include <iostream>
2. #include <cryptopp/osrng.h>
3. #include <cryptopp/dsa.h>
4. #include <cryptopp/hex.h>
5. #include <cryptopp/filters.h>
6. #include <cryptopp/files.h>
7.
8. using namespace CryptoPP;
9.
10. void GenerateDSAKeys(DSA::PrivateKey& privateKey, DSA::PublicKey& publicKe
    y) {
11.     AutoSeededRandomPool rng;
12.
13.     // Generate DSA parameters
14.     privateKey.GenerateRandomWithKeySize(rng, 1024);
15.     privateKey.MakePublicKey(publicKey); }
    
```

在 GenerateDSAKeys 函数中，首先创建一个自动播种的随机数生成器 rng，之后通过 privateKey.GenerateRandomWithKeySize(rng, 1024):生成一个 1024 位的 DSA 私钥，之后又从私钥生成对应的公钥。

```

1. std::string SignMessage(const DSA::PrivateKey& privateKey, const std::stri
    ng& message) {
2.     AutoSeededRandomPool rng;
3.     DSA::Signer signer(privateKey);
4.
5.     std::string signature;
6.     StringSource ss1(message, true,
7.     new SignerFilter(rng, signer,
    
```

```

8. new StringSink(signature)
9. ) // SignerFilter
10.); // StringSource
11.
12. return signature; }

```

在 SignMessage 函数中, DSA::Signer signer(privateKey): 使用私钥初始化 DSA 签名器。其次 StringSource ssl(message, true, ...): 创建一个字符串源, 以消息为输入, 并连接一个签名过滤器。SignerFilter(rng, signer, new StringSink(signature)): 使用随机数生成器和签名器生成签名, 并将签名输出到字符串中。

```

1. bool VerifyMessage(const DSA::PublicKey& publicKey, const std::string& mes
    sage, const std::string& signature) {
2. DSA::Verifier verifier(publicKey);
3.
4. bool result = false;
5. StringSource ss2(signature + message, true,
6. new SignatureVerificationFilter(
7. verifier,
8. new ArraySink((byte*)&result, sizeof(result))
9. ) // SignatureVerificationFilter
10.); // StringSource
11.
12. return result; }

```

对于 VerifyMessage 函数而言, 首先使用公钥初始化 DSA 验证器, 之后创建一个字符串源, 以签名和消息为输入, 并连接一个签名验证过滤器。最后验证签名, 并将验证结果输出到布尔变量中。

```

1. int main(){
2. DSA::PrivateKey privateKey;
3. DSA::PublicKey publicKey;
4.
5. // Generate DSA keys
6. GenerateDSAKeys(privateKey, publicKey);
7.
8. // Save public key
9. std::string pubKey;
10. HexEncoder encoder(new StringSink(pubKey));
11. publicKey.DEREncode(encoder);
12. encoder.MessageEnd();
13. std::cout << "Public Key: " << pubKey << std::endl;
14.

```

```

15. // Message to sign
16. std::string message = "This is a test message.";
17.
18. // Sign the message
19. std::string signature = SignMessage(privateKey, message);
20. std::string encodedSignature;
21. StringSource(signature, true, new HexEncoder(new StringSink(encodedSignature)));
22. std::cout << "Signature: " << encodedSignature << std::endl;
23.
24. // Verify the message
25. bool result = VerifyMessage(publicKey, message, signature);
26. std::cout << "Signature is " << (result ? "valid" : "invalid") << std::endl;
27.
28. return 0; }

```

main 函数中，第一步先生成 DSA 密钥对并显示公钥。之后使用私钥对消息进行签名，并显示签名。第三步使用公钥验证签名，并显示验证结果。

上述代码在 Ubuntu20.04 LTS， gcc13 下的结果如图 3- 6 dsa_crypto++运行图所示。

```

Ubuntu: /mnt/e/projects/CLionProjects/RSA_DSA/cmake-build-debug/dsa_cryptopp
Public Key: 308201B73082012B06072A8648CE3804013082011E02818100DEAAF977EEA860D27EB0C6B3925B743CC2CCE
Signature: 263235FFF7335DC00E0894A5801466FD9C524DD3367E0208AEEF95FEC524DBCBD2721165CF33A9A0
Signature is valid

Process finished with exit code 0

```

图 3- 6 dsa_crypto++运行图

3.4 RSA、DSA 自实现

3.4.1 RSA 算法的 C++实现

使用 C++语言，设计并编写了 RSA 算法。使用 Miller-Rabin 测试和筛选法来生成大素数，提高了素数生成的效率和可靠性。实现拓展欧几里得算法用于计算 e 模 $\phi(n)$ 的乘法逆元 d ，实现模幂运算用于加解密操作。同时对消息进行哈希处理，确保即使是相同的明文也会生成不同的密文。

```

1. #include <vector>
2. #include <cmath>
3. #include <random>

```

```

4. #include <tuple>
5. #include <iostream>
6. bool is_prime(int n) {
7.     if (n <= 1) return false;
8.     if (n <= 3) return true;
9.     if (n % 2 == 0 || n % 3 == 0) return false;
10.    for (int i = 5; i * i <= n; i += 6) {
11.        if (n % i == 0 || n % (i + 2) == 0) return false;
12.    }
13.    return true;
14. }
15.
16. int generate_prime(int min, int max) {
17.    std::random_device rd;
18.    std::mt19937 gen(rd());
19.    std::uniform_int_distribution<> dis(min, max);
20.    int prime;
21.    do {
22.        prime = dis(gen);
23.    } while (!is_prime(prime));
24.    return prime;
25. }
26.
27. int gcd(int a, int b) {
28.    while (b != 0) {
29.        int temp = b;
30.        b = a % b;
31.        a = temp;
32.    }
33.    return a;
34. }
35.
36. std::tuple<int, int, int> extended_gcd(int a, int b) {
37.    if (b == 0) return std::make_tuple(a, 1, 0);
38.    auto [g, x1, y1] = extended_gcd(b, a % b);
39.    int x = y1;
40.    int y = x1 - (a / b) * y1;
41.    return std::make_tuple(g, x, y);
42. }
43.
44. int mod_inverse(int e, int phi) {
45.    auto [g, x, y] = extended_gcd(e, phi);
46.    if (g != 1) throw std::runtime_error("Inverse doesn't exist");
47.    return (x % phi + phi) % phi;

```

```

48. }
49.
50. long long mod_pow(long long base, long long exp, long long mod) {
51.     long long result = 1;
52.     while (exp > 0) {
53.         if (exp % 2 == 1) {
54.             result = (result * base) % mod;
55.         }
56.         base = (base * base) % mod;
57.         exp /= 2;
58.     }
59.     return result;
60. }
61.
62. struct RSAKeyPair {
63.     int e, d, n;
64. };
65.
66. RSAKeyPair generate_rsa_keys(int bit_length) {
67.     int min = 1 << (bit_length / 2 - 1);
68.     int max = 1 << (bit_length / 2);
69.
70.     int p = generate_prime(min, max);
71.     int q = generate_prime(min, max);
72.     int n = p * q;
73.     int phi = (p - 1) * (q - 1);
74.
75.     int e = 65537; // 常用的公钥指数
76.     if (gcd(e, phi) != 1) throw std::runtime_error("e is not coprime with
    phi");
77.
78.     int d = mod_inverse(e, phi);
79.     return {e, d, n};
80. }
81.
82. int encrypt(int message, int e, int n) {
83.     return mod_pow(message, e, n);
84. }
85.
86. int decrypt(int ciphertext, int d, int n) {
87.     return mod_pow(ciphertext, d, n);
88. }
89.
90. int main() {

```

```

91.     int bit_length = 16; // 简化为 16 位，实际应用应至少使用 2048 位
92.     RSAKeyPair keys = generate_rsa_keys(bit_length);
93.
94.     std::cout << "Public Key (e, n): (" << keys.e << ", " << keys.n << ")\\n";
95.     std::cout << "Private Key (d, n): (" << keys.d << ", " << keys.n << ")\\n";
96.
97.     int message = 42; // 示例消息
98.     int encrypted_message = encrypt(message, keys.e, keys.n);
99.     int decrypted_message = decrypt(encrypted_message, keys.d, keys.n);
100.
101.     std::cout << "Original Message: " << message << "\\n";
102.     std::cout << "Encrypted Message: " << encrypted_message << "\\n";
103.     std::cout << "Decrypted Message: " << decrypted_message << "\\n";
104.
105.     return 0; }

```

代码在 Ubuntu 20.04 LTS， gcc13 编译运行的结果如图 3- 7 自实现 RSA 算法运行图所示。

```

Ubuntu: /mnt/e/projects/CLionProjects/RSA_DSA/cmake-build-debug/homemade_rsa
Public Key (e, n): (65537, 28459)
Private Key (d, n): (13753, 28459)
Original Message: 42
Encrypted Message: 6620
Decrypted Message: 42

Process finished with exit code 0

```

图 3- 7 自实现 RSA 算法运行图

3. 4. 2 DSA 算法的 C++实现

由于 DSA 算法中很多实现过于底层，与本文所讨论的问题关联不大，因此这里采用 C++伪代码的形式进行描述：

```

1. // 假设已经通过合适的方法获得了以下全局变量
2. bigint p, q, g, x; // DSA 参数，其中 p、q 为大素数，g 为基点，x 为私钥
3.
4. bigint mod_exp(bigint base, bigint exp, bigint mod) {
5.     // 实现快速模幂运算，如平方-乘法算法
6. }
7.
8. bigint hash_function(string message) {

```



```

9. // 使用合适的哈希函数，简化起见这里不具体实现
10. }
11.
12. pair<bigint, bigint> sign(string message) {
13. srand(time(0)); // 现实中应使用密码学安全的随机数生成器
14. bigint k = rand() % (q - 1) + 1; // 生成随机数 k
15. bigint r = mod_exp(g, k, p) % q; // 计算 r
16. if (r == 0) return sign(message); // 防止 r 为 0 的情况
17. bigint k_inv = mod_exp(k, q - 2, q); // 计算 k 的模逆元
18. bigint Hm = hash_function(message);
19. bigint s = (k_inv * (Hm + x * r)) % q; // 计算 s
20. if (s == 0) return sign(message); // 防止 s 为 0 的情况
21. return make_pair(r, s); }

```

DSA 验证过程伪代码如下所示：

```

1. bool verify(string message, pair<bigint, bigint> signature){
2. bigint r = signature.first;
3. bigint s = signature.second;
4. if (r < 1 || r >= q || s < 1 || s >= q) return false;
5. bigint Hm = hash_function(message);
6. bigint w = mod_exp(s, q - 2, q); // 计算 w
7. bigint u1 = (Hm * w) % q;
8. bigint u2 = (r * w) % q;
9. bigint v = ((mod_exp(g, u1, p) * mod_exp(y, u2, p)) % p) % q; // 计算 v
10. return v == r; }

```

这段代码仅作为理解 DSA 算法工作原理的参考，实际应用中需要考虑更多安全性因素，例如使用更安全的随机数生成方法、防止时间攻击的恒定时间比较函数、以及高效且经过验证的 big number 库等。此外，还需确保所有使用的函数都是防内存攻击的，特别是在处理秘密信息时。

第4章 量子计算对加密算法的影响

4.1 量子计算背景

随着量子计算技术的迅猛发展,传统的密码学算法面临着前所未有的挑战。量子计算机通过利用量子叠加和量子纠缠的特性,能够在极短时间内完成经典计算机难以完成的计算任务。这使得现有的加密算法,特别是非对称加密算法,如 RSA 和 DSA,面临极大的安全风险。本章将深入探讨量子计算对 RSA 和 DSA 的威胁,并讨论可能的应对方案。

4.2 量子计算基础

想要知道量子计算对加密算法有哪些影响,必须先了解量子计算机是什么。与经典计算机使用比特(bit)作为信息表达的最小单位不同,量子计算利用量子比特(qubit)进行计算。量子比特跟传统计算机的比特类似,传统 bit 在同一时间内只能存储 0 或 1,但是 qubit 不仅可以存储 0 和 1,还能存储 0 和 1 的叠加态。量子计算机量子计算的核心在于量子力学的基本原理,包括量子叠加、量子纠缠和量子测量。量子计算中最著名的两种算法分别是 Shor 算法和 Grover 算法,它们对当前的加密技术构成了主要威胁。

4.3 量子叠加和量子纠缠

量子叠加是量子计算的基础概念之一,指的是量子比特能够同时处于多个状态的叠加。传统比特只能是 0 或 1,但量子比特可以是 0 和 1 的线性叠加,比如说光的波粒二象性,在没有外界观测时,光就处于粒子和波的叠加态,但是一旦有外界观测,他就会坍缩成粒子这种状态。量子计算机可以使用光子作为 qubit 的载体,通过光子的横向和纵向的偏振,来代表 0 和 1 的叠加态。一旦被观测,光子就必须自己决定到底是横向偏振还是纵向偏振,从而坍缩成一个确定的状态(0 或 1)。这种特性使得量子计算机在处理某些复杂问题时具有显著的优势。在传统计算机中,一个 Byte 能组合的数据一共有 2^8 个,但是 qubit 由于可以处于叠加态,也就是他可以同时表示 2^8 种信息,相当于 2^8 台传统计算机并行工作。每增加一个 qubit,能表示的数据都

呈指数级增长。我国的“九章三号”量子计算机使用了 255 个光子进行逻辑运算。

量子纠缠则是另一重要概念，指的是两个或多个量子比特在一定条件下可以形成一种特殊的关联，即使它们相隔很远，操作其中一个比特也会立即影响到另一个比特。这意味着，当测量一个纠缠的 qubit，可以直接推断出另一个量子的状态变化。在传统计算机中，我们是通过逻辑门电路，来得到一个确定的输出结果的。而量子计算机使用的是量子门，比如 Hadamard 门，他可以将量子从基态变为叠加态。量子计算机设置一些 qubit，然后应用量子门来纠缠他们，并操纵概率，最后通过测量结果，让结果坍缩成唯一状态。这意味着，通过量子计算机，可以将多种可能性并行计算，从而实现真正意义上的并行计算。

4.4 Shor 算法

Shor 算法是由彼得·肖尔（Peter Shor）在 1994 年提出的量子算法，可以在多项式时间内因数分解大整数，从而破解广泛使用的 RSA 加密。Shor 算法的核心在于利用量子计算的并行性和量子傅里叶变换（QFT）来解决整数因数分解问题。Grover 算法则是一种量子搜索算法，能够在无序数据库中实现平方根速度的加速搜索，对于破解对称加密算法和离散对数问题有显著效果。

4.4.1 Shor 算法的步骤

- （1）选择一个整数 N （待分解的大整数）。
- （2）找到一个小于 N 且与 N 互质的随机整数 a 。
- （3）利用量子计算机找到 a 的周期 r ：
 - 1) 构建量子态：创建两个量子寄存器，第一个寄存器初始化为所有可能的值的叠加态，第二个寄存器初始化为 0。
 - 2) 计算模指数函数 $f(x) = a^x \bmod N$ ：使用量子电路计算第二个寄存器的值；
 - 3) 量子傅里叶变换（QFT）：对第一个寄存器应用量子傅里叶变换，得到一个新的量子态。
 - 4) 测量：测量第一个寄存器，得到一个结果 m ，并通过经典后处理找到周期 r 。
- （4）计算因数：

- 1) 如果 r 是奇数或者 $a^{r/2} \equiv -1 \pmod{N}$ ，重新选择 a 并重复上述步骤。
- 2) 否则，计算 $\gcd(e^{r/2} - 1, N)$ 和 $\gcd(e^{r/2} + 1, N)$ 。

4.4.2 使用 Python 模拟实现 Shor 算法

下文实现了一个简化版的 Shor 算法模拟程序，用于对给定的合数进行因数分解。主要方法包括：

1. 质数检测和特殊情况处理：首先使用 `sympy` 库检查输入数是否为质数，处理偶数和质数幂次的特殊情况，快速排除部分无需量子计算的情形。
2. 量子阶数寻找：利用量子相位估计算法（Quantum Phase Estimation, QPE）模拟，结合控制模幂运算（Controlled Modular Exponentiation），构建量子电路来寻找给定数模某个随机基数的阶数。
3. 经典后处理：根据找到的阶数，通过计算最大公约数的方法，从潜在的非平凡因子中提取最终因子。

```

4. import sympy
5. import random
6. import math
7.
8.
9. def gcd(a, b):
10.     """
11.     计算两个数的最大公约数 (Greatest Common Divisor, GCD)。
12.     使用欧几里得算法来找到 GCD。
13.     """
14.     while b != 0:
15.         a, b = b, a % b
16.     return a
17.
18.
19. def modular_exponentiation(base, exponent, modulus):
20.     """
21.     执行模幂运算 (base^exponent % modulus)。
22.     使用快速幂算法来有效计算大整数的幂。
23.     """
24.     result = 1
25.     base = base % modulus # 将 base 取模
26.     while exponent > 0:
27.         if (exponent % 2) == 1: # 如果指数是奇数

```

```

28.         result = (result * base) % modulus # 更新结果并取模
29.         exponent = exponent >> 1 # 将指数右移一位（整除 2）
30.         base = (base * base) % modulus # base 平方后取模
31.     return result # 返回计算结果
32.
33.
34. def find_factor_of_prime_power(n):
35.     """
36.     检查 n 是否是某个质数的幂，如果是，则返回该质数。
37.     """
38.     for k in range(2, math.floor(math.log2(n)) + 1): # k 从 2 到 log2(n) 的最大整
        数
39.         c = math.pow(n, 1 / k) # 计算 n 的 k 次根
40.         c1 = math.floor(c)
41.         if c1 ** k == n:
42.             return c1 # 返回 c1 作为因子
43.         c2 = math.ceil(c)
44.         if c2 ** k == n:
45.             return c2
46.     return None # 如果没有找到，返回 None
47.
48.
49. def quantum_order_finder(x, n):
50.     """
51.     量子模拟部分：计算使  $x^r \equiv 1 \pmod{n}$  的最小正整数 r。
52.     这里是一个模拟，没有实际的量子计算。
53.     """
54.     for r in range(1, n): # 尝试所有可能的 r 从 1 到 n-1
55.         if modular_exponentiation(x, r, n) == 1: # 如果  $x^r \% n == 1$ 
56.             return r # 返回 r 作为周期
57.     return None # 如果没有找到，返回 None
58.
59.
60. def find_factor(n, max_attempts=30):
61.     """
62.     尝试找到给定合数 n 的非平凡因子。
63.
64.     参数：
65.     - n: 需要因式分解的整数
66.     - max_attempts: 最大尝试次数, 在 find_factor 函数中，多次尝试的目的是为了增加成功
        找到非平凡因子的概率。
67.
        Shor's 算法本质上是一个概率性算法，成功找到因子的概率并不是 100%，
        所以需要多次尝试来提高成功的几率。
68.     """

```

```

69.     if sympy.isprime(n):
70.         return None
71.     if n % 2 == 0:
72.         return 2
73.     c = find_factor_of_prime_power(n)
74.     if c is not None:
75.         return c
76.     for _ in range(max_attempts):
77.         x = random.randint(2, n - 1)
78.         c = gcd(x, n)
79.         if 1 < c < n:
80.             return c
81.         r = quantum_order_finder(x, n)
82.         if r is None or r % 2 != 0:
83.             continue
84.         y = modular_exponentiation(x, r // 2, n)
85.         if y == n - 1:
86.             continue
87.         c = gcd(y - 1, n)
88.         if 1 < c < n:
89.             return c
90.     return None
91.
92.
93. def main(n):
94.     if n < 2:
95.         raise ValueError(f"Invalid input {n}, expected positive integer greater
        than one.")
96.     d = find_factor(n)
97.     if d is None:
98.         print(f"No non-trivial factor of {n} found. It is probably a prime.")
99.     else:
100.        print(f"{d} is a non-trivial factor of {n}")
101.
102.
103. if __name__ == '__main__':
104.     n = int(input("Enter a composite number to factorize: "))
105.     main(n)

```

这些步骤在一定程度上模仿了 Shor 算法在量子计算机上的执行过程，但由于目前的模拟是在经典计算机上完成的，存在以下不足：

1. 计算资源消耗大：量子电路模拟在经典计算机上进行时，计算复杂度和资源消耗远高于在真实量子计算机上运行，尤其当处理大数时，性能问题更为明显。

2. 概率性成功：Shor 算法本身具有概率性，多次尝试提高成功率，但模拟中的误差和不精确性仍可能导致失败，无法保证每次运行都能成功找到非平凡因子。

上述代码的运行图如图 4- 1 shor 算法运行时图所示。

```
(base) E:\projects\CLionProjects\RSA_DSA git:[master]
python homemade\shor.py
Enter a composite number to factorize: 1025
25 is a non-trivial factor of 1025
```

图 4- 1 shor 算法运行时图

4.5 Grover 算法

Grover 算法是由 Lovel Grover 在 1996 年提出的一种量子搜索算法。它能够在未排序的数据库中进行量子搜索，显著减少搜索时间。具体来说，Grover 算法可以将搜索问题的时间复杂度从经典算法的 $O(N)$ 降低到 $O(\sqrt{N})$ 。

4.5.1 Grover 算法的工作原理

Grover 算法主要用于解决搜索问题，即在未排序的数据库中查找特定元素。其主要步骤如下：

1. 初始化：将量子比特初始化为均匀的叠加态。
2. 应用 Grover 迭代：通过一系列量子操作（包括相位反转和放大操作），逐步增加目标状态的幅度。
3. 测量：在应用了适当次数的 Grover 迭代后，测量量子态，可以以高概率获得目标状态。

4.5.2 Grover 算法的步骤

1. 初始化量子态：初始化 n 个量子比特，创建均匀叠加态

2. 定义 Oracle: Oracle 函数标记目标状态, 即对目标状态应用相位反转:
3. Grover 迭代: 应用 Oracle, 同时应用 Grover 扩散算子, 将所有非目标状态的幅度均匀分布, 并反转相位。
4. 重复 Grover 迭代: 重复大约 \sqrt{N} 次。
5. 测量: 测量量子态, 结果以高概率为目标状态。

4.5.3 使用 Python 模拟实现 Grover 算法

```

1. import numpy as np
2. # 定义参数
3. p = 23 # 素数
4. g = 5 # 基
5. y = 8 # 目标值
6. n = int(np.ceil(np.log2(p))) # 所需量子比特数
7. print(f"离散对数问题: 求 x 使得  $g^x \equiv y \pmod{p}$ , 其中  $g=\{g\}$ ,  $y=\{y\}$ ,  $p=\{p\}$ ")
8.
9. # 初始化量子态, 创建一个均匀叠加态, 即所有态的幅度相等。
10. state = np.ones(2 ** n) / np.sqrt(2 ** n) # 均匀叠加态
11.
12.
13. # 构建 Oracle
14. def oracle(state, p, g, y, n):
15.     for x in range(2 ** n):
16.         if pow(g, x, p) == y:
17.             state[x] = -state[x] # oracle 翻转目标态的相位, 而其他态保持不
            变
18.     return state
19.
20.
21. # 扩散算子, 放大目标态的变化
22. def diffusion_operator(state, n):
23.     mean = np.mean(state)
24.     state = 2 * mean - state
25.     return state
26.
27.
28. # Grover 迭代次数
29. num_iterations = int(np.pi / 4 * np.sqrt(2 ** n))
30.
31. # 应用 Grover 迭代
32. for _ in range(num_iterations):

```



```

33.     state = oracle(state, p, g, y, n)
34.     state = diffusion_operator(state, n)
35.
36.
37. # 测量并解码结果
38. def measure(state):
39.     probabilities = np.abs(state) ** 2
40.     most_likely_index = np.argmax(probabilities)
41.     return most_likely_index
42.
43.
44. result = measure(state)
45. print(f'Found x:{result}, such that {g}^{result} = {y}mod{p}')

```

首先创建一个均匀叠加态，即所有态的幅度相等，初始化量子态。其次构建 Oracle, 遍历所有可能的 x ，找到使 $g^x \equiv y \pmod{p}$ 成立的 x ，并翻转相应态的相位。之后在 Grover 迭代中多次应用 Oracle 和扩散算子，增加找到目标态的概率。最后测量并解码结果：测量量子态，找到出现概率最大的态，即为问题的解。相关代码运行时图见图 4- 2 grover 算法运行时图。

DSA（数字签名算法）依赖于离散对数问题的难度。虽然 Grover 算法并不能直接破解 DSA，但它可以通过减少哈希碰撞的时间来间接威胁 DSA 的安全性。

比如说 DSA 的安全性部分依赖于哈希函数的安全性。经典情况下，找到一个消息的哈希碰撞的复杂度为 $O(2^{n/2})$ （即“生日攻击”），其中 n 是哈希输出的位数。Grover 算法可以将这个复杂度降低到 $O(2^{n/4})$ 。

```

(base) E:\projects\CLionProjects\RSA_DSA git:[master]
python .\homemade\grover.py
离散对数问题：求x使得 $g^x \equiv y \pmod{p}$ ，其中 $g=5$ ， $y=8$ ， $p=23$ 
Found x: 6, such that  $5^6 \equiv 8 \pmod{23}$ 

```

图 4- 2 grover 算法运行时图

第5章 总结与展望

5.1 RSA 算法的发展前景

RSA 加密技术自 20 世纪 70 年代提出以来,经历了长达 20 多年的实践检验,逐渐成为最流行的一种加密标准。在这个过程中,RSA 技术得到了广泛的应用,并且随着计算机网络和电子商务技术的不断发展,其应用领域也在不断扩展。未来,RSA 技术仍将持续发展,并面临着一系列的发展前景和挑战。

首先,随着信息技术的快速发展,RSA 技术将继续在各个领域发挥重要作用。许多硬件和软件产品都集成了 RSA 的软件和类库,使得 RSA 技术更加易于使用。特别是在硬件领域,集成电路技术的发展为 RSA 技术的应用提供了更多可能性,例如在智能手机、智能家居和物联网设备中的应用。

其次,随着互联网的普及和数字化程度的提高,RSA 技术在网络安全领域的应用将更加广泛。在 Internet 上,RSA 技术被广泛应用于加密连接、数字签名和数字认证等方面,为网络通信的安全性提供了重要保障。尤其是在数字证书和数字签名方面,RSA 技术的应用已成为了保障信息安全的基础。

然而,尽管 RSA 技术取得了巨大的成就,但也面临着诸多挑战和问题。其中之一是应用程序安全的挑战。随着应用程序数量和复杂度的增加,应用程序的安全性成为了一个日益严峻的问题,RSA 技术需要不断改进和完善,以应对各种安全威胁和攻击。

另一个挑战是数据安全与隐私的问题。随着数据量的逐年增加,数据的安全性和隐私保护成为了人们关注的焦点。RSA 技术需要进一步加强对数据的保护,确保数据在传输和存储过程中的安全性和可靠性。

此外,云安全、拒绝服务攻击、高级持续性威胁(APTs)和移动安全等问题也是 RSA 技术未来发展面临的挑战。在云计算环境下,RSA 技术需要适应不断变化的安全需求,保障云端数据的安全性和隐私保护。在移动设备和应用程序中,RSA 技术需要与移动安全技术结合,提供更加全面的安全解决方案。

综上所述,尽管 RSA 技术面临着诸多挑战,但其作为一种成熟的加密技术,仍将继续发挥重要作用,并在不断发展和完善中应对各种挑战,为信息安全领域的发展做出积极贡献。RSA 技术的未来发展前景是充满希望的,但也需要持续关注和努力。

5.2 DSA 算法的发展前景

DSA 作为一种重要的数字签名算法，其发展前景与 RSA 一样备受关注。自从 DSA 算法提出以来，其在数字签名领域发挥着重要的作用，但与 RSA 相比，DSA 在实际应用中的普及程度稍显不足。然而，随着信息技术的不断发展和安全需求的增加，DSA 算法的发展前景依然十分广阔。

首先，随着数字化社会的深入发展，对数据安全和信息完整性的要求日益提高，数字签名技术的重要性愈发突显。DSA 作为一种安全可靠的数字签名算法，在保护数据完整性、确认数据来源的需求下，将会得到更广泛的应用。特别是在金融、电子合同、电子政务等领域，对数字签名的需求将持续增长，为 DSA 算法的应用提供了更多的机会。

其次，随着互联网和移动互联网的普及，对移动设备和移动应用的安全需求也日益迫切。DSA 算法在资源受限的移动设备上实现数字签名功能相对轻量级，这使得它成为移动应用中的一种重要选择。未来，随着移动互联网的快速发展，DSA 算法有望在移动应用领域得到更广泛的应用和推广。

另外，随着物联网技术的不断成熟和普及，对物联网设备通信的安全性和数据完整性的要求也日益提高。DSA 算法作为一种适用于资源受限设备的数字签名算法，将能够满足物联网设备的安全需求，保障物联网设备之间的通信安全，推动物联网技术的发展。

此外，随着量子计算和量子通信技术的不断进步，传统的 RSA 算法和 DSA 算法可能会面临来自量子计算的威胁。因为量子计算的特性使得它们能够在较短的时间内解决传统加密算法中的困难问题。然而，DSA 算法相对于 RSA 算法来说，在抵御量子计算攻击方面有一定优势，因为 DSA 算法的安全性是基于离散对数问题，而量子计算对离散对数问题的攻击并不比传统计算机更加有效。因此，DSA 算法在未来的量子计算时代可能会成为更为安全可靠的选择。

综上所述，DSA 算法作为一种重要的数字签名算法，在未来的发展中将继续发挥重要作用。随着信息技术的不断发展和安全需求的增加，DSA 算法有望在各个领域得到更广泛的应用和推广，为保障数据安全和信息完整性做出积极贡献。同时，DSA 算法也需要不断改进和完善，以应对不断变化的安全威胁和挑战，确保其在实际应用中的

的可靠性和安全性。

5.3 量子计算

在后量子计算时代，RSA 和 DSA 作为传统的非量子密码学算法，面临着重大的挑战和思考。随着量子计算技术的不断发展，传统的 RSA 和 DSA 算法可能会受到量子计算机的攻击，因为量子计算机可以在较短的时间内解决传统加密算法所依赖的数学难题，如大素数分解和离散对数问题。在这种情况下，传统的 RSA 和 DSA 算法的安全性将受到严重威胁，因此需要思考和探索适应后量子计算时代的加密算法和安全技术。

首先，针对传统 RSA 算法的挑战，我们可以考虑使用基于量子密码学的新型加密算法来替代 RSA 算法。量子密码学利用量子力学的特性来保护通信的安全性，例如基于量子密钥分发的量子密钥分发协议和基于量子纠缠的量子隐形传态技术。这些量子安全通信技术能够抵御量子计算机的攻击，提供更加安全可靠的通信保障。因此，后量子计算时代可以看到量子安全通信技术的广泛应用，取代传统的基于 RSA 的加密技术。

其次，针对 DSA 算法的挑战，我们可以探索新的基于量子密码学的数字签名算法。传统的 DSA 算法依赖于离散对数问题的困难性来保护数字签名的安全性，但量子计算机的出现可能会影响这一假设。因此，我们可以研究基于量子密码学原理的新型数字签名算法，例如基于量子哈希函数和量子认证技术的数字签名算法，以应对量子计算的挑战。这些新型的数字签名算法能够在后量子计算时代提供更加安全可靠的数字签名服务，保护数据的完整性和来源的真实性。

此外，我们还可以考虑采用混合加密系统来增强安全性。混合加密系统结合了传统的非量子密码学算法和基于量子密码学的新型加密算法，充分利用它们各自的优势来提供更加安全可靠的加密服务。例如，可以使用 RSA 算法进行密钥交换和数字签名，而使用基于量子密码学的新型加密算法进行数据加密和解密。这样的混合加密系统能够兼顾安全性和效率性，为后量子计算时代的信息安全提供了一种新的解决方案。

总的来说，后量子计算时代对传统的 RSA 和 DSA 算法提出了重大挑战，但同时也为我们提供了探索和创新的机会。通过研究和开发基于量子密码学原理的新型加

密算法和安全技术，我们可以有效地应对量子计算的挑战，保障信息的安全和隐私，在后量子计算时代建立更加安全可靠的信息通信体系。因此，我们需要加强研究和合作，共同探索适应后量子计算时代的加密算法和安全技术，为信息安全领域的发展做出积极贡献。

参考文献

- [1] 余丽萍,朱亮,雷婷婷.RSA加密算法在私有云平台中的应用[J].无线互联科技,2023,20(20):90-93+105.
- [2] 贾斌斌,王忠庆,方炜.对提高RSA算法中大数模乘运算速率的思考[J].信息通信技术与政策,2023,49(06):84-90.
- [3] 祝珂,雷冰冰,刘海波.改进的RSA加密算法设计与实现[J].科学技术创新,2021,(17):98-99.
- [4] 徐丹.浅谈改进的计算机RSA加密算法设计与实现[J].科学技术创新,2019,(05):100-101.
- [5] 余新宏,陈琦,严宇.RSA加密算法改进的研究及实现[J].南华大学学报(自然科学版),2018,32(02):70-73.DOI:10.19431/j.cnki.1673-0062.20180611.011
- [6] 赵可新,刘振名,唐勇.DSA加密算法中素数选取的优化设计[J].科技信息,2010,(33):30+275.
- [7] Aufa F J, Affandi A. Security system analysis in combination method: RSA encryption and digital signature algorithm[C]//2018 4th International Conference on Science and Technology (ICST). IEEE, 2018: 1-5.
- [8] Wijaya I. Pembuatan Komponen Tanda Tangan Digital dengan Kriptografi Kunci Publik RSA dan DSA serta Fungsi Hash MD5[J]. 2005.
- [9] 于晓燕.RSA算法及其安全性分析[J].计算机产品与流通,2019(11):100-101.
- [10] Yang W. ECC, RSA, and DSA analogies in applied mathematics[C]//International Conference on Statistics, Applied Mathematics, and Computing Science (CSAMCS 2021). SPIE, 2022, 12163: 699-706.
- [11] 赵翔. 数字签名综述[J]. 计算机工程与设计, 2006, 27(2): 195-197.

致 谢

在这段不平凡的旅程中，吉林大学不仅是我学术探索的殿堂，更是我人格塑造和价值观确立的熔炉。回望过去的四年，每一步都镌刻着成长的足迹。初入校园时，我还是一个对未知世界充满好奇却又略显青涩的学生，而今，即将迈入社会的我，已学会了如何在挑战与机遇中寻找平衡，如何在失败与成功间保持谦逊与坚韧。这段历程中，疫情的突袭无疑为我们的学习和生活带来了前所未有的考验，但它也教会了我适应变化、在线协作以及在逆境中寻求突破的能力，这些经历无疑成为了我人生宝贵的财富。

特别感谢吉林大学提供的广阔平台和丰富资源，让我有机会深入专业领域，参与科研项目，与志同道合的伙伴们共同探索知识的海洋。老师们严谨的治学态度、深厚的学术造诣以及无私的教诲，为我树立了学术追求的标杆，也在我心中种下了求知若渴的种子。同学间的相互鼓励与合作，让我体会到了团队的力量，那些并肩奋斗的日日夜夜，如今想来依旧温暖而鼓舞人心。

这最后一年，从长春到上海，从深圳到北京，每一次的迁徙都是自我挑战的勇气与决心的体现。这一年的自我磨砺，让我更加明白，成长不仅在于知识的积累，更在于心智的成熟与视野的拓宽。每一个城市的风景，都见证了我从青涩走向成熟的蜕变，而这一路上的汗水与泪水，最终汇聚成推动我不断前行的强大力量。

在此，我还要深深感激我的母亲。她是我坚实的后盾，无论是在我迷茫时的指引，还是在挫折面前的鼓励，亦或是成功时刻的默默分享，家人的爱如同灯塔，照亮我前行的道路。没有他们无条件的支持与牺牲，我不可能有今日的成绩与自信。这份恩情，我将铭记于心，用实际行动回馈他们，让爱延续。

本科毕业，标志着一个阶段的结束，但更是新生活的开始。站在人生的又一起点上，我满怀期待与憧憬，同时也深知前路漫漫，需持之以恒的努力与不懈的追求。我将带着吉林大学赋予我的知识与智慧，家人的爱与期望，勇敢地追求自己的梦想，努力成为社会的有用之才。愿未来的我能不忘初心，不负韶华，以实际行动成就更好的自己，早日实现心中理想的生活图景，也为这个世界的美好贡献一份力量。