

**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

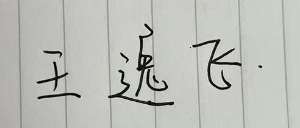
**专 业** 软件工程

**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：

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；

**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](#_Toc168080005)

[1.1 研究背景 1](#_Toc168080006)

[1.2 国内外研究现状 2](#_Toc168080007)

[1.2.1 RSA研究现状 2](#_Toc168080008)

[1.2.2 DSA研究现状 2](#_Toc168080009)

[1.3 论文的主要工作 3](#_Toc168080010)

[1.4 论文的组织架构 4](#_Toc168080011)

[第2章 相关数学基础、RSA算法与DSA算法 5](#_Toc168080012)

[2.1 相关数学基础 5](#_Toc168080013)

[1.1.1 大数因子分解问题 5](#_Toc168080014)

[1.1.2 离散对数难题 5](#_Toc168080015)

[2.2 密码学综述 6](#_Toc168080016)

[2.2.1 密码学基础 6](#_Toc168080017)

[2.2.2 加密算法描述 6](#_Toc168080018)

[2.3 RSA算法 7](#_Toc168080019)

[2.3.1 RSA算法内容 7](#_Toc168080020)

[2.3.2 RSA算法的安全性 9](#_Toc168080021)

[2.3.3 RSA算法的应用 10](#_Toc168080022)

[2.4 DSA算法 11](#_Toc168080023)

[2.4.1 DSA算法内容 11](#_Toc168080024)

[2.4.2 DSA算法的安全性 12](#_Toc168080025)

[2.4.3 DSA算法的应用 14](#_Toc168080026)

[第3章 OpenSSL、Botan、Crypto++算法与自实现算法 16](#_Toc168080027)

[3.1 OpenSSL库与RSA、DSA实现 16](#_Toc168080028)

[3.1.1 OpenSSL库 16](#_Toc168080029)

[3.1.2 OpenSSL中RSA实现源码分析 16](#_Toc168080030)

[3.1.3 OpenSSL RSA算法实现 17](#_Toc168080031)

[3.1.4 OpenSSL DSA算法实现 23](#_Toc168080032)

[3.2 Botan库与RSA、DSA实现 27](#_Toc168080033)

[3.2.1 Botan库 27](#_Toc168080034)

[3.2.2 Botan RSA算法实现 28](#_Toc168080035)

[3.2.3 Botan DSA算法实现 31](#_Toc168080036)

[3.3 Crypto++库与RSA、DSA实现 34](#_Toc168080037)

[3.3.1 Crypto++库 34](#_Toc168080038)

[3.3.2 Crypto++库RSA算法实现 35](#_Toc168080039)

[3.3.3 Crypto++库DSA算法实现 38](#_Toc168080040)

[3.4 RSA、DSA自实现 40](#_Toc168080041)

[3.4.1 RSA算法的C++实现 40](#_Toc168080042)

[3.4.2 DSA算法的C++实现 43](#_Toc168080043)

[第4章 量子计算对加密算法的影响 44](#_Toc168080044)

[4.1 引言 44](#_Toc168080045)

[4.2 量子计算基础 45](#_Toc168080046)

[4.3 量子叠加和量子纠缠 45](#_Toc168080047)

[4.4 Shor算法 46](#_Toc168080048)

[4.4.1 Shor算法的步骤 46](#_Toc168080049)

[4.4.2 使用Python模拟实现Shor算法 47](#_Toc168080050)

[4.5 Grover算法 50](#_Toc168080051)

[4.5.1 Grover算法的工作原理 50](#_Toc168080052)

[4.5.2 Grover算法的步骤 50](#_Toc168080053)

[4.5.3 使用Python模拟实现Grover算法 50](#_Toc168080054)

[第5章 总结与展望 52](#_Toc168080055)

[5.1 RSA算法的发展前景 52](#_Toc168080056)

[5.2 DSA算法的发展前景 53](#_Toc168080057)

[5.3 量子计算 54](#_Toc168080058)

[参考文献 57](#_Toc168080059)

[致 谢 58](#_Toc168080060)

# 绪论

## 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。

## 国内外研究现状

### 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。

被广泛用于信息安全领域。国内外对DSA算法的研究现状如下：

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。

## 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。

## 论文的组织架构

1. 绪论：介绍了本文研究背景、国内外研究现状、主要工作。
2. 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。
3. 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。
4. 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。
5. 展望对RSA和DSA算法在后量子时代的发展。

# 相关数学基础、RSA算法与DSA算法

## 相关数学基础

### 大数因子分解问题

因子分解问题是指将一个合数分解成质因子的乘积。例如，给定，其质因子分解为。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的质数，看看它们是否是的因子。这种方法的时间复杂度为，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。

### 离散对数难题

离散对数难题是指在给定一个有限域和一个生成元𝑔的情况下，找到一个整数𝑥，使得成立，其中ℎ是中的一个元素。例如，给定,，在中，找到𝑥，使得。在这种情况下，因为。

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，这种方法的时间复杂度是，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。

## 密码学综述

### 密码学基础

尽管密码学有着悠久的研究历史，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。

### 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。

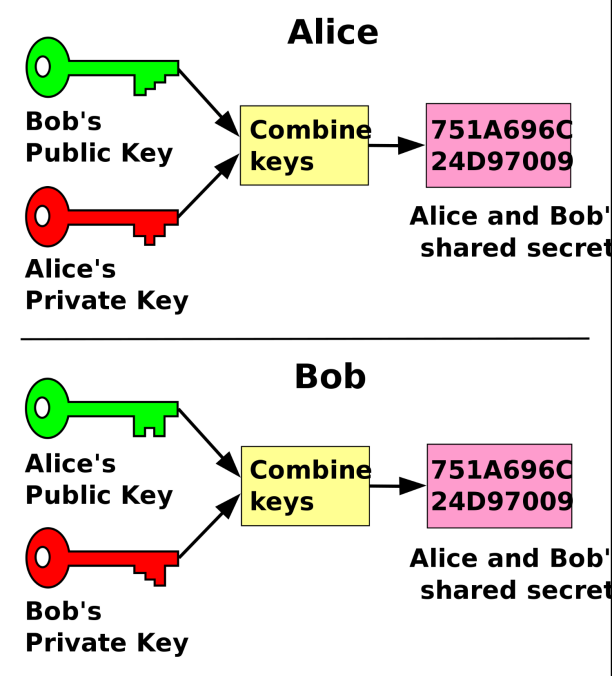
在简单的流程中，甲方生成一对密钥，并将其中一把作为公开密钥向其他方公开；乙方得到该公开密钥后使用它对机密信息进行加密，然后发送给甲方；甲方再使用自己保存的另一把专用密钥对加密后的信息进行解密。甲方只能使用其专用密钥来解密由其公开密钥加密后的任何信息。

图2- 1 Alice & Bob问题

## RSA算法

### 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**：
2. **选择两个大质数**：随机选取两个足够大的质数和 。
3. **计算模数**：计算这两个质数的乘积 ，将作为公钥和私钥的模数使用。
4. **计算欧拉函数φ(n)**：。
5. **选择公钥指数**：选取一个与互质的整数 ，满足通常  取值为65537（因为它满足互质且加密效率较高）。
6. **计算私钥指数**：找到一个整数 作为的逆模元，使得 。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。
7. **公钥**：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。
8. **私钥**：作为私钥保密存储，只有持有者可以用来解密信息。
9. **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使用其公钥对消息M进行加密，计算密文 。

1. **解密过程**：

用户A接收到密文C后，使用其私钥进行解密，计算原文 。

由于，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。

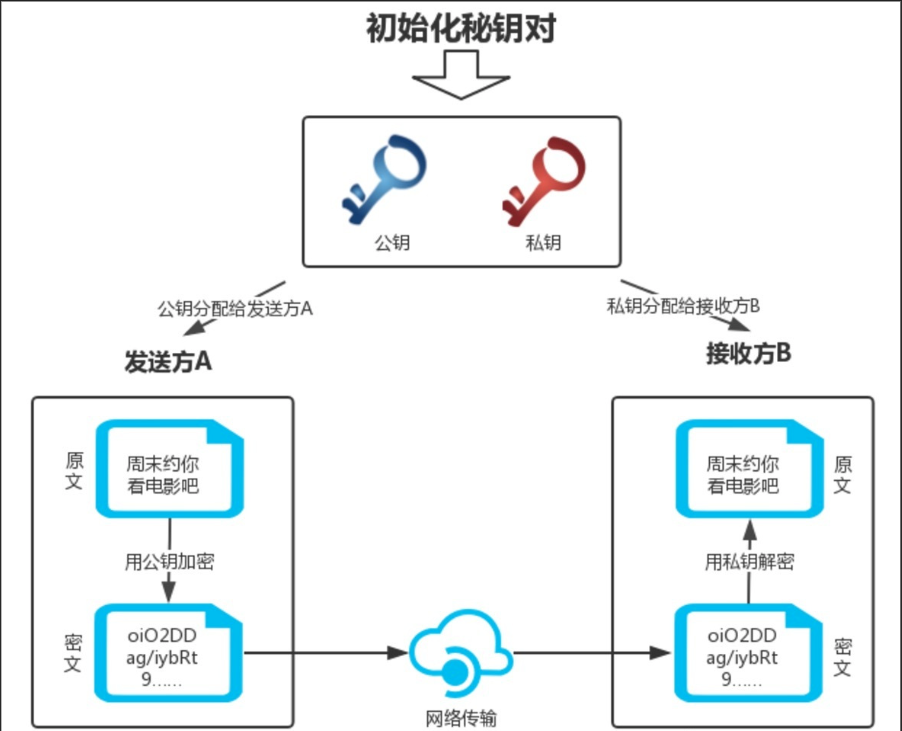


图2- 2 RSA应用图示

### RSA算法的安全性

在传统的数据加密体制中，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。

### 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。

## DSA算法

### 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**：
2. **选择两个大素数**：随机选取一个大素数和另一个较小的素数 ，其中 应该是 的因子。
3. **计算生成元**：选择一个整数 （通常 ），满足 ，是一个生成元。
4. **私钥**：随机选择一个小于 的整数 作为私钥。
5. **公钥**：计算 ，作为公钥发布。
6. **签名生成**：

发送方用私钥 和一个随机数 （，且 不能重复使用）来签署消息 。

计算 ，要求 。

计算，其中 是消息的哈希值，且  是 在模 下的乘法逆元。

签名对为。

1. **签名验证**：

接收方使用发送方的公钥 和消息 来验证签名的有效性。

计算 ，即 s在模 q 下的乘法逆元。

计算和 。

计算。

如果 ，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。

### 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 ，其中 y 是已知的。目前，没有已知的有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中具有重要地位，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。

### 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。

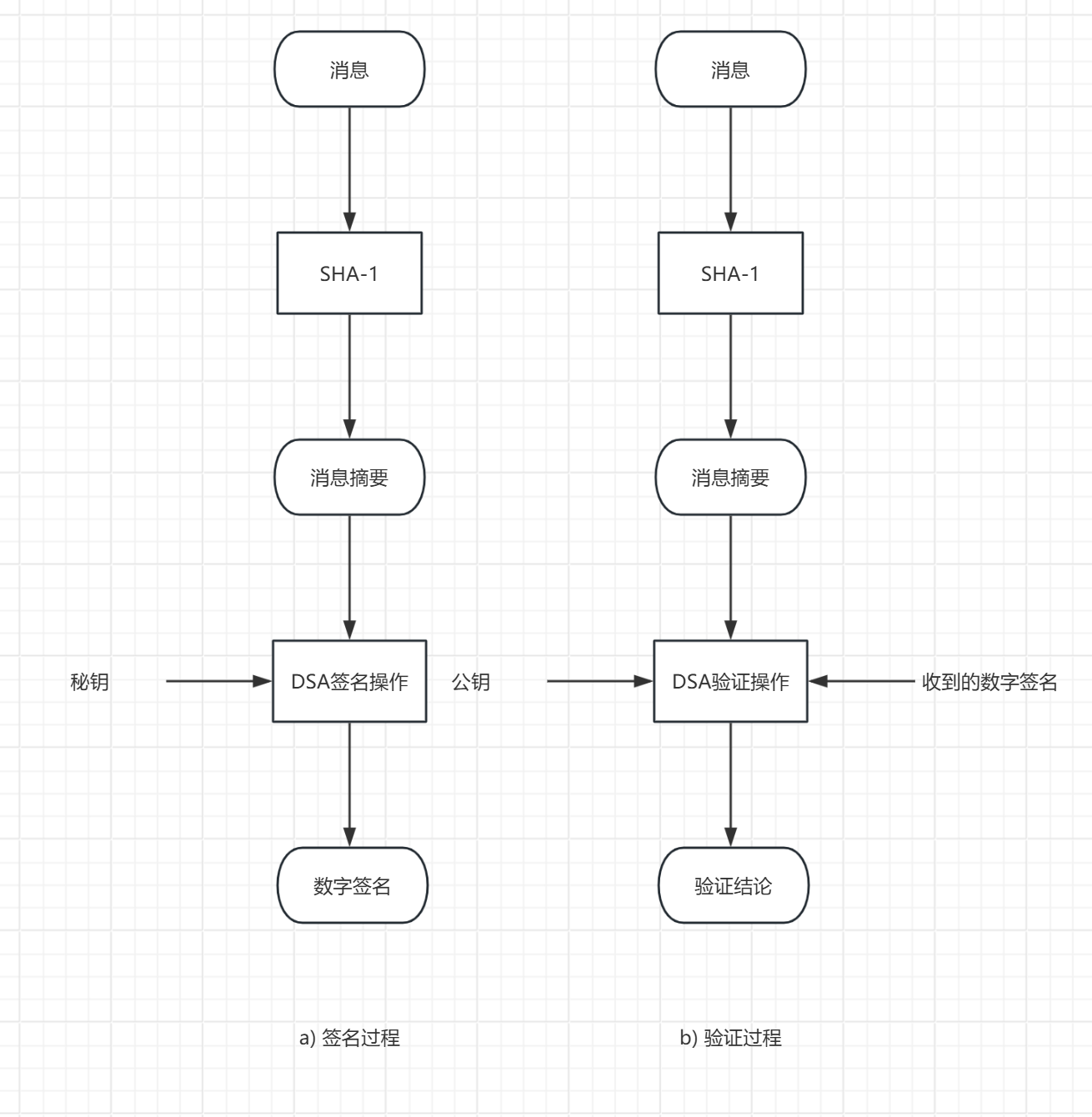


图2- 3 DSA签名算法流程

# 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。

## OpenSSL库与RSA、DSA实现

### 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。

### 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 \*dmpl;// d mod (p-1)
9. BIGNUM \*dmql;// 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。

### 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>
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. }
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. }
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. }
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\_PKCS1\_OAEP\_PADDING);
37. **if** (len == -1) {
38. handleErrors();
39. }
40. **return** ciphertext;
41. }
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\_PKCS1\_OAEP\_PADDING);
48. **if** (len == -1) {
49. handleErrors();
50. }
51. plaintext.resize(len);
52. **return** plaintext;
53. }
55. **int** main() {
56. // Initialize OpenSSL
57. OpenSSL\_add\_all\_algorithms();
58. ERR\_load\_BIO\_strings();
59. ERR\_load\_crypto\_strings();
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. }
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());
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);
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);
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);
84. BIO\_read(bp\_public, pub\_key, pub\_len);
85. BIO\_read(bp\_private, priv\_key, priv\_len);
87. pub\_key[pub\_len] = '\0';
88. priv\_key[priv\_len] = '\0';
90. std::cout << "Public Key: " << std::endl << pub\_key << std::endl;
91. std::cout << "Private Key: " << std::endl << priv\_key << std::endl;
93. // Clean up
94. BIO\_free\_all(bp\_public);
95. BIO\_free\_all(bp\_private);
96. free(pub\_key);
97. free(priv\_key);
99. // Encrypt message
100. std::string plaintext = "Hello, OpenSSL RSA!";
101. std::string ciphertext = rsaEncrypt(rsa, plaintext);
102. std::cout << "Encrypted: " << ciphertext << std::endl;
104. // Decrypt message
105. std::string decryptedtext = rsaDecrypt(rsa, ciphertext);
106. std::cout << "Decrypted: " << decryptedtext << std::endl;
108. // Free RSA key
109. RSA\_free(rsa);
111. // Clean up OpenSSL
112. CRYPTO\_cleanup\_all\_ex\_data();
113. ERR\_free\_strings();
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。

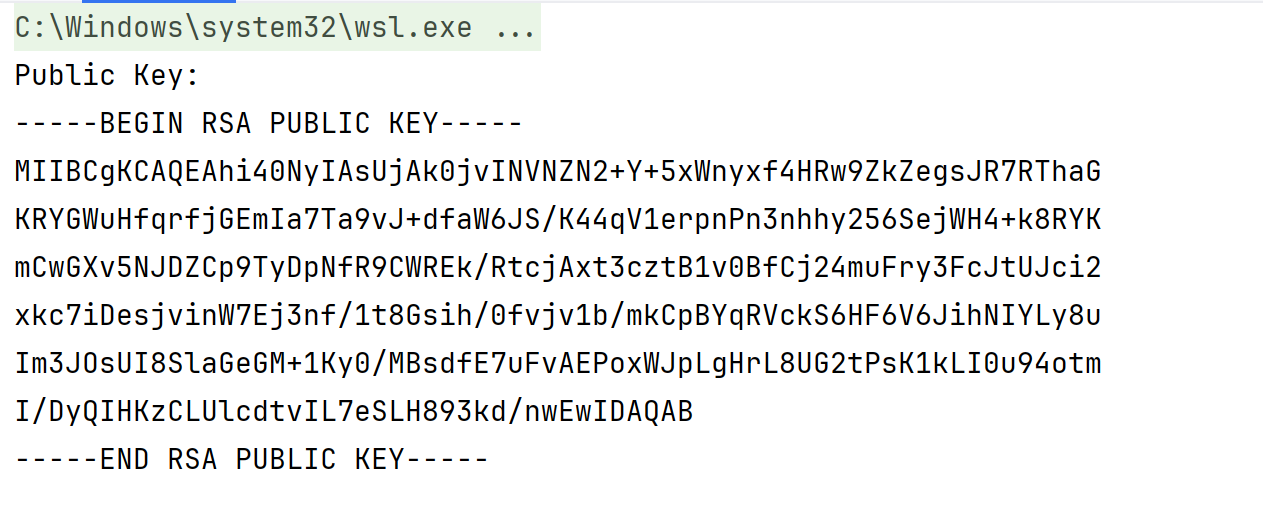
上述的代码在Ubuntu 20.04 LTS上的运行结果如图3- 1和图3- 2所示。

图3- 1 rsa\_openssl运行图1



图3- 2 rsa\_openssl运行图2

### 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>
8. **void** handleErrors() {
9. ERR\_print\_errors\_fp(stderr);
10. abort();
11. }
13. DSA\* generateDSAKeyPair() {
14. DSA\* dsa = DSA\_new();
15. **if** (!DSA\_generate\_parameters\_ex(dsa, 2048, NULL, 0, NULL, NULL, NULL)) {
16. handleErrors();
17. }
18. **if** (!DSA\_generate\_key(dsa)) {
19. handleErrors();
20. }
21. **return** dsa;
22. }
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);
28. std::vector<unsigned **char**> sig(DSA\_size(dsa));
29. unsigned **int** sig\_len;
31. **if** (DSA\_sign(0, hash, SHA256\_DIGEST\_LENGTH, sig.data(), &sig\_len, dsa) == 0) {
32. handleErrors();
33. }
35. sig.resize(sig\_len);
36. **return** sig;
37. }
39. **bool** dsaVerify(DSA\* dsa, **const** std::string& message, **const** std::vector<unsigned **char**>& sig) {
40. unsigned **char** hash[SHA256\_DIGEST\_LENGTH];
41. SHA256((unsigned **char**\*)message.c\_str(), message.size(), hash);
43. **return** DSA\_verify(0, hash, SHA256\_DIGEST\_LENGTH, sig.data(), sig.size(), dsa) == 1;
44. }
46. **int** main() {
47. // Initialize OpenSSL
48. OpenSSL\_add\_all\_algorithms();
49. ERR\_load\_BIO\_strings();
50. ERR\_load\_crypto\_strings();
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. }
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);
65. **size\_t** pub\_len = BIO\_pending(bp\_public);
66. **size\_t** priv\_len = BIO\_pending(bp\_private);
68. **char**\* pub\_key = (**char**\*)malloc(pub\_len + 1);
69. **char**\* priv\_key = (**char**\*)malloc(priv\_len + 1);
71. BIO\_read(bp\_public, pub\_key, pub\_len);
72. BIO\_read(bp\_private, priv\_key, priv\_len);
74. pub\_key[pub\_len] = '\0';
75. priv\_key[priv\_len] = '\0';
77. std::cout << "Public Key: " << std::endl << pub\_key << std::endl;
78. std::cout << "Private Key: " << std::endl << priv\_key << std::endl;
80. // Clean up
81. BIO\_free\_all(bp\_public);
82. BIO\_free\_all(bp\_private);
83. free(pub\_key);
84. free(priv\_key);
86. // Sign message
87. std::string message = "Hello, OpenSSL DSA!";
88. std::vector<unsigned **char**> signature = dsaSign(dsa, message);
89. std::cout << "Signature: ";
90. **for** (unsigned **char** c : signature) {
91. printf("%02x", c);
92. }
93. std::cout << std::endl;
95. // Verify signature
96. **bool** is\_valid = dsaVerify(dsa, message, signature);
97. std::cout << "Signature is " << (is\_valid ? "valid" : "invalid") << std::endl;
99. // Free DSA key
100. DSA\_free(dsa);
102. // Clean up OpenSSL
103. CRYPTO\_cleanup\_all\_ex\_data();
104. ERR\_free\_strings();
106. **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- 3和图3- 4是相关代码在Ubuntu 20.04上运行的结果。



图3- 3 dsa\_openssl

图3- 4 dsa\_openssl运行图2

## Botan库与RSA、DSA实现

### 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。

### 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. **int** main() {
11. **try** {
12. // 初始化随机数生成器
13. Botan::AutoSeeded\_RNG rng;
15. // 生成RSA密钥对
16. Botan::RSA\_PrivateKey private\_key(rng, 2048);
17. Botan::RSA\_PublicKey public\_key = private\_key;
19. // 获取私钥和公钥的PEM格式字符串
20. std::string private\_key\_pem = Botan::PKCS8::PEM\_encode(private\_key);
21. std::string public\_key\_pem = Botan::X509::PEM\_encode(public\_key);
23. // 打印私钥和公钥
24. std::cout << "Private Key:\n"
25. << private\_key\_pem << std::endl;
26. std::cout << "Public Key:\n"
27. << public\_key\_pem << std::endl;
29. // 要加密的消息
30. std::string message = "Hello, Botan!";
32. // 加密消息
33. Botan::PK\_Encryptor\_EME encryptor(public\_key, rng, "EME1(SHA-256)");
34. std::vector<uint8\_t> ciphertext = encryptor.encrypt(
35. **reinterpret\_cast**<**const** uint8\_t \*>(message.data()), message.size(), rng);
37. std::cout << "Encrypted message: " << Botan::hex\_encode(ciphertext) << std::endl;
39. // 解密消息
40. Botan::PK\_Decryptor\_EME decryptor(private\_key, rng, "EME1(SHA-256)");
41. Botan::secure\_vector<uint8\_t> decrypted = decryptor.decrypt(ciphertext.data(), ciphertext.size());
43. std::string decrypted\_message(decrypted.begin(), decrypted.end());
44. std::cout << "Decrypted message: " << decrypted\_message << std::endl;
46. } **catch** (std::exception &e) {
47. std::cerr << "Exception: " << e.what() << std::endl;
48. **return** 1;
49. }
51. **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- 5图3- 6所示。



图3- 5 rsa\_botan运行图1



图3- 6 rsa\_botan运行图2

### 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>
13. **void** handle\_errors(**const** std::string &msg) {
14. std::cerr << "Error: " << msg << std::endl;
15. exit(1);
16. }
18. **int** main() {
19. **try** {
20. // 初始化随机数生成器
21. Botan::AutoSeeded\_RNG rng;
23. // 生成DSA参数
24. Botan::DL\_Group group("dsa/botan/2048");
26. // 生成DSA密钥对
27. Botan::DSA\_PrivateKey private\_key(rng, group);
28. Botan::DSA\_PublicKey public\_key = private\_key;
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);
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;
38. // 要签名的消息
39. std::string message = "Hello, Botan DSA!";
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);
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);
49. std::cout << "Signature: " << Botan::hex\_encode(signature) << std::endl;
51. // 验证签名
52. Botan::PK\_Verifier verifier(public\_key, "EMSA1(SHA-256)");
53. **bool** valid = verifier.verify\_message(message\_hash, signature);
55. std::cout << "Signature is " << (valid ? "valid" : "invalid") << std::endl;
57. } **catch** (std::exception &e) {
58. handle\_errors(e.what());
59. }
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- 7图3- 8所示。



图3- 7 dsa\_botan运行图1

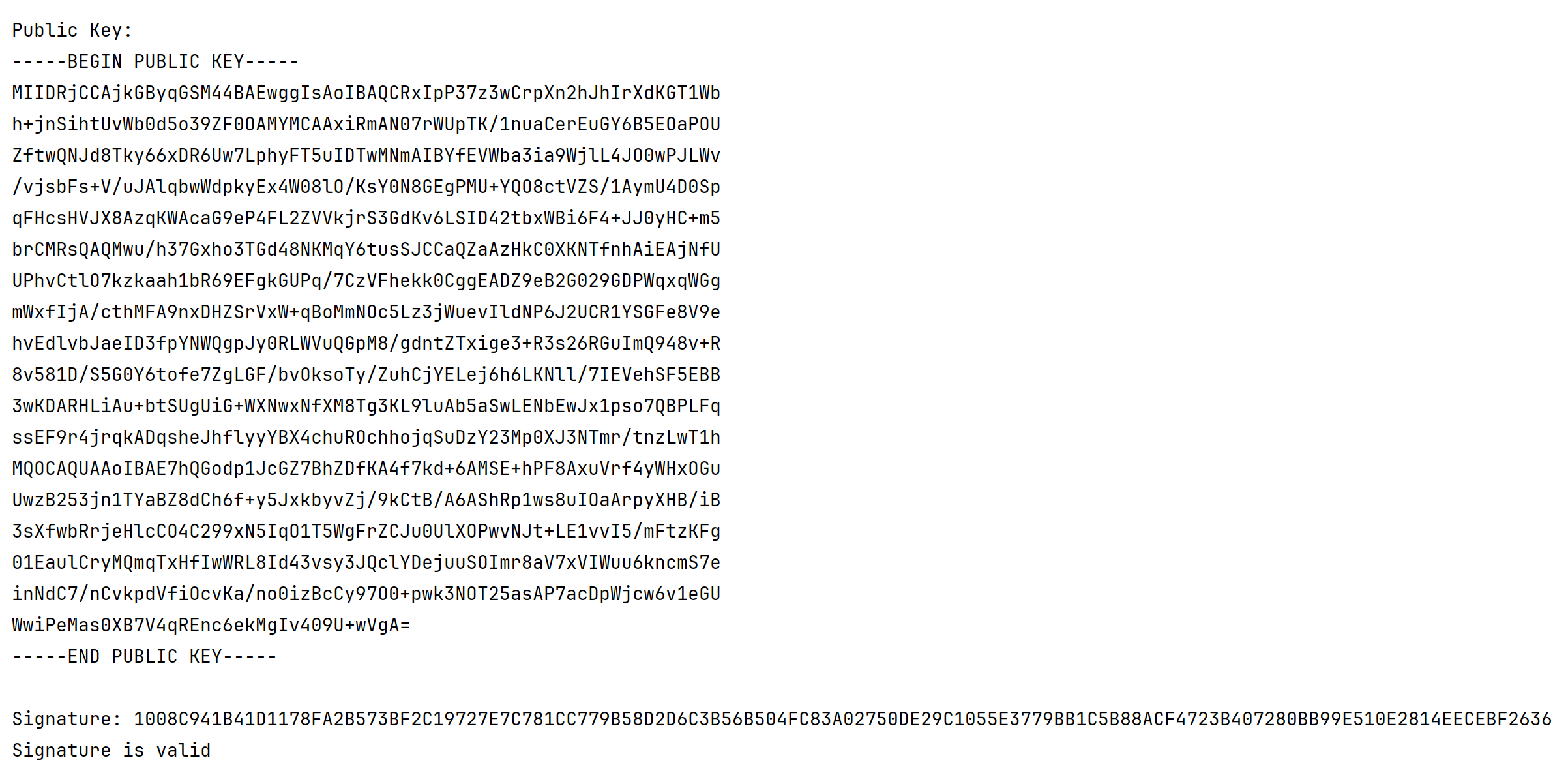


图3- 8 dsa\_botan运行图2

## Crypto++库与RSA、DSA实现

### 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。

### 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& publicKey)
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);
6. std::string cipherText;
8. StringSource(plainText, **true**,
9. **new** PK\_EncryptorFilter(rng, encryptor,
10. **new** StringSink(cipherText)
11. )
12. );
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\_EncryptorFilter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);
6. std::string recoveredText;
8. StringSource(cipherText, **true**,
9. **new** PK\_DecryptorFilter(rng, decryptor,
10. **new** StringSink(recoveredText)
11. )
12. );
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;
8. // 生成RSA密钥对
9. GenerateRSAKeyPair(privateKey, publicKey);
11. // 待加密的文本
12. std::string plainText = "Hello, world !";
14. std::cout << "plainText: " << plainText << std::endl;
16. // RSA加密
17. std::string cipherText = RSAEncrypt(publicKey, plainText);
18. std::cout << "Cipher Text: " << cipherText << std::endl;
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. }
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- 9所示。

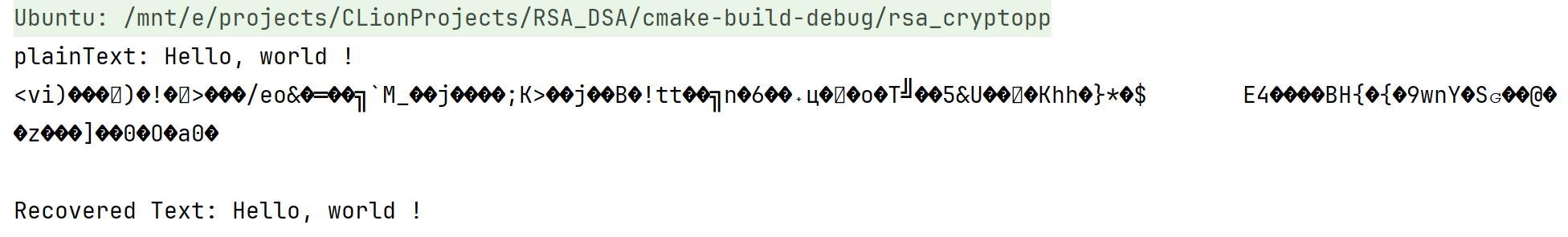


图3- 9 rsa\_crypto++运行图

### 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>
8. **using** **namespace** CryptoPP;
10. **void** GenerateDSAKeys(DSA::PrivateKey& privateKey, DSA::PublicKey& publicKey) {
11. AutoSeededRandomPool rng;
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::string& message) {
2. AutoSeededRandomPool rng;
3. DSA::Signer signer(privateKey);
5. std::string signature;
6. StringSource ss1(message, **true**,
7. **new** SignerFilter(rng, signer,
8. **new** StringSink(signature)
9. ) // SignerFilter
10. ); // StringSource
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 ss1(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& message, **const** std::string& signature) {
2. DSA::Verifier verifier(publicKey);
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
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;
5. // Generate DSA keys
6. GenerateDSAKeys(privateKey, publicKey);
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;
15. // Message to sign
16. std::string message = "This is a test message.";
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;
24. // Verify the message
25. **bool** result = VerifyMessage(publicKey, message, signature);
26. std::cout << "Signature is " << (result ? "valid" : "invalid") << std::endl;
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- 10所示。

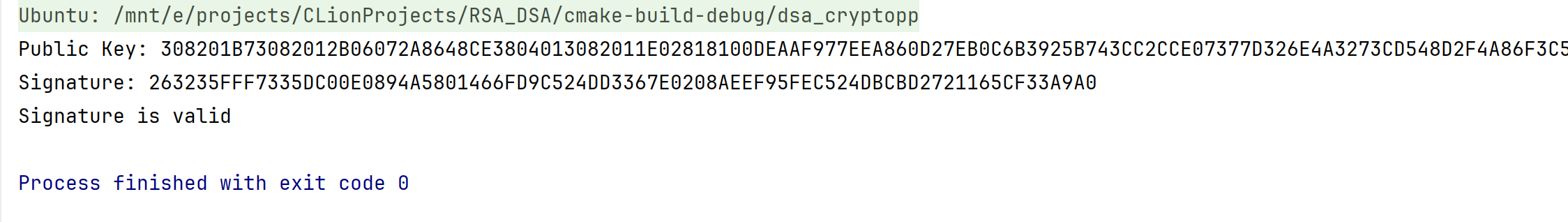


图3- 10 dsa\_crypto++运行图

## RSA、DSA自实现

### 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. }
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. }
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. }
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. }
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. }
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. }
62. **struct** RSAKeyPair {
63. **int** e, d, n;
64. };
66. RSAKeyPair generate\_rsa\_keys(**int** bit\_length) {
67. **int** min = 1 << (bit\_length / 2 - 1);
68. **int** max = 1 << (bit\_length / 2);
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);
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");
78. **int** d = mod\_inverse(e, phi);
79. **return** {e, d, n};
80. }
82. **int** encrypt(**int** message, **int** e, **int** n) {
83. **return** mod\_pow(message, e, n);
84. }
86. **int** decrypt(**int** ciphertext, **int** d, **int** n) {
87. **return** mod\_pow(ciphertext, d, n);
88. }
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);
94. std::cout << "Public Key (e, n): (" << keys.e << ", " << keys.n << ")\n";
95. std::cout << "Private Key (d, n): (" << keys.d << ", " << keys.n << ")\n";
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);
101. std::cout << "Original Message: " << message << "\n";
102. std::cout << "Encrypted Message: " << encrypted\_message << "\n";
103. std::cout << "Decrypted Message: " << decrypted\_message << "\n";
105. **return** 0;  }

代码在Ubuntu 20.04 LTS， gcc13编译运行的结果如图3- 11 自实现RSA算法运行图所示。

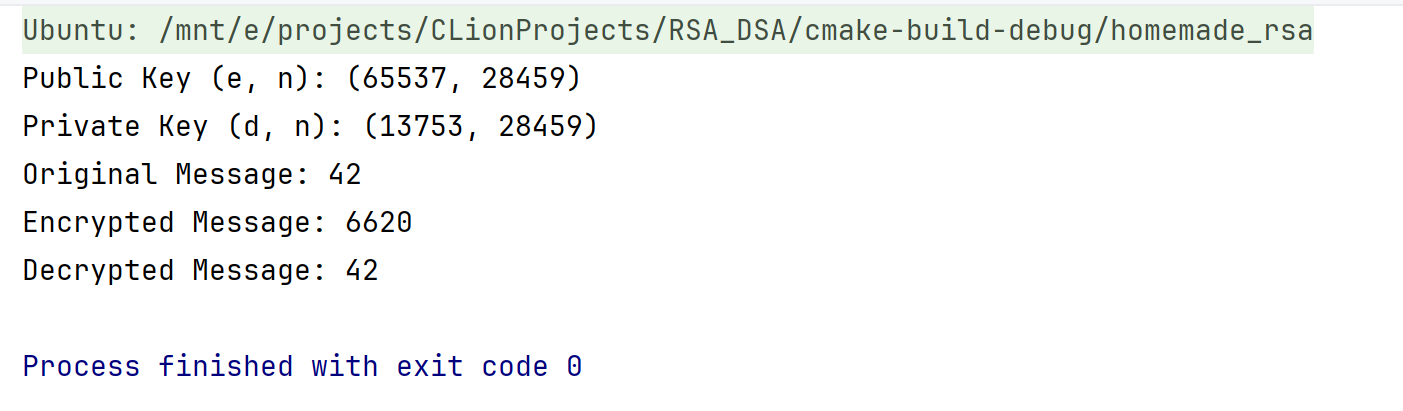


图3- 11 自实现RSA算法运行图

### 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：

DSA签名过程伪代码如下所示：

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
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. }
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. }
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。

# 量子计算对加密算法的影响

## 引言

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。

## 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。

## 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个，但是qubit由于可以处于叠加态，也就是他可以同时表示种信息，相当于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。

## 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。

### 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**：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 是奇数或者，重新选择 a并重复上述步骤。
     2. 否则，计算 和 。

### 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

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

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

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

49. **def** quantum\_order\_finder(x, n):
50. """
51. 量子模拟部分：计算使 x^r ≡ 1 (mod 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

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。
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

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}")

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。

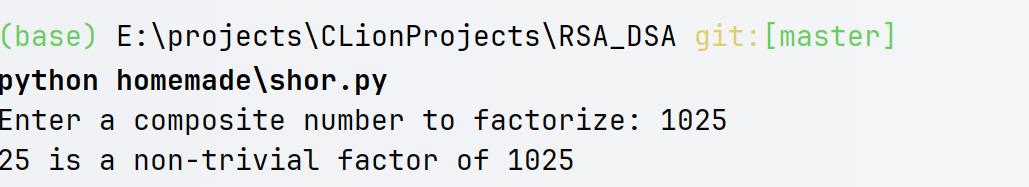


图4- 1 shor算法运行时图

## 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 降低到 。

### 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。

### 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：重复大约次。
5. 测量：测量量子态，结果以高概率为目标状态。

### 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 ≡ y (mod p), 其中g={g}, y={y}, p={p}")
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

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

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

28. # Grover迭代次数
29. num\_iterations = int(np.pi / 4 \* np.sqrt(2 \*\* n))
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)

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

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 （即“生日攻击”），其中是哈希输出的位数。Grover算法可以将这个复杂度降低到 。

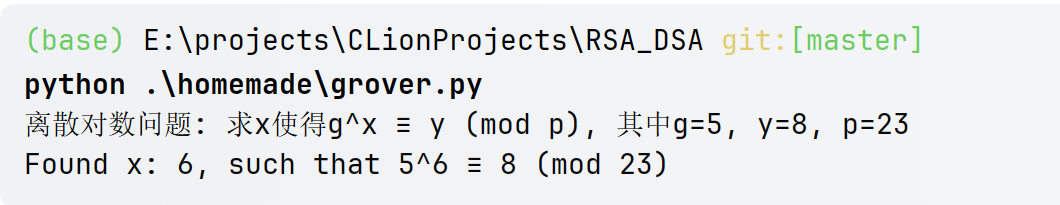


图4- 2 grover算法运行时图

# 总结与展望

## 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。

## 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可靠性和安全性。

## 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。

# 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。她是我最坚实的后盾，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。

# 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.
12. 王传俊.基于C语言的DES与RSA数据加密算法实现与分析[J].电子测试,2015,(03):40-41+35.DOI:10.16520/j.cnki.1000-8519.2015.03.015
13. 戚娜.基于C语言的RSA算法的实现[J].电子设计工程,2015,23(17):62-65.DOI:10.14022/j.cnki.dzsjgc.20151013.001.