

**本科生毕业设计（论文）大纲**

**学生姓名：刘思宇**

**导师姓名、职称：肖敏(副教授)**

**所属学院：计算机与人工智能学院**

**专业班级：计算机zy2001**

**设计（论文）题目：RSA加解密算法在FPGA虚拟仿真平台上的实现与应用**

2024年3月19日

摘要

最后写

关键词

英文摘要

中文摘要

关键词

英文

# 绪论.

## 1.1论文的选题背景

选题背景:

在当今数字化时代,信息安全已成为一个至关重要的议题.随着互联网技术的广泛应用,如何保障数据的安全性和完整性成了一个急需解决的问题.RSA算法,作为一种公钥加密技术,广泛应用于数据加密和数字签名等领域.其安全性基于大数分解的计算难题,能够有效抵抗各种密码攻击,是现代加密技术中的重要组成部分.

然而,RSA算法在软件中的实现往往受限于处理速度和安全性的双重挑战.与此同时,硬件加密技术,特别是采用现场可编程门阵列(FPGA)的加密实现,因其高效性和灵活性,成为研究的热点.FPGA提供了并行处理的能力和可重配置的优势,使得在硬件级别上实现RSA加密算法成为可能.此外,FPGA在安全性方面具有内在的优势,如防止侧通道攻击等,进一步增强了加密系统的整体安全性.

因此,研究RSA算法在FPGA上的实现不仅对推动加密技术的硬件化发展具有重要意义,也对提升数据加密系统的安全性和效率提供了新的视角和技术支持.本研究旨在探索RSA算法在FPGA上的高效实现方式,通过设计和优化加密模块,达到高安全性和高性能的双重目标.

## 1.2国内外研究现状

理论研究

RSA算法自提出以来,各种理论研究不断深入,以应对日益增长的安全需求和计算挑战.1992年,Jan-HendrikEvertse在<JCRYPTOL>发表的论文[1]指出,在特定条件下,可以从一种RSA签名推导出另一种签名,即便在签名者无法直接计算RSA根的假设下也是如此.此发现对理解RSA签名的潜在弱点提供了重要视角.随后,AmosFiat于1997年提出了批量RSA[2],这是一种通过批处理提升RSA性能的变体,其增强了RSA的效率和安全性.

到了1999年,DonCoppersmith探讨了在采用中国剩余定理算法的公钥密码系统中可能出现的安全风险[3],揭示了单个错误签名可能导致密钥泄露的问题.2000年,RosarioGennaro和TalRabin引入了一种鲁棒的门限RSA签名方法[4],该方法可以容忍恶意行为而不牺牲效率,并提出了基于RSA的不可否认签名方案.此外,DanielR.L.Brown于2016年和CarmitHazay于2017年分别就RSA算法的安全性和分布式RSA生成的安全性提出了重要的研究成果[5].

实际应用

RSA算法的实际应用也颇为广泛.早在1990年,J.R.Sherwood和V.A.Gallo就探讨了使用智能卡进行RSA数字签名的应用[6].华中科技大学[7]和大连理工大学[8]的研究团队分别在2008年和2013年,探索了基于FPGA的RSA加密芯片设计与实现,以及利用FPGA动态重构技术与RSA加密技术的结合,展示了RSA算法在硬件加密领域的应用潜力.

改进实现

在改进RSA实现方面,ThorstenKleinjung等人2010年使用数域筛法成功对768位数RSA-768进行因式分解[9],这对理解RSA算法面临的实际挑战具有指导意义.此外,西安交通大学的研究团队针对低速微型芯片设计了小面积的RSA硬件加密模块[10],展示了通过优化算法实现硬件友好的RSA加密操作的可能.

这些理论探索和实际应用的累积,构成了对RSA算法在FPGA上实现的深厚背景,为本研究提供了丰富的前沿科技和实践经验参考.

## 1.3主要研究内容

要实现RSA算法,其工程量和计算量是很大的,其实现方法也有很多,各种方法都有各自的优缺点及其独特的适用场景.在综合考虑实现目标,速度要求,FPGA硬件资源条件之后,对于RSA加密解密算法予以实现,主要研究工作总结如下:

1.学习已经发表的被广泛认可的相关研究论文进行学习,了解其中的一些基本原则.例如选取指数e一般取65537,对于常见的3,17,虽然计算速度更快,但是在没有适当的消息填充的情况下,攻击者可能利用低加密指数攻击,如果相同的数据被发送到多个接收者而没有足够的消息填充,攻击者可以使用中国剩余定理来恢复原始消息.

2.在广泛学习论文后,参考他们的设计模式,综合考虑本项目目标的速度资源要求,给出多个理论上可行的实现路径,然后对比分析,选择合适的一条作为最终的理论路线.例如在对模幂算法进行学习后,采用RL快速模幂算法将问题分解为模乘,然后使用蒙哥马利算法来提升模乘的速度,用节省进位乘法器进一步优化乘法计算速度,用高位相减法实现快速取模运算.

3.理论层面自顶向下,将问题划分为许多小模块,实现方面自底向上,用verilog逐一实现.在全程同时用其他语言完成相似功能,对比中间结果是否正确,模块功能是否正常.例如,为了测试蒙哥马利乘法器的输入输出是否正常,用Python完成相似功能后将结果进行比对.

4.对整体进行综合,然后仿真测试,完成黑盒白盒测试,编写使用和维护文档.

研究的具体内容包括:RSA算法及其变体的实现流程研究,RSA参数选取的研究,RSA算法安全性的研究,RSA应用场景的研究,RSA计算过程用数学理论限定范围以及加速,适合RSA实现的常见快速算法,1024位大整数四则运算和取模运算硬件加速,RSA的RTL设计与验证,仿真实现,RSA具体应用场景.

研究重点:适合RSA实现的快速算法,适合RSA实现的硬件优化,RSA算法的综合,仿真,验证,RSA具体应用

## 1.4论文内容安排

# 第2章基于FPGA实现RSA算法的基本理论分析与技术路线选型

## 2.1RSA算法的基本理论分析

RSA是一种广泛使用的公钥加密算法,其安全性依赖于大整数因数分解的难度.具体来说,RSA的安全性基于以下数学事实:给定两个大质数的乘积,想要将其重新分解为这两个质因数是非常困难的.这个问题被称为大整数因数分解问题,目前还没有发现在多项式时间内解决该问题的算法.

然而,随着计算机技术和数学方法的发展,对RSA的攻击能力也在不断提高.根据公开的研究结果,目前已经成功分解了768位的RSA密钥[11].这意味着,低于768位的RSA密钥已经不再安全.为了保证安全性,现在RSA密钥的长度通常选择1024位或更高.这也是本文选择1024位RSA进行实现的原因.

RSA算法的核心操作是模幂运算,具体过程如下:

1.随意选择两个大的指数p和q,p不等于q,计算.

2.计算欧拉函数.

3.选择一个小于r的整数e,使e与r互质,并求得e关于r的模反元素,命名为d(求d令).(模反元素存在,当且仅当e与r互质)

4.将p和q的记录销毁:公钥为(N,e),私钥为(N,d)

5.加密过程:

将明文M转换为整数C.计算密文.

6.解密过程:

使用私钥计算.将密文C转换回明文M.

可以看到,整个RSA计算中最核心的部分就是如何处理模幂,即.而模幂计算的核心是模乘,即,因此,模乘是整个RSA最为核心的部分

## 2.2RSA算法参数

由上述描述可知,RSA算法基于欧拉定理,其安全性依赖于大数分解的难度.选择两个不同的大素数p和q后,计算模数n=pq相对简单,而要通过公开的n分解出p和q,当p和q很大时,仍然是一个未解决的数学难题.如果能够成功分解n得到p和q,那么根据公式phi(N)=(p-1)(q-1),可以轻易得到phi(N)的值,从而轻松计算出密钥d.攻击者无法通过分解n得到p和q,也就无法计算欧拉函数phi(N)=(p-1)(q-1),从而不能通过公开的加密指数e(公钥)和公式d=e^(-1)modphi(N)计算出解密指数d(私钥),因此无法通过正常解密途径从截获的密文c中恢复出明文m.合法接收者因为知道解密指数d(私钥),能够根据公式M=C^dmodN正确解密.

因此,为了防止被破解,选择模数N和公钥e等参数非常重要.

为了保证安全性,根据当前技术水平,两个不同的大素数p和q至少应为512比特或约154位十进制数,这样模数n为1024比特或约308位十进制数.研究表明,模数n为1024比特的RSA算法的安全性大致相当于80比特密钥的分组密码算法.

密码学家建议,在使用RSA算法时,除了确保p和q足够大之外,还应注意以下几点:

1.素数p和q必须是非常大的强素数,且位数差不能太大也不能太小,一般差几个二进制位为佳,且p-1和q-1、p+1和q+1都应有一个大的素因子.p/q不应接近带有两个小分子或小分母的有理数.

2.相对于公开密钥e,私钥d的长度应更大.在安全性要求不高的场合,私钥d的值可以适当小一些,以缩短计算时间.

3.公钥e不能随意取值,选择过大会增加加密运算时间,但也不能太小,太小时容易受到低指数攻击.在通过一系列计算后,应使私钥d>n^(1/4).为满足这一要求,通常先选择私钥d,使n<d<phi(N)且gcd(d,phi(N))=1,然后再由私钥d计算公钥e:e=d^(-1)modphi(N).

## 2.3模乘算法

对于乘法abmodn来说,如果按照普通方法,先计算乘积然后再计算modn,从左端高位开始减n,这样的代价是:

1.估算不准确,可能要多做2次乘法,2次比较;

2.不能流水,需要先计算完高位再计算低位

3.对存储区的要求比较大,存储器读写次数多.

RSA加解密运算的加解密的过程简单地说实际上就是模幂运算的过程,幂运算可以转化为乘法运算,所以RSA密码算法的关键就变为模乘运算.为了保证信息的安全性,模数都取得较大,如512-2048bit正整数,且一般为两个素数的乘积,而非2的幂(若是2的幂则可用移位操作代替乘、除运算),直接进行大整数模幂运算C=MEmodN是不可能的,必须先将其分解为基本的模乘运算ABmodN,然后寻找ABmodN的高效算法

目前简化模乘运算主要有两种方式:(1)先乘法后模简化;(2)边乘法边模简化.

下文将对一些具有代表性的各种大数模乘算法进行简单分析,从基本的原理到实际应用,对各种模乘算法进行分类整理,并总结各种大数模乘算法的优缺点、性能及其应用环境等方面的信息.

① 加法性算法

Blakley于1983年根据上述计算式提出了一种加法型模乘简化算法[35],他的优化方式是把大数的模乘运算转换成一系列小的加法运算

② 估商型算法

1960年Stein和Pope提出了一种全新的模简化的思想,这种方法利用了估商技术.直到1981年这种思想才被切实可行的算法实现.Quisquater于1992年提出了一种新的估商型模乘算法,它能加快商的计算过程

1985年,蒙哥马利[38]提出了一种全新的算法,相对于前面的加法型算法和估商型算法,其针对大数模乘运算极大的提高了运算效率.蒙哥马利算法是目前在计算机软硬件中通用的模乘算法,也是目前被人们认为是最好的一种模乘算法.

作为目前最快的模乘算法,发展出了许多种基于蒙哥马利算法的实用型软硬件实现模乘算法.下文将简略的描述蒙哥马利算法的数学原理,及其针对原始蒙哥马利算法的优化.

蒙哥马利乘法是一种用于高效执行模乘运算的算法,能完成ABmodN的计算,特别适用于大整数运算.它广泛应用于RSA等公钥加密算法的加速计算中.蒙哥马利乘法的核心思想是通过将传统的模乘运算转换为一种更容易计算的形式,从而提高运算效率.例如,12345mod97不好心算,但是12345mod100好算,只需要取后两位45就行了,因为在十进制表示下,模10的幂比模非10的幂要好算得多,我们需要做的是引入100这个值,通过某些数学方法,将原本的mod97转化为mod100,从而加速对结果的计算.参考这种思想,蒙哥马利乘法利用了数学中模运算的一些性质,将输入A,B,N后,对A\*BmodN的计算转变为在蒙哥马利表现形式下其他变量的计算,并且能保持输出结果与A\*BmodN相等,一次蒙哥马利乘法可能因为引入了大量中间值而导致效果上相比于直接计算并没有显著提升,但是在连续的乘法取模运算中,蒙哥马利算法因为其本身支持并行计算的特征以及FPGA也支持并行计算的条件下能起到很好的作用.

蒙哥马利算法基本原理

下面的内容是对原始的蒙哥马利算法的介绍,对于模数N,选择适合大小的基数r,满足gcd(N,r)=1,为了方便硬件实现,这里取r=2^w(w=1,2,3..),模数N可表示为:

$$N=\sum\_{i=0}^{s-1}n\_{i}r^{i}$$

对于任意正整数A、B(0≤AB≤RN,R=𝑟𝑠),利用蒙哥马利算法可以求P=AB𝑅^−1𝑚𝑜𝑑𝑁,算法的具体形式如下:

输入:A,B<N;N是n位的奇数;

输出:R\_n=A\*B\*2^{-n}modN

FunctionMMM(A,B,N)

R\_0=0

For(i=0 to n-1)do

q\_i=(R\_{[0]}+a\_{i}\*B[0])modr

R\_{i+1}=(R\_i+a\_i\*B+q\_i\*N)/r

IfR\_n>=N,R\_n=R\_n-N

RETURNR\_n

以上就是原始的蒙哥马利算法,但是算法本身并不能直接得出两个大数模乘运算的结果,蒙哥马利算法是计算两个数模乘运算的一个核心过程之一.除此之外还需要进行预运算(即对两个大数a、b进行预处理)和后续调整(对MMM算法的运算结果进行后处理),最终得到想要的结果.

综上所述,采用蒙哥马利算法作为两数模乘运算核心算法的计算过程总共可以分成三个部分:

(1)预处理,使用下面两个公式,将两个1024位大数a和b映射到蒙哥马利域:

A=Mont(a,(R^2modN),N)=a\*(R^2modN)\*R-1modN=a\*RmodN

B=Mont(b,(R^2modN),N)=b\*(R^2modN)\*R-1modN=b\*RmodN

(2)模乘运算过程的核心计算步骤:

P=Mont(A,B,N)=A\*B\*R^-1modN=a\*R\*b\*R\*R^-1modN=a\*b\*R^-1modN

(3)对MMM算法的计算结果进行后续调整,调整后为最终结果:

P\_n+1=Mont(P\_n,1,N)=P\_n\*1\*R^-1modN=a\*b\*R\*R^-1modN=a\*b\*modN

由上述算法的计算过程可以看出,采用蒙哥马利算法的模乘运算总共分成三个部分,这三个部分的计算都要涉及到式MMM(A,BN)的运用.若仅仅进行一次模乘运算,则需要分三次进行蒙哥马利算法,其中对数据处理的预运算和对计算结果的后续处理就分别占据了两次,这两次运行消耗了实际所需模乘计算的时间,可见蒙哥马利模乘算法并不适用于低次数的模乘运算[40].当需要计算的模乘的次数越多,其中数据处理的预运算和对计算结果的后续处理所占的比例就越小,这样蒙哥马利模乘算法就能体现出高性能的优势,大大提供计算的速度.随着加密位数的逐渐提高,所需要进行的模乘运算的次数也越来越多,因此蒙哥马利算法的模乘运算的使用就更加广泛但是原始的蒙哥马利算法也有缺点:(1)运算得到的结果有个多余的2^-n;(2)最终得到的结果位于两个整数0到2N之间,每次需要调节大小,即:比较与N的大小关系大于N,那么减去N的值就是最终结果,否则不用减.

## 2.5模乘算法的优化

高效的进位保存加法器(Carry-SaveAdder,CSA)通过减少进位传播延迟显著改善了蒙哥马利算法的性能.传统加法器在每次加法操作后都需要处理进位,这会带来较大的延迟.CSA通过分离进位和和数(sum)来解决这一问题,使得多次加法操作可以并行进行,进而提高计算速度.

进位保存加法器(CSA)的原理

CSA的基本原理是将多个二进制数相加时,进位和和数分开存储,在多个加法步骤之后再统一处理进位.CSA通常由一个全加器(FullAdder)阵列构成,每个全加器同时处理三位输入:两个加数位和一个进位位.

CSA加法步骤:

1.部分和(PartialSum)和部分进位(PartialCarry):

对于每一位的加法操作,计算部分和𝑆S和部分进位𝐶C.

例如,对于三位输入𝑎a、𝑏b、𝑐c,全加器计算得到𝑆=𝑎⊕𝑏⊕𝑐S=a⊕b⊕c,𝐶=(𝑎∧𝑏)∨(𝑏∧𝑐)∨(𝑐∧𝑎)C=(a∧b)∨(b∧c)∨(c∧a).

2.延迟进位处理:

多次加法操作后,将所有部分和和部分进位汇总,最终处理所有进位.

CSA在蒙哥马利算法中的应用

在蒙哥马利乘法中,需要大量的加法操作来完成模乘运算.使用CSA可以显著提高这些加法操作的效率.

蒙哥马利乘法步骤中的优化

1.初始化:

选择参数R和N.

计算R−1modN和N′=−N−1modR.

2.计算中间值:

设T=A⋅B.

使用CSA计算m=(TmodR)⋅N′modR.

3.CSA处理多个加法操作:

计算t=(T+m⋅N)/R时,使用CSA进行部分和和部分进位的处理.

将T和m⋅N的每一位加法操作分解为部分和和部分进位计算,避免每次加法操作后的进位传播.

4.最终结果:

处理所有部分和和部分进位,得到最终结果.

如果t≥N,则t=t−N.

## 2.6模幂算法

快速幂算法

如何计算a^n,对于朴素算法，时间复杂度为O(n),而快速幂算法，则时间复杂度为O(logn),其思想大致如下:

特殊情况：n为2的幂,根据倍增原理，与其一个一个乘以a,不如每次将a的数量翻倍,例如a^64=a^32\*a^32=a^16\* a^16\* a^16\* a^16\*……

当n不为2的幂的时候，对n进行二进制展开,例如计算a^105,由105=1+8+32+64可得a^105=a\*a^8\*a^32\*a^64



伪代码

function BINEXP(a, n)

r=1

while n≠0

if n mod 2 ==1

r=r\*a

a=a\*a

n = n/2

return r



而且算法中

Nmod2可以替换为n&1

N=n/2可以替换为n>>1

适合硬件实现

应用:计算a^nmodN

朴素算法

function MODEXPNAIVE(a, n, m)

r=1

for i=l to n

r =(r \* a) mod m

return r

备注:(a\*b)modn=((amodn)\*(bmodn))modn

快速幂取模

function MODEXPFAST(a,n, m)

a =a mod m

r=1

while n ≠0

ifn&1-=1

r=(r \* a) mod m

a=(a \* a) mod m

n=n>>1

return r

在快速幂取模算法的加持下:计算A^BmodN可以分拆成(A·B)modN和(A^2)modN的计算,最终利用模乘的方式来实现模幂的运算.这里根据对E指数扫描方向的不同,可分为R-L和L-R两种方式.下面对这两种方式进行详细的分析.

(1)R-L模式算法:是指幂指数的二进制位从右到左即从低位(Right)到高位(Left)进行扫描.模幂运算的基本形式为C=M-modN.在R-L运算模式算法中,两次模乘运算M=M\*M(modN)与C=M\*C(modN)在当前轮运算中没有数据的顺序关系,也就是说平方与乘法操作是相互独立的,可以采用两个模乘器,在一个时钟周期内并行实现平方与乘法操作.R-L模式中可以采用并行运算,以达到提高处理速度,节约运算时间的目的,但需要两个模乘器,实现时增加了硬件资源的占用.

算法2.1:R-L算法ME1(M,E,N)

输入:N(模),E(幂),M(明文)

输出:C(密文)

伪代码:

初始化变量E和C,其中E是幂的二进制位数之和,C初始为1.

循环从0到n−1(幂的二进制位数).

如果当前位ei为1,则更新C=M∗CmodN.

更新M=M∗MmodN.

返回结果C.

(2)L-R 算法:指幂指数的二进制位从右到左即从高位(Left)到低位(Right)进行查询.模幂运算的基本形式为C=MmodN,.在L-R 算法模式中,硬件实现只需要一个模乘器串行执行乘法操作.在循环体内的每轮运算中进行两次模乘运算,且两次模乘运算的数据相关,必须顺序进行.

算法2.2:L-R算法ME2(M,E,N)

输入:N(模),E(幂),M(明文)

输出:C(密文)

伪代码:

初始化变量 E 和 C,其中 E 是幂的二进制位数之和,C 初始为 1.

从最高位(n−1)到最低位(0)进行循环.

更新 C=C∗CmodN.

如果当前位 ei 为 1,则更新 C=M∗CmodN.

返回结果 C.

两种模幂方案比较

R-L 算法和 L-R 算法均是通过对指数的位查询方式来完成相应的幂乘运算,这种按位查询的方式十分方便硬件实现.它们的模乘运算次数是相同,也就是说它们的运算量是一样的,均需要n+n次模乘运算,但是它们各有利弊,所需要的运算时间不同.R-L算法需要三个寄存器,而L-R算法只需要两个寄存器;R-L算法可以采用两个模乘器进行并行运算,因此就大大缩短了操作时间,算法所需的运算周期与指数 E 二进制位中1的个数无关,需要n个运算周期.而L-R算法中的两个模乘运算必须顺序执行,每次迭代所需的时间根据幂指数当前位e决定.当e=0时,只进行一次模平方运算,当e=1时,除一次模平方运算外,额外还要进行次模乘运算,即e=1时运行的时间是e,-0时的两倍.

综上所述,本文拟采用RL快速幂取模算法结合采用了CSA加速的蒙哥马利算法来实现模幂计算和模乘计算.

# 第3需求分析与设计

## 3.1总体设计

## 3.2详细设计

# 第四章实现

# 第五章测试

# 第六章维护

参考文献

致谢

术语表

图表目录

部分代码