Is RSA Really Secure?
Using Repunits We Prove Otherwise

Mehran Davoudi Ayaz Isazadeh
Department of Computer Science
Tabriz University, Tabriz, IRAN
mehran.davoudi@yahoo.com
isazadeh@tabrizu.ac.ir

Abstract

Prime numbers plays a critical role in encryption algorithms. The security of RSA relies on an assumption: We hope there is no algorithm capable of factorize a big number in polynomial time. A RSA public key constitute of a few prime factors for more security. We use this fact as a constraint to simplify the problem of factorizing RSA public key as follows: "In a RSA public key ***some factors*** *is near to* ***all factors****.*

In this paper we find ***Repunit numbers***[1] *as an interesting set of numbers. We show that the set of all repunits contains all of prime numbers. We provide some theorems to show the relation between repunits and prime numbers. As an advantage of using abbreviated form of repunits we notice their low space complexity. Then we introduce our method to factorize big numbers based on repunits and their abbreviated form.*

1. Introduction

Primes, primes, primes. Why are they so important for us? Why factorizing numbers to primes are so interesting. We live in a digital world and we need to feel secure. I want to emphasize the fact that we owe this security to some features of prime numbers which even has not proofed yet! We are feeling secure while there is no proof to deny existence of polynomial algorithm to factorize a number mathematically.

1.1 Problem

In RSA we trust that there is no fast algorithm to factorize a big number to prime numbers in a polynomial time. Factorization algorithms are designed to factorize numbers into primes. But in RSA, we are satisfied even if we have some factors of the big number [2]. In fact in RSA public keys **some factors** is near to **all factors**, because factors are so large and the number of them is really few(We can consider it 2). Providing this idea we declare the problem as:

> *How can we factorize a big number to some factors?*

The idea provided by this paper claim a new method (***Repunit Method***) for factorization algorithms. We show the relation between repunits and prime numbers. Having introduced this relation we propose some methods to factorize a big number which usually used as RSA public keys.

There are some few papers published in the domain of repunit numbers. Most of discussion around the repunits focuses on *Prime Repunits*. In fact the race of finding bigger prime repunit attracts the most energy of repunit workers! [1][5][6]

Chris K. Caldwell and Harvey Dubner [1] used repunits to find *Unique period primes*. S. Yates [5, 6] also have done lots of jobs for Unique period primes.

A. Slinko [3] used repunits to find *Absolute primes*. Also he presented some useful properties for repunits. He used repunits to find absolute primes and also he presented some theorems which are useful to test primeness of a repunit.

Unfortunately there is not any paper to show the strong relation between repunits and primes. This is the topic which we are going to invest in this paper.

1.2 Terminology

In this section we introduce elementary definitions and abbreviations which we use in the paper.

[1] Numbers like $1, 11, \ldots$ or formally $R_n = (10^n - 1)/9 \; for \; n \geq 1$

[2] A big number is considered as a big composite number with few big factors.

- **Repunit number.** In recreational mathematics, a repunit is a number like 11, 111, or 1111 that contains only the digit 1. The term stands for repeated unit and was coined in 1966 by A.H. Beiler. The repunits are defined mathematically as:

$$R(n) = \frac{10^n - 1}{9} \quad for\ n \geq 1$$

- **Big number.** A big number is considered as a big composite number with few big factors. In this paper we assume we have big numbers with no factor of 2, 3 and 5. In fact finding this factors is easy so we can ignore them easily.

2 Previous Works

This chapter contains the basic algorithms and theorems which the idea of this paper is based on. First of all we explore some of the most important available *prime factorization* algorithms. Then we propose a few familiar theorems with their proofs so we can cite them in the next sections.

2.1 Simple Factorization Algorithm

The simplest algorithm to factorize a number is a *brute force* algorithm. Algorithm 2.1 checks if the number is divisible to any of the numbers lower than it.

```
SimpleFactorizing(n:integer):Array
    AnswerList = ∅
    i=2
    Do
        If i mod n = 0 Then
            AnswerList = AnswerList ∪ i and
        End If
        i = i+1 // Get Next factor to test.
    Loop While( i < n )
    Return AnswerList
```

Algorithm 2.1 *The famous algorithm from Euclid to factorize numbers is very simple and construct the base of our methods.*

Proposition 2.1 *Consider about optimizations we can do on the algorithm 2.1 to improve its running time. First we can decrease the upper bound of probable divisors to $\sqrt{n}$ instead of n. But a better optimization is to select more appropriate divisors. There is no need to test all numbers lower than n to find its factors. If we choose just the primes lower than n(or even $\sqrt{n}$) we satisfy the correctness of algorithm and nothing is lost yet. So we should have a list of primes ready, but we are in middle of finding primes in this algorithm how could we have the prime list? This is the problem which forces us to test all divisors lower than n.*

The Repunit Method tries to use an *implicit list of primes* in middle of finding primes.

2.2 Finding GCD Algorithms

Calculating GCD of two numbers is so important for us. We use it as a primary operation in next sections. So we need to know its computing complexity and also have an appropriate algorithm to do it.

Theroem 2.2 *The complexity of finding the $GCD(n, m)$ for $n \leq m$ is $O(\log n)$.*

Here comes the famous *Euclid's algorithm* to calculate $GCD(m, n)$.

```
EuclidGCD(m, n)
    If n = 0 Return m
    Else Return EuclidGCD(m, m mod n)
```

Algorithm 2.2 *Euclid's algorithm for calculating GCD.*

Theroem 2.3 *Complexity of calculating $GCD(n, m)$ (using Euclid's algorithm) for $n \leq m$ is $O(\log n)$.*

$$O(GCD(m, n)) = O(\log n)$$

Proof. It can be proved in various ways. All proofs are really straight and interesting. Also there is a good proof at [4] using Fibonacci numbers. The *Binary GCD algorithm* described by Knuth [2] as a practically fast algorithm:

> "The binary GCD algorithm is an algorithm which computes the greatest common divisor of two nonnegative integers. It gains a measure of efficiency over the ancient Euclidean algorithm by replacing divisions and multiplications with shifts, which are cheaper when operating on the binary representation used by modern computers. This is particularly critical on embedded platforms that have no direct processor support for division..."

2.3 Period of a Number

The reciprocal of every prime p (other than two and five) has a period, that is the decimal expansion of 1/p repeats in blocks of some set length [1]. This period is *period of p*. for example:

$$\frac{1}{11} = 0.\overline{09} \qquad \frac{1}{7} = 0.\overline{142857} \qquad \frac{1}{13} = 0.\overline{076923}$$

The first and basic algorithm to find period of a number is to do the usual divide method just like high school, after finding a repetitive remainder the quotient is the period of divisor. But how long should we stay in the loop. Or even, is endless loop possible? In theorem 3.4 we prove that any number have its own period.

3 The Repunit Method

In this section we introduce our factorization method based on repunit numbers, We then, present some related algorithms to facilitate to work with big repunit numbers.

3.1 Usages of Repunits

This section starts proposing theorem 3.1 which shows the relation between repunits and prime numbers. Then we define a new definition called *Admissible Repunit* to a number. Finally we introduce *Repunit Method* while necessary definitions are declared.

3.1.1 Repunits and Primes Relationship

Theroem 3.1 *For a given number p having prime factors except 2,3 and 5 there is at least one repunit number R_n which $p \mid R_n$.*

Proof. Let's declare x as follows:

$$x = \frac{1}{p}$$

As $x \in Q$ so we can write x in decimal form as follows:

$$x = 0.b_1b_2...b_n\overline{a_1a_2...a_m}$$

Theorem 3.3 shows that m exists and is not infinity. Let's do the simple high school method to find p from its decimal form x, it is as simple as follows:

$$10^{n+m}x - x = b_1 \dots b_n a_1 \dots a_m.\overline{a_1 \dots a_m}$$

$$x = \frac{b_1 \dots b_n a_1 \dots a_m}{10^{n+m} - 1}$$

We know that $x = \frac{1}{p}$, so $p = \frac{1}{x}$ and p is a natural number.

$$p = \frac{10^{n+m} - 1}{b_1 \dots b_n a_1 \dots a_m}$$

So

$$10^{n+m} - 1 = (b_1 \dots b_n a_1 \dots a_m)p$$

$$9R_{n+m} = (b_1 \dots b_n a_1 \dots a_m)p$$

Due to assumption p has not 3 as its divisors, so it can not count 9. Therefore period of p count 9. In another words $\frac{b_1 \dots b_n a_1 \dots a_m}{9}$ is a natural number, let's name it k and then we have:

$$R_{n+m} = kp$$

So we constructed a repunit R_n that $p \mid R_n$. We use the notation **RelatedRepunit(n).** as a function that returns the related repunit for a given number n. Algorithm 3.2 shows how to compute $RelatedRepunit(n)$.

Corollary 3.2 *Lets probe theorem 3.1 contrary. If there is a repunit for each prime number, so the **set of all repunits** contains all prime numbers. As a result of theorem 3.1 the following set(***Repuniset***) covers all prime numbers in the world.*

$$Repuinset = \{11, 111, 1111, \dots\}$$

and for each prime p we have this:

$$p \mid \prod_{i=1}^{\infty} R_i$$

3.1.2 Related Repunit Algorithms

In this section we probe the algorithms to find Related Repunit of a number. First in algorithm 3.1 we notice how to find the number of digits of related repunit. Having *RelatedRepunitIndex(n)* function makes it easy to construct the related repunit algorithm. Algorithm 3.2 shows its simple steps. It is obvious that bottlenecks of Algorithm 3.1 are these lines:

$$If(Remainder \in RemaindersList)$$

$$RemaindersList = RemaindersList \cup \{Remainder\}$$

The complexity of checking whether $Remainder$ is already a member of $RemainderList$ depends on being the list sorted or unsorted. Considering it sorted it takes $O(\log n)$ using binary search, otherwise it is $O(n)$.

The fact of $RemaindersList$ being sorted or unsorted depends on the second line, the method we add $Remainder$ to $RemaindersList$ in implementation. If we want it to keep sorted we can use *Insertion Sort* method at each step. So at each step we do insertion sort with complexity of $O(n)$ in worst case (we hope it to be much better in real). But in simple adding the Complexity is $O(1)$.

```
RelatedRepunitIndex(n:integer):integer
    RemaindersList = ∅
    Dividend = 10
    Divisor = n
    RemaindersList = RemaindersList ∪ {Devidend}
    QuotientLength = 1
    While(true)
        Remainder = Dividend/Divisor
        If ( Remainder ∈ RemaindersList ) Then
            Break
        Else
            RemaindersList = RemaindersList ∪ {Remainder}
        End If
        QuotientLength = QuotientLength + 1
    Return Quotient
```

Algorithm 3.1 *This algorithm returns the number of digits of* $RelatedRepunit(n)$.

```
RelatedRepunit(n:integer):integer
    Index = RelatedRepunitIndex(n)
    Return (10^Index − 1)/9
```

Algorithm 3.2 *This algorithm returns* $RelatedRepunit(n)$.

So using sorted or unsorted list force us to have at least $O(n)$ complexity at each division step. Theorem 3.3 shows that in the worst case we need to repeat division n times. So final complexity of algorithm using former methods is $O(n^2)$

Theroem 3.3 *For a given number* n *there exist a repunit* R_k *which* $k \leq n$.

Proof. In algorithm 3.1 we search for repetitive remainders. On the other hand we know that if $r = m \bmod n$ then r can be one of the numbers of following set:

$$PossibleRemainders = \{0, 1, \ldots, n-1\}$$

The cardinality of $PossibleRemainders$ set is equal to n. This shows that at the worst case we will find a repetitive remainder at least after n divisions.

Corollary 3.4 *The related repunit index equals the length of* period of a number *introduced before. In fact theorem 3.3 proofs that period of a number's length is limited.*

Having theorem 3.3 limited the floor of remainders count, we can use a faster algorithm. The faster way is to keep a bit map for remainders have seen so far. In this way the computational complexity for each step is $O(1)$ and a total computation complexity of $O(n)$. Reaching complexity of $O(n)$ makes us to have space complexity too. In fact we need to store an array of n bits to check if a remainders is seen before. So the space complexity for Algorithm 3.1 is $O(n)$.

3.1.3 Repunit Method

Before introducing Repunit Method we define ***Admissible Repunit***. For a composite number $n = n_1 n_2 \ldots n_k$ admissible repunit of n is a repunit R_i which R_i is related repunit of n_t for some $1 \leq t \leq k$. Theorem 3.5 shows an important property of admissible repunits.

Theroem 3.5 *Suppose* R_i *as an admissible repunit of n.* $GCD(R_i, n)$ *is a factor of* n.

Proof. Consider $n = n_1 n_2 \ldots n_k$, and due to definition of admissible repunits, there exits a number t which $n_t \mid R_i$. So $GCD(R_i, n)$ is a factor of n_t. Let's take a look back at the set introduced at Corollary 3.2. Repuniset have some great properties we which categorize them as follows:

- **This set contains all prime numbers within it. There is no need to generate them first.** As we mention at proposition 2.1 it would be better to have an *implicit list of primes* to improve the factorization algorithm. Here it is! Instead of creating a list of primes which is limited and time consuming, we have a set contains all primes and no need to initiate or generate it.

  Algorithm 3.3 shows a method to factorize a number using benefits of reupints. In this algorithm we use GCD instead of division. In fact GCD is a great function. When $GCD(n, R_i) = 1$ for some i, we understand that number n has not any factor of R_i. In this way we use on GCD instead of doing lots of divisions(for big numbers of course). Algorithm 3.3 uses the concept of **admissible repunits** implicitly. In fact it looks up for the lowest admissible repunit of n. But we can ignore one limitation of this algorithm to achieve more performance.

  Why looking for lowest admissible repunit?

  Corollary 3.6 *One important feature of this algorithm is that we can surpass our algorithm starting from* k *instead of* 2*, and having nothing lost yet. This is one benefit of using* Repuniset. *For example we know RSA key designers do not use little primes as divisors, so why we should start checking from 2. we can start checking*

```
FactorizeUsingRepunits(n:integer):integer
    For i=2 to n
        gcd = GCD(R_i, n)
        If gcd ≠ 1
            return gcd
```

Algorithm 3.3 *Factorizing numbers using repuniset as an implicit list of primes. This algorithm returns when the first admissible repunit of numbers found.*

from R_i which i can be selected appropriately to the problem situations.

Now we want concentrate on an specific kind of numbers, *big numbers* which used in ciphering algorithms like RSA. There is an important fact about these numbers:

> **These numbers contains two big primes for more security.**

The reason is that the security of cipher depends on **factorizing difficulty** which is bound to *lowest* prime used in the number. So the cipher developers try to use big primes with nearly same size. This lead us that prime numbers are near to:

$$\sqrt[n]{BigNumber}$$

n is the number of constructor primes of big number. In this example we used $n = 2$. We present algorithm 3.4 based on this idea.

```
RelatedRepunitIndexRSA(n:integer, modulo:integer):integer
    EstimatedPrime = √n
    For i = EstimatedPrime To n
        If ( GCD(R_i, n) ≠ 1 )
            return i
```

Algorithm 3.4 *This algorithm works better when RSA assumptions are considered about n. In this case we can restrict boundaries to look up for admissible repunit of n*

- **Storing repuniset consumes considerably low memory.** Consider you want to store all prime factors of a repunit. There is no need to save all repunit number. To store R_n you should just store its length n. For example instead of storing:

  11111: 41, 271.
  111111: 3, 7, 11, 13, 37.
  1111111: 239, 4649.

  we can use their abbreviated form:

  5: 41, 271.
  6: 3, 7, 11, 13, 37.
  7: 239, 4649.

  In fact space complexity of storing R_n is $O(n)$ while it was $O(10^n)$ before.

3.2 Special Algorithms for Repunits

We see the usage of abbreviated forms to maximize memory performance. But it is useless in real calculation unless we have appropriate algorithms. Consider GCD calculation presented at algorithm 2.2. If one parameter is a big abbreviated repunit, we should construct it first, then start to divide. This sounds really bad for big repunits. Here we present some useful algorithms for repunits which use the abbreviated form for calculations without need of constructing them.

Remember the algorithm 2.2 for calculating GCD. It was good and fast. But consider finding GCD of a R_n and a number n. If we use Euclid's algorithm we can not use abbreviated form of repunits. It is obvious that the only problem is the first division and finding first remainder. After the first division and finding first remainder the algorithm can be done in usual manner. But for the first division we need to calculate R_n and use it to find the remainder which can be impossible due to memory limitations. Algorithm 3.5 calculates the remainder using a *Divide&Conquer* method.

Theroem 3.7 *The complexity of algorithm 3.5 is $O(\log n)$.*

Proof. Let's compute R_n modulo m considering two cases, first assume $(n = 2k)$:

$$R_n = R_{\frac{n}{2}} \times 10^{\frac{n}{2}} + R_{\frac{n}{2}}$$

$$= R_{\frac{n}{2}} \times (10^{\frac{n}{2}} + 1)$$

$$R_n \bmod m = (R_{\frac{n}{2}} \bmod m) \times ((10^{\frac{n}{2}} + 1) \bmod m)$$

$$R_n \bmod m = (R_{\frac{n}{2}} \bmod m) \times (((10^{\frac{n}{2}} \bmod m) + 1) \bmod m)$$

This shows what happened at first part of *RepunitRemainder* function. Now consider if $n = 2k + 1$: In this case we

```
RepunitRemainder(n:integer, modulo:integer):integer
    If n mod 2 = 0 Then
        remExp = ExpRemainder(n/2, modulo)
        remRep = RepunitRemainder(n/2, modulo)
        Return ((remExp + 1) × remRep) mod modulo
    Else
        remRep = RepunitRemainder((n−1)/10, modulo)
        Return (remRep × 10 + 1) mod modulo

{A utility function}
ExpRemainder(n:integer, modulo:integer):integer
    If n mod 2 = 0 Then
        rem = ExpRemainder(n/2, modulo)
        Return (rem^2) mod modulo
    Else
        rem = ExpRemainder(n − 1, modulo)
        Return (rem × 10) mod modulo
    End If
```

Algorithm 3.5 *A useful algorithm to do the first step of Euclid's algorithm.*

modify problem to use former case in a recursion manner. We know that R_{n-1} satisfies the former case conditions.

$$R_n = (10 \times R_{n-1})/10$$

$$R_n \bmod m = 10 \times (R_{n-1} \bmod m) + 1$$

In this way we show what happens at the latter case at *RepunitRemainder* function.

The proof for *ExpRemainder* is simply like this.

The complexity of this algorithm is $O(\log n)$ because it is a *Divide and Conqueror* method which divides each problem into 2 problems with size of $\frac{n}{2}$.

Proposition 3.8 *We can apply* Memoization *[4] on algorithm 3.5 to construct an algorithm based on* Dynamic Programming *which could run faster on the situation.*

4 Conclusion

We complete the paper arguing about the achievements we reached. Then comparison of our work with the others help you to sense its progress. As an ending we suggest some topics for future research.

4.1 Achievements

In this paper we argue about repunits and their important relation with prime numbers. This relationship holds at theorem 3.1. After introducing this relevancy we declared a set called *Repuniset* which was very useful. We validated the fact that this set consists of all primes, so we used it as an *implicit prime list* to factorize numbers. Also we noted two important property of this set:

1. We can store it on computer using its abbreviated form to decrease space complexity.
2. There is no need to initialize the set, it is already filled.

We then, continued by introducing **Related Repunit** and **Admissible Repunit**. As a matter of fact we found Admissible Repunit definition very capable to use in factorization algorithms. The Repunit Method used admissible repunits to find factors of big numbers.

However we presented an algorithm to factorize numbers in a way that each step (each GCD) we throw lots of possible prime factors out. This gives us the power which we can start our algorithm with a big offset having nothing lost. In Repunit method does not work efficient on small numbers. In fact it is developed specially for big numberswhich are equivalent to RSA public keys.

4.2 Future Topics of Research

As an ending of our paper we want to make it endless! Here's some topics which we consider you can take it and work on it after reading this paper. We will be happy if you let us know if you do so.

- Arguing about algorithms for factorizing repunits.
- Implementation of algorithm using powerful hardware and optimized codes.
- Making guesses about $RelatedRepunit(n)$ and also *Admissible Repunit* boundaries can improve performance of factorization algorithm as we do a little at algorithm 3.4.

References

[1] H. D. Chris K. Caldwell. Unique-period primes. *Recreational Maths*, 1(29):43–48, 1998.

[2] D. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 3rd edition.

[3] A. Slinko. Absolute primes.

[4] R. L. R. C. S. Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms*. McGraw Hill, 2nd edition, 2001.

[5] S. Yates. Periods of unique primes. 53(5):314, 1980.

[6] S. Yates. *Repunits and Repetends*. Star Publishing Co., Inc., Boynton Beach, Florida, 1982.