

Implementation of Fast RSA Key Generation on Smart Cards

Chenghuai Lu
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
lulu@cc.gatech.edu

Andre L. M. dos Santos
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
andre@cc.gatech.edu

Francisco R. Pimentel
Departamento de Matematica
Universidade Federal do Ceara
Fortaleza, CE 60455, Brazil
pimentel@mat.ufc.br

ABSTRACT

Although smart cards are becoming used in an increasing number of applications, there is small literature of the implementation issues for smart cards. This paper describes the issues and considerations that need to be taken into account when implementing the key generation step of a cryptographic algorithm widely used nowadays, RSA.

Smart cards are used in many applications that require a tamper resistant area. Therefore, smart cards that use cryptography have to provide encryption, decryption, as well as key generation inside its security perimeter. RSA key generation is a concern for on-card implementation of RSA cryptosystem, as it usually takes a long time. In this paper, two simple but efficient key generation algorithms are evaluated, in addition to a simple but not very efficient algorithm. The paper discusses in detail how to build fast implementations for the three algorithms presented, using smart cards with crypto-coprocessor.

Keywords

Smart card, Coprocessor, RSA key generation, Prime finding

1 INTRODUCTION

Smart cards are plastic cards that resemble those usually employed by credit card companies and bank debit cards. The main feature that sets smart cards apart is that while most of credit and debit cards use a magnetic strip to store static information, smart cards have an integrated circuit embedded in the card. A smart card integrated circuit implements a processing unit, optionally a mathematical accelerator unit, and areas of persistent and non-persistent memory. The possibility of performing computation using data stored internally, associated with the tamper-resistant characteristic

[4] of the integrated circuit, make smart cards well suitable for a variety of applications [4, 18, 23, 25] that require high degree of security. In particular, the tamper resistance and computing power of smart cards can be exploited to perform cryptographic operations without depending on potentially vulnerable external resources.

Public key cryptography has gained extreme popularity since it was first published to the unclassified community [6]. A wide variety of schemes have been designed that use public key algorithms, e.g. digital signature [9, 17, 19] and key exchange [7]. One of the most popular public key cryptographic algorithms is called RSA [19]. Although there are some controversy about the reasons of its popularity [15, 26], RSA is a very simple and easy to implement algorithm. In addition, many implementations of RSA cryptosystem have been studied extensively [11, 13, 21]. On September 7, 2000, two weeks before the patent expired, RSA security relinquished its patent on the RSA algorithm. All these characteristics combined make RSA very attractive for use on the near future, unless a powerful attack against RSA is discovered. Because of this, manufacturers of microcontrollers (the integrated circuits that are used in smart cards) and application developers have been implementing and optimizing the RSA algorithm on these microcontrollers. However, the computational power of smart cards is very limited and the on-card implementations are usually much slower than that in desktops. Because of this, the high-end smart card microcontrollers are equipped with special hardware, called crypto-coprocessor [8], which can accelerate the crypto computations for a class of public key cryptographic algorithms. The crypto-coprocessor is a specialized circuitry that is able to perform fast modular exponentiation. Therefore, the crypto-coprocessor accelerates encryption and decryption of public key cryptographic algorithms that use the very computing intensive modular exponentiation. RSA uses modular exponentiation for encryption and decryption of data and as such can benefit from a crypto-coprocessor for these two operations.

Although crypto-coprocessors help the RSA key generation by accelerating the modular exponentiation operation, it alone cannot let smart card achieve the desired efficiency for on-card key generation. Thus, and due to the nature of the procedure for key generation, which will be discussed later in this paper, the generation of a key pair takes several seconds to complete. Some companies try to solve this problem by generating the RSA key pairs on a desktop and upload the pair, or only the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

private key, into a smart card. However, the use of this scheme gives attackers an easy way to compromise the security of applications using smart cards by compromising desktops providing RSA keys. Thus, the RSA key generation is preferably performed totally inside the smart card secure microcontroller in order to guarantee the efficiency and the security of the applications that use smart cards.

Although smart cards have been gaining popularity, there is only a small literature on the implementation of cryptographic algorithms for smart card microcontrollers. In particular, there are very few articles that discuss the trade-offs and techniques for generating RSA key pairs inside smart cards. Some commercial companies, like Schlumberger and DataKey, publish timings for 1024-bit RSA key pair generation, but they do not provide any information of how they were able to achieve the claimed performance.

The RSA key generation can be mapped to the problem of finding large primes. Because of this, the time to generate an n -bit RSA key pairs is mostly due to the time to find two $(N/2)$ -bit primes. Marc Joy et al. [10] describe an efficient prime-finding algorithm for smart card microcontrollers. Although this is a step in the right direction for dissemination of knowledge of the problems faced when implementing cryptographic algorithms on smart cards, the paper is vague on some details of their implementation. This paper provides detailed technical information on trade-offs of implementations of RSA key generation applications for smart card microcontrollers.

RSA key lengths are increased every few years to ensure that the improved factoring algorithms do not compromise the security of messages encrypted with RSA. Thus, it is important to investigate the performance of algorithms for generation of RSA key pairs larger than 1024 bits, which is one of the most used key length currently. This paper gives the performances of some discussed algorithms when generating 1024-bit primes, which correspond to 2048-bit RSA key pairs. All timing measurements are taken using the SLE66CX160S microcontroller manufactured by Infineon Technology at 3.57 MHz internal clock frequency. The SLE66CX160S microcontroller has a crypto coprocessor, which is used to illustrate how some of the special features of crypto coprocessors can improve the efficiency of the overall implementations.

The rest of paper is organized as follows. Section 2 briefly describes the RSA key pair generation. Section 3 details the designing issues related to prime finding algorithms. The algorithms analyzed in this paper are compared with that given in [10] in section 4. Section 5 discusses the generation of public and private key pairs. Finally, in section 6, the conclusions and future work are discussed.

2 RSA KEY GENERATION

The RSA cryptographic algorithm [19], which was discovered in 1977, is a public-key algorithm that can be used in many schemes, e.g. digital signatures. As any other public key cryptographic algorithm, RSA uses a key for encryption that is different from the decryption key. A key pair must be

generated before any encryption or decryption can occur. The procedure for generating a k -bit key pair is as below.

1. Find two primes, p and q , of length $k/2$ bits;
2. Compute $n = p \cdot q$;
3. Choose a public key e co-prime to $(p-1)(q-1)$ and compute the inverse, say d , of $e \pmod{(p-1)(q-1)}$;
4. The pair (e, n) is published as the public key, and the pair (d, n) , or (p, q, n) is kept secret as private key by the owner.

Finding the two large primes p and q is the most costly operation in generating the keys, being roughly the total time for RSA key pair generation.

3 LARGE PRIME FINDING ALGORITHMS

It was argued on section 2 that the total time for generating an RSA key pair is almost totally due to the time of finding two large primes. Therefore, it is necessary to optimize the performance of the algorithm used for finding large primes in order to optimize the performance of the algorithm for generating the RSA key pair. The next subsections describe several factors that need to be considered when optimizing the performance of an algorithm for finding large primes. Initially a general prime finding algorithm is presented, what shows that designing an efficient sieve algorithm is critical for prime finding. Then, two efficient sieve algorithms are described, comparing them to existing algorithms. Smart cards hardware presents particular challenges. Thus, techniques useful for efficiently implementing algorithms in smart cards are discussed.

3.1 Prime Distribution

According to [12], the number of primes less than a natural number N is asymptotically equal to $N / \log N$. Thus, if a x less than N is chosen randomly, the probability of x being a prime number is approximately $(N/\log N)/N$ or $1/\log N$. If x is an n -bit number, then $\log x \approx n \cdot \log 2 \approx 0.69n$. Therefore, the ratio of primes among n -bit numbers is $1/0.69n$, i.e. one out of $0.69n$ n -bit numbers is a prime. Table 1 shows the number of n -bit numbers that contains one prime on the average.

	256	512	1024	2048
Number of n -bit random numbers that contain one prime on the average	176	355	710	1420

Table 1. Number of randomly generated numbers needed to obtain one prime on the average

3.2 Primality Test

The generation of keys must use numbers that are as close to random as possible. However, after generating a random number, the generated number must be tested for primality in order to be useful for the generation of a RSA key pair.

There is a very simple method to test if a given number is prime, known as sieve of Eratosthenes [5]. The method is efficient to test the primality of small numbers. But for prime finding of RSA key pairs where the prime numbers need to be several hundreds bit long, sieve of Eratosthenes is impractical

[12]. Other primality tests able to deal with large number were developed due to the difficulty of applying the sieve of Eratosthenes for large numbers.

Primality tests can be divided into two categories: primality test and probabilistic primality test. Using probabilistic primality tests, a number is found to be composite with probability 1 or prime with some probability < 1 . Hence, by repeatedly running the test one gains more and more confidence on the result. The most common probabilistic primality tests are the Fermat, Solovay-Strassen, and Miller Rabin tests [5]. Primality tests [2, 16] will find if a number is prime with probability 1. Although primality tests may seem the most appropriate technique when finding if a number is prime, they are much more complex and computing power intensive than probabilistic primality tests. Therefore, most of the algorithms used for testing a number for primality use probabilistic primality testing, which is much faster than primality testing. A number is a prime number with a high probability if it passes a probabilistic primality a certain number of times. The number of iterations necessary for the probabilistic primality test implemented will be discussed later in this paper.

3.3 Prime Finding Algorithms

A naïve approach to find an n-bit prime is to randomly choose an n-bit odd number, and call a probabilistic primality test function T using the odd number as input (T in this paper includes the number of iterations necessary to achieve a comfortable level of confidence on the primality of a number). In case the probabilistic primality test returns that the number is not a prime number, another random odd number is chosen and the same procedure is repeated, until a prime number is found. The tested number is output if the probabilistic primality test returns that the number is a probable prime. This algorithm, shown in Figure 1, is referred to as the naïve prime finding algorithm.

1. Pick a random n-bit odd number q
2. If $T(q) = \text{false}$ then goto 1
3. Output q and halt

Figure 1: Naïve prime finding algorithm

An average of 176 calls to the probabilistic primality test function T is required, from Table 1, to find a 512-bit prime using the naïve prime finding algorithm. An average of 355 calls to T is required to find a 1024-bit prime. Therefore, it can be concluded that the implementation of the primality test function T must be optimized and the number of calls to T must be the lowest possible in order to optimize the performance of the prime finding algorithm. The optimization of T is discussed on session 3.4 of this paper.

One way to reduce the number of calls to the probabilistic primality test function T is to use a variation of sieve of Eratosthenes called before calling T. The variation of the sieve of Eratosthenes (sieve for abbreviation), called before T, is able to detect a portion of composites that contain small factors and is shown in Figure 2.

1. Let p_i be the i-th smallest odd prime ($p_1 = 3, p_2 = 5, \dots$)
2. Let $S(k)$ be a set of small primes such that $S(k) = \{p_i \mid p_i \leq k, i \in \mathbb{N}\}$, where k can be any positive number
3. For a given number q, divide q by all the elements in $S(k)$
4. If q is not divisible by all the elements in $S(k)$, q is said to survive the sieve. Otherwise q is said to fail the sieve, i.e., q is a composite number.

Figure 2. Variation of sieve of Eratosthenes

Although nothing can be said about the primality of q if it survives the sieve, the sieve function avoids calls to T for numbers found composite. The sieve function is very time efficient for $S(k)$ when k is small, requiring much less processing time than T. Therefore the sieve function can help with an early detection of a portion of composite candidates, increasing the overall performance of the prime finding algorithm.

An experiment was conducted using 50,000 random numbers to illustrate the efficiency of the sieve function in detecting composite number. Initially a set of 50,000 512-bit random numbers is chosen. Then, the sieves with different small prime sets $S(k)$, are applied to the set. The number of survivors is recorded for each prime set. Figure 2 shows the percentage of numbers that survive the sieve. It is easy to note from Figure 2 that even for small sets, like $S(29)$, almost 70% of the numbers tested are detected as composite.

	S(29)	S(256)	S(512)	S(2560)	S(5120)
Ratio	30.9%	20.0%	17.8%	14.3%	13.1%

Table 2. Rate of sieve survivors over candidates tested

The algorithm in Figure 1 is changed to use the sieve. The new algorithm is shown in Figure 3. In the new algorithm, $q^{(i)}$ for $i = 0, 1, \dots$, are tested until a probable prime is found. Another small optimization on the algorithm of Figure 3 is that a new $q^{(i)}$ is generated by adding $2d$ to $q^{(i-1)}$. Hence, the algorithm doesn't need to generate an n-bit random number on each iteration, saving the time required for random number generation. The value of d on the algorithm above can be any number. For simplicity, d is chosen to be 1.

1. Pick a random n-bit odd number q and let $q^{(0)} = q, i = 0$
2. Call sieve procedure, if $q^{(i)}$ fails, goto 4
3. If $T(q^{(i)}) = \text{TRUE}$ then output $q^{(i)}$ and halt.
4. $q^{(i+1)} = q^{(i)} + 2d, i = i + 1$, goto 2 (d is a chosen integer)

Figure 3. Naïve algorithm modified to use sieve

Only a small portion of the prime candidates is able to reach step 3 on the algorithm of Figure 3 because of the sieve procedure. Therefore, the average number of calls to T is reduced. Table 3 shows the expected number of calls to T, taking into account the values from Tables 1 and 2.

	S(29)	S(256)	S(512)	S(2560)	S(5120)
512-bit	54.7	35.5	31.0	25.3	23.2
1024-bit	109.3	71.0	62.0	50.1	46.4

Table 3. Expected average number of calls to the probabilistic primality test T

A bigger size of the set $S(k)$ results in fewer calls to primality test T. Thus it is desirable to have $S(k)$ as big as possible. However, bigger size of $S(k)$ causes an increase of both storage space to keep pre-computed primes and processing time for the sieve procedure. Thus, one of the goals when optimizing a prime finding algorithm is to find the optimal point of using a sieve algorithm that uses the largest $S(k)$ possible keeping a small overhead.

One of the most used sieve methods is the trial division method [3]. The prime finding algorithm using the trial division method is shown in Figure 4, with p_i defined the same way it was defined in Figure 1.

1. Choose a set $S(k)$. Pick a random n-bit odd number q and let $q^{(0)} = q, i = 0$
2. Let $w_j^{(i)} = q^{(i)} \bmod p_j$. If $w_j^{(i)} = 0$, for any $j, 1 \leq j \leq k$, goto 4
3. If $T(q^{(i)}) = \text{TRUE}$, output $q^{(i)}$ and halt.
4. $q^{(i+1)} := q^{(i)} + 2, i := i + 1$, goto 2

Figure 4. Prime finding algorithm using trial division

The sieve procedure shown on the Figure 4 uses modular reduction operations. Although smart cards crypto co-processors are designed to speed up modular reductions, these operations are still very expensive. Therefore it is desirable to keep the number of modular reduction operations on the sieve procedure to a minimum. The modular reduction operation of the trial division algorithm is shown on step 2 of Figure 4 ($w_j^{(i)} = q^{(i)} \bmod p_j$). Although modular reduction can be used to compute $w_j^{(i)}$, there is an obvious way to improve it. If $q^{(i+1)} = q^{(i)} + 2$, and $w_j^{(i)} = q^{(i)} \bmod p$, then $w_j^{(i+1)} = w_j^{(i)} + 2 \bmod p$. Thus, one can calculate $w_j^{(i+1)}$ from $w_j^{(i)}$. In addition, if p_j is assumed to be 8-bits long, so is $w_j^{(i)}$. Therefore the computation of $w_j^{(i+1)}$ only uses two 8-bit operands, resulting in a performance much faster than modular reduction operations. The algorithm just described is shown in Figure 5.

1. Choose a set $S(k)$. Pick a random n-bit odd number q and let $q^{(0)} = q, i = 0$
2. Compute $w_j^{(0)} = q^{(0)} \bmod p_j, 1 \leq j \leq k$
3. If $w_j^{(i)} = 0$, for any $j, 1 \leq j \leq k$, goto 5
4. If $T(q^{(i)}) = \text{TRUE}$, output $q^{(i)}$ and halt.
5. $w_j^{(i+1)} = w_j^{(i)} + 2 \bmod p_j, 1 \leq j \leq k$
6. $q^{(i+1)} = q^{(i)} + 2, i = i + 1$, goto 3

Figure 5. Prime finding algorithm using table look-up

The algorithm shown on Figure 5 requires a table [w_1, w_2, w_3, \dots] to keep all the residues $w_j^{(i)}$ of the previous iteration. Therefore, this algorithm is referred to as table look-up algorithm. In order to optimize the performance, $w_j^{(i+1)} = w_j^{(i)} + 2 \bmod p_j$ can be calculated as shown in Figure 6.

1. $w_j^{(i+1)} = w_j^{(i)} + 2$
2. $w_j^{(i+1)} = w_j^{(i+1)} - p_j$
3. If $w_j^{(i+1)} < 0, w_j^{(i+1)} = w_j^{(i+1)} + p_j$.

Figure 6. Algorithm for finding $w_j^{(i+1)}$

The table look up sieve was evaluated and the results will be discussed later in this paper. In addition, another method [22] for sieve was evaluated. In order to understand the other algorithm, let us consider an interval of l test candidates, say $\{q^{(0)}, q^{(1)}, \dots, q^{(l-1)}\}$. Let's then define a bit array $A = [a_0 a_1 \dots a_{l-1}]$, where a_i is the i -th bit, initially set to 0 with a_i representing the number $q^{(i)}$. For each prime p from the set $S(k)$, we find out a starting point $g(p) = \min\{i \mid q^{(i)} \text{ is divisible by } p, i \in \mathbb{N}\}$. If $g(p)$ is less than l , set $a_{g(p)}$ to 1, and then every p -th bit of A is set to 1 since the values of q that are represented by these bits would also be divisible by p . After this sieve, a_i is zero if and only if $q^{(i)}$ is not divisible by any of the numbers in $S(k)$. The sieve is then concluded and the prime finding algorithm must only scan the bit array A and try the primality test T for each $q^{(i)}$ such that a_i is zero. This algorithm is called the bit-array algorithm. In order to find out the start points for the bit-array algorithm, it is used the fact that for $q^{(0)} = q$, we let $r = q \bmod p$. Then¹, if r is odd, $g(p) = (p - r)/2$. If $r = 0, g(p) = 0$. Else r is even and $g(p) = (2p - r)/2$.

The bit-array algorithm will find a probable prime if there is one in the chosen interval. However the interval may be such that there is no probable prime. In the case there is no probable prime on the chosen interval, one can either randomly choose another odd q and let $q^{(0)} = q$, or simply let $q^{(0)} = q^{(l-1)} + 2$, repeating the same procedure. The bit array algorithm is shown in Figure 7.

1. Set $a_i = 0$, for $0 \leq i \leq l-1$;
2. Pick a random n-bit odd number q and let $q^{(0)} = q, i = 0$
3. For each $p_j \in S(k)$, do
 - a. Compute $w_j^{(0)} = q^{(0)} \bmod p_j$
 - b. Compute $g(p_j)$
 - c. Set $a_{g(p_j)+mp_j} = 1, 0 \leq m \leq \lfloor (l - g(p_j))/p_j \rfloor$
4. for $i := 0$ to $l-1$, do
 - a. If $(a_i = 0)$ and $(T(q^{(i)}) = \text{TRUE})$, output $q^{(i)}$ and halt.
 - b. $q^{(i+1)} = q^{(i)} + 2$
 - c. $q^{(0)} = q^{(i)}, i = 0$, goto 3

Figure 7. Prime finding algorithm using bit array

¹ Assuming r is odd and $v = (p - r)/2$, it is easy to verify that v is an integer and $v < p$. $q^{(v)} = q^{(0)} + 2 * (p - r)/2 = q + p - r$. Since $r \equiv q \bmod p, q - r \equiv 0 \bmod p, q^{(v)} \equiv q - r + p \equiv 0 \bmod p$. Hence, v is the smallest positive integer such that $p \mid q^{(v)}$ which means $g(p) = v = (p - r)/2$. Conclusion follows. Proofs for $r = 0$ and r even are similar.

3.4 Fast Implementation of Probabilistic Primality Tests

The existing probabilistic primality test algorithms are already well studied. Therefore, the algorithms were optimized for smart cards that have crypto co-processors, without any effort of optimizing the underlying algorithms.

Modular exponentiations are the most computing expensive operations of probabilistic primality test algorithms. Hence, it is important to design a fast implementation for modular exponentiation.

In our implementation, binary method for exponentiation [12] is used. In order to compute $B = A^E \pmod M$, E is represented in binary by $E = [e_{k-1} \dots e_0]$ with $e_{k-1} = 1$. Figure 8 shows how to compute $B = A^E \pmod M$ using the binary representation of E .

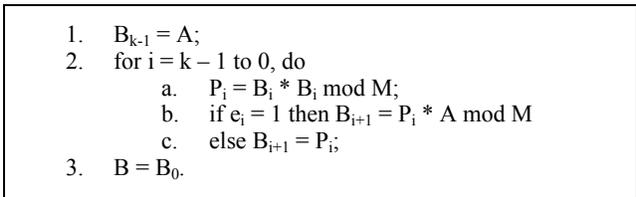


Figure 8: Algorithm to compute $B = A^E \pmod M$.

Usually the exponent E on probabilistic primality test algorithms is as large as the module M , while the basis A is an arbitrary number called witness. A smart card crypto co-processor implements the modular multiplication as an operation of n -bit multiplicand and m -bit multiplier, where n and m can be different. A smaller bit-length of multiplier can achieve better speed for the calculation in step 2.b. on Figure 8. The witness A can be any number between 2 and $n - 2$. Thus, the number 2 is the best choice for A because with this choice the multiplier on step 2.b (A) is only 2-bit long. Using 2 as the witness can result in 33% time-saving than that using randomly picked up witness.

The crypto co-processor in the Infineon SLE66CX160S microcontroller supports two modes of operation: short mode for operations using numbers up to 560-bit long, and long mode for operations using numbers up to 1120-bit long. A characteristic of the Infineon SLE66CX160S is that the performance of modular exponentiation operation on its two modes of operation is different. For a modular exponentiation where both the module and exponent are 512-bit long and the witness is 2, the executing time is 110 ms. under the short mode and 220 ms. under the long mode.

3.5 Timings

Prime finding algorithms were implemented verify the performance and gains obtained using the proposed optimizations. The times for completion of the algorithms are described in this subsection.

As already discussed, the most costly part of an RSA key generation is the prime finding procedure. In addition, the biggest optimization on the performance of the prime finding algorithm is due to the optimizations performed on the sieve algorithms. Table 4 shows the overhead on using the sieve algorithm.

An odd number q is chosen randomly for the trial division and table look-up algorithms. The processing time for 177 consecutive odd candidates, beginning with q , is measured for a 512-bit q . The same process is repeated for a 1024-bit q , using 355 consecutive odd candidates. From Table 1, one needs 177 candidates on average to find a 512-bit prime and 355 candidates to find a 1024-bit prime. Thus, the measured timings are good approximations for the sieve procedures overhead.

The search interval for the bit array algorithm is set to be $\{q^{(0)}, q^{(1)}, \dots, q^{(1023)}\}$. The time to check the divisibility of $q^{(0)}, q^{(1)}, \dots, q^{(1023)}$ and mark the corresponding bits in A is measured, which is called a one-round processing time. The overhead in the sieve procedure for 512-bit prime finding is computed by $1.0075 * \text{one-round processing time}$, while the overhead for 1024-bit prime finding is computed by $1.0575 * \text{one-round processing time}$.²

The performance shown in Table 4 is an average of 400 example measurements as described above for each sieve implementation. All the timings are obtained by counting the microcontroller clock cycles and translating it into seconds, except for the trial division sieve using 1024-bit numbers. The timing for the trial division sieve using 1024-bit numbers is measured outside the card and the overhead is computed from this measurement.

	Memory Space Used	512-bit prime (sec)	1024-bit prime (sec)
Trial division algorithm with S(256)	53 bytes code memory	2.00	20.00
Table look-up algorithm S(256)	53 bytes code memory 53 bytes RAM	0.20	0.80
Bit array algorithm with S(256)	53 bytes code memory 128 bytes RAM	0.11	0.17
Bit array algorithm with S(512)	93 bytes code memory 128 bytes RAM	0.15	0.26
Bit array algorithm with S(2560)	374 bytes code memory 128 bytes RAM	0.37	0.84
Bit array algorithm with S(5120)	691 bytes code memory 128 bytes RAM	0.64	1.55

Table 4. Overhead of sieve procedures for different algorithms

² In our experiment of 400 examples, an average of 1.0075 round are needed for the search interval in each 512-bit prime finding and an average of 1.0575 round are needed in each 1024-bit prime finding.

The performance of the bit-array algorithm is shown in Table 4 using four different small prime sets, S(256), S(512), S(2560), S(5120). Although the algorithms that use a larger small prime sets, like S(512), S(2560), S(5120), will spend more time in sieve procedure, it will gain in the overall efficiency in prime finding by reducing the calls to the probabilistic primality test.

The probabilistic primality test to be implemented in a smart card must be both efficient and have a small memory footprint due to the restrictions of the smart card, in addition of providing a high confidence of primality for a number passing the test. Two probabilistic primality tests that present these characteristics are the Miller-Rabin and Fermat test, if the later is combined with the sieve routine. The Rabin-Monier theorem states that the probability that an odd composite number n can pass Miller-Rabin test with t iterations is at most 4^{-t} . Hence, Miller-Rabin test with five iterations ensures that the candidates passing the test are not prime with the chance less than $2^{-20} (< 10^{-6})$. However, the Fermat test is easier to implement and can practically provide very high level of primality confidence for a number passing the test and using a sieve. An experiment was conducted in [20] where approximately 718 million 256-bit numbers were tested by trial division with S(104) and one-round Fermat test with witness 2. All the numbers passing both the trial division test and one-round Fermat test with witness 2, pass an eight round Miller-Rabin test. Because of this the one-round Fermat test with witness 2 was used for getting the performance measurements.

Using the implementation described above, the probabilistic primality test takes 0.11 ms. to test if a 512-bit number is a probable prime and 0.88 ms. to test if a 1024-bit number is a probable prime in average. The total time for finding a probable prime for the different sieve procedures is shown in Table 5. The timings in table 5 are an average of the time to find 400 different probable primes. The timings include the overhead introduced by sending data and receiving data from the smart card, copying the input into certain destination and jump to the prime-finding routine. The average overhead is 0.39 for finding a 512-bit prime is 0.47 and for finding a 1024-bit prime.

	512-bit prime (sec)	1024-bit prime (sec)
Trial division algorithm with S(256)	8.50	89.54
Table look-up algorithm S(256)	5.20	59.84
Bit array algorithm with S(256)	4.99	65.39
Bit array algorithm with S(512)	4.68	58.40
Bit array algorithm with S(2560)	3.96	44.76
Bit array algorithm with S(5120)	4.05	45.76

Table 5. Performance of the prime finding algorithms using different sieve procedures

4 RELATED WORK

Although it seems simple, the prime finding algorithms are scarcely investigated. In particular, there are very few technical papers that describe the performance of prime finding algorithms in smart card and give out the detailed time measures for it. Because of this, the proposed optimizations and the prime finding algorithms are compared with the algorithm given by Marc Joy et al. [10].

The major advantages of the algorithm proposed in this paper are:

- **Performance** The bit-array algorithm with S(2560) makes 25.3 calls on average to a probabilistic primality test T for each 512-bit prime finding and 50.1 calls on average for each 1024-bit prime finding. The algorithm proposed in [10] makes 33.3 calls and 60.0 calls on average respectively.
- **Space** The bit-array algorithms only need a few hundred bytes of code memory and 128 bytes RAM as shown in Table 4. The table look-up algorithm needs 53 bytes code memory and 53 bytes RAM. The space requirement just described is the same for finding either 512-bit or 1024-bit probable primes. The algorithm described in [10] requires about 2.7 KB of code memory to store pre-computed data for finding a 512-bit probable prime. It doesn't mention the memory needed for finding a 1024-bit prime.
- **Flexibility** Users may need to generate RSA key pairs with different lengths to meet different security requirement. Thus, the prime finding algorithm should be able to find probable primes of different bit lengths. On the algorithm described in [10], each specific pre-computed data can only be used for finding probabilistic primes with a specific bit length. Both the table lookup algorithm and the bit array algorithm can be used to generate random primes of any bit length.

5 GENERATION OF AN RSA KEY PAIR

The RSA public key is usually either randomly generated or pre-defined to be small. Thus, all the time required for generating a RSA key pair is due to determining the RSA private key. The private key is the modular inverse of the public key as described in section 2. A very common method for finding modular inverses is the extended Euclidean algorithm [24].

Finding the public key and private key after finding the prime numbers is trivial. The particular method for computing modular inverse chosen has little impact on the overall key generation. With the implementation of the Extended Euclidean Algorithm, the overall time to find a 1024-bit RSA key pair is less than 8 seconds with the bit-array algorithm S(2560) and less than 11 seconds with table look-up algorithm S(256).

```

1. Compute  $m = \Phi(n) = (p-1)(q-1)$ ;
2. Let  $g_0 = m$ ,  $g_1 = e$ ,  $u_0 = 1$ ,  $v_0 = 0$ ,  $u_1 = 0$ ,  $v_1 = 1$ ,  $i = 1$ ;
3. While  $g_i \neq 0$ 
  a.  $y := g_{i-1} \text{ div } g_i$ ;
  b.  $g_{i+1} := g_{i-1} - y * g_i$ ;
  c.  $u_{i+1} := u_{i-1} - y * u_i$ ;
  d.  $v_{i+1} := v_{i-1} - y * v_i$ ;
  e.  $i := i + 1$ ;
4.  $x := v_{i-1}$ ;
5. If  $x = 0$  Then  $\text{inv} = x$  else  $\text{inv} = x + m$ ;
6. Output  $\text{inv}$ .

```

Figure 9. Euclid's extended algorithm for computing inverses

6 CONCLUSION

This paper provides a detailed description of the steps on optimizing the performance of RSA key generation in smart cards. The paper provides the factors that must be taken into account when designing cryptographic algorithms for smart cards. Although there is an alternative way to generate RSA key pair [1], it is not suit for that in smart card due to the inefficiency [14].

The on-card key generation problem is, in fact, a large prime finding problem. This paper proposes a prototype for fast prime finding algorithms. From it, we are able to derivate two efficient prime finding algorithms. The paper discusses in detail how to build fast implementation for the algorithms in smart card with crypto-coprocessor and provides the timings for finding large primes based on our implementation of the algorithms on an Infineon SLE66CX160S microcontroller. It is shown in this paper that the algorithms presented are more flexible, efficient and space-saving than other algorithms. The algorithms presented are especially suit for on-card RSA key generation. From the paper, one can see that designing efficient sieve algorithms is very critical to the prime finding algorithm.

The performance and optimizations for keys larger than 1024-bits are going to be investigated in a future work. Larger key sizes present new challenges since smart cards crypto-coprocessors have fixed size registers that are not able to accommodate large numbers.

Elliptic curve cryptography (ECC) can provide the same level of security using smaller key lengths than RSA. However, smart card crypto-coprocessors were not designed with ECC in mind. A future work is to optimize and investigate the performance of several ECC algorithms.

7 REFERENCE

- [1] Blackburn, S., Black-Wilson, S., and Burmester, M. Shared Generation of Shared RSA Keys. Technical Report CORR 98-19, Dept. of C&O, Univ. of Waterloo, Canada, 1998.
- [2] Bosma, W. Primality Proving with Cyclotomy. Doctoral Dissertation, University of Amsterdam, 1990.
- [3] Bressoud, D. M. Factorizations and Primality Testing. Springer-Verlag, New York, 1989.

- [4] Chan, S. C. An Overview of Smart card Security, <http://www.hkstar.com/~alanchan/papers/smartCardSecurity/>.
- [5] Dhem, J. F. Design of an Efficient Public-key Cryptographic Library for RISC-based Smart Cards. Universit Catholique de Louvain - Facultdes Sciences Appliqus - Laboratoire de microelectronique, Louvain-la-Neuve, 1998.
- [6] Diffie, W., and Hellman, M. Multiuser Cryptographic Techniques. Proceedings of AFIPS National Computer Conference, 1976, 109-112.
- [7] Diffie W., and Hellman, M. New Directions in Cryptography. IEEE Transactions on Information Theory, IT-22n. 6, Nov. 1976, 644-654.
- [8] Handchuh, H., and Paillier, P. Smart Card Cryptocoprocessors for Public-Key Cryptography. Crypto Bytes, RSA Laboratories, 4 (summer 1999), 6-10.
- [9] Johnson, D., and Menezes, A. The Elliptic Curve Digital Signature Algorithm (ECDSA). Technical Report CORR 99-34, Dept. of C&O, Univ. of Waterloo, Canada, 1999.
- [10] Joye, M., Palliar, P., and Vandenedy, S. Efficient Generation of Prime Numbers. CHES 2000, 340-354.
- [11] Kaliski, B., and Robshaw, M. The Secure Use of RSA. CryptoBytes, RSA Laboratories, (Autumn 1995), 7-13.
- [12] Knuth, D.E. The Art of Computer Programming. Semi numerical Algorithms of Computer Science and Information Processing, Addison-Wesley, 3rd ed., Vol. 2, 1997.
- [13] Koç, Ç. K. High-Speed RSA Implementation. Technical Report TR-201, version 2.0, RSA Laboratories, November 1994.
- [14] Malkin, M., Wu, T., and Baneh, D. Experimenting with Shared Generation of RSA Keys. SNDSS'99, 1999, 43-56.
- [15] Meneze, A. Comparing the Security of ECC and RSA. <http://cacr.math.uwaterloo.ca/~ajmeneze/misc/cryptogram-article.html>.
- [16] Morain, E. Implementation of the Goldwasser-Killian-Atkin Primality Testing Algorithm. Mathematics Computation, Vol. 54, 1990, 839-854.
- [17] NIST, US Department of Commerce, Digital Signature Standard, FIPS PUB 186, May 1994.
- [18] Peyret, P. Which Smart Card Technology Will You Need to Ride the Information Highway Safely? Head of Research & Development, Gemplus Corporation, 1995.
- [19] Rivest, R., Shamit, A., and Adleman, L. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. Communications of ACM, Vol. 21, No. 2, Feb. 1978, 158-164.
- [20] Rivest, R.L. Finding Four Million Random Primes. Advances in Cryptology, Crypto '90, Springer-Verlag, 1991, 625-626.
- [21] Rivest, R.L. Response to NIST's proposal. Communications of ACM, 35, 1992, 41-47.
- [22] Silverman, R. D. Fast Generation of Random, Strong RSA Primes. Crypto Bytes, RSA Laboratories, 3 (spring 1997), 9-13.
- [23] Schumberger Limited. Advantages, Smart Cards: Inherent Advantages. http://www.slb.com/et/inherent_advantage.html.
- [24] Stallings, W. Cryptography and Network Security: principle and practice, 2nd ed., Prentice-Hall, New Jersey, 1999.

[25] University of Texas at Austin. Enabling Smart Commerce in the Digital Age. <http://cism.bus.utexas.edu/works/articles/smartcardswp.html>.

[26] Wiener, M. Performance Comparison of Public-Key Cryptosystems. Crypto Bytes, RSA Laboratories, 4 (summer 1999), 1-5.