Report of the project

Generation of strong primes for ssh keys

Julien Caron Jeremie Stordeur

ABSTRACT

This paper review the interest of using strong prime numbers to generate RSA Keys.
To generate efficiently prime numbers we need an efficient primality test. This is why the first part of our work is a study on Miller-rabin's primality test and the Agrawal-Kayal-Saxena test. We compare those tests to determine if a determistic test like AKS can be used to generate large prime numbers.
In the second part we implement an algorithm to generate prime numbers with a minimum of calls to the oracle (a primality test, Miller-Rabin for our implementation) [5].
In the final part we describe how it is possible to use this generator of prime numbers to create valid SSH-RSA Keys using the open source libraries : *openssh openssl*.

Keywords

strong primes, AKS, SSH

1. INTRODUCTION

In cryptology every efficient codes is based on a hard operation, "hard" meaning it takes a very long time to complete it. The factorisation of large prime numbers is the key of the RSA system where n=p.q with p and q two large prime numbers.
A prime number is said to be *strong* when :

$$\begin{cases} \text{q-1 has a large prime divisor u,} \\ \text{q+1 has a large prime divior s,} \\ \text{u-1 has a large prime divisor t.} \end{cases}$$

Strong primes have certain properties that make the product n hard to factor by specific factoring methods : the Pollard p-1 and p+1 methods are especially suited to primes p such that p-1 or p+1 has only small factors [8]. For this reason, strong primes are required by the ANSI X9.31 standard for use in generating RSA keys for digital signatures. However, strong primes do not protect against modulus factorisation using newer algorithms such as Lenstra elliptic curve factorization [7] and Number Field Sieve algorithm [6].
However, there is no danger in using strong, large primes, though it may take slightly longer to generate a strong prime than an arbitrary prime. The C++ cryptographic library Givaro is currently using the Gordon's algorithm to produce strong primes [4]. We implemented another algorithm theoricaly better than Gordon's [5] to compare their speed.

2. PRIMALITY TEST

The algorithm used in the next section to generate strong primes requires several calls to a primality test. In order to improve the efficiency of our generation algorithm we compared two primality tests. The first one is the probabilistic test of Miller-Rabin which is currently used in the Givaro library and the second one is the Agrawal-Kayal-Saxena test.

2.1 Miller-Rabin

This test is based on a property :

Let n be a prime number : $n = 2^s * d$ with d odd.
Then $\forall a \in (\mathbb{Z}/n\mathbb{Z})$,
$a^d \equiv 1\ mod(n)$
or
$\exists r, 0 \leq r \leq s - 1 / a^{2^r * d} \equiv -1\ mod(n)$

Figure 1: property (1)

So if we can find an a such that
$a^d \neq 1\ mod(n)$ and $\forall r, 0 \leq r \leq s - 1 / a^{2^r * d} \neq -1\ mod(n)$ then a is called a witness for the compositeness of n, which means that a has proven that n wasn't prime.Otherwise n is a strong probable prime to base a. Unfortunatly for a given n composite it is possible to find some a that will satisfy the property (1).
Such an a is called a strong liar and make the test answering "n is prime" whereas n is actually composite.
It can be shown that for any odd composite n, at least $\frac{3}{4}$ of the bases a are witnesses for the compositeness of n [9]. It means that if we run the test with only one value for a there is a probability $p = \frac{1}{4}$ to be wrong. This is why the test is only probabilistic and the more bases a we test, the better the accuracy of the test is.

However if we admit the Generalized Riemann Hypothesis it is possible to build a deterministic test by testing every a $\in$ [2, min $(2(logn)^2$, n-1)] [2].
Our implementation of the test gives the users the choice

of the percentage of bases tested. For example for 50% the programm will run the test $log(n)^2$ times with a randomly choosen in [2, min $(2(logn)^2$, n-1)].

2.2 AKS

This algorithm was first presented in the paper *Primes is in P* [1].

INPUT : integer n>1.

1. If ($n = a^b$ for $a \in N$ and b>1) output COMPOSITE
2. Find the smallest r such that order of n modulo r $> log^2 n$
3. If 1< (a,n) <n for some $a \leq r$ output COMPOSITE
4. If $n \leq r$ output PRIME
5. for a=1 to $\sqrt{\phi(r)}logn$ do
   if $(X+a)^n \neq X^n + a(modX^r - 1, n)$ output COMPOSITE
6. output PRIME

r is the smallest number k such that $a^k = 1$ (mod r).
The theorical asymptotic time complexity of the algorithm is $O(log^{21/2}n)$. However this complexity can be seriously reduced to O $(log^6 n)$ with some optimisations [3].
Our implementation of the algorithm offers the user the choice of the percentage of bases a tested in [1, $\sqrt{\phi(r)}logn$].

2.3 Comparison of the two algorithms

We first compare the two algorithms in their deterministic versions (percentage=100), results are stored in the **Fig. 2**. Miller is clearly faster than AKS but in fact both algorithms are very deceptive, Miller-Rabin can't produce a fast answer for a size of n beyond 300 bits.

To have a better idea of the capacity of those algorithms we compared their speed in a probabilist version with two differents value for the percentage of bases a tested : 5% and 1%
Results are on the **Fig. 3** and **Fig. 4**.
Miller-Rabin is clearly faster than AKS but this is essentially because our implementation doesn't have all the optimisations for AKS (there is a way to speed up the squaring in $(\mathbb{Z}/n)[x]/(x^r - 1)$).

3. EFFICIENT GENERATION OF STRONG PRIME NUMBERS

In this section, we study the generation of strong prime numbers. We implemented the algorithm, described in *Efficient Generation of Strong Prime Numbers* [5]. We give here the different algorithms that we used in order to generate strong prime numbers.

3.1 Theory

We first need a generator **g** of invertible numbers modulo Π. Let $\Pi = \prod_{i=1}^{k} p_i^{\delta_i}$ be a n-bit product of the first k primes with some small exponents. Moreover we have a generator of t-bit numbers, **random(t)** with $t = 2\max_i(p_i^{\delta_i})$
The algorithm (**Fig. 5**) generates invertible numbers modulo Π. We did not implement the algorithm as it was done [5], we remove the θ_i and replace it with the chinese reminder theorem.

1. c = 0
2. for i =1 to k
   (a) $\alpha = random(t)$
   (b) if ($\alpha^{\delta_i} \mod p_i^{\delta_i} = 0$) goto (a)
   (c) $\hat{n_i} = \frac{\Pi}{p_i^{\delta_i}}$, find u_i and v_i such as $u_i\Pi + v_i\hat{n_i} = 1$
   $e_i = v_i\hat{n_i}$
   $c = c + \alpha e_i$
3. output c

Figure 5: generator g of invertible numbers mod Π

Generated primes are expected to lie in some target window $[w_{min}, w_{max}]$. Now let η be the product of the first prime numbers so that $\eta \times p_{i+1} \geq w_{min}$. We define $\epsilon_{\min}$ and $\epsilon_{\max}$ such as : $\epsilon_{\min} = \lfloor \frac{w_{\min}}{\eta} \rfloor + 1$ and $\epsilon_{\max} = \lfloor \frac{w_{\max} - w_{\min}}{\eta} \rfloor$.
We then set $\Pi = \epsilon_{\max}\eta$ and $\rho = \epsilon_{\min}\eta$.
In the next algorithms, T denotes the primality oracle. In order to have a fast transition step, we multiply c by 2 and so we need to exclude 2 from Π's factorization.
The algorithm (**Fig. 6**) generates n-bit prime number.

1. $c = \mathbf{g}()$
2. $q = c + \rho$
3. if q is even, $q = q + \eta$
4. if $T(q) = false$ then $c = 2c \mod \Pi$ and goto (2)

Figure 6: $GPrime(n)$ - A prime number generator

We now focus on the problem of generating an uniformly distributed random n-bit prime $p = 1 + qr$ for a given n_q-bit prime q. Trial divisions are intended to check that the candidate p has no prime factor p_i for $i = 1, .., k$. We can advantageously generate r so that p automatically fulfills this condition. It suffices that
$p \neq 0 \mod p_i \Leftrightarrow r \neq -q^{-1} \mod p_i$ for $i = 1, .., k$
We choose $r = -q^{-1} + c \mod \Pi$

size of n (bit)	30	40	50	100	200	300
Miller (s)	0.03	0.07	0.1	0.63	9.73	43.6
AKS (s)	40.5	342	361	-	-	-

Figure 2: comparison between Miller-Rabin and AKS with the deterministic versions

size of n (bit)	50	75	100	150	512	700
Miller (s)	0.02	0.03	0.05	0.14	24.3	100
AKS (s)	25	111	384	-	-	-

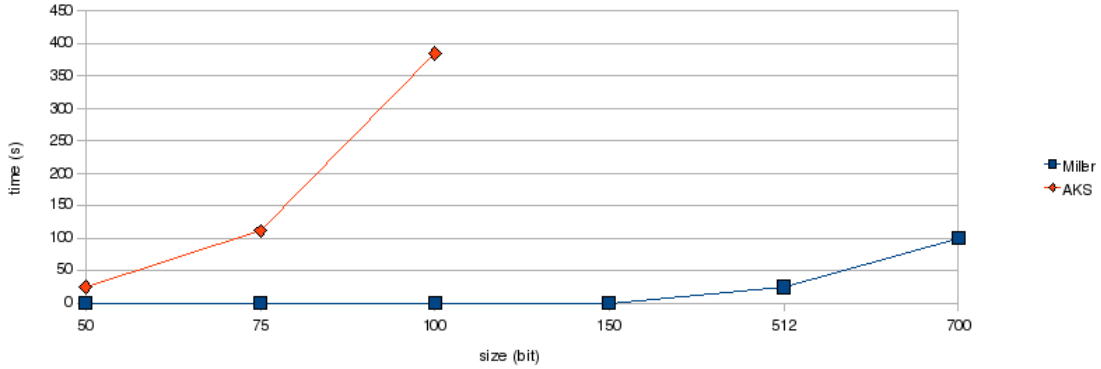

Figure 3: Comparison between AKS and Miller-Rabin for 5% of witnesses tested

size of n (bit)	50	100	125	150	512	1024
Miller (s)	0.01	0.01	0.03	0.07	4.9	122
AKS (s)	6	69	252	-	-	-

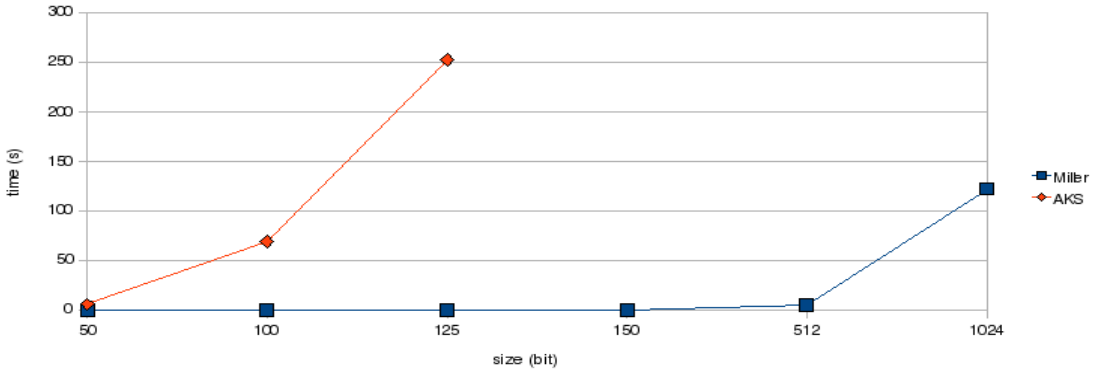

Figure 4: Comparison between AKS and Miller-Rabin for 1% of witnesses tested

For this algorithm, we set $\Pi \geq 2^{n-n_q+2}$ and when r is calculated, we check that $r \geq 2^{n-n_q}$ so that $p = 1 + qr \geq 2^{n-1}$

The algorithm (**Fig. 7**) generates such prime numbers. In this implementation of the algorithm, we calculate the inverse of q with the extended Euclidean algorithm, and not with the Carmichael's function as it was done in the paper of Marc Joye, Pascal Paillier and Serge Vaudenay [5], because our tests demonstrate that the extended Euclidean algorithm is faster.

1. calculate q^{-1} with the extended Euclidean algorithm
2. $c = \mathbf{g}()$
3. $r = (c - q^{-1}) \mod \Pi$
4. if $r \leq 2^{n-n_q}$ goto (3)
5. $p = 1 + qr$
6. if $T(p) = false$ then $c = 2c \mod \Pi$ and goto (3)
7. output p

Figure 7: $GDSA(n, q)$ **-DSA prime generator**

We still have to generate a n-bit prime q such that $q - 1$ has a large prime divisor u, and $q + 1$ has a large prime divisor s.

$$q = 1 + r_1 u = -1 + r_2 s$$

Hence $r_1 = -2u^{-1} \mod s$ and there must be an integer r_3 such that

$$q = 1 + u(-2u^{-1} \mod s + r_3 s)$$

$$r_3 = -(su)^{-1} - (-2u^{-1} \mod s)s^{-1} + c \mod \Pi$$

we denote

$$\kappa = 1 + u(-2u^{-1} \mod s)$$

$$\mu = -\kappa(su)^{-1} \mod \Pi$$

In order to generate strong prime numbers, we first generate s and t with $GPrime(n/3)$ and then u with $GDSA(n/2, t)$

We check that $r \geq 2^{\frac{n}{6}-1}$ so that $q = \kappa + sur \geq 2^{n-1}$

Eventually, the algorithm (**Fig. 8**) generates strong prime numbers.

3.2 Comparison with Gordon's algorithm

We stress the fact that this technique features a dramatic performance improvement in terms of the average number of calls to T executed by *GStrong* and the classical method, Gordon's algorithm. We give in **Fig. 9** a comparison of the heuristic expected number of calls to the primality oracle T.

However, even if the average number of calls to T is divided by around 4.4, in terms of time the gain isn't so obvious with our oracle (we used the function is_prime from the library *Givaro*). We compared in **Fig. 10** the time of the generation of the two algorithms, we see that generation's times are quite equivalent and that even for small integers (under 1000 bits) Gordon's algorithm is faster, at 2000 bits we only gain half of a second.

1. generate s and t using $GPrime(\frac{n}{3})$ generate u using $GDSA(\frac{n}{2}, t)$
2. $\kappa = 1 + u(-2u^{-1} \mod s)$ and $\mu = -\kappa(su)^{-1} \mod \Pi$
3. $c = \mathbf{g}()$
4. $r = \mu + c \mod \Pi$ and $q = \kappa + sur$
5. if $T(q) = false$ then $c = 2c \mod \Pi$ and goto (4)
6. output q

Figure 8: $GStrong(n)$ **- A Strong prime generator**

We conclude that the gain in time is not equivalent to the gain in number of calls, this is in fact due to the time needed by T, which is higher in the case of GStrong than Gordon (**Fig. 11**) (in average four times higher). This can be explained due to the oracle we used, the function *is_prime* from the library *givaro* does not start directly with the Miller's algorithm but try to find if the gcd of the number with the first prime numbers isn't one. The numbers that Gordon's algorithm generates and tries to find whether or not they are prime often have small prime divisors such as 3, 5 and 7, whereas GStrong's numbers often have bigger prime divisors : this is why the function *is_prime* goes faster to say whether or not they are prime numbers in the case of Gordon.

size of n (bit)	128	256	512	800	1024	1500	2000
GStrong(n)	11.32	24.12	48.83	65.52	69.09	99.1	124.22
Gordon	42.6	84.17	206.14	267.93	369.01	477.22	629.26

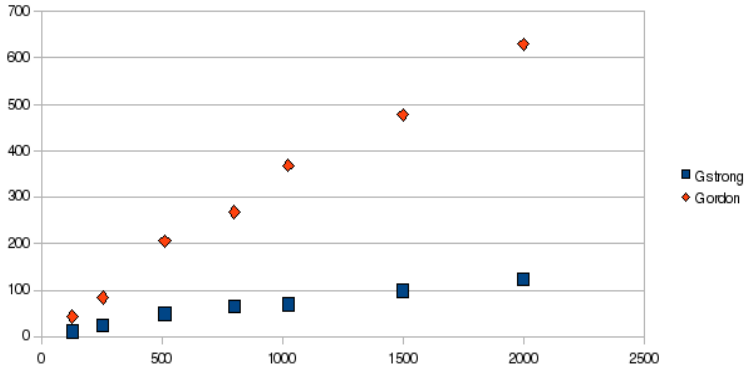

Figure 9: heuristic expected number of calls to the primality oracle T functions of size of n

size of n (bit)	128	256	512	800	1024	1500	2000
GStrong(n)	0.0031	0.012	0.071	0.2570	0.497	1.946	5.43
Gordon	0.0016	0.0073	0.071	0.254	0.62	2.163	6.09

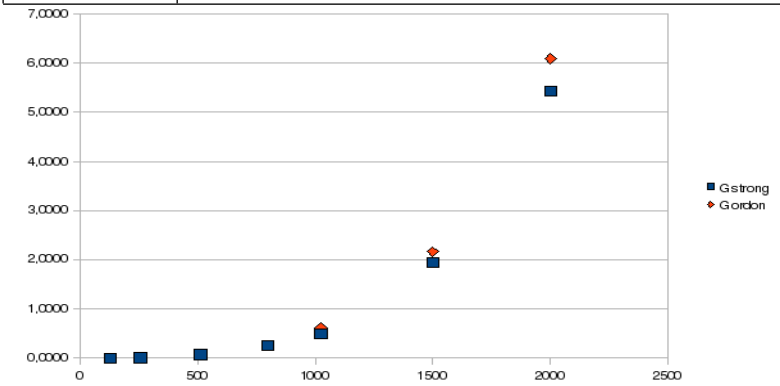

Figure 10: generation's time comparison (in sec.) between GStrong and Gordon

n	128	256	512	800	1024	1500	2000
time for the call of T in GStrong(n)	74 μs	213 μs	1025 μs	3.2 ms	6 ms	17 ms	37 ms
time for the call of T in Gordon	24 μs	66 μs	269 μs	1.3 ms	0.62 ms	3.5 ms	7.6 ms

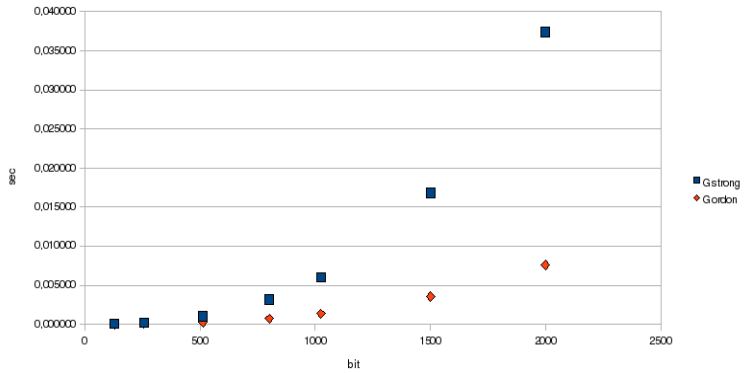

Figure 11: oracle's time comparison between GStrong and Gordon

4. GENERATION OF SSH-RSA KEYS

We describe here how to generate two files *id_rsa* and *id_rsa.pub* in order to use the algorithm described in **section [3]** to generate SSH-RSA keys.

Once p and q are calculated with *GStrong*, of size respectively $\lfloor \frac{n}{2} \rfloor - 1$ and $\lfloor \frac{n}{2} \rfloor + 1$, then we get $n = pq$. We remind here to our readers the RSA algorithm. We fixe e-private key and calculate d-public key so that e is coprime with $\varphi(n) = (p-1)(q-1)$ and $d = e^{-1} \mod \varphi(n)$.

In order to produce the two files needed for ssh, *openssh* also needs $d \mod [p-1]$, $d \mod [q-1]$ and $q^{-1} \mod p$. This can be easily done by modifying the class $IntRSADom$ of the library *Givaro*, you will also need to convert the integers from *Givaro*, this can be done using the function BN_dec2bn.

Then we can create an object of type *RSA* from *openssh*, fullfilling those informations. Eventually we can use the function *PEM_write_RSAPrivateKey(privateKeyFile, key, NULL, NULL, 0, NULL, NULL)* to produce your private key file (usually named *id_rsa*). In order to write the public key, you need to create an object *Key* from openssl of type *KEY_RSA* and with the key in parameter, to finish use the function (from *openssl*) *key_write(maCle, publicKeyFile)* to write the public key file. You will also need to add *ssh-rsa* before the key, and the name of the computer at the end of the file, if you want to put it in the file *authorized_keys* of the remote computer. You can now use the public/private key you generated with your own strong prime numbers.

5. CONCLUSION

The conclusion of our work is that the fast generation of strong primes relies on the generation algorithm **AND** on the primality test called by this algorithm. We have seen in the first part of this paper that a primality test is nothing more than a compromise between the correctness of the answer and the speed of the test. The results of our implementation of this new algorithm to generate strong primes shows that you can't compare the efficiency of two algorithms by comparing their number of calls to the oracle T. Indeed, the time needed by T may be different in function of the composition of the numbers n that we submit to it. This is why we think that to really improves the generation of strong prime, you first have to build a specific oracle T depending of the composition of the numbers that you need to test.

6. REFERENCES

[1] M. Agrawal, N. Kayal, and N. Saxena. Primes is in p. 2002.

[2] E. Bach. Explicit bounds for primality testing and related problems. *Mathematics of Computation*, 55:355–380, 1990.

[3] D. J. Bernstein. Proving primality after agrawal-kayal-saxena. *Mathematics Subject Classification*, 2003.

[4] J. Gordon. Strong primes are easy to find. *Proceedings of EUROCRYPT*, 84:216–223, 1986.

[5] M. Joye, P. Paillier, and S. Vaudenay. Efficient generation of prime numbers. *Lecture Notes in Computer Science*, 1965:340–354, 2000.

[6] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard. The number field sieve. *Proc. 22nd ACM Symposium on theory of Computing*, pages 564–572, 1990.

[7] H. W. Lenstra and Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126:649–673, 1987.

[8] J. M. Pollard. Theorems on factorization and primality testing. *Proceedings Cambridge Philosophical Society*, 76:521–528, 1974.

[9] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12:128–138, 1980.