Replace this file with prentcsmacro.sty for your meeting,
or with entcsmacro.sty for your meeting. Both can be
found at the ENTCS Macro Home Page.

Experimenting with Distributed Generation of RSA Keys

Thierry Congos[1]

SUPELEC
SSIR Group
Rennes, France

François Lesueur[2]

SUPELEC
SSIR Group
Rennes, France

Abstract

Threshold cryptography allows a message to be enciphered through the cooperation of several entities among which no one knows the whole secret key. However, if the key pair is generated using an usual centralized algorithm, then there is an entity which knows the whole secret key. In this paper, we present an implementation of a distributed algorithm to generate RSA keys, previously proposed by Boneh and Franklin. Using this implementation, we analyze how many parties can jointly generate the key and we show that a few dozens of ordinary nodes distributed worldwide can achieve this task.

Keywords: RSA, Threshold Cryptography, Distributed Key Generation

1 Introduction

Public key cryptography is widely used for encryption and authentication to ensure security in distributed protocols. Usually, each entity is provided with two public/secret key pairs. The first key pair is used for signature : the secret key allows a message to be signed and the signature is then verified with the public key. The second key pair is used for encryption : the public key allows messages to be enciphered and messages can only be deciphered with the private key.

With the development of autonomous networks such as ad hoc networks or peer-to-peer networks, there is a renewed interest in threshold cryptography among many trustees. Threshold cryptography allows a private key to be shared between some nodes which can then sign a message only if a number of t of them cooperate.

[1] Email: tcongos@rennes.supelec.fr

[2] Email: francois.lesueur@supelec.fr

©2009 Published by Elsevier Science B. V.

The secret key is never reconstructed to sign messages but nodes create t partial signatures which are then aggregated to obtain the signature with the secret key.

For instance, Kong *et al.* propose in [7] to provide certificates to nodes admitted in a ad hoc network. These certificates are cooperatively established by the neighbors of the nodes. Lesueur *et al.* propose in [8] a distributed certification mechanism adapted to large peer-to-peer networks where certificates are obtained through the collaboration of a large fraction of the nodes. There are numerous other examples of autonomous networks using threshold cryptography.

With threshold cryptography, the initial key generation is critical. If the key pair of the network is initially generated by a centralized entity, then there is a single entity which knows the secret key and users need to trust this entity. In that case, the benefit of a self-healed system is greatly reduced. The key pair should thus be generated using a distributed algorithm, in which case the secret key is never known by anyone. This key pair is originally generated by some founding members. We focus here on the distributed generation of RSA keys since it is the most used signature algorithm.

Boneh and Franklin [3] as well as Algesheimer, Camenisch and Shoup [1] proposed distributed algorithms to generate RSA keys, at the end of which each participant knows a share of d but no one knows d. These two protocols are $\lfloor \frac{k-1}{2} \rfloor$ private, that is no coalition of less than $\lfloor \frac{k-1}{2} \rfloor$ servers on the k servers involved can learn information about secret values. This implies that the more servers there are, the harder it is to learn information about secret values.

However, the complexity increases with the number of participants and previous evaluations from Malkin, Wu and Boneh [9] as well as Wright and Spadling [13] are limited to 5 participants. In larger networks such as ad hoc or peer-to-peer ones, there can be more members to generate the key and it is thus interesting to benefit from a larger initialization in which there is less trust to put on each initial member. In this paper, we evaluate distributed generation using current hardware and our main result is the possibility to distributively generate RSA keys in a worldwide network composed of nearly 40 nodes in a few hours. Since the generated key is a long-lived key which is then used to sign individual certificates, the generation can last a few hours.

In this paper, we use the scheme proposed by Boneh and Franklin which seems simpler and which is also described in previous implementations. We note that the scheme of Algesheimer *et al.* generates N as a product of strong primes, which is sometimes seemed as more secure; in fact, Rivest and Silverman [12] showed that this is not true for the standard non-distributed RSA cryptosystem. Still, strong primes may be needed by some distributed signature schemes.

In Section 2, we present the distributed generation scheme from Boneh and Franklin as well as previous evaluations. In Section 3, we present our implementation and the improvements we made on the original model. In Section 4, we present and analyze the results obtained on both local network and PlanetLab. Finally, we conclude and suggest some future work in Section 5.

2 Related Work

A RSA key pair is usually created as follows. First, the user generates two large random primes p and q and sets the modulus $N = p \times q$. The totient $\varphi(N)$ is $(p-1)(q-1)$. The user chooses a positive integer $e < \varphi(N)$ coprime with $\varphi(N)$ (e can be chosen independently of N and is usually a small prime such as 65535) and determine d such as $d \times e \equiv 1 \pmod{\varphi(N)}$. The public key is (e, N) and the secret key is (d, N).

In a simple additive threshold scheme [4,5], the shares d_i are such that $d = \sum d_i$. In that case, the RSA ciphering of some message m with the secret key, which is $m^d \pmod N$, is also equal to $(\prod m^{d_i} \pmod N) \pmod N$ and each $m^{d_i} \pmod N$ can be calculated separately. In an additive sharing scheme, every share are needed to obtain a signature but redundancy can then be obtained by transforming the scheme to a redundant one [6].

In this section, we briefly describe the distributed key generation algorithm used in our implementation as well as its previous evaluations. We refer the interested reader to [3,9] for more details.

2.1 *Distributed Key Generation Algorithm*

We describe here the distributed RSA key generation algorithm proposed by Boneh and Franklin in [3].

There are k participants labeled $1, \ldots, k$ and the aim is to generate a public key and shares of the private key. The public exponent e is fixed and the algorithm thus generates $N = p \times q$ with p and q prime numbers as well as $d_1, \ldots, d_k$ such as $d = \sum_{i=1}^{k} d_i$. At the end of the algorithm, each participant knows only one share of the secret key: no one knows the whole d. Moreover, each participant is convinced that N is a product of two prime numbers yet no one knows either p or q.

The algorithm is composed of four steps described in the following.

2.1.1 *Step 1: Choose Candidates p_i and q_i*

Each node i randomly chooses p_i and q_i. p is then $\sum_{i=1}^{k} p_i$ and q is $\sum_{i=1}^{k} q_i$, although p and q are never explicitly computed. To maximize the probability that p and q are prime numbers, the generation method of p_i and q_i ensures that there are no small divisors of p and q. In this paper, we extend one of these methods -*distributed sieving*- to handle a larger number of parties.

2.1.2 *Step 2: Compute N*

Parties jointly compute $N = p \times q = \sum_{i=1}^{k} p_i \times \sum_{i=1}^{k} q_i$. The *BGW* protocol, not detailed here, is used to compute N without revealing p and q. This algorithm is $\lfloor \frac{k-1}{2} \rfloor$ private, which means that p and q are not revealed as long as there are less than $\lfloor \frac{k-1}{2} \rfloor$ attackers.

2.1.3 *Step 3: Test for bi-primality*

Parties apply a distributed bi-primality test on N, here derived from Euler's Theorem. At this point, each party i knows its p_i and q_i and $N = pq$ is publicly known.

Parties choose a common random $g \in \mathbb{Z}_N^*$. Party 1 computes $v_1 = g^{N-p_1-q_1+1} \pmod{N}$. Parties $1 < i \leq k$ compute $v_i = g^{p_i+q_i} \pmod{N}$. Then each party broadcasts its v_i to all other and test if $v_1 = \prod_{i=2}^{k} v_i$. This tests whether $g^{\varphi(N)} = g^{N-p-q+1} \equiv 1 \pmod{N}$. If the test fails, N is not a product of two primes and thus parties return to step 1 (choose candidates).

It should be noted that there are some N which are not products of two primes yet pass the test. However, if tested numbers do not have small divisors, which is the case due to method used on step 1, the density of such integers is very small (less than 1 on 10^{40}, as stated in [11,10]).

2.1.4 Step 4: Generate Shares

Finally, parties jointly compute shares of the private key d. Each party i obtains a share d_i such that $d = \sum_{i=1}^{k} d_i$. No party can compute d.

2.2 Previous Evaluations

This scheme has previously been implemented and evaluated independently by Malkin, Wu and Boneh [9] and Wright and Spadling [13] in 1999.

In [9], Malkin *et al.* used up to 5 servers and SSL for communications. They also use multithreading to mitigate latencies induced by synchronization. The computers are 333MHz Pentium II and are either connected on a LAN or on a WAN across the USA. With 3 computers on a LAN, they are able to generate 1024-bit keys in 1 minute and a half and 2048-bit keys in 18 minutes. 5 computers on a LAN can generate a 1024-bit key in 5 and a half minutes. Finally, 3 computers on a WAN across the USA generate a 1024-bit key in 5 and a half minutes.

In [13], Wright and Spadling mostly evaluated the impacts of some parameters in the case of 3 servers. Experiments are performed on a single machine and thus do not provide clues on the scalability of the key generation algorithm.

In this paper, we evaluate this algorithm on a worldwide network using a few dozens of modern computers. We show that for current applications, keys can be generated on such a network. We made our own implementation since we were not able to download one of the previous ones.

3 Implementation

Our implementation is realized in C for Unix systems. We use OpenSSL libraries for SSL communications (usual SSL/TLS library), multi-precision integers and random number generation (Crypto library). A thread abstracts I/O communications in order to allow simple function calls with only server's id and data to send or receive. All parameters and important information like the list of servers IP are grouped in a unique configuration file. Each server must have this configuration file. At the beginning, every node has the list of the public keys of the other nodes.

We present here the three particularities of our implementation. First, we generalized the distributed sieving, used to filter *good* p_i and q_i, for larger networks. We then describe the parallelization and failure tolerance.

3.1 Distributed sieving

In order to decrease the required number of iterations on average, [9] introduce distributed sieving at step 1 of the algorithm. We implemented this protocol to which we make a change to make it effective with a larger number of servers. The idea is to pick up not so random p_i such that $p = \sum p_i$ is not divisible by any small prime. This way is much more efficient than perform a distributed trial division on p without leaking information if the number of servers is not too large.

To do this, [9] recursively uses a distributed protocol named *BGW with additive sharing* (BGWA) proposed in [3], derived from a more general protocol proposed in [2]. For each party i on the k parties, this protocol takes in input a couple $\langle c_i, d_i \rangle$ and an integer M common for all parties. It returns r_i to each party such as

$$\sum_{i=1}^{k} r_i = \left(\sum_{i=1}^{k} c_i \right) \cdot \left(\sum_{i=1}^{k} d_i \right) \pmod{M}$$

However, M must not admit a prime divisor smaller than k. This constraint leads [9] to fix p value modulo $30 = 2 \times 3 \times 5$, which is possible in their experiments because the number of servers is always lower than 7. We propose here a more general technique which works on the same principle.

Let k be the number of servers, n the number of bits we want for p, M_2 the product of all small prime numbers up to a certain limit. The limitation is that M_2 must be smaller than p. Let M_p be the product of all factors of M_2 lower than k. Let M be the product of all factors of M_2 greater or equal to k. So $M_2 = M \times M_p$ and M respects the constraint on the input modulus in *BGWA*.

Step 1: Each party i pick up a random a_i in $[1, M]$ such that a_i is relatively prime to M.

Step 2: Thus $\prod_{i=1}^{k} a_i \bmod M$ is relatively prime to M, but we want an additive sharing among the parties, i.e. $(b_i)_{1 \leq i \leq k}$ such that $\sum_{i=1}^{k} b_i = \prod_{i=1}^{k} a_i \pmod{M}$. To do this, parties recursively call the *BGWA* protocol.

Suppose there is some $1 \leq j \leq k$ and k integers $b_{1,j}, ..., b_{k,j}$ such that $b_{1,j} + \ldots + b_{k,j} = \prod_{i=1}^{j} a_i$. Then Each party i calls BGWA protocol with the couple $\langle b_{i,j}, u_{i,j} \rangle$ and the modulus M with $u_{i,j} = a_{j+1}$ if $i = j + 1$ and $u_{j,i} = 0$ otherwise. BGWA returns value r_i. One can note that $\sum_{i=1}^{k} r_i = (b_{1,j} + \ldots + b_k, j) \cdot (a_{j+1}) = (\prod_{i=1}^{j} a_j) \cdot a_{j+1}$ and so parties set $u_{i,j+1} = r_i$.

For initialization, $j = 1 \quad ; \quad b_{1,1} = a_1 \quad ; \quad \forall 1 < i \leq k \quad b_{i,1} = 0$ fit. After $k - 1$ iterations, parties set $b_i = b_{i,k}$ and so that is the requested additive sharing.

Step 3: Each server chooses $r_i < \frac{2^n}{M}$ and sets $p_i = r_i \cdot M + b_i$. By this way, $\sum p_i = (\sum r_i)M + \sum b_i$ so p and M are relatively primes.

To ensure that p is not divisible by a factor of M_p (that is, a prime lower than k), we must choose good values for r_i. Our strategy consist to choose $r_{i,i \neq 0}$ such that $p_i \bmod M_p = 0$ and r_1 such that p_1 and M_p are relatively prime. In details:

Let $r = r_m \cdot r'_i + M_p \cdot r''_i + r'''_i$ with

$r_m = M^{-1}$ in $\mathbb{Z}_{M_p}$; $r'_i = M_p - (b_i \bmod M_p)$
$r''_i =$ a random integer in $[0, \frac{2^n}{M_2}]$
$r'''_i =$ a random integer in $[1, M_p]$ relatively prime to M_p if $i = 1, 0$ otherwise.

Proof.

Let some $i \neq 1$:
$p_i = (r_m \times r'_i + M_p \times r''_i)M + b_i \pmod{M_p}$
$p_i = r'_i \times r_m \times M + b_i \pmod{M_p}$
$p_i = r'_i + b_i \pmod{M_p}$
$p_i = 0 \pmod{M_p}$

Similarly, for $i = 1$, $p_1 = r'''_i \times M \pmod{M_p}$. Since r'''_i and M_p are relatively prime and M and M_p are also relatively prime, p_1 and M_p are relatively prime.□

3.2 Parallelization

There are very frequent communications of small packets between each pair of servers and so, on a large WAN with significant latency, a lot of time can be wasted. In order to reduce the problem, our implementation is multithreaded. Basically, threads are numbered and each thread on a server runs separately the algorithm with all corresponding threads on other servers. When an instance ends, all others end up and the algorithm finish. On a server, all threads are synchronized in order to reduce the number of I/O operations and send large packets containing the values of all threads. This way, the waste of time caused by latency is divided by the number of threads.

3.3 Failure Tolerance

In order to launch the program on some dozens of nodes with significant risks of individual failures, program must handle failures. When a connection to another server closes abnormally, it tries to re-launch the algorithm from the beginning without the deficient node after a short waiting time.

If one node dies (thus all connections to this node shut down abnormally), all the other nodes re-launch the algorithm with $k-1$ parties. More complicated failures on the network, such as a one to one connection breaking, are not supported yet. Experiments have however shown that this simple approach is sufficient.

Since success probability is constant for each iteration, only the time of the current iteration plus the time to re-launch is lost.

4 Results

In this section, we present our experimental results and we show that key generation can now be used on several dozens of nodes distributed worldwide. We tested our implementation both on a local network and over internet, using the PlanetLab[3] worldwide testbed.

On a local network, we mainly studied the impacts of the multithreading and of the key size. We also evaluated the number of iterations of steps 1 to 3 (generating

[3] http://www.planet-lab.org

p_i, computing N and then testing whether the nearly random N is a product of primes) which represent the main cost of the algorithm. This number only depends on the key size and not on the latency nor on the number of nodes: we use the number of iterations to infer precise results from only a few runs on PlanetLab.

Over internet, we studied the scalability of the key generation algorithm. We were able to generate 1024-bit keys with 37 nodes and we think that we could go even further. The difficulty resides on the unreliability of PlanetLab nodes which are shared among experiments and are often overloaded or disconnected. We think that PlanetLab nodes represent finally a worst-case analysis and that end-user nodes can behave better.

In following measurements, all communications are encrypted with SSL. The sieving bound is automatically fixed to its experimental optimum, that is the largest possible value satisfying $M < p$.

We first estimate the number of iterations needed by the algorithm to generate a 1024-bit key. We then study performance on a local network and we finally propose some results on PlanetLab.

4.1 Number of Iterations

We present in Figure 1 the experimental cumulative distribution function obtained for a 1024-bit modulus. This figure shows the probability of having finished in function of the number of iterations done. For instance, after 1000 iterations of steps 1 to 3, there is a 0.6 probability of having found a N which is a product of two prime numbers. Since the probability of success for each iteration is constant, the number of iterations follows an exponential distribution.

The average number of iterations is 1099 and the median value is 750.

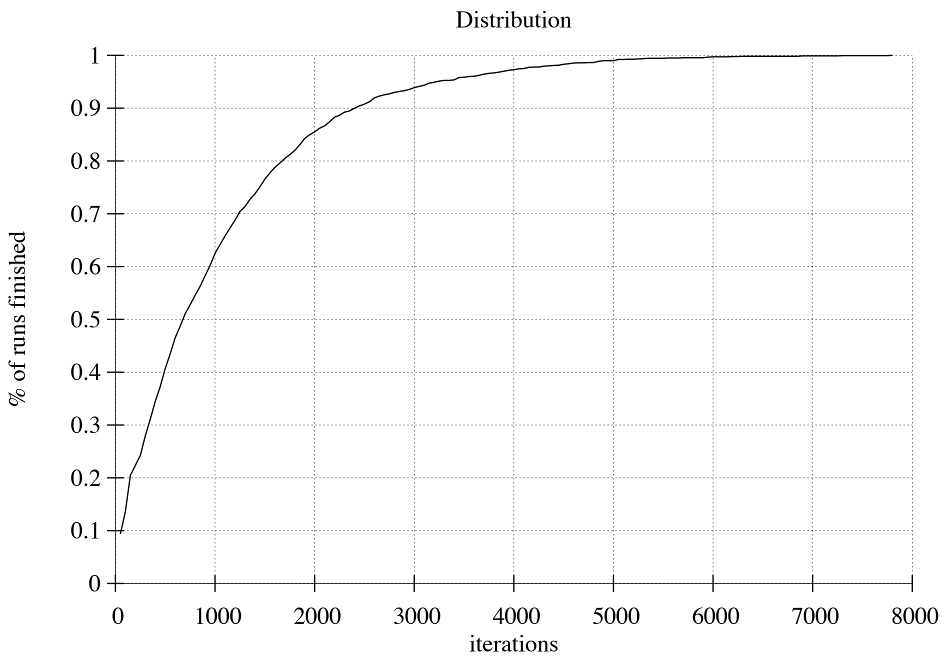

Fig. 1. Cumulative distribution for a 1024-bit modulus

4.2 Analysis on a Local Network

In this subsection, we present results obtained on a local network with a few machines. These machines are modern but not high-end, typically from 2005.

Table 1 presents some performance values in function of the modulus size, using 3 servers. The *number of iterations* is the average number of iterations needed to find N as a product of two primes. The *data sent* is the data sent by each node during the whole algorithm, without the protocol overhead. The *time per iteration* represents the time needed to run a single iteration. The *total time* is the time needed by the whole algorithm.

The generation of a 1024-bit key involves on average 1099 iterations of the algorithm, each node sends 3.1MB, each iteration lasts 0.02 seconds and the whole generation takes 25 seconds.

Since the probability to randomly find p and q simultaneously primes is in $O(\frac{1}{n^2})$, doubling the size of the key means multiplying by 4 the average number of iterations. Furthermore, computations and communications are made with larger numbers and so each iteration is longer.

Modulus size	# iterations	Data sent	Time per iteration	Total time
1024 bits	1099	3.1 MB	0.02s	25s
2048 bits	4213	22.2 MB	0.05s	3 min 43s
4096 bits	16227	166.5 MB	4.82s	56 min 6s

Table 1
Performance in function of the modulus size, using 3 servers on a LAN

Table 2 presents the difference between an execution on a local network and an execution on PlanetLab, both using the same parameters. There are 10 servers using 30 threads each and they generate a 1024-bit modulus.

We can see that the time per iteration goes from 0.07 seconds on a LAN to 2.27 seconds on PlanetLab, mainly because of the latency between the servers. The *normalized total time* corresponds to the time per iteration multiplied by the average number of iterations, calculated previously, which is 1099 for 1024-bit keys.

Network	Time per iteration	Normalized total time
LAN	0.07s	1 min 18 sec
PlanetLab	2.27s	50 min

Table 2
Performances in function of the network (10 servers, 30 threads, 1024 bit modulus)

4.3 PlanetLab runs

Table 3 shows the effect of multi-threading on 3 PlanetLab servers, for a 1024-bits modulus. The aim of multi-threading is to compensate the high latency between nodes. Optimal choice depends on each situation, but in general multithreading makes a huge improvement, especially for networks with significant round trip times. Experimentally, for a few PlanetLab servers, best values are around 100 threads. The normalized total time is the expected total time with the average number of iterations.

# threads	Time per iteration	Normalized total time
11	0.44 s	8 min 3s
50	0.20 s	3 min 40s
100	0.15 s	2 min 45s

Table 3
The effect of multi-threading with 3 servers on PlanetLab

Finally, Table 4 presents measures of a few sample executions on PlanetLab with different number of servers. The increase in number of nodes results in an increased time per iteration. The main origins are the distributed sieving and an increased time of synchronization. Indeed, there are k BGW function calls in distributed sieving, and the BGW algorithm itself depends on k.

The main result is that one can calculate a 1024-bits key on a few dozens of servers in a few hours. One can note that the data sent, which rises in Ok^2, reaches important values. In the experience with 37 servers, the sending bandwidth is 44 kB/s and this can reach the limit of a small connection.

# servers	# threads	Norm. data sent	Time per iteration	Norm. total time
10	30	39 MB	2.72s	50 min
21	100	181 MB	6.44s	118 min
37	300	572 MB	11.79s	215 min

Table 4
Some example runs on PlanetLab with a 1024-bit modulus

5 Conclusion

In this paper, we implemented and evaluated distributed RSA key generation. Our implementation follows the original scheme with the exception of the generalized distributed sieving, necessary to handle a large number of nodes. This implementation can be downloaded at `http://www.rennes.supelec.fr/ren/perso/flesueur/`.

Experiments showed that modern computers can generate a key over internet with a few dozens of parties.

We are now looking forward to generating 2048-bit keys with 50 nodes or 1024-bit keys with 100 nodes. PlanetLab instabilities make such long computations hard to deploy and we are thus thinking about a P2P substrate allowing dynamic membership during the generation.

References

[1] Algesheimer, J., J. Camenisch and V. Shoup, *Efficient computation modulo a shared secret with application to the generation of shared safe-prime products*, Report 2002/029, Cryptology ePrint Archive (2002).
URL http://eprint.iacr.org/2002/029.pdf

[2] Ben-Or, M., S. Goldwasser and A. Wigderson, *Completeness theorems for non-cryptographic fault-tolerant distributed computation*, in: *Proceedings of the 20th Annual Symposium on Theory of Computing (STOC)* (1988), pp. 1–10.

[3] Boneh, D. and M. Franklin, *Efficient generation of shared RSA keys*, in: B. S. Kaliski, Jr., editor, *Advances in Cryptology – CRYPTO ' 97*, Lecture Notes in Computer Science **1294**, International Association for Cryptologic Research (1997), pp. 425–439.
URL http://theory.stanford.edu/~dabo/papers/sharing.ps.gz

[4] Boyd, C., *Digital multisignatures*, in: H. Baker and F.Piper, editors, *Cryptography and Coding* (1989), pp. 241–246.

[5] Frankel, Y., *A practical protocol for large group oriented networks*, in: J.-J. Quisquater and J. Vandewalle, editors, *Proceedings of Workshop on the Theory and Application of Cryptographic Techniques (EUROCRYPT)*, Lecture Notes in Computer Science **434** (1989), pp. 56–61.

[6] Frankel, Y., P. Gemmell, P. D. MacKenzie and M. Yung, *Optimal-resilience proactive public-key cryptosystems*, in: *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science (FOCS)* (1997), pp. 384–393.

[7] Kong, J., P. Zerfos, H. Luo, S. Lu and L. Zhang, *Providing robust and ubiquitous security support for mobile ad hoc networks*, in: *Proceedings of the 9th IEEE International Conference on Network Protocols (ICNP)* (2001), pp. 251–260.
URL http://csdl.computer.org/dl/proceedings/icnp/2001/1429/00/14290251.pdf

[8] Lesueur, F., L. Mé and V. Viet Triem Tong, *A distributed certification system for structured P2P networks*, in: D. Hausheer and J. Schönwälder, editors, *Proceedings of the 2nd International Conference on Autonomous Infrastructure, Management and Security (AIMS)*, Lecture Notes in Computer Science **5127** (2008), pp. 40–52.

[9] Malkin, M., T. Wu and D. Boneh, *Experimenting with shared generation of RSA keys.*, in: *Internet Society's 1999 Symposium on Network and Distributed System Security (SNDSS)* (1999), pp. 43–56.
URL http://theory.stanford.edu/~dabo/papers/ShareExp.ps

[10] Pomerance, C., *On the distribution of pseudoprimes*, Mathematics of Computation (1981), pp. 587–593.

[11] Rivest, R., *Finding Four Million Large Random Primes*, in: *Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, Springer-Verlag London, UK, 1990, pp. 625–626.

[12] Rivest, R. L. and R. D. Silverman, *Are 'strong' primes need for RSA?*, Technical report, RSA Data Security, Inc., pub-RSA:adr (1998).
URL ftp://ftp.rsasecurity.com/pub/ps/sp2.ps;ftp://ftp.rsasecurity.com/pub/pdfs/sp2.pdf

[13] Wright, R. N. and S. Spalding, *Experimental performance of shared RSA modulus generation*, in: *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'99 (Baltimore, Maryland, January 17-19, 1999)*, ACM SIGACT, SIAM (1999), pp. 983–984.