Proceedings of "Algorithms and Experiments" (ALEX98) Trento, Italy, Feb 9–11, 1998
R. Battiti and A. A. Bertossi (Eds.) pp. 34-43

Experimental Performance of Shared RSA Modulus Generation

(Preliminary Report)

Sara Spalding*
Department of Mathematics
University of Indiana
Bloomington, IN 47405, USA
e-mail: sspaldin@indiana.edu

and

Rebecca N. Wright
AT&T Labs – Research
180 Park Avenue
Florham Park, NJ 07932-0971, USA
e-mail: rwright@research.att.com

Abstract

Many distributed protocols require the participants to have secret shares of an RSA modulus in order to perform distributed cryptographic computations. Until recently, a trusted party was required to generate and distribute these secret shares before the start of the protocol. Recently, Boneh and Franklin introduced a protocol whereby participants could themselves generate the secret shares without revealing any information about their shares to each other. We experimentally evaluate the performance of their protocol and we recommend good choices for certain parameters of the protocol.

1 Introduction

Public key cryptography has proven extremely useful for providing security in distributed protocols. The RSA cryptosystem has interesting number-theoretic properties that make it suitable for certain kinds of distributed applications. In particular, many distributed protocols require the participants to have secret "shares" of an RSA modulus in order to perform distributed cryptographic computations. Until recently, a trusted party was required to generate and distributed these secret shares. Recently, however, Boneh and Franklin [2] showed the participants could generate the shares themselves.

Boneh and Franklin's protocol allows a group of parties to determine an RSA modulus $N = pq$, a public encryption exponent e, and a private decryption exponent d, such that p and q are large primes, all parties know N and e, and each party knows only partial information about, or *shares* of, the factors p and q and the private exponent d. More specifically, three or more parties can jointly generate $N = pq$, e, and d such that at the end of the computation, each party has a share of d, none of the parties knows d, none of the parties knows the factors p and q of N, and yet all parties are convinced that the unknown factors p and q are prime and that e and d are inverses modulo $(p-1)(q-1)$. (For more information about RSA, see [15] or [5]).

*Work done while in the AT&T Undergraduate Research Program.

Such a protocol is useful in a number of places, most notably threshold cryptography [7, 6]. In threshold cryptography, the parties can use their shares of d to jointly create a digital signature or to jointly decrypt messages without revealing any information about their own shares. The signature and decryption operations will succeed if and only if a sufficient number of the shareholders cooperate. In addition, simply allowing the parties to generate an RSA modulus without anyone learning the factors is useful for avoiding the need for a trusted party in many protocols [9, 8, 11, 13, 14, 10].

In this work, we concentrate on the generation of the shared RSA modulus and experimentally evaluate its performance. We have implemented a three-party version of the shared modulus generation part of the Boneh/Franklin protocol; ours is the only implementation of the protocol that we are aware of. Our results show that the computational overhead of their protocol is not prohibitive. For a 1024-bit modulus and a "good" choice of certain parameters, our implementation of their protocol takes approximately 10 minutes to generate a shared modulus. Although this is considerably slower than single trusted user generation, which takes an average of 6 seconds, it is fast enough to be useful in some practical settings. In particular, many applications that require shared RSA use it precisely to limit the trust required of any particular party. Furthermore, usually it is only infrequently necessary to generate a new shared key, for example as an initialization step. Thus, it is likely that in many settings this will not be too high a computational price to pay for eliminating the need for a single trusted party. Of course, there will still be some settings in which the cost of the distributed protocol is too high or having a trusted party is not a problem; in these cases the single user solution is more appropriate.

We also investigate optimal values, in terms of reducing the expected amount of computation required, for two parameters of the protocol that determine how many times certain steps of the protocol should be performed, and give some recommendations for these values.

In Section 2, we more fully describe the Boneh/Franklin protocol and state more precisely the experimental questions we wish to address. We describe our implementation and experimental results in Section 3. We conclude in Section 4.

2 The Protocol

We describe here the Boneh/Franklin protocol as it works for modulus generation between two parties, Alice and Bob, plus a helper, Henry, who is required for the two-party version. At the end of the protocol, Alice, Bob, and Henry each know $N = pq$ such that p and q are prime with extremely high probability. In addition, Alice knows p_a and q_a and Bob knows p_b and q_b such that $p = p_a + p_b$ and $q = q_a + q_b$. Other than this information, none of Alice, Bob, and Henry learns p or q. This last property is important for the security of the distributed protocols that use the resulting RSA modulus, and is also what makes this task more difficult than might appear at first glance.

Figure 1 gives an overview of the protocol Alice, Bob, and Henry use to generate an m-bit modulus N. Depending on the way N is to be used, Alice and Bob may then continue with the key generation part of the protocol (see [2]), which we do not describe in this paper. The subprotocols for distributed trial division, distributed computation of N, and distributed primality testing are described subsequently.

Note that in step 3b of the shared RSA modulus generation protocol described in Figure 1, Alice and Bob both know N and it it is already known that p and q are not divisible by the first f primes. Alice checks whether N is divisible by any of the next s primes simply by computing N modulo each prime. Hence, after this step it is known that p and q are not divisible by the first $f + s$ primes. An obvious optimization in step 3b would be to have Alice and Bob split the work of the trial divisions between themselves. Note also that it would clearly be desirable to test p and q themselves directly for primality, rather than only doing trial division against some primes. Unfortunately, it is not known how to do this without revealing p and q.

In the distributed trial division protocol (Figure 2), Alice, who knows a value x_a, and Bob, who knows a value x_b, wish to learn whether $x = x_a + x_b$ is divisible by a prime p without revealing x_a

Shared RSA Modulus Generation:

1. Find candidate $p = p_a + p_b$:
   (a) Alice picks a random $\frac{m}{2}$-bit number p_a.
   (b) Bob picks a random $\frac{m}{2}$-bit number p_b.
   (c) Alice, Bob, and Henry run the distributed trial division protocol to determine if $p_a + p_b$ is divisible by any of the first f primes.

   Step 1 is repeated until no divisors of $p_a + p_b$ are found in step 1c.
2. Find candidate $q = q_a + q_b$:
   (a) Alice picks a random $\frac{m}{2}$-bit number q_a.
   (b) Bob picks a random $\frac{m}{2}$-bit number q_b.
   (c) Alice, Bob, and Henry run the distributed trial division protocol to determine if $q_a + q_b$ is divisible by any of the first f primes.

   Step 2 is repeated until no divisors of $q_a + q_b$ are found in step 2c.
3. Find candidate N:
   (a) Alice, Bob, and Henry run the distributed protocol to compute $N = (p_a + p_b)(q_a + q_b)$.
   (b) Alice does local trial division of N by the next s primes to determine whether p or q are divisible by any of these primes.

   If a divisor is found in step 3b, execution returns to step 1 to find new candidates for p, q, and N.
4. Check whether N is the product of two primes:
   (a) Alice and Bob run the distributed primality test to determine whether N is the product of two primes. If not, execution starts over at step 1.

Figure 1: **Shared RSA Generation Modulus Protocol**

Distributed Trial Division:

- Alice picks a random $c \in \mathbf{Z}_p$ and $d \in \mathbf{Z}_p^*$, and computes $u = c + dx_a \bmod p$. She sends c and d to Bob and sends u to Henry.
- Bob computes $v = c - dx_b \bmod p$ and sends v to Henry.
- Henry sends "divisor" to Alice and Bob if $u = v$, and "not divisor" otherwise.

Figure 2: **Distributed Trial Division Protocol**

or x_b. Henry will help Alice and Bob with the computation, but will not learn x_a or x_b. Note that x is divisible by p if and only if $x_a \equiv -x_b \pmod{p}$.

An optimization to reduce the overall computation spent in distributed trial division would be to work with the product of a few small primes at one time rather than working with each prime individually; this will be explored as a part of our ongoing work. In addition, to reduce communication, multiple primes can be tested in parallel.

The protocol for distributed computation of N (Figure 3) is adapted from the secure function computation protocol of Ben-Or *et al.* [1]. Alice, who knows p_a and q_a, and Bob, who knows p_b and q_b, wish to compute $N = (p_a + p_b)(q_a + q_b)$ without revealing any of p_a, p_b, q_a, q_b, $p_a + p_b$, and $q_a + q_b$. Again, Henry will help with the computation, but also will be prevented from learning these values.

Intuitively, the protocol operates by Alice picking two random lines $P_a(x)$ and $Q_a(x)$ that intersect the y-axis at p_a and q_a, respectively, and a random quadratic $r(0)$ such that $r(0) = 0$. Similarly, Bob picks two random lines $P_b(x)$ and $Q_b(x)$ that intersect the y-axis at p_b and q_b, respectively. Alice and Bob each send Henry the values of their lines and of r evaluated at $x = 1$,

Distributed Computation of N:

- Alice picks random values $c_a, d_a, p_b^a, q_b^a \in \mathbf{Z}_P^*$. She sets $P_a(x) = c_a x + p_a$ and $Q_a(x) = d_a x + q_a$. Alice computes $N_a = (P_a(1) + p_b^a)(Q_a(1) + q_b^a) + r(1)$. She sends $Pa(2)$, $Q_a(2)$, p_b^a, q_b^a, and $r(2)$ to Bob, and sends $P_a(3)$, $Q_a(3)$, $r(3)$, and N_a to Henry.
- Bob computes $c_b = p_b^a - p_b$ and $d_b = q_b^a - q_b$, and sets $P_b(x) = c_b(x) + p_b$ and $Q_b(x) = d_b(x)$. Bob computes $N_b = (P_a(2) + P_b(2))(Q_a(2) + Q_b(2)) + r(2)$, and sends $P_b(3)$, $Q_b(3)$, and N_b to Henry.
- Henry computes $N_h = (P_a(3) + P_b(3))(Q_a(3) + Q_b(3)) + r_3$. Then, Henry interpolates $N_a = \alpha(1)$, $N_b = \alpha(2)$, and $N_h = \alpha(3)$ to obtain $\alpha(0)$, which is equal to N. Henry sends N to Alice and Bob.

Figure 3: **Protocol for Distributed Computation of N**

$x = 2$, and $x = 3$. Henry interpolates the values he has received to compute the quadratic $\alpha(x) = (P_a(x) + P_b(x))(Q_a(x) + Q_b(x)) + r(x)$, and computes $N = \alpha(0)$.

In order to optimize the number of messages that must be sent, the protocol as described in Figure 3 has Alice choose some of the values that should be chosen by Bob. In the protocol, P is a publicly known prime larger than any possible value of N, and all computations are done modulo P.

It is notable that the BGW protocol for general secure distributed computation on which this is based has significant theoretical interest, but has long been considered infeasible in practice. Our results indicate that this is not the case—a single iteration of this version of the BGW protocol takes well under half a second.

Distributed Primality Test:

- Alice computes $v_a = 2^{(N - p_a - q_a + 1)}$ and sends it to Bob.
- Bob computes $v_b = 2^{(-q_a - q_b)}$ and sends it to Alice.
- Alice and Bob check whether $v_a \cdot v_b = 1 \bmod N$. If so, they accept N as the product of two primes. Otherwise, they reject.

Figure 4: **Distributed Primality Test Protocol**

Boneh and Franklin describe two possible distributed "primality" tests [2]. In their case, the computed value N is being checked for whether or not it is of the form $N = pq$ where p and q are both prime. Their tests are probabilistic, in the sense that they may, with provably low probability, give the incorrect answer.

In Figure 4, we describe the distributed primality test that we have implemented. It is a practical optimization based on Fermat's theorem (cf. [5]), suggested by Boneh and Franklin [3]. Although this protocol is not provably correct as above, it is widely accepted as being correct in practice for sufficiently large N. In fact, we checked the resulting $N = pq$ of several thousand runs against a standard (non-distributed) primality test on p and q, and only ever encountered incorrect results when N was very small (16 bits).

In order to fully specify the full shared RSA modulus generation protocol, it is necessary to specify f and s, respectively determining how many distributed trial divisions of p and q in steps 1c and 2c and how many local trial divisions of N in step 3b should be done. Both types of trial divisions attempt to find "good" candidates p and q by weeding out some of the non-primes before carrying out the expensive distributed primality test of N. However, the trial divisions themselves come at some cost.

Note that distributed trial divisions and local trial divisions have a significant difference in effect: distributed trial divisions of p and q ask *separately* for p and q: "Does this appear to be a good p?" and "Does this appear to be a good q?", requiring only the values that do not pass the test to be discarded. In contrast, the local trial division of N is asking: "Does it appear that *both* p and q are good?", requiring both values to be discarded if the test is not passed, even though one of them may in fact be prime. Thus, the distributed trial divisions have a much greater effect than the local trial divisions towards increasing the chances that the distributed primality test will turn out successful.

On the other hand, the computation and communication costs of the distributed trial divisions are also much greater than that of the local trial divisions. To complicated matters further, note that trial division by a prime t will catch $1/t$ of the non-primes. Thus, trial division by a smaller prime has a greater effect and a lesser computation cost than trial division by a larger prime.

Hence, it is not at all clear how many of each kind of trial division, if any, should be done. We show in Section 3 that even a small number of distributed trial divisions give a significant increase in performance over doing none at all, while the number of local trial divisions is less important. Our results suggest that doing a few hundred each of distributed and local trial divisions is optimal for our implementation.

3 Experimental Results

To focus on computation costs (rather than communication costs), we have implemented the protocol described in Section 2 as a sequential, rather than distributed, program. As a result, our running times differ from a distributed implementation in two significant ways: (1) computations that would be done in parallel are done in sequence, making our running times slower; (2) communication costs are not taken into account, making our running times faster. Nevertheless, our results are useful for comparing running times for runs with different parameters. In addition, our results provide bounds on the total time spent on computation for a distributed implementation: a distributed implementation would take no less than 1/3 of the time (since at least one of the three participants must do at least a third of the work) and no more than the total time taken by our sequential implementation (since this is equivalent to one participant doing all the work). Further, since each subprotocol only requires only a small constant number of rounds of communication, communication overhead could easily be estimated as a small multiple of the time to send one message between two parties.

Our implementation is in C++ using the Cryptolib library [12] to perform the modular arithmetic operations on large numbers. In order to mimic the structure of the distributed protocol, Alice, Bob, and Henry are each implemented as C++ classes, with the only communication between Alice, Bob, and Henry occurring via explicit function calls. All experiments were carried out on SPARC 20 machines running SUNOS. All our results are for the generation of a 1024-bit modulus, since this is the current security recommendation for RSA moduli. All running times are given in seconds.

Our experiments run as follows. For a given number f of distributed trial divisions of p and q, and a given number s of local trial divisions of N, we iterate steps 1 through 4 of the shared RSA modulus generation protocol only until step 4 is run once, possibly repeating steps 1, 2, and 3 multiple times before step 4 is reached. We record the running time of the run, along with whether the candidate N that was found was a success or a failure. Using the results from many runs with the same parameters f and s, we can calculate the expected run time m for a single run with those parameters. Similarly, we can calculate the expected probability p of success. Then the number of expected trials until success is $1/p$. Since m and p are independent once f and s are fixed, we can then calculate the expected run time until success for parameters f and s by m/p.

The prime number theorem states that for large n, the probability that a random n-bit number is prime is approximately $1/n \log_e 2$ [5]. For a 1024-bit modulus, p and q are 512-bit numbers, so a random candidate has probability approximately 1/355 probability of being prime. Hence, we would expect that runs with no trial divisions at all, success would be found only around $1/355^2$ of the time, since p and q must both be chosen prime in order for N to pass the distributed primality test. This would suggest that an expected number of approximately $126,000$ runs would be needed for a success to be found. As more trial divisions are done, the success rate will increase. In particular, DeBruijn [4] showed that the probability that a random n-bit number is prime given that is known not to be divisible by primes less than b is approximately $2.57 \log_e b/n$. This says, for example, that if 300 trial divisions of p and q are done, the probability of success is improved to approximately 1/688. Indeed, our results are well within this range. However, trial divisions of N do not have this same rate of improvement. Moreover, the expected running time will depend both on the expected

number of runs until success and on the computation time to carry out the trial divisions. Our results show that the expected running time until success decreases dramatically as distributed trial divisions are added, while local trial divisions are not as significant.

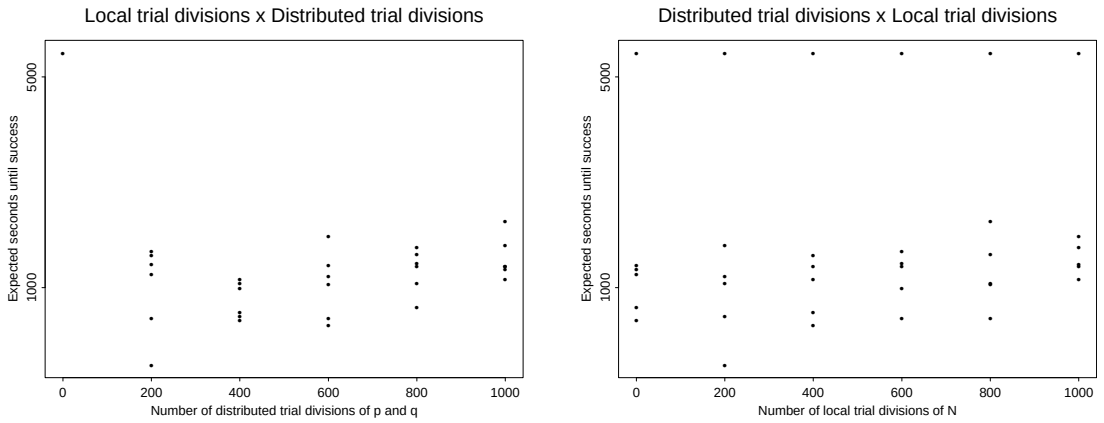

Figure 5: **Expected running times until success.**

Figure 5 shows the expected running times until success, calculated as described above based on 10,000 runs each for numbers of distributed and local trial divisions varying from 0 to 1000 by increments of 200. The graph on the left shows the running times grouped by the number of distributed trial divisions, while the graph on the right shows the running times grouped by the number of local trial divisions. Both use a log scale on the y-axis. The full set of mean running times used to create this graph can be found in Appendix A.1.

In the graphs, the points at 6000 represent ∞, where no successes were found. Note that this occurs only when the number of distributed trial divisions of p and q is 0, and in fact occurs whenever the number of distributed trial divisions is 0, regardless of the number of local trial divisions. This clearly demonstrates the power of the distributed trial divisions of p and q, which can rule out bad p's and q's separately, compared to local trial divisions of N, which require that both p and q are picked to be good at once.

Also notable is that except for when no distributed trial divisions are done, the running times on average increase as more distributed trial divisions are performed. The pattern is less clear with the local trial divisions, but again shows higher running times on average as the number of local trial divisions increases. At their best, the expected running times until success are around 13 minutes.

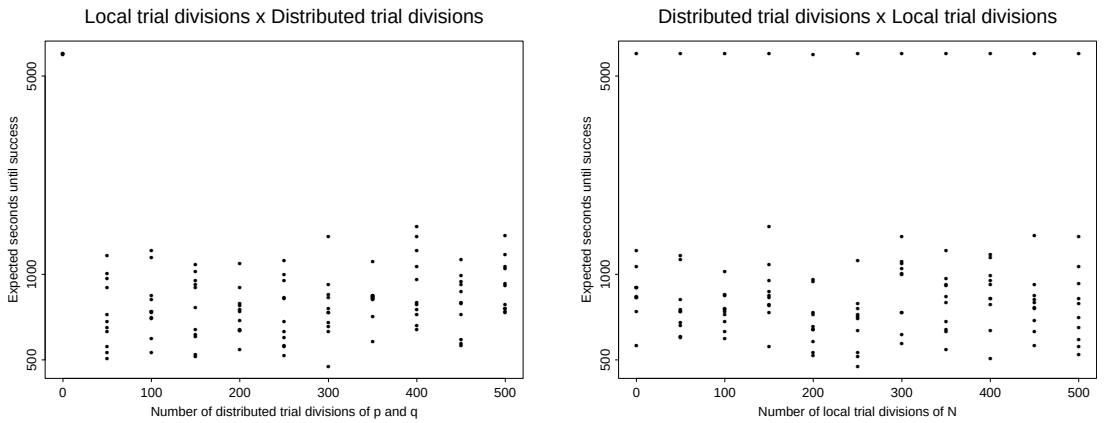

Figure 6: **Expected running times until success**

Figure 6 shows a closer look at smaller values of f and s. The graphs are of the same format as in Figure 5, except that now f and s each vary from 0 to 500 by increments of 50. Again, 10,000 runs were done for each pair of parameters. The full data can be found in Appendix A.2. Again, note that in every case except for one, no successes were found if no distributed trial divisions were done. As before, the running times are lowest when the number of distributed trial divisions is small but nonzero. The number of local trial divisions does not make a significant difference, but it appears best to do 200–300. At their best, expected running times until success are around 10 minutes.

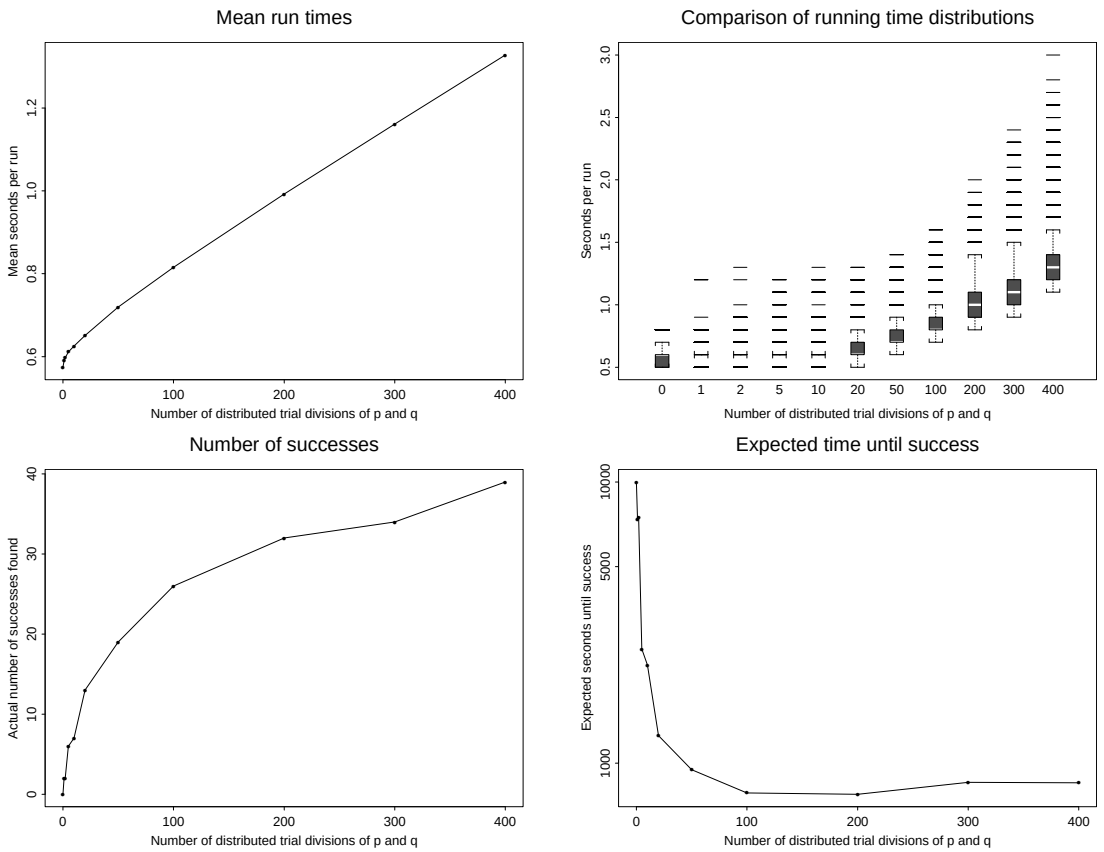

Figure 7: **Distributed trial divisions only**

In Figure 7, we focus entirely on distributed trial divisions. These graphs show the results of doing 25,000 runs with each of 0, 1, 2, 5, 10, 20, 50, 100, 200, 300, and 400 distributed trial divisions, with no local trial divisions. The four graphs show the mean run times for each number of distributed trial divisions; a boxplot comparing the distribution of run times for each value; the number of successes that were found; and the expected run time until a success is found (log scale). In the boxplot, each column shows the running times for all 25,000 runs with a fixed number of distributed trial divisions. The box contains the middle half of the runs, with brackets extending to non-extreme runs; extreme runs are plotted individually[1]. A table showing the aggregated data for these runs can be found in Appendix A.3. Note the sharp increase in the number of successes found as even just a handful of distributed trial divisions are added, and the corresponding sharp drop in expected run time until success. We conclude that 100–200 distributed trial divisions are optimal.

[1] The explanation for the small boxes for small numbers of trial divisions is that the variance is extremely low when few trial divisions are done because candidate p's and q's are much more likely to pass through on the first try. For 1–10 trial divisions, the run times for individual runs are closely clustered around 0.6 seconds, while for 0 they are almost entirely split between 0.5 and 0.6 seconds.

4 Conclusions and Future Work

Our implementation illustrates that the total computation cost of shared RSA is around 10 minutes, well within the realm of practicality for certain applications. Our data shows that even a small number of distributed trial divisions significantly improves the performance of the protocol, with around 100–200 being optimal. Local trial divisions are not as important to the running time of the protocol; 200–400 appears optimal. It is of independent interest that a protocol utilizing the BGW secure distributed computation protocol [1] can run this fast.

As ongoing work, we are devising new experiments applying variance reduction techniques in order to be able to obtain more conclusive data with less computational overhead.

In the future, we plan to investigate the result of combining several primes into one trial division, thereby increasing the effect of a single trial division computation by a constant factor, while only slightly increasing its cost. Another clearly interesting direction for future work would be to use mathematical analysis techniques to characterize the effects of distributed and local trial divisions exhibited here.

Acknowledgements

We thank Matthew Franklin and Dan Boneh for suggesting and encouraging this work. We thank David Johnson for useful discussions about experimental tools and methodology.

References

[1] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computing. In *Proc./ 20th ACM Symposium on the Theory of Computing*, 1988.

[2] D. Boneh and M. Franklin. Efficient generation of shared RSA keys. In *Advances in Cryptology – CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 424–439. Springer-Verlag, 1997.

[3] D. Boneh and M. Franklin. Private conversation, 1997.

[4] N. De Bruijn. On the number of uncanceled elements in the sieve of Eratosthenes. In *Proc./ Neder. Akad. Wetensch.*, volume 53, pages 803–812. 1950.

[5] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 1989.

[6] A. DeSantis, Y. Desmedt, Y. Frankel, and M. Yung. How to share a function securely. In *Proc./ 23rd ACM Symposium on the Theory of Computing*, pages 522–533, May 1994.

[7] Y. Desmedt. Threshold cryptography. *European Transactions on Telecommunications and Related Technologies*, 5(4):35–43, July–August 1994.

[8] U. Feige, A. Fiat, and A. Shamir. Zero-knowledge proofs of identity. *Journal of Cryptology*, 1:77–94, 1988.

[9] A. Fiat and A. Shamir. How to prove yourself: Practical schemes to identification and signature problems. In *Advances in Cryptology – CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–189. Springer-Verlag, 1987.

[10] M. Franklin and S. Haber. Joint encryption and message-efficient secure computation. *Journal of Cryptology*, 1996.

[11] L. Guillou and J. Quisquater. A practical zero-knowledge protocol fitted to security microprocessor minimizing both transmission and memory. In *Advances in Cryptology – EUROCRYPT '88*. Springer-Verlag, 1988.

[12] J. B. Lacy, D. P. Mitchell, and W. M. Schell. Cryptolib: Cryptography in software. In *Proceedings of the 4th USENIX Security Workshop*, pages 208–221, Oct 1993.

[13] K. Ohta and T. Okamoto. A modification of the fiat-shamir scheme. In *Advances in Cryptology – CRYPTO '88*. Springer-Verlag, 1990.

[14] H. Ong and C. Schnorr. Fast signture generation with a fiat shamir-like scheme. In *Advances in Cryptology – EUROCRYPT '90*. Springer-Verlag, 1991.

[15] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatues and public key cryptosystems. *Comm. of the ACM*, 21:120–126, Feb 1978.

A Appendix

A.1

Data for Figure 5 showing expected running times in seconds until an RSA modulus is found, based on 10,000 runs for each combination local and distributed trial divisions varying from 0 to 1,000 by increments of 200.

	# local					
# dist.	0	200	400	600	800	10000
0	∞	∞	∞	∞	∞	∞
200	1103	553	1275	1314	792	1195
400	780	805	824	996	1033	1065
600	1181	1089	751	790	1027	1481
800	860	1030	1172	1199	1287	1354
1000	1150	1378	1063	1180	1662	1174

A.2

Data for Figure 6 showing expected running times in seconds until an RSA modulus is found, based on 10,000 runs for each combination local and distributed trial divisions varying from 0 to 500 by increments of 50.

	# local										
# dist.	0	50	100	150	200	250	300	350	400	450	500
0	∞	∞	∞	∞	5945	∞	∞	∞	∞	∞	∞
50	901	1163	683	556	721	530	1004	968	504	631	648
100	741	740	594	842	533	698	731	1213	1143	818	705
150	903	606	1027	1084	637	516	616	920	949	763	523
200	903	751	738	776	640	635	1091	543	636	689	792
250	829	598	627	952	517	1119	996	683	825	564	559
300	829	675	849	736	653	476	732	628	922	761	1366
350	833	813	845	827	579	712	1109	835	822	840	820
400	1210	662	755	1480	959	722	1356	640	781	796	1066
450	563	1129	721	869	947	790	572	797	995	920	589
500	1065	738	759	783	734	759	1050	916	1177	1375	930

A.3

Data for Figure 7 showing 25,000 runs for each of various numbers of distributed trial divisions and no local trial divisions. Values shown are average run times, number of runs, number of successes found, success rate per thousand runs, expected number of runs until success, and expected time until success.

# dist.	mean seconds	number of runs	number success	success rate/k	exp runs	exp time
0	0.5744	25000	0	0	∞	∞
1	0.5900	25000	2	0.080	12500	7374.5
2	0.5972	25000	2	0.080	12500	7464.8
5	0.6112	25000	6	0.240	4166	2546.5
10	0.6253	25000	7	0.280	3571	2233.1
20	0.6511	25000	13	0.520	1923	1252.1
50	0.7198	25000	19	0.760	1315	947.1
100	0.8153	25000	26	1.040	961	783.9
200	0.9925	25000	32	1.280	781	775.4
300	1.1612	25000	34	1.360	735	853.8
400	1.3282	25000	39	1.560	641	851.4