Experimental Performance of Shared RSA Modulus Generation

Rebecca N. Wright* Sara Spalding†

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. Such "threshold cryptography" protocols [8, 7] are extremely important to many electronic commerce applications because threshold cryptography allows trust and/or control to be distributed among a number of parties rather than centralized. Some important applications include joint authorization and auditing of key usage. Specifically, in threshold cryptography, the parties can use shares of a digital signature key or a decryption key 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. Until recently, for RSA keys, 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 ("BF") 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, two 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 [21] or [6]). In the case of two parties, a third party is also required as a helper, but this party does not need to be trusted with any secrets. The protocol will only produce correct results if all parties follow

*AT&T Labs – Research, 180 Park Avenue, Florham Park, NJ 07932. E-mail: rwright@research.att.com.

†Department of Mathematics, University of Indiana, Bloomington, IN 47405. E-mail: sspaldin@indiana.edu. Work done while in the AT&T Undergraduate Research Program.

In *Algorithmica*, Vol. 33, No. 1, 2002, pp. 88-103.

the protocol properly. An alternate protocol has also been given by Cocks [5]. In more recent work, Poupard and Stern [20] and Gilboa [13] provide two-party solutions that do not require a helper. Frankel, MacKenzie and Yung [11] give a robust solution that works even if some of the parties misbehave. Poupard and Stern's protocol [20] is also robust against the misbehavior of one player. Distributed RSA key generation protocols have a number of applications, most notably threshold cryptography. In addition, simply allowing the parties to generate an RSA modulus $N = pq$ without anyone learning the factors is useful for avoiding the need for a trusted party in many protocols [10, 9, 14, 18, 19, 12], even without a key pair d and e.

In this work, we concentrate on the generation of the shared RSA modulus and experimentally evaluate its performance. We have implemented a two-party version of the shared modulus generation part of the BF protocol. Our results show that the computational overhead of the two-party protocol is not prohibitive. For a 1024-bit modulus and a "good" choice of certain parameters, our implementation of their protocol takes approximately 13 minutes to generate a shared modulus. Although this is considerably slower than single trusted user generation, which takes around 6 seconds on the same machines, 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.

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.

Ours is the first implementation of the protocol that we are aware of; since our initial implementation, a full multiparty version has been implemented by Malkin, Wu, and Boneh [16]. Their implementation uses a new technique of "distributed sieving", resulting in a significant performance improvement, to running times around 3 minutes. Nonetheless, we feel our (earlier) results are still interesting because they provide insight into the interaction between different parts of the protocol.

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

2 The Protocol

We describe here the BF 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 $|p| = |q| = |N|/2$ and 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 any information about 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 address in this paper. The subprotocols for distributed trial division, distributed computation of N, and distributed duoprimality testing are described subsequently.

In step 3b of the shared RSA modulus generation protocol described in Figure 1, Alice and Bob

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 d 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 d 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 ℓ 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 duoprimality 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**

both know N and it is already known that p and q are not divisible by the first d primes. Alice checks whether N is divisible by any of the next ℓ 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 $d + \ell$ primes. An obvious optimization in step 3b would be to have Alice and Bob split the work of the trial divisions between themselves. Note 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.

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**

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 or x_b. Henry helps Alice and Bob with the computation, but also does 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. An optimization to reduce communication would be to test multiple primes in parallel. We do not consider either of these optimizations.

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**

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 $x = 1$, $x = 2$, 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 would be chosen by Bob in the above description; this does not affect the security of the protocol. 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 necessarily the case—a single iteration of this version of the BGW protocol takes well under half a second.

Distributed Duoprimality Test:

- Alice computes $v_a = 2^{(N - p_a - q_a + 1)}$ and sends it to Bob.
- Bob computes $v_b = 2^{(-p_b - q_b)}$ 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 Duoprimality Test Protocol**

Boneh and Franklin describe two possible distributed duoprimality tests [2] to test whether or not the computed value N is of the form $N = pq$ where p and q are both prime. The tests are probabilistic, in the sense that they may, with provably low probability, give the incorrect answer. In Figure 4, we describe the distributed duoprimality test that we have implemented. It is a practical optimization based on Fermat's theorem (cf. [6]), suggested by Boneh and Franklin [3]. Although this protocol is not provably correct as the two mentioned above are, it is widely accepted as being correct in practice for sufficiently large N. In fact, we checked the resulting $N = pq$ of several hundred thousand runs against a standard (probabilistic, non-distributed) primality test on p and q, and only ever encountered contradictory 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 d and ℓ, 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 duoprimality test of N. However, the trial divisions themselves come at some cost.

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 duoprimality 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.

The prime number theorem states that for large n, the probability that a random n-bit number is prime is approximately $1/n \log_e 2$ [6]. For a 1024-bit modulus, p and q are 512-bit numbers, so a random candidate p or q has probability approximately $1/355$ probability of being prime. Hence, we would expect that runs with no trial divisions at all, a successful N 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 duoprimality test. Hence, an expected number of approximately $126,000$ runs would be needed for a successful N to be found. As more trial divisions are done, the success rate of the duoprimality test increases. 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 (distributed and/or local) trial divisions are done before testing N for duoprimality, the probability of success of N is improved to approximately $1/688$. Indeed, our experimental results are well within this range, as is later shown in Figure 7. However, the expected running time depends both on the expected number of runs until success and on the computation time to carry out the distributed and/or local trial divisions. Furthermore, an unsuccessful local trial division has the unfortunate side effect that both p and q must be discarded, even though most likely only one is actually a multiple of the prime being tested.

Hence, it is not at all clear how many of each kind of trial division, if any, should be done. We address this question experimentally, and we show in Section 3 that

- even a small number of distributed trial divisions yields a significant increase in performance over doing none at all
- the number of local trial divisions is less important to the performance of the protocol

- around 50 distributed and 1000 local trial divisions is optimal for our implementation, resulting in running times of around 13 minutes.

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 by keeping track of the number of subprotocol iterations and multiplying this by an estimate of the time to send the appropriate, fixed, number of messages between two parties.

Our implementation is in C^{++} using the Cryptolib library [15] 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. The experiments were carried out on 200 MHz Pentium Pro computers. All our results are for the generation of a 1024-bit modulus, the current security recommendation for RSA moduli. All running times are given in seconds.

In these experiments, the protocol is broken into three stages, each run and measured separately. The performance results of the stages can then be combined to provide performance results for the whole protocol. This method, called splitting, has been described by C. McGeoch [17], and provides two advantages over simply running the entire protocol. First, it reduces the variance, thereby reducing the number of runs needed to get conclusive results and allowing additional experimentation time to be spent only in the stages where it is needed. Second, it provides a better understanding of where the time in the protocol is being spent.

Specifically, for each pair of parameters d and ℓ, we separately run distributed trial division (steps 1 and 2 of Figure 1), local trial division (step 3), and duoprimality testing (step 4). For each single iteration of any of these three stages, we record the results (e.g. p_a, p_b), whether the result was a successful candidate, and the computation time. Using the results from many runs of each stage with the same parameters d and ℓ, we calculate the expected run time m for a single run of each stage with those parameters. Similarly, we 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 d and ℓ are fixed, we can then calculate the expected run time until success for parameters d and ℓ by m/p. We combine the results from each stage to obtain results for the entire protocol. The details of the computation and the full data for all graphs in this section can be found in Appendix A.

Our results show that the expected running time until success decreases dramatically as a small number of distributed trial divisions are added, and then increase if additional distributed trial divisions are added, with around 50 being optimal. Local trial divisions are not as significant to the overall running time of the protocol, with around 1000 being optimal.

Performance results for the distributed trial division stage are shown in Figure 5, for $d =$

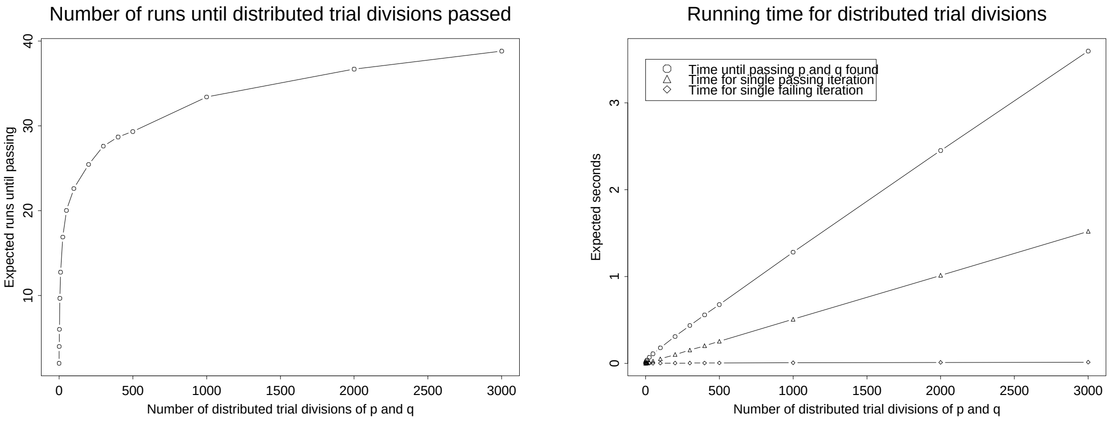

Figure 5: **Distributed trial division stage**

$0, 1, 2, 5, 10, 25, 50, 100, 200, 300, 400, 500, 1000, 2000$ and 3000, based on 150,000 distributed trial division runs for each value of d. The graph on the left shows the expected number of times new candidates p_a, p_b, q_a, and q_b will need to be picked until both $p = p_a + p_b$ and $q = q_a + q_b$ do not have any of the first d primes as a factor.[1] Clearly, the larger d is, the more candidates will be required until p and q are found that are not a multiple of any of the first d primes. The graph on the right shows the expected time until a passing p and q are found, as well as the expected time for a single passing run and a single failing run. Note that the failing runs take much less time than the passing runs. This is because a run terminates as soon as any divisor is found, so will stop half of the time after dividing by 2, one sixth of the time after dividing by 3, etc., only rarely requiring many trial divisions before failure. In contrast, a successful run always tries all d primes, so takes longer as d grows. To account for the difference between the time for a successful run and a failed run, the expected time until a passing p (or q) is found is calculated as $t_p(r-1)+t_f$, where t_p is the average time for a passing run, t_f is the average time for a failing run, and r is the expected number of runs until success (exactly $r-1$ of which are failing runs by definition); hence, the expected time until a passing p and a passing q are found is $2(t_p(r-1)+t_f)$.

Performance results for the local trial division stage are shown in Figure 6 for $\ell = 0, 250, 500, 750$ and 1000, and $d = 0, 1, 2, 5, 10, 25, 50, 100, 200, 300, 400, 500, 1000, 2000$ and 3000. For each pair ℓ, d, 150,000 distributed trial division runs were done, and those that were successful were passed on to the local trial division stage.

The top two graphs in Figure 6 show the expected number of local trial division runs until an $N = pq$ is found that does not have as a factor any of the primes from the $(d+1)$st prime to the $(d+\ell)$th prime (i.e. the next ℓ primes). The left graph is broken out by the value of ℓ and the right graph is broken out by the value of d. These graphs show that the expected number of local trial divisions until success rises slightly as ℓ increases and drops dramatically as d increases. The former phenomenon is intuitive, since it would be expected to take more trials to find an N passing more trial divisions. The latter phenomenon can be explained by noting that if d is small, then a local trial division is looking for small, and therefore likely, factors, while if d is large, then a local trial division will only fail if the candidate N has specific "obscure" factors, such as 4969, 4973, 4987.... In particular, this graph clearly shows that the effect of local trial divisions can not be considered independently of the number of distributed trial divisions.

[1] The distributed trial divisions are done independently for p and q. We show the results here for p and q summed together to allow better comparison with the other stages, which work with p and q together.

The middle two graphs in Figure 6 show the expected running time for a single trial division stage. On the left, the running time is broken out by the value of ℓ and shown for both passing runs and failing runs, with one point for each ℓ, d pair. As with distributed trial divisions, a passing run takes longer than a failing run, since the passing run always has to try each of the ℓ divisors, while the failing run stops as soon as any divisor is found. The spread of the expected failing run times for each value of ℓ can be explained more clearly by looking at the graph on the right, breaking it out by ℓ and d, which shows the rate of increase of the expected failing run time as either ℓ or d increases. Again, for increasing ℓ, this is because a "pass" will not be declared until more divisions have been done, while for increasing d, this is because the primes being checked in local divisions are less likely to be factors.

The lower two graphs in Figure 6 show the total expected time spent in both local and distributed trial division stages until an $N = pq$ passing both trial division stages is found. Here, we see that again the computation time increases as ℓ grows, while values of d between 10 and 200 yield the lowest computation times. The right graph reflects the fact that the computation time spent in distributed trial divisions is more if d is large, while the computation time spent in local trial divisions is more if d is small. In particular, for higher values of d, the time required to complete both trial division stages increases because of the higher cost of the distributed trial division computation. On the other hand, for lower values of d, the computation time is high because, as seen in the top right graph, the number of local trial divisions required until passing is very high in this case. Note, however, that just a small increase in the number of distributed trial divisions is enough to make a dramatic difference. Of course, N's that have passed more divisions are more likely to pass the final duoprimality test, so it is not clear a priori whether it might be worthwhile to perform the additional computation of more trial divisions. However, we will see that this is not the case.

We next consider the duoprimality test stage. Here, it is possible to analytically determine the expected number of iterations of the duoprimality test before a successful N is found using the DeBruijn estimate [4]. As a partial validation of our experimental results, we compare in Figure 7 the expected number of candidate N's entering the duoprimality test stage before a successful N is found as based on our experiments and as predicted by DeBruijn, for a total number $t = d + \ell$ of trial divisions. As the table shows, the two are quite close.

In Figure 8, we determine which values of ℓ and d result in the best overall performance of the protocol. The left graph considers the total number $t = d + \ell$ of trial divisions for $d = 0, 1, 2, 5, 10, 25, 50, 100, 200, 300, 400, 500, 1000, 2000$ and 3000 and $\ell = 0, 250, 500, 750, 1000, 2000$ and 3000. For each $\ell > 0$, the performance curve starts out high, then lowers to a minimum, and then slowly grows again, with the minimum in each curve occurring when $d = 50$. Since only the total number of trial divisions affects the probability that the result will pass the duoprimality test, this implies that whatever the best total number t of trial divisions is, it is optimal to take $d = 50$ and $\ell = t - 50$. Hence, it remains only to determine which value of ℓ is optimal when $d = 50$. The right graph shows the expected running time until a modulus is generated taking $d = 50$ and $\ell = 0$, 250, 500, 750, 1000, 1250, 1500, 2000, and 3000, and shows it is optimal to take ℓ around 1000. With these parameters, expected running time for the protocol is around 13 minutes.

4 Conclusions

Our implementation illustrates that the total computation cost of shared RSA modulus generation can be as low as 13 minutes, well within the realm of practicality for many high-security applications, especially considering that new modulus generation may only be needed infrequently. Our data

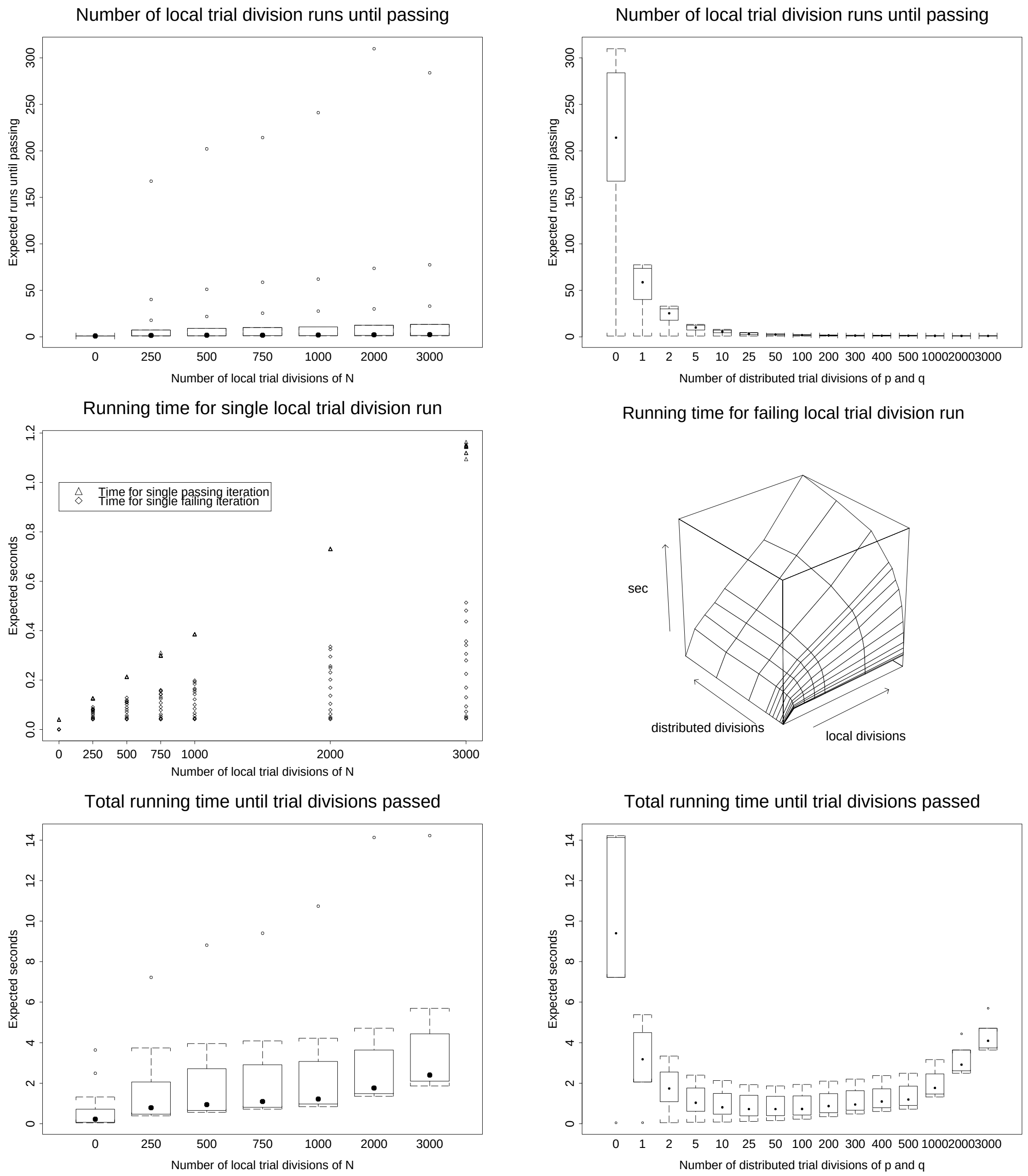

Figure 6: **Local trial division stage**

t	Our results	DeBruijn
50	1587	1336
300	699	688
550	559	577
800	498	522
1050	450	486
1300	427	466
1550	457	445
2050	441	414
3050	388	379

Figure 7: Expected number of candidate N's before success

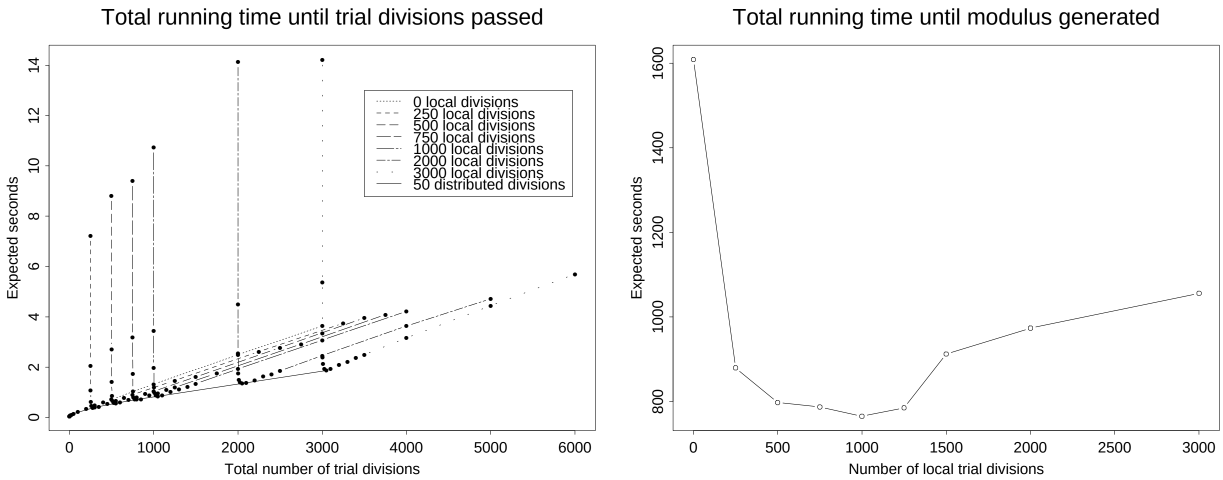

Figure 8: **Duoprimality test**

shows that even a small number of distributed trial divisions significantly improves the performance of the protocol, with around 50 distributed divisions being optimal. Local trial divisions do not have as significant an effect on the running time of the protocol; around 1000 appears optimal. It is of independent interest that a protocol utilizing the BGW secure distributed computation protocol [1] can run this fast.

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. The participants of ALEX98 provided many useful comments and suggestions on an earlier version of this work. We thank Doug Blewett for providing several computers on short notice to support these experiments.

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] C. Cocks. Split knowledge generation of RSA parameters. In *Proc. IMA Conference on Cryptography and Coding*, December 1997.

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

[7] 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.

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

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

[10] 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.

[11] Y. Frankel, P. MacKenzie, and M. Yung. Robust efficient distributed RSA key generation. In *Proc. 30th ACM Symposium on the Theory of Computing*, pages 663–672, May 1998.

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

[13] N. Gilboa. Two party RSA key generation. In *Advances in Cryptology—CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 116–129. Springer-Verlag, 1999.

[14] 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.

[15] 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.

[16] M. Malkin, T. Wu, and D. Boneh. Experimenting with shared generation of RSA keys. In *Proceedings of the Internet Society's 1999 Symposium on Network and Distributed System Security (SNDSS)*, 1999.

[17] C. C. McGeoch. Analyzing algorithms by simulation: variance reduction techniques and simulation speedups. *Computing Surveys*, June 1992.

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

[19] H. Ong and C. Schnorr. Fast signature generation with a Fiat-Shamir-like scheme. In *Advances in Cryptology – EUROCRYPT '90.* Springer-Verlag, 1991.

[20] G. Poupard and J. Stern. Generation of shared RSA keys by two parties. In *Advances in Cryptology—AISACRYPT '98*, volume 1514 of *Lecture Notes in Computer Science*, pages 11–24. Springer-Verlag, 1998.

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

A Appendix

A.1

Data for Figure 5, Figure 6, and the left graph of Figure 8 showing showing 150,000 runs of the distributed trial division stage for each pair d, ℓ. (Figure 5 uses only the values for $\ell = 0$). Values shown are d, ℓ, the mean recorded running time in seconds for a passing distributed trial division runs (pq.pass), the mean recorded running time in seconds for a failing distributed trial division runs (pq.fail), the recorded number of passing distributed trial division runs (#pq.pass), the computed expected number of distributed trial division runs until passing distributed trial divisions for p or q (pq.exp, computed as 150000/#pq.pass), the mean recorded running time in seconds for a passing local trial division runs (Nl.pass), the mean recorded running time in seconds for a failing local trial division runs (Nl.fail), the recorded number of passing distributed trial division runs (#Nl.pass), and the computed expected number of local trial divisions runs until an N is found passing local trial divisions (Nl.exp, computed as (#pq.pass/2)/#Nl.pass). A value of N/A indicates that there were no runs of that type (e.g. N/A for pq.fail when $d = 0$ is because every distributed trial division run passes in this case.)

d	ℓ	pq.pass	pq.fail	#pq.pass	pq.exp	Nl.pass	Nl.fail	#Nl.pass	Nl.exp
0	0	0.001130	N/A	150000	1.00	0.039531	N/A	75000	1.00
1	0	0.001823	0.001824	75314	1.99	0.039467	N/A	37657	1.00
2	0	0.002561	0.002005	49980	3.00	0.039673	N/A	24990	1.00
5	0	0.004251	0.002247	31065	4.83	0.040259	N/A	15532	1.00
10	0	0.006884	0.002439	23553	6.37	0.039431	N/A	11776	1.00
25	0	0.014687	0.002787	17779	8.44	0.039464	N/A	8889	1.00
50	0	0.027452	0.003113	14983	10.01	0.039745	N/A	7491	1.00
100	0	0.052861	0.003628	13271	11.30	0.039480	N/A	6635	1.00
200	0	0.103480	0.004420	11790	12.72	0.039630	N/A	5895	1.00
300	0	0.154102	0.005025	10871	13.80	0.039760	N/A	5435	1.00
400	0	0.204660	0.005628	10462	14.34	0.039698	N/A	5231	1.00
500	0	0.255265	0.006133	10229	14.66	0.038467	N/A	5114	1.00
1000	0	0.509137	0.008350	8983	16.70	0.039081	N/A	4491	1.00
2000	0	1.014330	0.012171	8178	18.34	0.039882	N/A	4089	1.00
3000	0	1.520200	0.015051	7732	19.40	0.040043	N/A	3866	1.00

d	ℓ	pq.pass	pq.fail	#pq.pass	pq.exp	Nl.pass	Nl.fail	#Nl.pass	Nl.exp
0	250	0.001134	N/A	150000	1.00	0.124070	0.040378	448	167.41
1	250	0.001831	0.001828	74799	2.01	0.126596	0.041710	930	40.21
2	250	0.002563	0.002007	50521	2.97	0.125961	0.043020	1411	17.90
5	250	0.004244	0.002243	31285	4.79	0.126038	0.046944	2141	7.31
10	250	0.006888	0.002451	24258	6.18	0.126305	0.050627	2758	4.40
25	250	0.014637	0.002777	18319	8.19	0.125751	0.058609	3662	2.50
50	250	0.027475	0.003105	15553	9.64	0.125458	0.064244	4026	1.93
100	250	0.052853	0.003608	13144	11.41	0.126475	0.069722	4401	1.49
200	250	0.103418	0.004365	11777	12.74	0.125043	0.073562	4621	1.27
300	250	0.154031	0.005006	10989	13.65	0.125841	0.077798	4621	1.19
400	250	0.206886	0.005729	10522	14.26	0.126721	0.078507	4613	1.14
500	250	0.255504	0.006186	9951	15.07	0.125689	0.080078	4426	1.12
1000	250	0.508263	0.008351	9197	16.31	0.126178	0.083301	4361	1.05
2000	250	1.014260	0.011883	8225	18.24	0.125324	0.080710	4013	1.02
3000	250	1.520190	0.015192	8026	18.69	0.125987	0.090067	3954	1.01

d	ℓ	pq.pass	pq.fail	#pq.pass	pq.exp	Nl.pass	Nl.fail	#Nl.pass	Nl.exp
0	500	0.001130	N/A	150000	1.00	0.213908	0.040492	371	202.16
1	500	0.001835	0.001833	74628	2.01	0.212094	0.042273	729	51.19
2	500	0.002567	0.002015	50085	2.99	0.211935	0.043980	1142	21.93
5	500	0.004245	0.002239	31132	4.82	0.211896	0.050171	1714	9.08
10	500	0.006882	0.002444	23802	6.30	0.212452	0.056446	2324	5.12
25	500	0.014637	0.002779	18021	8.32	0.213348	0.070205	2943	3.06
50	500	0.027514	0.003124	15251	9.84	0.212193	0.078854	3340	2.28
100	500	0.052868	0.003636	12893	11.63	0.212071	0.088387	3701	1.74
200	500	0.103423	0.004449	11540	13.00	0.213131	0.100863	4017	1.44
300	500	0.154040	0.005057	11013	13.62	0.212384	0.108881	4227	1.30
400	500	0.206923	0.005810	10605	14.14	0.212762	0.108995	4277	1.24
500	500	0.255283	0.006147	10129	14.81	0.212658	0.111072	4276	1.18
1000	500	0.508279	0.008339	9119	16.45	0.213263	0.117578	4175	1.09
2000	500	1.014270	0.011874	8229	18.23	0.211804	0.117182	3915	1.05
3000	500	1.520200	0.015953	7705	19.47	0.213230	0.127967	3748	1.03

d	ℓ	pq.pass	pq.fail	#pq.pass	pq.exp	Nl.pass	Nl.fail	#Nl.pass	Nl.exp
0	750	0.001134	N/A	150000	1.00	0.299022	0.040408	350	214.29
1	750	0.001831	0.001834	74824	2.00	0.300213	0.042400	637	58.73
2	750	0.002566	0.002009	50245	2.99	0.297896	0.045303	989	25.40
5	750	0.004252	0.002238	31333	4.79	0.299087	0.052813	1560	10.04
10	750	0.006883	0.002449	23803	6.30	0.298799	0.060739	2106	5.65
25	750	0.014635	0.002768	18086	8.29	0.298599	0.077714	2671	3.39
50	750	0.027557	0.003140	15474	9.69	0.298582	0.090896	3040	2.55
100	750	0.052885	0.003633	13253	11.32	0.299315	0.107642	3590	1.85
200	750	0.103412	0.004460	11668	12.86	0.298120	0.124480	3680	1.59
300	750	0.154029	0.005020	11188	13.41	0.299169	0.130823	4065	1.38
400	750	0.206131	0.005774	10658	14.07	0.310338	0.141978	4075	1.31
500	750	0.255269	0.006131	10213	14.69	0.299187	0.143529	4044	1.26
1000	750	0.508275	0.008337	9035	16.60	0.299435	0.154124	4005	1.13
2000	750	1.014200	0.011817	8318	18.03	0.299613	0.158002	3881	1.07
3000	750	1.520130	0.015384	7775	19.29	0.298926	0.158742	3719	1.05

d	ℓ	pq.pass	pq.fail	#pq.pass	pq.exp	Nl.pass	Nl.fail	#Nl.pass	Nl.exp
0	1000	0.001135	N/A	150000	1.00	0.384496	0.040834	311	241.16
1	1000	0.001822	0.001823	75290	1.99	0.385075	0.042813	607	62.02
2	1000	0.002563	0.002006	50071	3.00	0.384881	0.045705	902	27.75
5	1000	0.004254	0.002244	31561	4.75	0.385272	0.055330	1462	10.79
10	1000	0.006886	0.002450	23598	6.36	0.385497	0.066055	1911	6.17
25	1000	0.014618	0.002771	17848	8.40	0.386194	0.083447	2419	3.69
50	1000	0.027475	0.003132	15347	9.77	0.385285	0.099981	2927	2.62
100	1000	0.052861	0.003634	13290	11.29	0.385340	0.121456	3289	2.02
200	1000	0.105110	0.004513	11820	12.69	0.386377	0.142284	3533	1.67
300	1000	0.154087	0.005044	10879	13.79	0.385376	0.152729	3615	1.50
400	1000	0.204650	0.005582	10558	14.21	0.385375	0.160226	3833	1.38
500	1000	0.255286	0.006163	10078	14.88	0.384677	0.165326	3807	1.32
1000	1000	0.508322	0.008358	9070	16.54	0.385425	0.183899	3852	1.18
2000	1000	1.014250	0.012342	8318	18.03	0.385198	0.192276	3824	1.09
3000	1000	1.522190	0.015718	7960	18.84	0.385202	0.197472	3742	1.06

d	ℓ	pq.pass	pq.fail	#pq.pass	pq.exp	Nl.pass	Nl.fail	#Nl.pass	Nl.exp
0	2000	0.001135	N/A	150000	1.00	0.729924	0.041109	242	309.92
1	2000	0.001815	0.001818	75246	1.99	0.731617	0.044393	510	73.77
2	2000	0.002567	0.002007	49879	3.01	0.731287	0.048891	830	30.05
5	2000	0.004252	0.002243	31395	4.78	0.730965	0.062139	1264	12.42
10	2000	0.006879	0.002444	23729	6.32	0.730831	0.078674	1667	7.12
25	2000	0.014621	0.002767	18084	8.29	0.730252	0.103612	2021	4.47
50	2000	0.027497	0.003140	15339	9.78	0.731161	0.136422	2518	3.05
100	2000	0.052853	0.003639	13322	11.26	0.730246	0.168727	2859	2.33
200	2000	0.105172	0.004569	12197	12.30	0.731317	0.201307	3302	1.85
300	2000	0.154104	0.005006	10894	13.77	0.730011	0.230525	3224	1.69
400	2000	0.204704	0.005655	10356	14.48	0.731213	0.249041	3378	1.53
500	2000	0.255263	0.006240	10139	14.79	0.730184	0.256144	3442	1.47
1000	2000	0.508256	0.008296	9182	16.34	0.731260	0.294840	3566	1.29
2000	2000	1.014230	0.012122	8482	17.68	0.731134	0.334577	3636	1.17
3000	2000	1.520140	0.015225	7878	19.04	0.731673	0.324184	3572	1.10

d	ℓ	pq.pass	pq.fail	#pq.pass	pq.exp	Nl.pass	Nl.fail	#Nl.pass	Nl.exp
0	3000	0.001207	N/A	150000	1.00	1.163580	0.043738	201	284.05
1	3000	0.001968	0.001945	74917	2.00	1.153640	0.047271	483	77.55
2	3000	0.002759	0.002166	50426	2.97	1.147150	0.053593	761	33.13
5	3000	0.004539	0.002417	31215	4.81	1.149690	0.071778	1172	13.32
10	3000	0.007283	0.002635	23810	6.30	1.150050	0.092654	1496	7.96
25	3000	0.014733	0.002846	18064	8.30	1.094870	0.129584	1885	4.79
50	3000	0.028028	0.003281	15394	9.74	1.120030	0.169315	2394	3.22
100	3000	0.054030	0.003784	13493	11.12	1.119390	0.224759	2655	2.54
200	3000	0.107545	0.004615	11546	12.99	1.119580	0.279234	2781	2.08
300	3000	0.161969	0.005302	10865	13.81	1.145820	0.305585	3051	1.78
400	3000	0.216400	0.005872	10354	14.49	1.145660	0.340981	3059	1.69
500	3000	0.273195	0.006400	10048	14.93	1.146680	0.356506	3200	1.57
1000	3000	0.545107	0.008952	9161	16.37	1.145340	0.436768	3373	1.36
2000	3000	1.087160	0.012847	8324	18.02	1.146570	0.481129	3418	1.22
3000	3000	1.627590	0.016500	7829	19.16	1.144480	0.513535	3378	1.16

A.2

Data for the right graph of Figure 8 showing the outcome of 100,000 duoprimality tests. For computed values, data used is from Appendix A.1 with $\ell = 50$, plus the following additional values:

d	ℓ	pq.pass	pq.fail	#pq.pass	pq.exp	Nl.pass	Nl.fail	#Nl.pass	Nl.exp
50	1250	0.027463	0.003131	15383	9.75	0.472811	0.112529	2789	2.76
50	1500	0.027456	0.003128	15295	9.81	0.558381	0.124385	2560	2.99

Values shown are the mean recorded running time in seconds for a passing duoprimality test (Nd.pass), the mean recorded running time in seconds for a failing duoprimality test (Nd.fail), the recorded number of passing duoprimality tests (#Nd.pass), the computed expected number of duoprimality tests until a passing modulus N is found (Nd.exp, computed as 100000/#Nd.pass), the computed expected total amount of seconds spent until distributed trial divisions are passed for p and q (pq.time, computed as 2 $\times$ (pq.pass + (pq.exp - 1) $\times$ pq.fail)), the computed expected total amount of seconds spent until local trial divisions are passed for N (Nl.time, computed as (Nl.exp - 1) $\times$ (Nl.fail + pq.time) + (Nl.pass + pq.time)), and the computed expected total amount of seconds spent until the duoprimality test is passed (Nd.time, computed as (Nd.exp - 1) $\times$ (Nd.fail

+ Nl.time) + (Nd.pass + Nl.time)). Note that Nd.time is the expected running time for the entire protocol to execute.

d	ℓ	Nd.pass	Nd.fail	#Nd.pass	Nd.exp	pq.time	Nl.time	Nd.time
50	0	0.863951	0.862625	63	1587.30	0.1110003	0.1507453	1608.5239
50	250	0.862094	0.862705	143	699.30	0.1086044	0.3948114	879.3806
50	500	0.862128	0.862641	179	558.66	0.1102603	0.5645196	797.2971
50	750	0.864119	0.862681	201	497.51	0.1096872	0.7191732	786.9897
50	1000	0.864548	0.862692	222	450.45	0.1098853	0.8351537	764.7964
50	1250	0.862204	0.862780	234	427.35	0.1097185	0.9736851	784.8128
50	1500	0.862307	0.862664	219	456.62	0.1100274	1.1348890	912.1223
50	2000	0.863383	0.862680	227	440.53	0.1101324	1.3467299	973.3121
50	3000	0.861087	0.862584	258	387.60	0.1134079	1.8610827	1055.6917