Experimental Performance of Shared RSA Modulus Generation

Rebecca N. Wright* Sara Spalding†

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 [1] showed how the participants could generate the shares themselves.

Using Boneh and Franklin's protocol ("BF"), two or more parties can jointly generate an RSA modulus $N = pq$, a public encryption exponent e, and a private decryption exponent 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 $(p-1)(q-1)$. 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.

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 takes approximately 13 minutes to generate a shared modulus. Our main contribution is to determine 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 are performed.

Following is an overview the protocol that Alice, Bob, and a helper, Henry, who is required for the two-party version, 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 [1]), which we do not address in this paper. At the end of the protocol, Alice, Bob, and Henry each know an m-bit number $N = pq$ 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 $q = q_a + q_b$. 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.

First, to find a candidate $p = p_a + p_b$, Alice picks a random $\frac{m}{2}$-bit number p_a and Bob picks a random $\frac{m}{2}$-bit number p_b. Then, 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, without revealing p_a or p_b to each other. This step is repeated with new random numbers until no divisors of p are found. Similarly, a candidate $q = q_a + q_b$ that is not divisible by any of the first d primes is found. Second, to find a candidate N, Alice, Bob, and Henry run the *distributed computation of N protocol* to compute $N = (p_a + p_b)(q_a + q_b)$ without revealing p, q, p_a, p_b, q_a, or q_b. Third, 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, N, p, and q are discarded, and the protocol starts over. Finally, when an N passes the local trial division step, Alice and Bob run the *distributed duoprimality test* to determine whether N is the product of two primes, again without revealing any of p, q, p_a, p_b, q_a, or q_b. If N is found to be the product of two primes, the protocol terminates; otherwise, p, q, and N are discarded and the protocol starts over. The subprotocols for distributed trial division, distributed computation of N, and distributed duoprimality testing are omitted due to space requirements.

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 and how many local trial divisions of N are to be done. Both types of 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. Note that distributed

*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.

divisions and local divisions have a significant difference in effect: distributed divisions 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 divisions of N ask: "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. However, distributed divisions have greater communication and computation costs than local divisions.

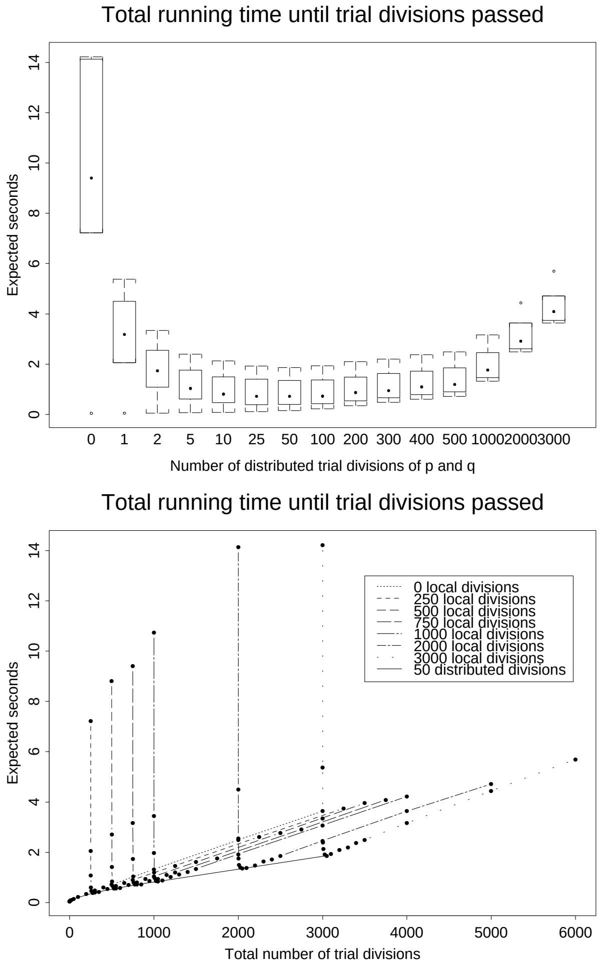

Figure 1: **Trial division performance**

We have implemented the BF protocol in C++ as a sequential, rather than distributed, program. 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. Our experiments were carried out on 200 MHz Pentium Pro computers using a 1024-bit modulus, the current security recommendation for RSA moduli. All running times are given in seconds.

Figure 1 contains two graphs showing the expected running time until a single $N = pq$ passing both the distributed and the local divisions is found, 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. The upper graph shows that even a small increase in the number of distributed divisions substantially decreases the running time, with the lowest running times occurring around $d = 50$. Note however that the resulting N's have a higher probability of passing the duoprimality test if more trial divisions have been done. The lower graph considers the total number $t = d + \ell$ of trial divisions. 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 $d = 50$. Figure 2 gives the expected running time until a modulus is generated taking $d = 50$ and $\ell = 0$, 250, 500, 750, 1000, 2000, and 3000, and shows it is optimal to take ℓ around 1000.

d	ℓ	seconds	d	ℓ	seconds
50	0	1609	50	1000	765
50	250	879	50	2000	973
50	500	797	50	3000	1056
50	750	787			

Figure 2: **Modulus generation performance**

In summary, our experimental results show that even a small number of distributed trial divisions yield 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 around 50 distributed and 1000 local trial divisions is optimal for our implementation. At their best, running times are around 13 minutes.

The full version of this paper is available from `http://www.research.att.com/~rwright/`.

References

[1] 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.