Experimenting with Shared Generation of RSA keys[*]

Michael Malkin
mikeym@stanford.edu

Thomas Wu
tjw@cs.stanford.edu

Dan Boneh[†]
dabo@cs.stanford.edu

Computer Science Department,
Stanford University,
Stanford, CA, 94305-9045

Abstract

We describe an implementation of a distributed algorithm to generate a shared RSA key. At the end of the computation, an RSA modulus $N = pq$ is publicly known. All servers involved in the computation are convinced that N is a product of two large primes, however none of them know the factorization of N. In addition, a public encryption exponent is publicly known and each server holds a share of the private exponent. Such a sharing of an RSA key has many applications and can be used to secure sensitive private keys. Previously, the only known method to generate a shared RSA key was through a trusted dealer. Our implementation demonstrates the effectiveness of shared RSA key generation, eliminating the need for a trusted dealer.

1 Introduction

To protect an RSA private key, one may break it into a number of pieces (shares) and store each piece at a separate location. Sensitive private keys, such as Certification Authority (CA) keys, can be protected in this way. Fortunately, for the RSA cryptosystem it is possible to share a private key among several sites without having to reconstruct the key in order to use it – a CA can issue certificates without reconstructing its private key at a single location [8]. To explain how this is done we briefly recall the details of the RSA system. Let $N = pq$ be a product of two large primes. Let e be a public signature verification exponent and d be the corresponding private signing exponent. A private RSA key is the pair $\langle d, N\rangle$ (other information such as p and q is often included for efficient signature generation). To share its private key among four sites, the CA does the following: it picks four random integers $d_1, \dots, d_4$ in the range $[-N, \dots, N]$ such that $d_1 + d_2 + d_3 + d_4 = d$. Each of these shares is then stored in a separate location. To issue a certificate, the CA creates a signature on a message m by asking each site to compute $s_i = m^{d_i} \bmod N$. Each site then sends s_i to the entity requesting the certificate. The requesting entity multiplies the four s_i's to obtain $s = m^{d_1+d_2+d_3+d_4} = m^d \bmod N$. Thus, a valid signature is obtained. The beauty of this simple procedure is that the private key d is never reconstructed at a single location. If a hacker penetrates even three of the four sites, she learns no information about the private key. To achieve fault-tolerance the scheme may be slightly modified so that any 3-out-of-4 sites can issue a certificate. Indeed, in SET [14] the private key belonging to the root CA is shared in this fashion among four entities [5]. Shared keys are also used in the Omega (Ω) public key management system developed at AT&T [11].

An important issue left out of the above discussion is key generation. Who generates the modulus N and the private key d? A simple solution is to ask a *trusted dealer* to generate two primes p and q, multiply them to get N and then generate e and d. Finally, the dealer splits d into four pieces and sends one piece to each of the sites. Unfortunately, a trusted dealer introduces a *single point of attack*: the dealer, or anyone who compromises the dealer, has the private key and can issue false certificates. Recently, Boneh and Franklin [4] showed how three (or more) servers can generate a shared RSA key *without* a trusted dealer. They describe an efficient distributed algorithm that enables a number of sites to jointly generate a shared key so that none of them know the private key d or the factorization of N. Once the key is generated it can be used for distributed certificate generation as described above.

[*]Project URL: `http://www.stanford.edu/~dabo/ITTC`
[†]Supported by DARPA contract #F30602-97-C-0326.

Distributed generation of an RSA modulus with an unknown factorization is useful in other contexts as well. For instance, in the Fiat-Shamir authentication scheme [7] a number of entities may use a common modulus $N = pq$ for authentication, but none of them should know the factorization of N. The original Fiat-Shamir scheme calls for a trusted dealer to generate N. The Boneh-Franklin algorithm eliminates the need for a trusted dealer in this context as well.

In this paper we study the practicality of distributed generation of shared RSA keys. In order to achieve optimal performance we implemented a number of practical optimizations that greatly reduce the algorithm's running time while still (provably) preserving security. The paper describes our implementation, including the optimizations we used, and gives detailed timing measurements on the algorithm's performance. Our results demonstrate the effectiveness of shared RSA key generation. We hope these results will reduce the reliance on trusted dealers for generating shared keys.

In our experiments we used up to five servers to generate the shared key. Communication between the servers is protected using SSL as implemented in the SSLeay package by Eric Young [17]. To store the shared RSA key on disk we designed an appropriate ASN.1 structure and store the key in PEM format. Once the shared key is generated we perform a number of tests to verify proper sharing. We describe the implementation in detail in Section 3.

Detailed timing measurements are given in Section 5. Here we briefly summarize the results. Our optimized implementation takes under 91 seconds on average to generate a 1024 bit key among three servers using 333MHz Pentium machines on a 10Mbps Ethernet. During the execution of the algorithm each server sends approximately 1.2Mb of data across the network. As an experiment we also ran our distributed computation across the continent by using servers located at various places in the US. We were able to generate a 1024 bit shared RSA key in under 6 minutes. Such a configuration may be used when the shares of a shared RSA key must be stored in remote locations. The optimizations we used to achieve this performance are described in Section 4.

The algorithm can be easily made robust against an attacker who is able to corrupt one of the servers during key generation. The attacker can disrupt key generation (and consequently be detected). However, if a shared key is generated, the attacker will not know the factorization of N or the private key d. We discuss this issue in Section 6.

2 Overview

Before describing our implementation and the practical optimizations we briefly describe the algorithm for generating shared RSA keys. The algorithm is somewhat complex and here we only give a high-level description. For a detailed explanation along with proofs of security see [4].

The goal is to enable k servers to generate a modulus $N = pq$ and exponents e and d. At the end of the computation all servers should be convinced that N is the product of two primes, however none of them should know the factorization. Furthermore, e should be public while d should be shared among the servers in a way that enables t-out-of-k threshold signature generation. That is, any t servers should be able to issue a certificate (without reconstructing the key d). At the same time, an attacker who penetrates at most $t-1$ servers should not be able to obtain any information about the private key. Our key generation algorithm proceeds in a number of steps. At a high level these steps are as follows:

(1) pick candidates: Each of the k servers picks two random n-bit integers p_i, q_i and keeps them secret.

(2) compute N: Using a private distributed computation the k servers compute

$$N = (p_1 + \ldots + p_k) \cdot (q_1 + \ldots + q_k)$$

Other than the value of N, this step reveals no information about the secret values $p_1, \ldots, p_k$ and $q_1, \ldots, q_k$. Now that N is public, the k servers apply trial division to ensure that N is not divisible by small primes.

(3) primality test: The k servers engage in a private distributed computation to test that N is the product of two primes. If the test fails, the protocol is restarted from step 1. As before, this step reveals no information about the private shares.

(4) key generation: Given a public encryption exponent e, the servers use a private distributed computation to generate a *shared* secret decryption exponent d.

Before describing how each of the steps is implemented we explain the security features achieved by the protocol.

Collusion The algorithm is $\lfloor \frac{k-1}{2} \rfloor$ private. That is, even if $\lfloor \frac{k-1}{2} \rfloor$ servers share the information they learn during the protocol, they will still not be able to recover the factorization of N or the private key d. Hence, when three or four servers are involved, no single server has any information. When five servers are used, no pair learns anything.

Honest parties In this section we assume all servers are honestly following the protocol. This model is often called *honest but curious parties* — curious parties learn nothing from the protocol. In Section 6 we explain how the protocol can be made robust against malicious participants. Techniques for achieving stronger robustness properties are described in [9].

Private and authenticated channels The connection between server i and server j must be secure. Otherwise, an adversary can tap all communication and expose critical information. Our system implements private channel using the SSL protocol as described in Section 3.

2.1 Distributed computation of N

For completeness, the next three subsections describe steps (2) – (4) of the algorithm. The reader may skip to Section 3 and refer back these subsections as necessary.

We begin by describing the private computation of N (Step 2). Each server has a secret p_i, q_i. They wish to make the product $N = (\sum p_i)(\sum q_i)$ public without revealing any information about their private shares beyond what is revealed by the knowledge of N. Our technique is a practical adaptation of a generic secure circuit evaluation protocol due to BenOr, Goldwasser and Wigderson (BGW) [1]. From here on, let $P > N$ be some prime. Unless otherwise stated, all arithmetic operations are done modulo P. The protocol works as follows:

Step 1: Let $l = \lfloor\frac{k-1}{2}\rfloor$. For all $i = 1, \ldots, k$ server i picks two random degree l polynomials $f_i, g_i \in \mathbb{Z}_P[x]$ satisfying $f_i(0) = p_i$ and $g_i(0) = q_i$. In other words, the constant term of f_i, g_i are set to p_i, q_i and all other coefficients are chosen at random. In addition, each server i picks a random degree $2l$ polynomial $h_i \in \mathbb{Z}_P[x]$ satisfying $h_i(0) = 0$.

Step 2: For all $i = 1, \ldots, k$ server i computes

$$\forall j = 1, \ldots, k: \qquad \begin{aligned} p_{i,j} &= f_i(j) \\ q_{i,j} &= g_i(j) \\ h_{i,j} &= h_i(j) \end{aligned}$$

Server i then privately sends the triple $\langle p_{i,j}, q_{i,j}, h_{i,j}\rangle$ to server j for all $j \neq i$. Note that the $p_{i,j}$ for $j = 1, \ldots, k$ are standard l-out-of-k Shamir secret sharings of p_i. The same holds for q_i.

Step 3: At this point, each server i has all of $\langle p_{j,i}, q_{j,i}, h_{j,i}\rangle$ for $j = 1, \ldots, k$. Server i computes:

$$N_i = \left(\sum_{j=1}^{k} p_{j,i}\right) \cdot \left(\sum_{j=1}^{k} q_{j,i}\right) + \sum_{j=1}^{k} h_{j,i} \pmod P$$

Server i broadcasts N_i to all other servers.

Step 4: At this point each server j has all values N_i for $i = 1, \ldots, k$. Let $\alpha(x)$ be the polynomial

$$\alpha(x) = (\sum_j f_j(x))(\sum_j g_j(x)) + \sum_j h_j(x) \pmod P$$

Observe that $\alpha(i) = N_i$ and by definition of f_i, g_i and h_i we have $\alpha(0) = N$. Furthermore, $\alpha(x)$ is a polynomial of degree $2l$. We note that l is defined so that $k \geq 2l+1$. Since all servers have at least $2l+1$ points on $\alpha(x)$ they can interpolate it and discover its coefficients. Finally, each server evaluates $\alpha(0)$ and obtains $N \bmod P$. Since $N < P$ the servers learn the correct value of N.

From the description of the protocol it is clear that all servers learn the value N. The protocol requires that at least three servers be involved, in which case linear polynomials are used and the protocol is 1-private. The following lemma states that a coalition of $\lfloor\frac{k-1}{2}\rfloor$ servers learns no other information about the private shares. This statement holds in the information theoretic sense – no complexity assumptions are needed.

Lemma 2.1 *Given N, any coalition of $\lfloor\frac{k-1}{2}\rfloor$ servers can simulate the transcript of the protocol. Consequently, the protocol is $\lfloor\frac{k-1}{2}\rfloor$ private.*

2.1.1 Sharing the final outcome

In some cases (as in Section 4.1) we wish to have the servers evaluate the function $N = (\sum p_i)(\sum q_i)$; however the result should be additively shared among the servers rather than become publicly available. That is, at the end of the computation each server should have an M_i such that

$$\sum_{i=1}^{k} M_i = (\sum_{i=1}^{k} p_i)(\sum_{i=1}^{k} q_i) \pmod P$$

and no information is revealed about the private shares or the final result.

The modification to BGW in order to achieve the above goal is immediate. The servers do not perform Step 4 of the protocol and do not perform the broadcast described at the end of Step 3. Consequently, they each end up with a point on a polynomial $\alpha(x)$ of degree $2l$ that evaluates to N at $x = 0$. Using Lagrange interpolation we know that

$$N = \alpha(0) = \sum_{i=1}^{k} \lambda_i(0) N_i \bmod P$$

where $\lambda_i(x) = \prod_{j \neq i}(x - j)/(i - j)$ is the appropriate Lagrange coefficient. Therefore, rather than broadcast N_i at the end of Step 3, server i simply sets $M_i = \lambda_i(0)N_i$. The resulting M_i's are an additive sharing of N as required.

2.2 Distributed primality test

We describe a simplified version of the distributed primality test (Step 3). Server i has two secret n-bit integers p_i, q_i. At this point, all servers know N where $N = pq = (\sum p_i)(\sum q_i)$. They wish to determine if N is the product of two primes without revealing any information about the factors of N.

Distributed primality test:

Step 1: The servers pick a random $g \in \mathbb{Z}_N^*$. The value g is known to all k servers.

Step 2: Server 1 computes $v_1 = g^{N-p_1-q_1+1} \bmod N$. All other servers compute $v_i = g^{p_i+q_i} \bmod N$. The servers exchange the v_i values with each other and verify that

$$v_1 = \prod_{i=2}^{k} v_i \pmod{N}$$

If the test fails then the servers declare that N is not a product of two primes. Otherwise, they declare success.

We refer to the above test as a Fermat test for testing that a number is a product of two primes. Essentially, what is being tested is that

$$g^{N-p-q+1} = 1 \pmod{N}$$

Note that there exist N that are *not* a product of two primes yet they always pass the test. The density of such integers is extremely small (less than 1 in 10^{40}) [12]. A full probabilistic primality test (that only admits integers that are a product of two primes) was designed by Boneh and Franklin [4]. Once an integer N that passes the Fermat test above is found, the full Boneh-Franklin test can be applied to ensure that the number is indeed a product of two primes. Currently our implementation only tests the number using the simple Fermat test protocol above.

When N is the product of two distinct primes, the primality test protocol reveals no information about the private shares of the participants.

2.3 Shared generation of public and private keys

Once the servers successfully construct an RSA modulus $N = pq = (\sum p_i)(\sum q_i)$ they wish to compute *shares* of $d = e^{-1} \bmod \phi(N)$ for a given encryption exponent e (Step 4). At the end of the computation each server should have a d_i such that $d = \sum d_i$. There are two approaches for doing so. The first approach works for small e (say $e < 2^{20}$) but is very efficient requiring very little communication between the servers. The second works for any e and is still efficient, however it requires more communication. Since signature generation usually makes use of a small RSA public exponent (so that signature verification is fast) we chose to implement the first approach as described below.

Step 1: Server 1 locally computes $\phi_1 = N - p_1 - q_1 + 1$. All other servers locally compute $\phi_i = -p_i - q_i$. Observe that $\phi(N) = \sum \phi_i$.

Step 2: The servers jointly determine the value of $\ell = \phi(N) \bmod e$. Since $\ell = \sum \phi_i \bmod e$, it is possible to compute ℓ without revealing any other information about the private shares. To do so we use a simple protocol due to Benaloh [3] which is $k-1$ private: each server i creates an additive sharing of ϕ_i, namely $\phi_i = \sum_j \gamma_{i,j} \bmod e$ for random $\gamma_{i,j}$. It then sends $\gamma_{i,j}$ to server j. Server j now has $\gamma_{i,j}$ for all i. It computes the sum $\alpha_j = \sum_i \gamma_{i,j} \bmod e$ and sends α_j to all other servers. Then each server locally computes $\sum_j \alpha_j$ which satisfies $\sum \alpha_j = \sum_j \phi_j = \ell \bmod e$.

Step 3: Let $\zeta = \ell^{-1} \bmod e$. Then, it is not difficult to see that $d = (-\zeta \cdot \phi + 1)/e$. Each party i locally computes:

$$d_i = \left\lfloor \frac{-\zeta \cdot \phi_i}{e} \right\rfloor$$

As a result we have $d = \sum d_i + r \bmod \phi(N)$ where $0 \le r < k$.

Step 4: The above sharing of d enables shared decryption [8] using the equality $c^d \equiv c^r \prod c^{d_i} \mod N$. Server 1 determines the value of r by trying all possible values of $0 \le r \le k$ during a trial decryption. It then subtracts r from its own share d_1.

The above approach leaks $\ell = \phi(N) \bmod e$ and r. This is a total of $\log_2 e + \log_2 k$ bits. As a result this approach only works for small e. In our implementation we use $e = 65537$, a standard small public exponent. We emphasize that an alternate approach works for all e and doesn't leak any information (see [4]). We chose not to implement it since it is more costly.

2.4 t-out-of-k sharing

The previous subsection explains how we obtain a k-out-of-k sharing of d. However, to provide fault-tolerance it is often desirable to have a t-out-of-k sharing enabling any subset of t servers to apply the private key. We explain how to achieve 2-out-of-3 sharing of the RSA key. This approach generalizes to any t-out-of-k as long as k is not too big (e.g. $k < 20$). Standard Shamir secret sharing [15] is inadequate since the private key would have to be reconstructed at a single location in order for it to be used.

We first explain the structure of a 2-out-of-3 RSA signature generation scheme. Write the private key d as $d = d_1 + d_2 = d_3 + d_4$ where d_1, d_2, d_3, d_4 are random integers in $[-N, N]$. Each of the three server is given shares according to the following table:

S1	S2	S3
d_1	d_2	d_1
d_3		d_4

Observe that any pair of servers can generate a signature (without having to reconstruct the key). No single server has any information about d. The private share of the key given to each server is composed of N and a list of d_i's. As a result, in the implementation we define a new ASN.1 private key structure, as described in Section 3.2. We note that an alternate approach to a t-out-of-k sharing of an RSA key is described in [13].

We can now explain how a t-out-of-k sharing of d as above can be generated by the servers themselves (without a trusted dealer). Once the servers generate a k-out-of-k sharing (as explained in the previous section) they can easily convert it into a t-out-of-k sharing as follows: each server i constructs a t-out-of-k sharing (as the above table) of its own share d_i. It then sends to server j the shares of d_i that belong to j. Finally, each server adds up all the shares it received from the other servers. The resulting d_i's are a t-out-of-k sharing of d.

3 Implementation details

Our implementation consists of two independent components. The first is a communications package (COM) that abstracts low level communications. It provides encrypted links between servers as well as a convenient interface for sending abstract data types, such as large numbers, over the network. The second component is the algorithm module (GEN) that implements the key generation algorithm. Our code is written in C for high performance and easy integration into existing products.

3.1 Communications package

The clients and servers use a communications package based on SSL to ensure the authenticity and confidentiality of connections. The communications package handles the following tasks internally:

- Tunneling communications using an underlying secure transport, like SSL.
- Providing an intuitive, platform-independent interface for reading and writing abstract data types, like large integers.
- Managing a large number of simultaneous network connections and presenting a simplified networking API for higher-level code.
- Handling transparent end-to-end authentication using techniques such as certificates and sequence numbers.
- Maximizing efficiency by taking advantage of buffering and nonblocking I/O.

In short, the package ensures that the underlying protocol security assumptions mentioned earlier are met while abstracting away the complexities of asynchronous networking and any optimizations we implement at the communications level.

A configuration file, which is read upon program startup, contains network settings for the clients and servers, such as IP addresses, port numbers, and pathnames. Instead of requiring each program to know all the details of each server, the API allows servers to be referenced directly by number. The communication package takes care of the mappings between server number and address/port information. In addition, it handles peer identification transparently, so that a server knows the identity of any clients that contact it.

3.1.1 Authentication

As stated earlier, the communications package is based on Eric Young's SSLeay [17] package. Client and server certificates are issued by a private CA, whose public key is distributed by hand to all clients and servers beforehand. Each party has its own certificate, signed by this CA, which contains its own identity as part of the signed certificate data. For example, the certificate for server 0 has a subject field that looks like:

```
/C=US/ST=California/O=Stanford University/
    OU=ITTC Project/CN=[SERVER 0]
```

The name field contains the authenticated identity of the party, which can be verified by clients and other servers.

At the moment our approach to authentication is ad-hoc. After all, the method of authentication depends on the environment in which our system is used. For instance, when generating a CA private key, authentication can be done using certificates generated by a higher level CA. For a root CA authentication can be done using the current CA root key. In other environments where shared keys are used authentication can be done using standard certificates issued by a CA.

In addition to certificate authentication, the communications package keeps track of sequence numbers for each pair of parties. This allows the system to detect when a private key is compromised and used, since this would introduce a skew between the sequence numbers held by the servers and the legitimate client.

3.1.2 Multiparty I/O

There are many instances where conventional network programming can lead to considerable implementation inefficiencies, even under normal usage. Consider an application of threshold decryption when one of the servers has failed. One of the benefits of threshold decryption is the ability to tolerate the loss of one or more shares. However, if one naively attempts to connect to each server in series, the procedure will stall when the non-functioning server is reached.

Instead, the communications package uses *non-blocking* I/O underneath to alleviate this problem. In this mode, when the communications package makes an I/O request, it tells the operating system to return immediately instead of waiting until the operation can be completed. This is useful when communicating with multiple parties, because the application can open multiple connections and have the communications package deliver packets on several of the connections without waiting for acknowledgments. This reduces the amount of time spent waiting on the network to its theoretical minimum.

The core of the communications package is a state machine that tracks the status of each connection. This state machine handles the initial connection establishment and negotiates the initial authentication handshakes transparently to the application. This approach is needed because multiple connections, in different stages, may be in progress simultaneously. All data is buffered internally and delivered to the application as complete and well-formed packets.

3.2 Key storage

The SSLeay package supports reading and writing both public and private keys in PEM format. Our private shares and shared public keys are represented internally as extensions of the standard RSA key data structure[1]. On disk, we support a PEM-encoded ASN.1 format similar to that used for RSA keys. The private and public share formats are described in Table 1. Note that none of these files contains the optional values $d \bmod p-1$, $d \bmod q-1$, or $q^{-1} \bmod p$ normally used to optimize RSA computations because none of the parties can construct these values.

The values g and $g^{d_i} \bmod N$ stored in the public shared key are used to detect incorrect (or possibly compromised) private share operations by the share servers. Their function is not discussed in this paper.

3.3 Testing the shared keys

Once a shared key is generated among the servers it undergoes a number of tests to verify proper sharing. The first thing the servers do is trial decryption: each server picks a random message, encrypts it using the

[1]Because SSLeay supports some degree of object polymorphism, our "extended" RSA keys can be used interchangeably with "ordinary" RSA keys in SSLeay.

Private share file format

Data Type	Field
INTEGER	Version
INTEGER	N
INTEGER	e
INTEGER	k (number of sets)
INTEGER	d_1
$\vdots$	$\vdots$
INTEGER	d_k

Public key file format

Data Type	Field
INTEGER	Version
INTEGER	N
INTEGER	e
INTEGER	g (generator)
INTEGER	k (number of sets)
INTEGER	$g^{d_1} \bmod N$
$\vdots$	$\vdots$
INTEGER	$g^{d_k} \bmod N$

Table 1: Private and public shared key formats

public key, and sends the result to all other servers. Each server then applies its private share to the ciphertext and sends the result back to the originating server. The originating server combines all the results and compares the resulting plaintext to the original random message it chose.

When our system is run in a test mode, more aggressive testing is done on the output. Once the key is generated, all servers send their shares of the private key d to all other servers. They also send their private shares of the factors p and q. Each server then verifies that p and q are both prime, that $N = pq$, and that $e \cdot d = 1 \bmod \phi(N)$. This test should clearly not be done under normal system operation since it exposes the private key to each of the servers, defeating the main point of key sharing.

4 Practical optimizations

We describe several practical optimizations we use to improve the performance of distributed key generation. First, we explain the main reason why our implementation is slower than standard single user generation. To generate an RSA key, a single user repeatedly picks random numbers until two primes are found. These primes are multiplied to form $N = pq$. The probability that a random n-bit integer is prime is approximately $1/n$. Consequently, an average of n probes are needed until a prime is found; $2n$ are needed until two primes are found. This approach cannot be used in distributed key generation since the prime factors have to be kept secret. Instead, in our implementation the servers first share two random n-bit integers p and q. The shared numbers are multiplied to obtain $N = pq$ and a double-primality test is then directly applied to N. The probability that both p and q are *simultaneously* prime is asymptotically $1/n^2$. Therefore, naively one has to perform n^2 probes on average until a suitable N is found. This is much worse than the expected $2n$ probes needed in single user generation, resulting in a slowdown of a factor of 256 (!) for a 1024 bit RSA modulus. Our first and most significant optimization eliminates much of this slowdown by an approach we call *distributed sieving*. Other optimizations take advantage of the distributed environment in which the computation takes place.

4.1 Distributed sieving

The goal of distributed sieving is to ensure that in Step 1 (Section 2) of the algorithm, when the servers generate shares of two random integer p and q, these integers are not divisible by small primes. Unfortunately, since p must remain secretly shared among the servers as $p = p_1 + \ldots + p_k$ it is not possible to efficiently perform trial division on p. Instead, we use a technique that enables each server to pick a random share p_i and be guaranteed that $\sum p_i$ is not divisible by small primes. Our method leaks no information – server i learns nothing about the shares of p belonging to other servers.

This single optimization results in a 10-fold improvement in running time when generating a 1024-bit modulus. In what follows we let M be the product of all odd small primes up to some bound which we call the *sieving bound*. The only constraint is that M be smaller than p.

Step 1: Each server i picks a random integer a_i in the range $[1, \ldots, M]$ such that a_i is relatively prime to M. To do so, we use a classic sieving technique: the server picks a random integer r between 1 and M. It then initializes a small boolean array representing the integers $r, r+1, r+2, \ldots, r+30$. For each of the small prime divisors of M it loops through the array and crosses out the elements divisible by that prime. Finally, it sets a_i to be the first entry that was not crossed out. If all entries were crossed out the process is restarted and a new random a_i is chosen.

In general, the size of the array should be proportional to $\log\log M$. An array of size 30 was experimentally proven to be sufficient for our purposes.

Step 2: Since each a_i is a random integer relatively prime to M, their product $a = a_1 \cdots a_k \bmod M$ is also a random integer relatively prime to M. We need to convert this multiplicative sharing of a into an additive sharing. More precisely, each server should obtain a private b_i in the range $[0, \ldots, M]$ such that $a = b_1 + \ldots + b_k \bmod M$. No information about a should be leaked.

We convert the multiplicative sharing $a = a_1 \cdots a_k \bmod M$ to an additive sharing by considering one server at a time. Suppose for some $1 \le \ell < k$ the value $a_\ell = \prod_{i=1}^{\ell} a_1 \cdots a_\ell \bmod M$ is already converted into an additive sharing

$$a_\ell = b_{1,\ell} + \ldots + b_{k,\ell} \pmod M$$

Initially $\ell = 1$. To convert $a_{\ell+1} = a_1 \cdots a_{\ell+1}$ to an additive sharing we run the algorithm of Section 2.1.1 on the input

$$(b_{1,\ell} + \ldots + b_{k,\ell}) \cdot (u_1 + \ldots + u_k) \pmod M$$

where $u_{\ell+1} = a_{\ell+1}$ and $u_i = 0$ for $i \neq \ell + 1$. The algorithm produces the required additive sharing $a_{\ell+1} = b_{1,\ell+1} + \ldots + b_{k,\ell+1}$ of the product. After $k-1$ iterations of this procedure (for $\ell = 1, \ldots, k-1$) we obtain the desired additive sharing $a = b_1 + \ldots + b_k \bmod M$ of $a_1 \cdots a_k \bmod M$. The privacy achieved is identical to that of Section 2.1.1, hence no information about $a \bmod M$ is leaked.

Step 3: Finally, each server i picks a random r_i in the range $[0, \frac{2^n}{M}]$ and sets $p_i = r_i M + b_i$. Clearly, $p = \sum p_i \equiv a \bmod M$ and hence p is not divisible by any small prime factors.

One caveat in the procedure of Step 2 is that the algorithm of Section 2.1.1 is carried out modulo M which is not prime. This is not a problem as long as the smallest prime factor of M is not smaller than the number of parties k. Shamir secret sharing is not possible in $\mathbb{Z}_M$ when the smallest prime factor of M is less than k. The algorithm of Section 2.1.1 cannot be executed modulo such M. In our experiments the number of servers is always less than 7. Hence, we apply distributed sieving modulo $M = 7 \cdot 11 \cdot 13 \cdots p_\ell$ where p_ℓ is the sieving bound. To ensure that p and q are not divisible by $2, 3$ and 5 we fix their values modulo 30.

4.2 Testing candidates in parallel

While generating and testing a particular candidate, the algorithm is synchronous. All servers step from one phase to another in synchrony. As a result, time is wasted at various synchronization points. To improve performance we run multiple threads on each server. Thread 1 on each server talks to thread 1 on other servers, thread 2 talks to thread 2 on other servers, and so on. As a result, multiple candidates are tested at once and synchronization overhead is reduced. Once one of the threads finds a modulus, the search terminates and all other threads die. Currently, each set of threads communicate on a separate set of ports.

Multithreading the key generation process greatly improves performance. Section 5 gives timing measurements to illustrate its effect. As expected, multithreading gives the greatest benefits in situations where synchronization is taking a lot of time.

It is possible to estimate the number of iterations that a thread must complete before finding an RSA modulus. If ℓ is the number of iterations that a single-threaded implementation would expect to run before finding a modulus, then an implementation with n threads expects

$$\text{iterations} = \frac{1}{1 - (1 - \frac{1}{\ell})^n} \; .$$

Since $\frac{1}{\ell}$ is small, we have $(1 - \frac{1}{\ell})^n \approx 1 - \frac{n}{\ell}$, and hence

$$\text{iterations} \approx \frac{\ell}{n} \; .$$

The timing measurements in Section 5 generally follow this formula. They do not match exactly due to random fluctuations.

An interesting consequence is the diminishing gain of multithreading. Increasing the number of threads from n to $n+1$ causes the expected number of iterations to go from $\frac{l}{n}$ to $\frac{l}{n+1}$, resulting in a decrease of a factor of

$$\frac{l/n+1}{l/n} = \frac{n}{n+1} \; .$$

As the number of threads becomes larger, $\frac{n}{n+1}$ tends to 1, hence adding new threads has little effect on the number of iterations needed to find a modulus.

Adding new threads slows down the servers, but not as much as would be expected under normal circumstances. For example, when going from one thread to two threads, the servers do not operate at half their speed because one thread can utilize the CPU while the other is waiting for another server to synchronize. The optimal number of threads is the result of this tradeoff. Adding threads lowers the expected number of iterations, while slowing down each iteration. Since it is very hard to calculate this tradeoff, the optimal number of threads is found by experimentation (See Section 5).

4.3 Parallel trial division

Recall that once N is computed the servers perform trial division on it before invoking the distributed primality test (Step 2 in Section 2). The k servers can perform this trial division in parallel – each server is in charge of verifying that N is not divisible by some set of small primes. This can be efficiently done by hard-coding all small primes $p_1, p_2, \ldots, p_l$ (greater than the sieving bound) in a list. Server i is in charge of testing that N is not divisible by any of the primes p_j in the list for which $j = i \bmod k$. This factor of k speedup enables us to use a large trial division bound, increasing the effectiveness of trial division.

To further improve performance of trial division we divide by multiple primes at once. For instance, to test if p is divisible by 5 or 7 one can compute $p \bmod 5$ and then $p \bmod 7$ and verify that both values are not zero. However, one can compute $a = p \bmod 35$ at the cost of one division; by testing if $\gcd(a, 35) = 1$, it can be seen if either 5 or 7 (or both) divide a. To take advantage of this trick, we pack as many small primes as possible into a single 32 bits word W and compute $p \bmod W$. Hence, at the cost of one division we test multiple small primes. This packing of small primes into 32-bit words is done before the algorithm begins and is identical on all servers.

4.4 Load balancing

In Step (3) of the primality test (Section 2.2) server 1 has to compute $v_1 = g^{N+1-p_1-q_1} \bmod N$ while all other servers only have to compute $v_i = g^{p_i+q_i} \bmod N$. Notice that $N + 1$ is roughly $2n$ bits while $p_i + q_i$ is only about n bits. Consequently, server 1 has to work twice as hard as the other servers. Ideally, a server would be chosen at random for every iteration and given the role of server 1. However, choosing a random server would require communication between all of the servers, and is thus undesirable. A deterministic method to even things out is to assign the role of server 1 to a different server for each thread, and increment this server after each iteration. This way the computation of $g^{N+1} \bmod N$ is not always done on the same server leading to better load balancing. Furthermore, the randomized timing of the iterations will result in a fairly random and even distribution of the role of server 1. As a result, primality test time is reduced by up to a factor of two.

5 Timing measurements

We measured the performance of distributed key generation in a number of environments. First, we measured the performance of our implementation for a number of common RSA key sizes. Table 2 summarizes our results when running the system on three servers. Note that all of the measurements, except for network traffic, are given as an average per thread. For example, Table 2 shows that when computing a 512-bit modulus, each thread spent an average of 55.7ms per iteration executing the BGW protocol. All times are in milliseconds, unless otherwise noted. Network traffic measurements are given for the entire key-generation process, and reflect the total load on the network. The same amount of traffic is sent as is received, so only one statistic is given in the table.

We used 333MHz Pentium II's running Solaris 2.5.1. The servers are connected by a 10-Megabit Ethernet. Communication between the servers is protected by SSL, and the optimal (largest) sieving bound is used.

The first column measures the time for distributed sieving for both p and q (Section 4.1). The second column measures the average time per iteration for the BGW protocol, which is used four times in sieving and once in the distributed computation of N (Section 2.1). Next we measure the average time per iteration of trial division with a bound of 15,000. The next two columns give both the average running time of the primality test and number of times it was executed. Following this is the number of iterations until a modulus is found and the total average running time per iteration. Finally, the average network traffic per key generation is given in megabytes. The timings are averaged over 20 executions of the algorithm. Note that the total time to generate a 1024-bit key is approximately 90 seconds.

We do not give the time to generate a sharing of the signing exponent d once the modulus is found (Section 2.3) since it is negligible compared to the rest of

	Sieving time	BGW time	trial div. time	prime test		iterations		total time	net traffic	number of threads
				time	num	time	num			
512 bit	39.9	55.7	2.4	37.6	36.4	77.2	119	0.15 min	0.180 Mb	2
1024 bit	362.4	82.6	2.3	637.2	49.1	696.9	130	1.51 min	1.162 Mb	6
2048 bit	369.3	470.9	3.6	4177.3	233.9	2198.0	495	18.13 min	7.48 Mb	6

Table 2: Shared key generation time among three servers.

the computation. These times ranges from 20ms for a 512-bit key to 500ms for a 2048-bit key.

Experimentally, 2 threads per server is optimal for a 512-bit key and 6 threads per server is optimal for a 1024-bit key. For 2048 bits, 6 threads also appears to be optimal.

There is a large differences between the 1024-bit key and the 2048-bit key in the number of primality tests and the number of iterations. This difference is much smaller between the 512-bit key and the 1024-bit key because the 512-bit key uses 2 threads per server, while the other two use 6 threads per server. Since the numbers given are the average *per thread*, the differences between the 512-bit key and the 1024-bit key are actually quite large. For example, the 512-bit key requires an average of 238 iterations overall ($238 = 119 \cdot 2$), while the 1024-bit key requires an average of 780 iterations overall ($780 = 130 \cdot 6$).

It is interesting to note that for a 1024 bit key the number of primality tests is about a third of the number of iterations. Trial division up to 15000 is able to eliminate two thirds of the candidates without a primality test. Normally, trial division is much more effective. However, due to the distributed sieving, the effect of trial division is diminished. Nevertheless, eliminating two thirds of the candidates is significant. Note that, as expected, the effect of trial division is smaller for the 2048 bit modulus (only 1 in 2 moduli are filtered out by it).

Fixed key, varying conditions In Table 3 we fix the key size to 1024 bits and study our system's behavior under different conditions. We generate a shared key among different numbers of servers: 3,4 and 5. All servers run 6 threads.

Running time is measured when all communication is sent in the clear and when SSL is used to secure communication. Performance is measured over a wide area network (WAN), and when all servers run on a single machine. In all these experiments, the trial division bound is set as in the previous table. Again, all measurements are in milliseconds unless otherwise noted.

The first five rows show the results from running the algorithm on different numbers of servers, with and without SSL enabled. Disabling SSL results in a small increase in performance which grows as the number of servers increases. Increasing the number of servers results in a substantial decrease in performance. This is due to an increase in synchronization time, an increase in the distributed sieving time, and an increase in network traffic. We note that the two extra servers used for experiments with four and five servers are older and much slower than the other three.

The line labeled "WAN" is especially interesting. The three servers involved were two computers at Stanford and one at the University of Wisconsin-Madison. We have thus performed the first transcontinental, distributed RSA key generation. This takes an average of about 5.7 minutes. We were stymied in our wish to perform the first intercontinental, distributed RSA key generation by US export laws.

Effect of sieving To demonstrate the effectiveness of distributed sieving we measured the algorithm's running time on a single thread with different sieving bounds. Clearly, a larger sieving bound is better. The results in Table 4 are the times for generating a 512-bit modulus on three servers with communication protected by SSL. Sieving improves performance by more than a factor of 10 for a 1024 bit modulus.

The table shows how sieving dramatically reduces the number of iterations until a key is found. The larger the number, the more effective sieving becomes. Therefore, the key generation program automatically finds the largest acceptable sieving bound for each key size, given the constraint that $M < p$ (Section 4.1).

Multi-threading As discussed in Section 4.2, to reduce the time wasted on synchronization it makes sense to run multiple threads on each of the servers. While one thread is waiting to synchronize with its peers, other threads can execute. Table 5 shows that running several threads in parallel results in a large decrease in running time. As expected, each iteration takes longer, but fewer iterations are required until

	SSL	Sieve time	BGW time	trial div. time	prime test time	time per iteration	total time
3 Servers	No	326	413	2.2	628	695	1.51 min
3 Servers	Yes	362	83	2.3	637	697	1.51 min
4 Servers	No	689	861	1.5	2035	1707	3.70 min
4 Servers	Yes	804	1017	1.9	2173	1909	4.14 min
5 Servers	Yes	1466	1731	1.9	2013	2589	5.61 min
WAN	Yes	774	1012	6.3	1626	1704	5.69 min
Local	No	263	334	2.7	1702	1014	2.20 min
Local	Yes	267	337	3.3	1899	1115	2.42 min

Table 3: The effect of changing the number and locality of servers

	Sieving	trial div.	prime test		iterations		total
	time	time	time	num	time	num	time
Sieve 150	22.6	1.8	36.1	89	46.8	288	14.0 sec
Sieve 50	19.1	1.0	30.7	99	43.8	607	26.6 sec
No sieve	N/A	1.4	34	149	11	10794	117.0 sec

Table 4: The effect of sieving on running time

	BGW	trial div.	prime test	iterations		total
threads/serv.	time	time	time	time	num	time
one	46.5	2.2	201.3	319.5	1102	5.87 min
four	67.6	2.2	414.5	529.3	365	3.22 min
five	77.1	2.2	527.2	622.7	185	1.92 min
six	82.6	2.3	637.2	696.9	130	1.51 min
seven	87.5	2.5	810.8	781.4	129	1.68 min

Table 5: The effect of multiple threads

one of the threads finds a modulus. We generated the 1024-bit keys among three servers with SSL enabled.

Multithreading is most beneficial when synchronization takes a lot of time. This is most obvious in the WAN trials and in the trials with 5 servers. When running these tests single-threaded, key generation was very choppy; bursts of calculations were followed by periods of waiting. When running multithreaded, key generation proceeds smoothly, making much better use of the servers' processors. On the WAN, generating a 1024-bit key with a single thread took 26.5 minutes, while it took only 5.7 minutes using 6 threads per server.

6 Robustness

Up until now we assumed all parties are honestly following the key generation protocol. For some applications it is desirable to make the protocol robust against active adversaries that cheat during the protocol. Since the RSA function is verifiable (the parties can simply check that they correctly decrypt encrypted test messages) active adversaries are limited in the amount of damage they can cause. However, it may still be possible that a party cheat during the protocol and consequently be able to factor the resulting N. Similarly, a party can cheat and cause a non-RSA modulus to be incorrectly accepted.

We describe a simple method for making our non-robust protocol robust when the number of participants is small (e.g. less than ten). Consider the case of four parties where at most one of them is malicious. One can run the non-robust protocol until a candidate modulus N is found. At this point the protocol is run four more times, once for each triplet of users. In the first run, party 1 shares her values p_1, q_1 with the other three parties by writing $p_1 = p'_2 + p'_3 + p'_4$ and $q_1 = q'_2 + q'_3 + q'_4$ where p'_j, q'_j are random integers in the range $[0, N]$. Party 1 then sends p'_i, q'_i to party i for $i = 2, 3, 4$. Party i adds these values to its own p_i, q_i. Next, parties $2, 3$, and 4 run our non-robust protocol among the three of them (ignoring party 1). If the resulting N does not match the N computed when all four parties were involved, or if N turns out to not be an RSA modulus, the N is rejected and the parties announce that one of them is misbehaving. This experiment is repeated with all four triplets – each time exactly one party is excluded from the computation. Assuming at most one party is malicious, the resulting N must be a product of two large primes. Furthermore, the malicious party cannot know the factorization of N since at no point in the protocol does an honest party reveal any information about it's share to another single party. This approach enables the parties to detect cheating, but it does not help in detecting the malicious party.

In general, when k parties are engaged in our non-robust protocol, and c of them are malicious, the protocol can be made robust at the cost of $\binom{k}{c}$ extra invocations. The resulting computation is $\lfloor \frac{k-c-1}{2} \rfloor$ private. Clearly this approach can only be applied as long as both k and c are very small.

Recently, Frankel, MacKenzie and Yung [9] showed how our protocol can be made to withstand $\lfloor \frac{k-1}{2} \rfloor$ malicious parties. Their approach enables the parties to detect and exclude the malicious party. In practice, one could run our non-robust protocol until a modulus N is found which is believed to be a product of two primes. Then, the robust Frankel-MacKenzie-Yung protocol can be used to determine that no majority of parties cheated during the non-robust phase. For more results on robust generation of shared RSA keys see [2].

7 Conclusions

The goal of this paper is to demonstrate the effectiveness of shared RSA key generation. Our optimized implementation and timing measurements show that distributed key generation is a viable method for generating shared RSA keys. Using three 333MHz PC's on a 10Mbps Ethernet we were able to generate 1024 bit shared RSA keys in under 91 seconds. On a wide area network, using servers across the US, we were able to generate keys in under 6 minutes. These performance figures are achieved using a number of effective optimizations and by multi-threading the key generation process. We hope these results can be used to reduce the need for trusted dealers. Our code will be made available on the project's web site.

We note that Spalding and Wright [16] previously implemented a version of the Boneh-Franklin key generation algorithm. Their implementation *simulates* the distributed environment on a single machine in a single process. Consequently, the timing measurements don't reflect network latencies or the parallelism obtained by multiple servers. Their implementation does not use distributed sieving since the technique was unknown at the time.

To obtain distributed key generation in under 91 seconds we designed and implemented a number of practical optimizations. The most significant is distributed sieving, which is responsible for a 10-fold improvement in running time. Other optimizations take additional advantage of the distributed environment.

References

[1] M. Ben-Or, S. Goldwasser, A. Wigderson, “Completeness theorems for non-cryptographic fault tolerant distributed computation”, STOC 1988, pp. 1–10.

[2] S. Blackburn, S. Blake-Wilson, M. Burmester, S. Galbraith, “Secure construction of shared RSA keys”, Preprint.

[3] J. Benaloh (Cohen), “Secret sharing homomorphisms: keeping shares of a secret secret,” Crypto ’86, 251-260.

[4] D. Boneh, M. Franklin, “Efficient generation of shared RSA keys”, in Proceedings Crypto’ 97, pp. 425–439.

[5] CertCo, *Root CertAuthority*, `http://www.certco.com`

[6] C. Cocks, “Split knowledge generation of RSA parameters”, Available from the author `cliff_cocks@cesg.gov.uk`.

[7] A. Fiat and A. Shamir, “How to prove yourself: Practical solutions to identification and signature problems,” Crypto ’86, 186-194.

[8] Y. Frankel, “A practical protocol for large group oriented networks”, Eurocrypt 89, pp. 56–61.

[9] Y. Frankel, P. MacKenzie, M. Yung, “Robust efficient distributed RSA key generation”, STOC 1998, pp. 663–672.

[10] P. Gemmel, “An introduction to threshold cryptography”, in CryptoBytes, a technical newsletter of RSA Laboratories, Vol. 2, No. 7, 1997.

[11] M. Reiter, M. Franklin, J. Lacy, R. Wright, “The Ω key management service”, Proceedings of the 3rd ACM conference on Computer and Communication Security, 1996.

[12] R. Rivest, “Finding four million large random primes”, Proceedings of Crypto ’91.

[13] T. Rabin, “A simplified approach to threshold and proactive RSA”, Proceedings of Crypto’ 98.

[14] Secure Electronic Transactions (SET), `http://www.visa.com`

[15] A. Shamir, “How to share a secret”, Comm. of the ACM, Vol. 22, 1979, pp. 612–613.

[16] S. Spalding, R. Wright, ”Experimental Performance of Shared RSA Modulus Generation”, In proceedings of SODA ’99.

[17] E. Young, SSLeay, `http://www.ssleay.org/`

Appendix: Configuration file

All system configuration parameters are located in one configuration file. This includes IP addresses of all servers involved as well as type of RSA key to generate. To reduce administrative overhead, all servers use an identical configuration file. We include an example configuration file.

```
;---------------------------------------------------
;---   Part 1: General configuration paratmers   ---
;---------------------------------------------------

Num_Servers:  3
Threads:      4               ; Number of threads per server.

HomeDir:      /ITTC/Log/
Word_Size:    32              ; 32 or 64 bits per word.
Log_Level:    Notify          ; Minimum priority of logged messages.

; Sieving and trial division bounds
Sieve:        True
TrialDiv_End: 17800

Public_Key:   65537
Key_Length:   Normal
; Possible key-lengths:  Weak = 512 bits,  Normal = 1024,  Strong = 2048.

Test_Mode:    False


;--------------------------------------
;---   Part 2: Server parameters   ---
;--------------------------------------

; Server_Cert: location of server's certificate.
; Server_Key:  location of server's private key.
; Server_Transport:  Clear transport vs. SSL transport

Server_IP_Addr_0:        saga3.stanford.edu
Share_IP_Port_0:         8713
Server_Cert_0:           cert_s0.pem
Server_Key_0:            key_s0.pem
Server_Transport_0:      clear
Server_Sequence_File_0:  seq0

Server_IP_Addr_1:        cardinal4.stanford.edu
Share_IP_Port_1:         8713
Server_Cert_1:           cert_s1.pem
Server_Key_1:            key_s1.pem
Server_Transport_1:      clear
Server_Sequence_File_1:  seq1

Server_IP_Addr_2:        epic2.stanford.edu
Share_IP_Port_2:         8713
Server_Cert_2:           cert_s2.pem
Server_Key_2:            key_s2.pem
Server_Transport_2:      clear
Server_Sequence_File_2:  seq2

Server_IP_Addr_3:        amy5.stanford.edu
Share_IP_Port_3:         8713
Server_Cert_3:           cert_s2.pem
Server_Key_3:            key_s2.pem
Server_Transport_3:      clear
Server_Sequence_File_3:  seq2
```