Generating RSA Keys on a Handheld Using an Untrusted Server

Nagendra Modadugu
nagendra@cs.stanford.edu

Dan Boneh*
dabo@cs.stanford.edu

Michael Kim
mfk@cs.stanford.edu

Abstract

We show how to efficiently generate RSA keys on a low power handheld device with the help of an untrusted server. Most of the key generation work is offloaded onto the server. However, the server learns no information about the key it helped generate. We experiment with our techniques and show they result in up to a factor of 5 improvement in key generation time. The resulting RSA key looks like an RSA key for paranoids. It can be used for encryption and key exchange, but cannot be used for signatures.

1 Introduction

In recent years we have seen an explosion in the number of applications for handheld devices. Many of these applications require the ability to communicate securely with a remote device over an authenticated channel. Example applications include: (1) a wireless purchase using a cell phone, (2) remote secure synchronization with a PDA, (3) using a handheld device as an authentication token [2], and (4) handheld electronic wallets [3]. Many of these handheld applications require the ability to issue digital signatures on behalf of their users.

Currently, the RSA cryptosystem is the most widely used cryptosystem for key exchange and digital signatures: SSL commonly uses RSA-based key exchange, most PKI products use RSA certificates, etc. Unfortunately, RSA on a low power handheld device is somewhat problematic. For example, generating a 1024 bit RSA signature on the PalmPilot takes approximately 30 seconds. Nevertheless, since RSA is so commonly used on servers and desktops it is desirable to improve its performance on handhelds.

In this paper we consider the problem of *generating* RSA keys. Generating a 1024 bit RSA key on the PalmPilot can take as long as 15 minutes. The device locks up while generating the key and is inaccessible to the user. For wireless devices battery life time is a concern. Consider a user who is given a new cellphone application while traveling. The application may need to generate a key before it can function. Generating the key while the user is traveling will lock up the cellphone for some time and may completely drain the batteries.

The obvious solution is to allow the handheld to communicate with a desktop or server and have the server generate the key. The key can then be downloaded onto the handheld. The problem with this approach is that the server learns the user's private key. Consequently, the server must be trusted by the user. This approach limits mobility of the handheld application since users can only generate a key while communicating with their home domain. We would like to enable users to quickly generate an RSA key even when they cannot communicate with a trusted machine.

*Supported by NSF CCR-9732754.

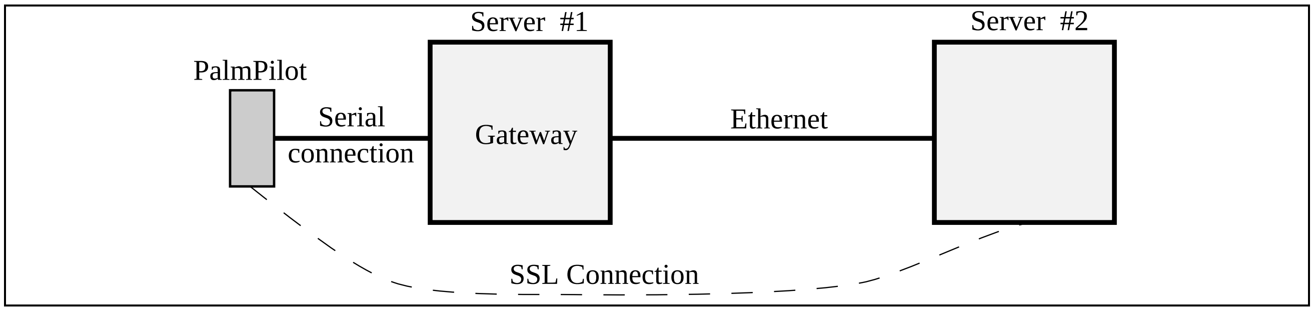

Figure 1: A two server configuration

We study the following question: can we speed up RSA key generation on a handheld with the help of an *untrusted server*? Our goal is to offload most of the key generation work onto the untrusted server. However, once the key is generated the server should have no information about the key it helped generate. This way the handheld can take advantage of the server's processing power without compromising the security of its keys.

Our best results show how to speed up the generation of *unbalanced* RSA keys. We describe these keys and explain how they are used in the next section. Our results come in two flavors. First, we show how to speed up key generation with the help of *two* untrusted servers. We assume that the two servers are unable to share information with each other. For instance, the two untrusted servers may be operated by different organizations. Using two untrusted servers we are able to speed up key generation by a factor of 5. We then show that a *single* untrusted server can be used to speed up key generation by a factor of 2. In Section 4 we discuss speeding up normal RSA key generation (as opposed to unbalanced keys).

We implemented and experimented with all our algorithms. We used the PalmPilot as an example handheld device since it is easy to program. Clearly our techniques apply to any low power handheld: pagers, cell phones, MP3 players, PDA's, etc. In our implementation, the PalmPilot connects to a desktop machine using the serial port. When a single server is used to help generate the key, the pilot communicates with the desktop using TCP/IP over the serial link. The desktop functions as the helping server. Note that there is no need to protect the serial connection. After all, since the desktop learns no information about the key it helped generate, an attacker snooping the connection will also learn nothing. When two servers are used, the desktop functions as a gateway enabling the pilot to communicate with the two servers. In this case, communication between the pilot and servers is protected by SSL to prevent eavesdropping by the gateway machine, and to prevent one server from listening in on communication intended for the other. Typically, the gateway machine functions as one of the two servers, as shown in Figure 1.

1.1 Timing cryptographic primitives on the PalmPilot

For completeness we list some running times for cryptographic operations on the PalmPilot. We used the Palm V which uses a 16.6MhZ Dragonball processor. Running times for DES, SHA-1, and RSA were obtained using a port of parts of SSLeay to the PalmPilot started by Ian Goldberg.

Algorithm	Time	Comment
DES encryption	4.9ms/block	
SHA-1	2.7ms/block	
1024 bit RSA key generation	15 minutes on average	
1024 bit RSA sig. generation	27.8 sec.	
1024 bit RSA sig. verify	0.758 sec.	$e = 3$
1024 bit RSA sig. verify	1.860 sec.	$e = 65537$

2 Preliminaries

2.1 Overview of RSA key generation

As a necessary background we give a brief overview of RSA key generation. Recall that an RSA key is made up of an n-bit modulus $N = pq$ and a pair of integers d, called the private exponent, and e, called the public exponent. Typically, N is the product of two large primes, each $n/2$ bits long. Throughout the paper we focus on generating a 1024 bit key (i.e. $n = 1024$). The algorithm to generate an RSA key is as follows:

Step 1: Repeat the following steps until two primes p, q are found:

- **a. Candidate** Pick a random 512 bit candidate value p.
- **b. Sieve** Using trial division, check that p is not divisible by any small primes (i.e. 2,3,5,7, etc.).
- **c. Test** Run a probabilistic primality test on the candidate. For simplicity one can view the test as checking that $g^{(p-1)/2} \equiv \pm 1 \pmod{p}$, where g is a random value in $1 \ldots p-1$. All primes will pass this test, while a composite will fail with overwhelming probability [10].

Step 2: Compute the product $N = pq$ (the product is 1024 bits long).

Step 3: Pick encryption and decryption exponents e and d where $e \cdot d = 1 \bmod \varphi(N)$ and $\varphi(N) = N - p - q + 1$.

The bulk of the key generation work takes place in step (1). Once the two primes p and q are found, steps (2) and (3) take negligible work. We note that trial division (step 1b) is frequently optimized by using a sieving algorithm. Sieving works as follows: once the candidate p is chosen in step (1a), the sieve is used to quickly find the closest integer to p that is not divisible by any small primes. The candidate p is then updated to be the integer found by the sieve. Throughout the paper we use a sieving algorithm attributed to Phil Zimmerman.

Our goal is to improve the performance of step (1). Within step (1), the exponentiation in step (1c) dominates the running time. Our goal is to offload the primality test to the server without exposing any information about the candidate being tested. Hence, the question is: how can we test that $g^{p-1} \bmod p = 1$ with the help of a server without revealing any information about p? To do so we must show how to carry out the exponentiation while solving two problems: (1) hiding the modulus p, and (2) hiding the exponent $p - 1$.

2.2 Unbalanced RSA keys

Our best results show how to speed up the generation of unbalanced RSA keys. An unbalanced key uses a modulus N of the form $N = p \cdot R$ where p is a 512 bit prime and R is a 4096 bit *random number*. One can show that with high probability R has a prime factor that is at least 512 bits long (the probability that it does not have such a factor is less than $1/2^{24}$). Consequently, the resulting modulus N is as hard to factor as a standard modulus $N = pq$.

An unbalanced key is used in the same way as standard RSA keys. The public key is $\langle e, N \rangle$ and the private key is $\langle d, N \rangle$. We require that $e \cdot d = 1 \bmod p - 1$. Suppose p is m-bits long. The system can be used to encrypt messages shorter than m bits. As in standard RSA, to encrypt a message M, whose length is much shorter than m bits, the sender first applies a randomized padding mechanism, such as OAEP [4, 9]. The padding mechanism results in an $m - 1$ bit integer P (note that $P < p$). The sender then constructs the ciphertext by computing $C = P^e \bmod N$. Note that the ciphertext is as big as N. To decrypt a ciphertext C, the receiver first computes $C_p = C \bmod p$ and then recovers P by computing $P = C_p^d \bmod p$. The plaintext M is then easily extracted from P. Since decryption is done modulo p it is as fast as standard RSA.

The technique described above for using an unbalanced key is similar to Shamir's "RSA for paranoids" [11]. It shows that unbalanced keys can be used for encryption/decryption and key exchange. Unfortunately, unbalanced keys cannot be used for digital signatures. We note that some attacks against RSA for paranoids have been recently proposed [5]. However, these attacks do not apply when one uses proper padding prior to encryption. In particular, when OAEP padding is used [4] the attacks cannot succeed since the security of OAEP (in the random oracle model) only relies on the fact that the function $f : \{0, \ldots, 2^{m-1}\} \to \mathbb{Z}_N$ defined by $f(x) = x^e \bmod N$ is a one-to-one trapdoor one way function.

3 Generating an unbalanced RSA key with the help of untrusted servers

We show how RSA key generation can be significantly sped up by allowing the PalmPilot to interact with untrusted servers. At the end of the computation the servers should know nothing about the key they helped generate. We begin by showing how *two* untrusted servers can help the Pilot generate RSA keys. The assumption is that these two servers cannot exchange information with each other. To ensure that an attacker cannot eavesdrop on the network and obtain the information being sent to both servers, our full implementation protects the connection between the Pilot and the servers using SSL. Typically, the machine to which the pilot is connected can be used as one of the untrusted servers (Figure 1). We then show how to speed up key generation with the help of a *single* server. In this case there is no need to protect the connection.

3.1 Generating keys with the help of two servers

Our goal is to generate a modulus of the form $N = pR$ where p is a 512-bit prime and R is a 4096-bit random number. To offload the primality test onto the servers we must hide the modulus p and the exponent $p - 1$. To hide the modulus p we intend to multiply it by a random number R and send the resulting $N = pR$ to the servers. The server will perform computations modulo $N = pR$. If it turns out that p is prime, then sending N to the servers does not expose any information about p or R. If

p is not prime we start over. To hide the exponent $p-1$ used in the primality test we intend to share it among the two servers. Individually, neither one of the servers will learn any information.

Our algorithm for generating an unbalanced RSA modulus $N = pR$ is as follows. The algorithm repeats the following steps until an unbalanced key is generated:

Step 1: Pilot generates a 512 bit candidate p that is not divisible by small primes and a 4096 bit random number R. We require that $p = 3 \bmod 4$.

Step 2: Pilot computes $N = p \cdot R$.

Step 3: Pilot picks random integers s_1 and s_2 in the range $[-p, p]$ such that $s_1 + s_2 = (p-1)/2$. It also picks a random $g \in \mathbb{Z}_N^*$.

Step 4: Pilot sends $\langle N, g, s_1 \rangle$ to server 1 and $\langle N, g, -s_2 \rangle$ to server 2.

Step 5: Server 1 computes $X_1 = g^{s_1} \bmod N$. Server 2 computes $X_2 = g^{(-s_2)} \bmod N$. Both results X_1 and X_2 are sent back to the pilot.

Step 6: Pilot checks whether $X_1 = \pm X_2 \bmod p$. If equality holds, then $N = pR$ is declared as a potential unbalanced RSA modulus. Otherwise, the algorithm is restarted in Step 1.

Step 7: The Pilot locally runs a probabilistic primality test to verify that p is prime. This is done to ensure that the servers returned correct values.

First, we verify the soundness of the algorithm. In step 6 the Pilot verifies that $g^{s_1} \cdot g^{s_2} = g^{(p-1)/2} = \pm 1 \bmod p$. If the test is satisfied then p is very likely to be prime. Then step 7 ensures that p is in fact prime and that the servers did not respond incorrectly. When generating a 1024 bit RSA key, a single primality test takes little time compared to the search for a 512 bit prime. Hence, Step 7 adds very little to the total running time.

During the search for the prime p, the only computation carried out by the pilot is the probable prime generation and the computation of s_1 and s_2. The time to construct s_1 and s_2 is negligible. On the other hand, generating the probable prime p requires a sieve to ensure that p is not divisible by small factors. As we shall see in Section 5 the sieve is the bottleneck. This is unusual since in standard RSA modulus generation sieving takes only a small fraction of the entire computation. We use a sieving method attributed to Phil Zimmerman. We note that faster sieves exist, but they result in an insecurity of our algorithm.

Security To analyze the security properties of the algorithm we must argue that the untrusted servers learn no information of value to them. During the search for the RSA modulus many candidates are generated. Since these candidates are independent of each other, any information the servers learn about rejected candidates does not help them in attacking the final chosen RSA modulus. Once the final modulus $N = pR$ is generated in Step 2, each server is sent the value of N and s_i where i is either 1 or 2. The modulus N will become public anyhow (it is part of the public key) and hence reveals no new information. Now, assuming servers 1 and 2 cannot communicate, the value s_1 is simply a random number (from Server 1's point of view). Server 1 could have just as easily picked a random number in the range $[-N, N]$ itself. Hence, s_1 reveals no new information to Server 1 (formally, a simulation argument shows that s_1 reveals at most two bits). The same holds for Server 2. Hence, as long as Server 1 and Server 2 cannot communicate, no useful information is revealed about the factorization of N. We note that if the servers are able to communicate, they can factor N.

Performance The number of iterations until a modulus is found is identical to local generation of an (unbalanced) modulus on the PalmPilot. However, each iteration is much faster than the classic RSA key generation approach of Section 2.1. After all, we offloaded the expensive exponentiation to a fast Pentium machine. As we shall see in Section 5.2, the total running time is reduced by a factor of 5.

3.2 Generating keys with the help of a single server

Next, we show how a *single* untrusted server can be used to reduce the time to generate an RSA key on the PalmPilot. Once the key is generated, the server has no information regarding the key it helped generate. Typically, the pilot connects to the helping server directly through the serial or infrared ports.

As before we need to compute $g^{(p-1)/2} \bmod p$ to test whether p is prime. Our technique involves reducing the size of the exponent using the help of the server and hence speeding up exponentiation on the pilot. The algorithm repeats the following steps until an unbalanced modulus is found:

Step 1: Pilot generates a 512 bit candidate p that is not divisible by small primes and a 4096 bit random number R. We require that $p = 3 \bmod 4$.

Step 2: Pilot computes $N = p \cdot R$. It picks a random $g \in \mathbb{Z}_N^*$.

Step 3: Pilot picks a random 160 bit integer r and a random 512 bit integer a. It computes $z = r + a(p-1)/2$.

Step 4: Pilot sends $\langle N, g, z \rangle$ to the server.

Step 5: The server computes $X = g^z \bmod N$ and sends X back to the Pilot.

Step 6: Pilot computes $Y = g^r \bmod p$.

Step 7: Pilot checks if $X = \pm Y \bmod p$. If so then the algorithm is finished and $N = pR$ is declared as a potential unbalanced RSA modulus. Otherwise, the algorithm is restarted in Step 1.

Step 8: The Pilot locally runs a probabilistic primality test to verify that p is prime.

To verify soundness observe that N will make it to step 8 only if $X = \pm Y$ i.e. $g^{r+a(p-1)/2} = \pm g^r \bmod p$. As before, this condition is always satisfied if p is prime. The test will fail with overwhelming probability if p is not prime. Hence, once step 8 is reached the modulus $N = pR$ is very likely to be an unbalanced modulus. The test is Step 8 takes little time compared to the entire search for the 512-bit prime.

Performance Since we are generating an unbalanced modulus the number of iterations until N is found is the same as in local generation of such a modulus on the PalmPilot. Within each iteration the Pilot generates p and R using a sieve and then computes $Y = g^r \bmod p$ (in step 6). However, r is only 160 bits long. This is much shorter than when a key is generated without the help of a server. In that case the Pilot has to compute an exponentiation where the exponent is 512 bits long. Hence, we reduce the exponentiation time by approximately a factor of three. Total key generation time is reduced by a factor of 2, due to the overhead of sieving.

Recall that in Step 6 the Pilot computes $Y = g^r \bmod p$ where r is a 160-bit integer. This step can be further sped up with the help of the server. Let $A = 2^{40}$ and write $r = r_0 + r_1 A + r_2 A^2 + r_3 A^3$ where r_0, r_1, r_2, r_3 are all in the range $[0, A]$. In Step 5 the server could send back the vector $\vec{R} = \langle g^A, g^{A^2}, g^{A^3} \rangle \bmod N$ in addition to sending X. Let $\vec{R} = \langle R_1, R_2, R_3 \rangle$. Then in Step 6 the Pilot only has to compute $Y = g^{r_0} \cdot R_1^{r_1} \cdot R_2^{r_2} \cdot R_3^{r_3} \bmod p$. Using Simultaneous Multiple Exponentiation [7, p. 617] Step 6 can now be done in approximately half the time of computing $Y = g^r \bmod p$ on the Pilot directly. This improvement reduces the total exponentiation time on the Pilot by an additional factor of 2 .

Security In the last iteration, when the final p and R are chosen, the server learns the value $z = a(p-1) + r$. Although z is a "random looking" 1024 bit number, it does contain some information about p. In particular, $z \bmod p-1$ is very small (only 160 bits long). The question is whether z helps an adversary break the resulting key. The best known algorithm for doing so requires $2^{r/2}$ modular exponentiations. Due to our choice of 160 bits for r, the algorithm has security of approximately 2^{80}. This is good enough since a 1024 bits RSA key offers security of 2^{80} anyhow. Nevertheless, the security of the scheme is heuristic since it depends on the assumption that no faster algorithm exists for factoring N given z. More precisely, the scheme depends on the following "$(p-1)$-multiple assumption":

$(p-1)$-*multiple assumption:* Let A_n be the set of integers $N = pq$ where p and q are both n-bit primes. Let m be an integer so that the fastest algorithm for factoring a random element $N \in A_n$ runs in time at least $2^{m/2}$. Then the two distributions: $\langle N, r + a(p-1)/2 \rangle$ and $\langle N, x \rangle$ cannot be distinguished with non-negligible probability by an algorithm whose running time is less than $2^{m/2}$. Here N is randomly chosen in A_n, a is randomly chosen in $[0, p]$, r is randomly chosen in $[0, 2^m]$, and x is randomly chosen in $[0, p^2/2]$.

Based on the $(p-1)$-multiple assumption, the integer z given to the server contains no more statistical information than a random number in the range $[0, p^2]$. Hence, the server learns no new useful information from z.

As before, since the generated key is an unbalanced key it can only be used for encryption/decryption and key exchange. It cannot be used for signatures.

4 Generating standard RSA keys

One could wonder whether the techniques described in the previous sections can be used to speed up generation of standard RSA keys. We show that at the moment these techniques do not appear to improve the generation time for a 1024 bit key. For shorter keys, e.g. 512 bits keys, we get a small improvement. In what follows we show how to generate a normal RSA key, $N = pq$, with the help of two servers.

We wish to generate an RSA modulus $N = pq$ where p and q are each 512-bits long. As before, we wish to offload the primality test to the servers. To do so we must hide the moduli p and q and the exponents $p-1$ and $q-1$. The basic idea is to *simultaneously* test primality of both p and q. For each pair of candidates p and q the Pilot computes $N = pq$ and sends N to the servers. The servers carry out the exponentiations modulo N. To hide the exponents $p-1$ and $q-1$ we share them among the two servers as in the last section.

The resulting algorithm is similar to that for generating unbalanced keys. In fact, the server-side is identical. The algorithm works as follows. Repeat the following steps until a standard RSA modulus is found:

Step 1: Pilot generates two candidates p, q so that neither one is divisible by small primes. We refer to p and q as probable primes.

Step 2: Pilot computes $N = p \cdot q$ and $\varphi(N) = N - p - q + 1$. Pilot picks a random $g \in \mathbb{Z}_N^*$.

Step 3: Pilot picks random integers φ_1 and φ_2 in the range $[-N, N]$ such that $\varphi_1 + \varphi_2 = \varphi(N)/4$.

Step 4: Pilot sends $\langle N, g, \varphi_1 \rangle$ to server 1 and $\langle N, g, -\varphi_2 \rangle$ to server 2.

Step 5: Server 1 computes $X_1 = g^{\varphi_1} \pmod{N}$. Server 2 computers $X_2 = g^{-\varphi_2} \pmod{N}$. Both results X_1 and X_2 are sent back to the pilot.

Step 6: Pilot checks if $X_1 = \pm X_2 \bmod N$. If so, the algorithm is finished and $N = pq$ is declared as a potential RSA modulus. Otherwise, the algorithm is restarted in Step 1.

Step 7: The Pilot locally runs a probabilistic primality test to verify that p and q are prime. This is done to ensure that the servers returned correct values.

First, we verify soundness of the algorithm. In step 6 the Pilot is testing that $X_1 = \pm X_2$, namely that $g^{\varphi_1} = g^{-\varphi_2} \bmod N$. That is, we check that $g^{\varphi_1 + \varphi_2} = g^{\varphi(N)/4} = \pm 1 \bmod N$. Clearly, this condition holds if p and q are both primes. Furthermore, it will fail with overwhelming probability if either p or q are not prime. Hence, Step 7 is reached only if $N = pq$ is extremely likely to be a proper RSA modulus. Step 7 then locally ensures that p and q are primes.

Security To analyze the security properties of the algorithm we must argue that the untrusted servers learn no information of value to them. During the search for the RSA modulus many candidates are generated. Since these candidates are independent of each other, any information the servers learn about rejected candidates does not help them in attacking the final chosen RSA modulus. Once the final modulus $N = pq$ is generated in Step 2, each server is sent the value of N and φ_i where i is either 1 or 2. The modulus N will become public anyhow (it is part of the public key) and hence reveals no new information. Now, assuming servers 1 and 2 cannot communicate, the value φ_1 is simply a random number (from Server 1's point of view). Server 1 could have just as easily picked a random number in the range $[-N, N]$ itself. Hence, φ_1 reveals no new information to Server 1. As long as Server 1 and Server 2 cannot communicate, no useful information is revealed about the factorization of N. If the servers are able to communicate, they can factor N.

Performance Each iteration in our algorithm is much faster than the classic RSA key generation approach of Section 2.1 — we offloaded the expensive exponentiation to a fast Pentium machine. Unfortunately, the number of iterations required until an RSA modulus is found is higher. More precisely, suppose in the classic approach one requires k iterations on average until a 512-bit prime is found. Then the total number of iterations to find two primes is $2k$ on average. In contrast, in our approach both p and q must be simultaneously prime. Hence, k^2 iterations are required. We refer to this effect as a *quadratic slowdown*. When generating a 1024 bit modulus the value of k is approximately 14. So even though we are able to speed up each iteration by a factor of 5, there are seven times as many iterations on average. Therefore when generating a standard 1024 bit key these techniques do not improve the running time. When generating a shorter key, e.g. a 512 bit key, the quadratic slowdown penalty is less severe since k is smaller (9 rather than 14). For such short keys we obtain a small improvement in performance.

Similarly, when generating keys with the help of a *single* server, the quadratic slowdown outweighs the reduction in time per iteration. It is an open problem to speed up server aided generation of standard RSA keys.

5 Experiments and implementation details

5.1 Implementation details

The two main components of our implementation were the cryptographic and networking modules. SSLeay provided for the cryptographic code on both the server (Pentium II 400Mhz) and PalmPilot side. In the case of the Pilot, we used SSLeay code that had been previously ported by Ian Goldberg.

5.1.1 Networking

We connect the pilot to a Windows NT gateway running RAS (Remote Access Service) through a serial-to-serial interface. The function of the gateway was to provide TCP/IP access to the network.

In our single server implementation, we used the gateway as our assisting server while in our dual server implementation, we used the gateway and another local machine.

Our networking layer abstracts the secure communication of BigIntegers to and from the PalmPilot. The network layer packs a number of BigIntegers into a buffer and sends the entire buffer at once. The receiving side unpacks the buffer and processes it as required.

5.2 Experiments

Tables 1 and 2 show the results we obtained when generating 512, 768 and 1024 bit RSA keys. The network traffic column measures the amount of data (in bytes) exchanged between the Pilot and the servers. We generate keys using three methods:

(1) Local: Key generated locally on the Pilot (no interaction with a server).

(2) One server: Pilot aided by a single untrusted server.

(3) Two servers: Pilot aided by two untrusted servers.

As expected we see that generating unbalanced keys with the aid of one or two servers leads to a performance improvement over generating keys locally on the PalmPilot. The rest of this section discusses these experimental results.

We note that the key factor that determines the time it takes to generate an RSA key is the time *per iteration* (the time to sieve and exponentiate *one* probable prime p). This number is more meaningful than the total running since since the total time has a high variance. More precisely, the number of iterations until a prime is found has high variance. Our tables state the average number of iterations we obtained.

In our experiments, we carried out trial division on a candidate prime using the first 2048 primes (upto approximately 17,000). In all our experiments we observed that the server's responses are

		Sieve time(ms)	Server time(ms)	Exp. time(ms)	total time/ iter.(ms)	average num. iter.	total time	net traf.
Local	unbal.	3,805		21,233	25,038	18.16	7.5min.	
Local	norm.	3,805		21,233	25,038	36.32	15.16min.	
One serv.	unbal.	3,516	955	6,995	11,467	14.5	2.7min.	5,568
Two serv.	unbal.	3,587	1,462		5,156	12.75	1.1min	8,160
Two serv.	norm.	7,850	820	0	8,720	406	59min	311,808

Table 1: Statistics for different key generation methods (1024 bit keys)

instantaneous compared to the Pilot's processing time. Consequently, improving server performance will only marginally affect the overall running time.

5.2.1 Generating a 1024 bit key

Table 1 shows detailed timing measurements for generating 1024 bit RSA keys. Our breakdown of timing measurements follows the description in Section 2.1. The first column shows the time to pick a probable prime, the second shows the time the Pilot spent waiting for the server to respond, the third shows the time to exponentiate on the PalmPilot (not used in the two-server mode). The last column shows the total network traffic (in bytes).

The first two rows in Table 1 measure the time to generate keys on the Pilot. The first column represents the time to generate an unbalanced key, the second represents the time to generate a normal $N = pq$ key. Since an unbalanced key requires only one prime (the other is a random number) the number of iterations for locally generating an unbalanced key is half that for generating a normal key.

When comparing the time per iteration for local generation and two server generation, we see that using two servers we get an improvement of a factor of 5. Using one server we obtain an improvement of a factor of 2. The average number of iterations is approximately the same in all three methods. Note that the improvements are a result of speeding up (or eliminating) the exponentiation step on the PalmPilot. Observe that when two servers are used the bottleneck is the sieving time — the time to generate a probable prime p.

On average, 406 iterations are needed to generate a normal RSA key ($N = pq$) with the aid of two servers. The large number of iterations is a result of the quadratic slowdown discussed in Section 4. Even though each iteration is much faster than the corresponding value for local generation, we end up hurting the total generation time.

Our algorithms require only a few kilobytes of data transfer between the Pilot and the servers. The traffic generated is linear in the number of iterations which explains the large figure for two server normal key generation.

5.2.2 Generating various key sizes

From Table 2 we see that the total iteration time increases almost linearly with key size for dual server aided generation. Indeed, the dominant component of each iteration is sieving, which takes linear time as a function of the key size. The expected total time for generating the key is the product of the time-per-iteration and the expected-number-of-iterations.

Observe that the improvement over local generation is less significant for shorter keys than for

	512 bits			768 bits			1024 bits		
	num. iter.	time/ iter.(ms)	net traf.	num. iter.	time/ iter.(ms)	net traf.	num. iter.	time/ iter.(ms)	net traf.
Local unbal.	9.15	3,550		10.53	8,215		18.16	25,038	
Local norm.	18.3	3,550		21.1	8,215		36.32	25,038	
One serv. norm.	9.3	4,546	1,785	14.8	7,644	4,262	14.5	11,467	5,568
Two serv. unbal.	9	2,492	2,880	12.55	3,687	6,024	12.75	5,156	8,160
Two serv. norm.	26	4,364	9,984	119.7	6,560	68,947	406	8,720	311808

Table 2: Statistics for different key sizes

longer keys. For instance, for a 512 bit key, two servers improve performance by only 50%. For a 1024 bit key the improvement is a factor of 5. The reason is that for smaller keys, the primality test is less of a dominating factor in the running time per iteration (we use the same size sieve, 2048 primes, for all key sizes). Hence, reducing the exponentiation time has less of an effect on the the total time per iteration.

6 Conclusions

At the present using RSA on a low power handheld is problematic. In this paper we study whether RSA's performance can be improved without a loss of security. In particular, we ask whether an untrusted server can aid in RSA key generation. We wish to offload most of the work to the server without leaking any of the handheld's secrets.

We showed a significant improvement in the time it takes to generate an *unbalanced* RSA key. With the help of two isolated servers we obtained a speed up of a factor of 5. With the help of a *single* server we obtained a speed up of a factor of 2. For normal RSA keys, $N = pq$, we cannot improve the running time due do the *quadratic slowdown* problem discussed in Section 4. It is an open problem to speed up the generation of a normal RSA key using a single server. In all our algorithms the load on the server is minimal; our experiments show that even though the server is doing most of the work, the PalmPilot does not produce candidates fast enough to fully occupy the server. Our code available for anyone wishing to experiment with it.

Acknowledgments

We thank Certicom for providing us with SSL libraries for the PalmPilot.

References

[1] N. Asokan, G. Tsudik and M. Waidner, "Server-Supported Signatures", Journal of Computer Security, Vol. 5, No. 1, pp. 91–108, 1997.

[2] D. Balfanz, E. Felten, "Hand-Held Computers Can Be Better Smart Cards", to appear in the 8th USENIX Security Symposium.

[3] D. Boneh, N. Daswani, "Experimenting with electronic commerce on the PalmPilot", in proc. of Financial-Crypto '99, Lecture Notes in Computer Science, Vol. 1648, Springer-Verlag, pp. 1–16, 1999.

[4] M. Bellare, P. Rogaway, "Optimal asymmetric encryption – How to encrypt with RSA", in proc. Eurocrypt '94.

[5] H. Gilbert, D. Gupta, A. M. Odlyzko, and J.-J. Quisquater, "Attacks on Shamir's 'RSA for paranoids," Information Processing Letters 68 (1998), pp. 197–199.

[6] T. Matsumoto, K. Kato, H. Imai, "Speeding up secret computations with insecure auxiliary devices", In proc. of Crypto '88, Lecture Notes in Computer Science, Vol. 403, Springer-Verlag, pp. 497–506, 1998.

[7] A. Menezes, P. van Oorschot and S. Vanstone, "Handbook of Applied Cryptography", CRC Press, 1996.

[8] P. Nguyen, J. Stern, "The Beguin-Quisquater Server-Aided RSA Protocol from Crypto'95 is not secure", in proc. of AsiaCrypt '98, Lecture Notes in Computer Science, Vol. 1514, Springer-Verlag, pp. 372–380, 1998.

[9] Public Key Cryptography Standards (PKCS), No. 1, "RSA Encryption standard", `http://www.rsa.com/rsalabs/pubs/PKCS/`.

[10] R. Rivest, "Finding four million large random primes", In proc. of Crypto '90, Lecture Notes in Computer Science, Vol. 537, Springer-Verlag, pp. 625–626, 1997.

[11] A. Shamir, "RSA for paranoids", CryptoBytes, Vol. 1, No. 3, 1995.