Experimenting with Shared Generation of RSA Keys

Michael Malkin
Thomas Wu
Dan Boneh

Stanford University
*Supported by DARPA
Who generates the shared key?

The private key is never reconstructed!
Drawbacks:

- Single point of failure
- May have to destroy dealer afterwards
Distributed Generation

Advantages:
- Nobody ever knows the entire key
- No single point of failure
RSA Keys

- An n-bit modulus, \( N = pq \)
- The encryption (public) key
- The decryption (private) key

Sharing of \( d : d = d_1 + d_2 + d_3 \)

- Can apply key without reconstructing \( d \)

\( d \) is the secret

\( p \) or \( q \to d \)
Distributed Generation*

1. $p_1, q_1$
2. $p_2, q_2$
3. $p_3, q_3$

$p_i, q_i$ are $n/2$ bit integers

$N = (p_1 + p_2 + p_3) \cdot (q_1 + q_2 + q_3) = pq$

3. Biprimality Test

Nobody ever knows $p$ or $q$!

(*Boneh-Franklin)
**How Do They Compare?**

**Non-Distributed:**
- Pick prime $p$
- Pick prime $q$
- Multiply

**Distributed:**
- Pick $N$
- Hope $N = pq$ is an RSA modulus
- Can’t test $p$ and $q$ separately

*Distributed generation takes more iterations*
Main Results

Initial time: 2.5 hours
(1024-bit key)

- Distributed Sieving $\times 10$
- Multithreading $\times 6$
- Load Balancing $\times 1.3$
- Parallel Trial Division $\times 1.3$

Final time: 1.5 minutes
Minding Your $p$’s and $q$’s

- Bad $N \rightarrow$ probably divisible by 3 or 5 or 7 or …
- Idea: Ensure that $N$ isn’t divisible by any small primes

Distributed Sieving

- Can pick $p_i, q_i$ so that $p, q$ are not divisible by small primes
  … But nobody actually knows $p$ or $q$!
• Synchronous algorithm → synchronization delays
• Under-utilizing CPU — idle 80% of time

Multithreading

• 6 threads optimal for 1024-bit key
• Almost 6 times faster!
  (On 300Mhz Pentium II’s running Solaris 2.6)
Costly Biprimality Test

- Biprimality test involves time-consuming calculation
- Idea: Only one server needs to do this

Load Balancing

- A different server does test for each iteration
- Probabilistic load balancing
• What about small primes not covered by sieving?
• Trial division on $N$ by small primes

Parallel Trial Division

• Each server does trial division on different small primes
Private Key Generation

- Implemented method for small $e$
- In RSA usually use a small $e$

- After $N$ is found, generate $d_1$, $d_2$, and $d_3$ so:

$$d_1 + d_2 + d_3 = d$$

... But do this so that nobody ever knows $d$

- There is an additional way to share $d$
- Only a fraction of servers need to be active
Implementation: Config File

<table>
<thead>
<tr>
<th>Configuration Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num_Servers</td>
<td>3</td>
</tr>
<tr>
<td>Key_Length</td>
<td>Normal</td>
</tr>
<tr>
<td>Threads</td>
<td>2</td>
</tr>
<tr>
<td>TrialDiv_End</td>
<td>10000</td>
</tr>
<tr>
<td>Sieve</td>
<td>True</td>
</tr>
<tr>
<td>Test_Mode</td>
<td>True</td>
</tr>
<tr>
<td>Sequence_Numbers</td>
<td>True</td>
</tr>
<tr>
<td>Transport</td>
<td>sslv3</td>
</tr>
<tr>
<td>Share_IP_Port_0</td>
<td>8080</td>
</tr>
<tr>
<td>Server_IP_Addr_0</td>
<td>ittc.stanford.edu</td>
</tr>
<tr>
<td>Server_Sequence_File_0</td>
<td>com_security/seq0</td>
</tr>
<tr>
<td>Server_Cert_0</td>
<td>com_security/cert_s0.pem</td>
</tr>
<tr>
<td>Server_Key_0</td>
<td>com_security/key_s0.pem</td>
</tr>
</tbody>
</table>
Implementation: COM

• Abstraction layer
• Fault tolerance - non-blocking I/O
• Private, authenticated channels
  • Based on SSLeay

• Authenticates share servers using a server certificate:

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

- Stored as PEM-encoded ASN.1 format

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Version</td>
</tr>
<tr>
<td>Integer</td>
<td>$N$</td>
</tr>
<tr>
<td>Integer</td>
<td>$e$</td>
</tr>
<tr>
<td>Integer</td>
<td>$k$</td>
</tr>
<tr>
<td>Integer</td>
<td>$d_1$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Integer</td>
<td>$d_k$</td>
</tr>
</tbody>
</table>
Performance

<table>
<thead>
<tr>
<th>Key Size</th>
<th>Threads</th>
<th>Primality Tests</th>
<th>Iterations</th>
<th>Total Time</th>
<th>Network Traffic</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 bit</td>
<td>2</td>
<td>36</td>
<td>119</td>
<td>0.15 min</td>
<td>0.18 Mb</td>
</tr>
<tr>
<td>1024 bit</td>
<td>6</td>
<td>49</td>
<td>130</td>
<td>1.5 min</td>
<td>1.16 Mb</td>
</tr>
<tr>
<td>2048 bit</td>
<td>6</td>
<td>234</td>
<td>495</td>
<td>18 min</td>
<td>7.48 Mb</td>
</tr>
</tbody>
</table>

On three 300Mhz Pentium II’s running Solaris 2.6

- Network bandwidth is reasonable
- 1024-bit works well
- 2048-bit is reasonable
**Effect of Number of Servers**

Time to generate a 1024-bit RSA key

**WAN:**
- Two servers at Stanford
- One server at University of Wisconsin at Madison
- Difficult to find PC’s running Solaris
Effect of Threads

- Synchronization/CPU tradeoff
- Minimize time with 6 threads

*Generating a 1024-bit RSA key
Effect of Distributed Sieving

- Sieve bound is largest prime sieved
- Larger sieve → fewer iterations
- Diminishing returns

*Generating a 512-bit RSA key*
Conclusions

- Distributed key generation is practical:
  - 1.5 minutes for 1024-bit key

- Several practical improvements to algorithm
  - Distributed Sieving
  - Multithreading
  - Load Balancing
  - Parallel Trial Division

- Optimized cryptographic algorithm
  - Requires security proofs

http://theory.stanford.edu/~dabo/ITIC