Simple Backdoors for RSA Key Generation

Claude Crépeau[1] and Alain Slakmon[2]

[1] School of Computer Science, McGill University,
3480 rue University, room 318, McConnell Eng. Bldg,
Montréal (Québec), Canada H3A 2A7
e-mail: `crepeau@cs.mcgill.ca`
[2] Département de mathématiques,
Collège de Bois-de-Boulogne,
10555 avenue de Bois-de-Boulogne,
Montréal (Québec), Canada H4N 1L4
e-mail: `Alain.Slakmon@bdeb.qc.ca`

Abstract. We present extremely simple ways of embedding a backdoor in the key generation scheme of RSA. Three of our schemes generate two genuinely random primes p and q of a given size, to obtain their public product $n = pq$. However they generate private/public exponents pairs (d, e) in such a way that appears very random while allowing the author of the scheme to easily factor n given only the public information (n, e). Our last scheme, similar to the PAP method of Young and Yung, but more secure, works for any public exponent e such as $3, 17, 65537$ by revealing the factorization of n in its own representation. This suggests that nobody should rely on RSA key generation schemes provided by a third party.

1 Introduction

As we all know, the RSA public-key Cryptosystem and Digital signature schemes are now in the public domain, which means that anybody may include them as means of confidentiality and authenticity in software products, smartcards, etc. The question that we raise here is how much can a user tell that an implementation of RSA he uses is safe and actually protects him? Recall the (in)famous "NSA-KEY" incident of Microsoft's CryptoAPI system. How easy is it for software developers and smartcard builders to embed a backdoor in their RSA key generation scheme that will be unnoticed by the user but allows the author to defeat the confidentiality and authenticity of the resulting RSA public-key Cryptosystem and Digital signature scheme ?

We present extremely simple ways of embedding a backdoor in the key generation scheme of RSA. Three of our schemes generate two genuinely random primes p and q of a given size, to obtain their public product $n = pq$. However they generate private/public exponents pairs (d, e) in such a way that appears very random while allowing the author of the scheme to easily factor n given only the public information (n, e). Our fourth scheme, similar to the PAP method of Young and Yung, but more secure, works for any public exponent

e such as $3, 17, 65537$ by embedding a backdoor to the factorization of n in its own representation. Our methods will modify only slightly the running time of the standard key generation process and thus will be unnoticeable from a timing point of view. Moreover, we conjecture that backdoored keys will be distributed in such a way that even with large samples they will remain indistinguishable from genuinely random keys. Our schemes do not expect the generation algorithm to have memory of its previous executions. We assume our algorithms are ran on a memoryless device, thus discarding even simpler methods based on pseudorandom generation that would be deterministically generating keys in a way reproducible by the software developer. Our methods use randomization in a way to generate a large set of keys, each of which is breakable by the developer.

For this purpose, one scheme relies on the well known attacks on RSA small private exponents by Wiener and Boneh and Durfee. The scheme creates a random weak pair of private/public exponents (δ, ϵ) with small δ, and transforms it into a random looking private/public exponents (d, e) which are related to (δ, ϵ) in a secret way that the author of the generation scheme may invert. From public knowledge of (e, n) only, the author may recover ϵ, break the easy instance (ϵ, n) and discover δ. Factorization of n is easily obtained using standard techniques, once δ, ϵ are known.

Our next two schemes rely on the more recent attacks on RSA small public exponents given parts of the private exponent by Boneh, Durfee and Frankel. The schemes create a random weak pair of private/public exponents (δ, ϵ) with small ϵ, and transforms this ϵ and parts of the corresponding δ into a random looking private/public exponents (d, e) which are related to (δ, ϵ) in a secret way that the author of the generation scheme may invert. From public knowledge of (e, n) only, the author may recover ϵ and parts of δ, break the easy instance (ϵ, n) given parts of δ and discover the whole δ. Factorization of n can easily be obtained using standard techniques, once δ, ϵ are known.

Our last scheme relies on the possibility of hiding at least half the bits of p in the representation of n. Factoring n is possible using Coppersmith's method once half the bits of p are recovered.

Our first scheme generates (d, e) pairs such that $|e| \approx |n|$. However, our second scheme generates (d, e) pairs such that $|e| \approx |n|/2$ while the third generates (d, e) pairs such that $|e| \approx |n|/4$. Our last scheme generates (d, e) pairs for any e of arbitrary size.

This suggests that nobody should rely on RSA key generation schemes provided by a third party. This is most striking in the smartcard model, unless some guarantees are provided that all such attacks to key generation cannot have been embedded. Restriction to RSA moduli with predetermined portions as proposed by Lenstra would not be of any help to prevent our methods. Even in software implementations, unless the actual source code is provided, it is non trivial to find out exactly what the key generation mechanism is. One can easily imagine that software companies who want to keep their code secret for market advantage will not make it easy to decompile their programs to make sense of their implementation know-how.

2 Reminders and Relation to Other Work

The RSA cryptosystem and digital signature schemes [10] are based on the generation of two random primes p, q of roughly equal size and generation of random exponents d, e such that $de \equiv 1 \pmod{\phi(n)}$, where $n = pq$.

Algorithm 2.1 (RSA key generation)

```
Pick random primes $p, q$ of the right size, let $n := pq$ be a $k$-bit integer.
repeat
    Pick a random odd $e$ such that $|e| \leq k$.
until $\gcd(e, \phi(n)) = 1$.
Compute $d := e^{-1} \bmod \phi(n)$.
return$(p, q, d, e)$.
```

The pair (n, e) may be made publicly available so that the function $m^e \bmod n$ be used for encryption, whereas $c^d \bmod n$ be used for decryption of messages m, c, $1 \leq m, c \leq n - 1$. Similarly, the private function $m^d \bmod n$ may be used to produce a signature c, whereas $c^e \bmod n$ may be compared to m as a signature verification procedure of messages m, c, $1 \leq m, c \leq n - 1$.

Wiener [13] demonstrated that small private exponents may be efficiently recovered if $d < n^{.25}/3$ and this result was recently improved by Boneh and Durfee [1] who showed a similar result for $d < n^{.292}$. Moreover, it is a well known fact [8] that given a multiple of $\phi(n)$ such as $de - 1$ satisfying $de \equiv 1 \pmod{\phi(n)}$, it is easy to factor n.

Boneh, Durfee and Frankel [2] recently demonstrated two interesting results allowing to recover the whole of d given a small e, n and parts of d. Let $n = pq$ such that $p/q < 4$ be an RSA moduli. We use the following two theorems from their work to construct our schemes:

Theorem 1 ([2], Theorem 1.2, part 1). *Let t be an integer in the range $[|n|/4, ..., |n|/2]$ and e be a prime in the range $[2^t, ..., 2^{t+1}]$. Suppose we are given (n, e), and the t most significant bits of d. Then we can compute the whole of d and factor n in time poly$(|n|)$.*

Theorem 2 ([2], Theorem 4.6). *Let t be an integer in the range $[1, ..., |n|/2]$ and e be an integer in the range $[2^t, ..., 2^{t+1}]$. Suppose we are given (n, e), the t most significant bits of d, and the $|n|/4$ least significant bits of d. Then we can factor n in time poly$(|n|)$.*

Some quite interesting results by Joye, Paillier and Vaudenay [6] may be used to speed up prime generation at Step 1 of the key generation protocols of section 3 since the primes are not chosen according to particular rules such as those in Section 5. Another useful sequence of result is the proof by Rivest and Shamir [9] that the $|n|/3$ most significant bits of p are sufficient to factor n efficiently, and an improvement due to Coppersmith [3] reducing the number of required bits to $|n|/4$.

De Weger [5] and Slakmon [11] have considered the algorithm of Wiener in a context where the primes p, q are partially known. In particular, we use the following result of Slakmon :

Theorem 3 ([11], Proposition 3.2.1). *Let t be an integer in the range $[1, ..., |n - \phi(n)|]$ and d be an integer in the range $[1, ..., 2^{|n-\phi(n)|-t/2}]$. Suppose we are given (n, e), and the $|n - \phi(n)| - t$ most significant bits of $n - \phi(n)$. Then we can factor n in time poly$(|n|)$.*

Our schemes are in the line of work by Young and Yung [14–16] on kleptography. Our schemes are different from theirs and involve a minimal amount of extra calculations to maintain the key generation time roughly the same as an honest RSA key generation scheme. Note however that our schemes of Section 5 are very similar to the PAP method of [14]. Although the PAP scheme would be foiled by the methods of Lenstra [7] to force certain bits of n to be chosen by the user, our schemes will resist to these countermeasures.

2.1 The Scenario

We assume that a legitimate user (called the distinguisher) is given access to the RSA key generation process as a black-box, where he is not allowed to see the code of the generator. However he can sample output tuples (e, d, p, q) from the generator to his will.

The source code of a valid RSA key generator is provided to the distinguisher as well as the code of our cheating generators, except for a secret key, unspecified in the code. The task of the distinguisher is to figure out which is which only from sample outputs of these generators and from running time analysis.

Our goal is to make this distinguishing task as difficult as possible with the simplest and most efficient modification to the standard key generation mechanism. To make this task more significant, we allow the distinguisher extra powers not usually provided to users of such key generation schemes. Our distinguisher may

1. keep one prime out of p, q and request the other one afresh
2. keep the primes p, q and request several valid pairs of exponents d, e.

We believe that despite these extra powers, our cheating mechanisms remain indistinguishable in output distribution and in (approximate) running time.

Notations We use the column ":" for concatenation. Let $m\rfloor_\ell$ be the ℓ least significant bits of integer m, and similarly $m\rceil^\ell$ be the ℓ most significant bits of integer m.

3 Hidden Exponent Key Generation Algorithms

3.1 Hidden Small Private Exponent δ

The basis of our "cheating" RSA-HSD$_\beta$ key generation scheme is to imbed a backdoor β in the scheme and use it to hide instances of small values of the private exponent δ. This is done as described in the following protocol, using an unspecified permutation π_β of odd integers smaller than n to themselves. Remember that $|n|$ is a k-bit integer. We discuss several classes of simple choices for π_β in Section 4.

Algorithm 3.1 (RSA-HSD$_\beta$ key generation)

```
1: Pick random primes $p, q$ of the appropriate size, set $n := pq$.
2: repeat
3:   Pick a random odd $\delta$ such that $\gcd(\delta, \phi(n)) = 1$ and $|\delta| \leq k/4$.
4:   Compute $\epsilon := \delta^{-1} \bmod \phi(n)$; $e := \pi_\beta(\epsilon)$.
5: until $\gcd(e, \phi(n)) = 1$.
6: Compute $d := e^{-1} \bmod \phi(n)$.
7: return$(p, q, d, e)$.
```

The instances produced by the above key generation scheme satisfy all properties required except for the fact that d, e are not entirely random, but are only random within a smaller set of possibilities specified by the images through π_β of inverses modulo $\phi(n)$ of small exponents δ. Notice that Step 3 may be made non-repetitive at the price of biasing the distribution of the δ's toward the smaller ones by setting $\delta := \delta / \gcd(\delta, \phi(n))$.

In terms of running time, this algorithm compares very well with the standard RSA key generation Algorithm 2.1: the running time of Steps 1 and 6 are identical to the original whereas the loop from Steps 2 to 5 will run a number of times roughly equal to the original loop; the number of gcd calculations inside the loop is about three time as much as the original (that is if we use the non-repetitive trick suggested above). As long as the computation of $\pi_\beta(\epsilon)$ is negligible with respect to calculations such as gcds, the difference in running time may be made quite negligible.

Once n, e are made public n may be factored as follows, given the secret backdoor β.

Algorithm 3.2 (RSA-HSD$_\beta$ attack (n, e))

```
1: Given $(n, e)$, compute $\epsilon := \pi_\beta^{-1}(e)$.
2: Compute $\delta$ from $(n, \epsilon)$ using Wiener's low exponent attack.
3: Given $(\epsilon, \delta)$ factor $n$ as p,q.
4: return$(p, q)$.
```

At extra cost in the attack, larger values of δ may be used, up to $n^{0.292}$ using the Boneh-Durfee cryptanalytic attack instead of Wiener's.

The main drawback of this method is that the generated e's will have roughly full size $|e| \approx |n|$. This means that any restriction on the size of e ($|e| < c|n|$ for $c < 1$) would foil the attack.

3.2 Hidden Small Prime Public Exponent ϵ

The basis of our "cheating" RSA-HSPE$_\beta$ key generation scheme is to imbed a backdoor β in the scheme and use it to hide instances of small values of the public prime exponent ϵ together with some partial information about the corresponding private exponent δ. This is done as described in the following protocol, using an unspecified permutation π_β of odd integers smaller than n. Remember that $|n|$ is a k-bit integer. We discuss several classes of simple choices for π_β in Section 4.

Algorithm 3.3 (RSA-HSPE$_\beta$ key generation)

```
1: Pick random primes $p, q$ of the appropriate size, set $n := pq$.
2: repeat
3:     Pick a prime $\epsilon$ such that $\gcd(\epsilon, \phi(n)) = 1$ and $|\epsilon| = k/4$.
4:     Compute $\delta := \epsilon^{-1} \bmod \phi(n)$; $\delta_H := \delta\rceil^{\frac{k}{4}}$; $e := \pi_\beta(\delta_H : \epsilon)$.
5: until $\gcd(e, \phi(n)) = 1$.
6: Compute $d := e^{-1} \bmod \phi(n)$.
7: return$(p, q, d, e)$.
```

The instances produced by the above key generation scheme satisfy all properties required except for the fact that d, e are not entirely random, but are only random within a smaller set of possibilities specified by the images through π_β of concatenations of δ_H, ϵ of small prime public exponents ϵ.

The size of the concatenations $(\delta_H : \epsilon)$ produced are $k/2$. Therefore, using extra random padding, we have freedom to generate exponents e in the range $\sqrt{n} < e < \phi(n)$.

In terms of running time, this algorithm compares poorly with the standard RSA key generation Algorithm 2.1: on one hand, the running time of Steps 1 and 6 are identical to the original while the loop from Steps 2 to 5 will run a number of times roughly equal to the original loop; the number of gcd calculations inside the loop is about three time as much as the original (since ϵ is prime). On the other hand, unfortunately, generating prime ϵ is going to be time consuming despite the fact that $|e| = |n|/4$.

The main advantage of this method is its simplicity and the fact that the keys produced may be as small as $|n|/2$.

Once n, e are made public n may be factored as follows, given the secret backdoor β.

Algorithm 3.4 (RSA-HSPE$_\beta$ attack (n, e))

```
1: Given $(n, e)$, compute $(\delta_H : \epsilon) := \pi_\beta^{-1}(e)$.
2: Compute $\delta$ from $(n, \delta_H, \epsilon)$ using BDF low public prime exponent attack (Theorem 1) with partial knowledge of private exponent.
3: Given $(\epsilon, \delta)$ factor $n$ as p,q.
4: return$(p, q)$.
```

3.3 Hidden Small Public Exponent ϵ

The basis of our "cheating" RSA-HSE$_\beta$ key generation scheme is to imbed a backdoor β in the scheme and use it to hide instances of small values of the public exponent ϵ together with some partial information about the corresponding private exponent δ. This is done as described in the following protocol, using an unspecified permutation π_β of odd integers smaller than n to themselves. Remember that $|n|$ is a k-bit integer. We discuss several classes of simple choices for π_β in Section 4. Let t be an integer in the range $[1, ..., |n|/2]$.

Algorithm 3.5 (RSA-HSE$_\beta$ key generation)

```
1: Pick random primes $p, q$ of the appropriate size, set $n := pq$.
2: repeat
3:    Pick a random $\epsilon$ such that $\gcd(\epsilon, \phi(n)) = 1$ and $|\epsilon| = t$.
4:    $\delta := \epsilon^{-1} \bmod \phi(n)$; $\delta_H := \delta\rceil^t$; $\delta_L := \delta\rfloor_{\frac{k}{4}}$; $e := \pi_\beta(\delta_H : \delta_L : \epsilon)$.
5: until $\gcd(e, \phi(n)) = 1$.
6: Compute $d := e^{-1} \bmod \phi(n)$.
7: return$(p, q, d, e)$.
```

The instances produced by the above key generation scheme satisfy all properties required except for the fact that d, e are not entirely random, but are only random within a smaller set of possibilities specified by the images through π_β of concatenations of $\delta_H, \delta_L, \epsilon$ of small public exponents ϵ. Notice that Step 3 may be made non-repetitive at the price of biasing the distribution of the ϵ's toward the smaller ones by setting $\epsilon := \epsilon / \gcd(\epsilon, \phi(n))$.

However, to avoid detection, it is necessary to randomize (or discard) the $\ell \geq 2$ least significant bits of δ_L where $2^\ell | \phi(n)$ and $2^{\ell+1} \nmid \phi(n)$. This is because $\epsilon\delta \equiv 1 \bmod 2^\ell$ and therefore $\delta\rfloor_\ell$ is always the inverse modulo 2^ℓ of $\epsilon\rfloor_\ell$. All such obvious redundancy must be removed in order to allow the permutation π_β to remain simple and fast to compute. Another such example is that the most significant bits of δ_H are not uniformly distributed because they come from an integer modulo n, where n is publicly known. It is certainly better to specify δ_H as a binary sequence describing the position of δ with respect to n (as in a binary search between 1 and n).

In general, if we request at Step 3 that $|\epsilon| = t$ for $t \in [1, ..., k/2]$, the total size of the concatenated input $(\delta_H, \delta_L, \epsilon)$ is $2t + k/4$. Asymptotically if we set $t := \gamma k$ for some small $\gamma > 0$ the size is $(1/4 + 2\gamma)k \approx k/4$. Therefore, using extra random padding or by using a larger t, we have freedom to generate exponents e in the range $\sqrt[4]{n} < e < \phi(n)$. Notice however that despite the fact that γ vanishes asymptotically, one should make sure that γk be at least, say 80, to prevent brute force attacks.

In terms of running time, this algorithm compares very well with the standard RSA key generation Algorithm 2.1: the running time of Steps 1 and 6 are identical to the original whereas the loop from Steps 2 to 5 will run a number of times roughly equal to the original loop; the number of gcd calculations inside the loop is about three time as much as the original (that is if we use the non-repetitive trick suggested above). As long as the computation of $\pi_\beta(\epsilon)$ is negligible with respect to calculations such as gcds, the difference in running time may be made quite negligible.

Once n, e are made public, n may be factored as follows, given the secret backdoor β.

Algorithm 3.6 (RSA-HSE$_\beta$ attack (n, e) **)**

```
Given $(n, e)$, compute $(\delta_H : \delta_L : \epsilon) := \pi_\beta^{-1}(e)$.
Compute $\delta$ from $(n, \delta_H, \delta_L, \epsilon)$ using BDF low public exponent attack (Theorem 2) with partial knowledge of private exponent.
Given $(\epsilon, \delta)$ factor $n$ as p,q.
return$(p, q)$.
```

4 Choices of π_β

Our main simple and very easy to compute permutation is

$$\pi_\beta(x) = x \oplus (2\beta)\rfloor_{|x|} .$$

It appears sufficient for schemes of Section 3 in the sense that the instance spaces are sufficiently large that even if a distinguisher tries to discover the fact that our schemes have been used they will likely fail. In scheme RSA-HSD for instance, the distinguisher is able to compute the XOR of ϵ's corresponding to small δ's by computing the XOR of the corresponding e's, but those will look very random.

Similarly in scheme RSA-HSE, XORing e's together is likely to look very random as long as they have been processed as described above, to remove obvious redundancy. In scheme RSA-HSPE, the XOR of e's will yield the XOR of primes ϵ's but this too is random enough to be indistinguishable.

4.1 More Examples

Of course one can always rely on different cryptosystems to generate efficient π's, for instance

$$\pi_\beta(x) = \mathrm{DES}_\beta(x) \text{ or } \pi_\beta(x) = \mathrm{AES}_\beta(x)$$

may be used. However it seems more desirable to use permutations π that use the same kind of arithmetic as RSA itself since these operation are readily available for normal key generation. It may be a problem in a smartcard scenario to use program space to implement DES or AES.

When working with fixed sizes k bit information to permute, a very simple way to create randomization is to choose β as a prime in the range $2^{k-1} \pm 2^{k/2}$ and then define

$$\pi_\beta(x) = x^{-1} \bmod \beta$$

which is computed with a single extended gcd calculation.

4.2 Permutations Using Operations Modulo $n + 1$

Another example uses an even translation of the odd exponents x modulo an even number such as $n + 1$. Let N be an upper bound on all the n's produced by the key generation scheme, and let β be a fixed parameter, picked at random such that $N \leq \beta \leq 2N$. The permutation

$$\pi_\beta(x) = (x + 2\beta) \bmod (n + 1)$$

maps the odd integers modulo $n+1$ to themselves. Notice that the probability that this permutation maps an element to a value greater than $\phi(n)$ is negligible, since $(n + 1) - \phi(n) = p + q$ which is exponentially small with respect to $\phi(n)$. This permutation is different for each n and so makes it even harder to notice the cheat. However, this specific permutation is not a good choice in the context of Section 3.1 since Vaudenay [12] found a way to identify our RSA-HSD generated keys within 24 hours of posting of our proposal on the web [4] !

This can be generalized in several directions using several extra hidden parameters. First notice that $n + 1 - 2\sqrt{n}$ is always an upper bound on $\phi(n)$ and thus

$$\pi_{\beta,\mu}(x) = (x + 2\beta) \bmod (n + 1 - 2m)$$

may also be used, where $m := \mu \bmod \left\lfloor \sqrt(n) \right\rfloor$ for any fixed μ, such that $\sqrt{N} \leq \mu \leq 2\sqrt{N}$. In other words, μ is an arbitrary constant at least half the size of the largest n's we want to generate.

Notice, however, that generalizing expression $e + 2\beta$ to affine functions using yet another secret parameter α, $1 \leq \alpha \leq N$:

$$\pi_{n,\alpha,\beta,\mu}(e) = ((2\alpha + 1)e + 2\beta) \bmod (n + 1 - 2m)$$

will cause a problem if $\gcd(2\alpha + 1, \phi(n), n + 1 - 2m) > 2$. In this case, given two different sets of exponents, but with the same modulus (p, q, e, d) and (p, q, e', d'), a user could compute $e' - e$ which is the same as $(2\alpha + 1)(\epsilon' - \epsilon)$ up to a multiple of $(n + 1 - 2m)$. The user could notice that $\gcd(e' - e, \phi(n)) > 2$ all the time which is unusual, despite the fact that μ and β are unknown.

4.3 Discussion: Avoiding and Securing Hidden Exponent Attacks

Of course, a very simple strategy will foil these attacks : make sure d (or e) is picked from an exponentially large subset S of possible values which is only an exponentially small fraction of all the d's, such that $\gcd(d, \phi(n)) = 1$. For example, consider the set of valid private exponents to be $S = \{d | \gcd(d, \phi(n)) = 1 \text{ and } d < \sqrt{n}\}$. It seems quite unlikely that one can find a simple permutation $\pi_\beta(e)$ that frequently maps inverses modulo $\phi(n)$ of d's, $d < n^{1/4}$, to inverses inverses modulo $\phi(n)$ of d's in S, in a random looking fashion. But, who knows really if such a thing is impossible ?

Bad choices of S can be no help to foil the attack. For example, $S = \{d|\, \gcd(d, \phi(n)) = 1 \text{ and } (d^{-1} \bmod \phi(n)) < \sqrt{n}\}$ is easily foiled by methods of Sections 3.2 and 3.3.

Alternatively, forcing some redundancy onto d's may help foil the attack. For instance,$S = \{d|\, \gcd(d, \phi(n)) = 1 \text{ and } d = (x : x)\}$ where x is a half size odd number, seems a good countermeasure to our hidden exponent schemes.

The indistinguishability of resulting schemes depend extensively on the permutation chosen but in many cases the simpler ones seem to suffice. Notice also that the security of our three schemes decrease as they produce e's of smaller and smaller size. Indeed, even more relevant is the fact that the set of possible e's gets smaller from the first to the third scheme. The size of the valid e's set is an important factor of security.

Finally, notice that the last of the three schemes, which produces the smallest e's may involve extra weaknesses similar to those already exposed because of the redundancy of the lsb's of ϵ and the corresponding lsb's of δ. Further analysis of this redundancy should be performed. It could otherwise lead to countermeasures.

5 Hidden Prime Factor

Our last proposal is very similar to the PAP (Pretty-Awful-Privacy) of [14] but we address and solve a number of deficiencies left unnoticed or unresolved by their scheme. We investigate the idea of imbedding some bits of the prime factor p in the product $n = pq$, thus choosing q to be a special prime satisfying a number of constraints. However, as long as the key β remains secret it should be hard to tell from the distribution of p, q, n's produced that cheating is going on. Our proposal differs from PAP in two major ways:

- After [3] we only hide the half most significant bits of p
- We make sure the distribution of numbers n, p, q is similar to the honest one.

The PAP method generates numbers n where the most significant bits are uniformly distributed which is not the proper distribution for products of two randomly selected (prime) integers of a fixed size. For instance, if one picks two random (prime) integers of 512 bits each, their product will be 1023 bits long with probability 38% whereas with probability 48% it will be 1024 bits long with leading bits "10" and with probability 14% it will be 1024 bits long with leading bits "11". This issue was ignored by [14] . Of course this problem does not happen if p and q are picked over the more appropriate interval $[\sqrt{2} \times 2^{511}, ..., 2^{512} - 1]$, but we see that the distribution of p, q influences the distinguishability of the result.

Let e be some fixed public exponent such as $3, 17, 65537$, etc, for which an appropriate n must be found. Our scheme RSA-HP$_\beta$ proceeds as follows:

```
Algorithm 5.1 ( RSA-HP$_\beta(e)$ key generation )

1: Pick a random prime $p$ of the right size, s.t. $\gcd(e, p-1) = 1$.
2: Pick a random odd $q'$ of the appropriate size, set $n' := pq'$.
3: Compute $\tau := n'\rceil^{\frac{k}{8}}$, $\mu := \pi_\beta(p\rceil^{\frac{k}{4}})$ and $\lambda := n'\rfloor_{\frac{5k}{8}}$.
4: Set $n := (\tau : \mu : \lambda)$ and $q := \lfloor n/p \rfloor + (1 \pm 1)/2$ so that it is odd.
5: while $\gcd(e, q-1) > 1$ or $q$ is composite do
     – Pick a random even $m$ such that $|m| = \frac{k}{8}$,
     – Set $q := q \oplus m$ and $n := pq$.
6: Compute $d := e^{-1} \bmod \phi(n)$.
7: return$(p, q, d, e)$.
```

The instances produced by the above key generation scheme is the product of a truly random p and a somewhat random q such that

- the top $k/8$ bits of n have the correct distribution of such a product
- the next $k/4$ bits of n are an "encryption" of the $k/4$ most significant bits of p
- the least $k/8$ bits of q are randomly chosen so that q is prime.

The running time of the above algorithm is more or less the same as the standard algorithm where p and q are individually picked at random until they are prime. Prime p is produced exactly in the same way, whereas prime q is picked according to the method of Step 5, which on average takes the same number of steps as Step 1. All other extra computations are negligible with respect to a single primality test.

```
Algorithm 5.2 ( RSA-HP$_\beta$ attack $(n, e)$ )

1: Given $n$, compute $p\rceil^{\frac{k}{4}} := \pi_\beta^{-1}(n\rceil^{\frac{3k}{8}}\rfloor_{\frac{k}{4}})$.
2: Factor $n$ as $p$,$q$ using Coppersmith's partial information attack.
3: return$(p, q)$.
```

Unlike the methods of Section 3, the simple permutation $\pi_\beta(x) = x \oplus (2\beta)\rfloor_{|x|}$ is definitely not secure; upon receiving two pairs (p, q) and (p', q') generated as above, one can easily check that

$$(n' \oplus n)\rceil^{\frac{3k}{8}}\rfloor_{\frac{k}{4}} = (p' \oplus p)\rceil^{\frac{k}{4}}$$

which should not happen normally. For this method we recommend permutations from Sections 4.1 and 4.2. Again the last permutation of Section 4.1 $\pi_\beta(x) = x^{-1} \bmod \beta$ is definitely not secure by itself; upon receiving a pair (p, q) generated as above, one can easily compute

$$n\rceil^{\frac{3k}{8}}\rfloor_{\frac{k}{4}} p\rceil^{\frac{k}{4}} - 1$$

which should be a multiple of the secret prime β. Running this experiment several times will lead to several multiples of β and a simple gcd calculation will yield β. Notice however that if $p\rceil^{\frac{k}{4}}$ is padded with a large enough number of extra random bits, the above attack is foiled.

Our favorite permutation is computing of a modular inverse mod a fixed predetermined prime near $2^{\frac{k}{4}}$ as proposed at the end of Section 4.1 *and* XORing with a fixed string:

$$\pi_{\beta,\mu}(x) = \Big(x \oplus (2\mu)\rfloor_{|x|}\Big)^{-1} \bmod \beta \quad \text{or} \quad \pi_{\beta,\mu}(x) = \big(x^{-1} \bmod \beta\big) \oplus (2\mu)\rfloor_{|\beta|}$$

which seem to foil both attacks presented above.

The unfortunate drawback of this method is that, while the first prime p can be picked according to any rule, the second prime q is picked according in a way that will not modify too much of its bits, once a first approximation is found. This opens the door to some attacks that may use this deficiency. Also it requires that a more elaborate encryption be used to hide the half of p in n. If by accident a "bad" prefix is selected, the amount of attempts to reach a prime may be very high. However on average the number of such attempts is the same as standard generation. Moreover we must find a valid portion of n that is uniformly distributed as long as p and q are picked according to any particular distribution. In the above example, we make the assumption that despite the fact that the first few and last bits of n are not uniformly distributed, the $k/4$ positions past the first $k/8$ are.

5.1 Combining with Exponent Method

Using the result of Slakmon (Theorem 3) we combine the above method with the hidden small exponent method with larger exponents ! For instance, if n is used to subliminally transmit the $|n|/6 \approx |n-\phi(n)| - |n|/3$ most significant bits of $n-\phi(n)$ (instead of $|n|/4$ msb's of p required by Coppersmith's method), then a variation of Wiener's method is able to recover hidden exponents d up to size $|n-\phi(n)| - |n|/3/2 \approx |n|/3$ which is better than both Wiener and Boneh-Durfee methods. Thus exponents d up to $n^{0.333}$ may be used and broken.

Increasing the size of acceptable d's increases the security of the hidden small exponent method and reducing the amount of information transmitted subliminally through n increases the security of the hidden prime method. However, the resulting method suffers the deficiencies of both methods.

5.2 Discussion: Avoiding and Securing Hidden Prime Attacks

Providing constraints in the way of Lenstra's work [7] do not seem to be an effective way of stopping the attack. Our method offers sufficient freedom in the choice of p and q that fixing the msb's or lsb's of n is not strong enough to foil our attack.

However, if we try to reduce the running time to be comparable to the primes generation algorithm of Joye, Paillier and Vaudenay [6] our method falls short because we could not find a way to generate p *and* q efficiently using their method while subliminally sending through n some portions of the bits of p.

An important issue to secure the hidden prime methods is to identify parts of n that are sufficiently uniformly distributed when p and q are picked from their respective distributions. Making very strict restrictions on the distributions of p and q can make that task very difficult.

6 Conclusions

We have introduced in Sections 3 and 5 a variety of *simple* backdoors that can be used to generate apparently normal RSA keys (p, q, d, e) in such a way that the owner of a secret key imbedded in the generation scheme may recover the private primes p and q from the public information e, n. Each of these schemes use (n, e) as a subliminal channel to carry special information useful to factor n. The security of the subliminal channel is parametrized by the choice of a secret mapping π_β. Several possibilities of π_β were proposed in Section 4.

We are well aware that no proof of security of our schemes have been provided or even hinted. Indeed, introducing a backdoor is somewhat like introducing a new computational assumption. Only time will tell whether these backdoors resist to cryptanalysis. The scheme of Section 3.1 is the only one that has been made public so far (on a web page) and has survived cryptanalysis for more than a year.

We challenge the cryptology community to break the several schemes/permutations possibilities proposed in this paper.

Acknowledgements

We thank Dan Boneh, Don Coppersmith, Jean-Marc Robert, Serge Vaudenay, and Moti Yung for helpful discussions and for breaking some early schemes. We are grateful to Martin Courchesne and Simon Wong for their web programming assistance.

References

1. D. Boneh and G. Durfee, *Cryptanalysis of RSA with private key d less than* $n^{0.292}$, Information Theory, IEEE Transactions on, 46 (2000), pp. 1339–1349.
2. D. Boneh, G. Durfee, and Y. Frankel, *An attack on RSA given a small fraction of the private key bits*, in Advances in Cryptology - AsiaCrypt '98, K. Ohta and D. Pei, eds., Berlin, 1998, Springer-Verlag, pp. 25–34. Lecture Notes in Computer Science Volume 1514.
3. D. Coppersmith, *Finding a small root of a bivariate integer equation; factoring with high bits known*, in Advances in Cryptology - EuroCrypt '96, U. Maurer, ed., Berlin, 1996, Springer-Verlag, pp. 178–189. Lecture Notes in Computer Science Volume 1070.

4. C. Crépeau and S. Wong, *The RSA hidden small exponent method*, in `http://crypto.cs.mcgill.ca/~crepeau/RSA`, 2001.
5. B. de Weger, *Cryptanalysis of RSA with small prime difference*, Applicable Algebra in Engineering, Communication and Computing, 13 (2002), pp. 17–28.
6. M. Joye, P. Paillier, and S. Vaudenay, *Efficient generation of prime numbers*, in CHES 2000, Ç. K. Koç and C. Paar, eds., Berlin, 2000, Springer-Verlag, pp. 340–354. Lecture Notes in Computer Science Volume 1965.
7. A. K. Lenstra, *Generating RSA moduli with a predetermined portion*, in Advances in Cryptology - AsiaCrypt '98, K. Ohta and D. Pei, eds., Berlin, 1998, Springer-Verlag, pp. 1–10. Lecture Notes in Computer Science Volume 1514.
8. G. L. Miller, *Riemann's hypothesis and tests for primality*, J. Comput. System Sci., 13 (1976), pp. 300–317.
9. R. L. Rivest and A. Shamir, *Efficient factoring based on partial information.*, in Advances in Cryptology - EuroCrypt '85, F. Pichler, ed., Berlin, 1985, Springer-Verlag, pp. 31–34. Lecture Notes in Computer Science Volume 219.
10. R. L. Rivest, A. Shamir, and L. M. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Comm. ACM, 21 (1978), pp. 120–126.
11. A. Slakmon, *Sur des méthodes et algorithmes de factorisation et leur application en cryptologie*, Master's thesis, Université de Montréal, dépt. IRO, 2000.
12. S. Vaudenay, *Private e-mail communication.*, 2 may 2001.
13. M. Wiener, *Cryptanalysis of short RSA secret exponents*, Information Theory, IEEE Transactions on, 36 (1990), pp. 553–558.
14. A. Young and M. Yung, *The dark side of "black-box" cryptography, or: Should we trust Capstone?*, in Advances in Cryptology - Crypto '96, N. Koblitz, ed., Berlin, 1996, Springer-Verlag, pp. 89–103. Lecture Notes in Computer Science Volume 1109.
15. ———, *Kleptography: Using cryptography against cryptography*, in Advances in Cryptology - EuroCrypt '97, W. Fumy, ed., Berlin, 1997, Springer-Verlag, pp. 62–74. Lecture Notes in Computer Science Volume 1233.
16. ———, *The prevalence of kleptographic attacks on discrete-log based cryptosystems*, in Advances in Cryptology - Crypto '97, B. Kaliski, ed., Berlin, 1997, Springer-Verlag, pp. 264–276. Lecture Notes in Computer Science Volume 1294.