RSA Attacks

Kevin Clark, Rainier Rabena, Tahseen Rabbani

April 20, 2015

1 Public Key Cryptosystems

Public key, or asymmetric, cryptosystems are a class of protocols that make use of two seperate procedures, one for encryption, E, and one for decryption, D. E is revealed to the public, while the details of D are kept private by each user.
The following four properties are essential for such a system.

1. Decrypting the encrypted version of a message m produces m

$$D(E(m)) = m$$

2. E and D are easy to compute

3. Possesion of E does not reveal an easy way to compute D.

4. Encrypting the decrypted version of a message m produces m

$$E(D(m)) = m$$

A function, E, that satisfies these 4 properties is known as a "trap-door one-way permutation" This concept was introduced by Diffie and Hellman [DH76], but they did not provide an example of such a function. The security of such systems is based on mathematical problems for which there no efficient method of solving them in polynomial time. They are "one-way" functions because they are easy to compute in one direction but difficult to compute in the other direction. They are "trap doors" becuase the inverse is easy to compute once some private piece information is known.

1.1 Privacy and Key Distribution

Asymmetric key cryptosystems allow users to establish privacy without *a priori* exchange of information. Symmetric key cryptosystems, which lack property 3 from earlier, suffer from what is called "the key distribution problem." Before a private communication session can begin, *another* private transaction must occur; the parties involved must send each other the private keys, by some form of courrier, that will be used for encryption and decryption. As such, the security of the system is dependant upon the security of the key distrbution procedure. Systems that require the rapid production and distrubution of a large number of keys quickly become infeasible.

Public-Key cryptosystems do not suffer from this problem. Alice and Bob can establish a secure communication session without private exchange of encryption keys. Furthermore, public-key schemes can be used to "bootstrap" a standard encryption scheme. For example, Alice can send Bob a message encrypted with his public key, but that contains a private key the two can then use to encrypt future messages using a standard encryption scheme, such as AES. This "bootstrap" process of negotiating a symmetric key via messages echanged using public key algorithms is how modern transport layer security works.

1.2 Signatures

Digital signatures address the problem of proving that a message originated from the sender. Property 4 from earlier makes such verification possible. When Alice sends Bob a message, she computes the "signature" by applying her private decryption algorithm to the original message.

$$D_a(m) = s$$

Alice then encrypts s using Bob's public encryption algorithm E_b.

$$E_b(s) = c$$

She the sends the message to Bob, who can obtain s by applying his private decryption algorithm

$$D_b(c) = s$$

He can then obtain the original message by applying Alice's public encryption method.

$$E_a(s) = m$$

Bob now has the message key pair (m, s). He cannot modify the message and then later deny doing so because he could not compute the corresponding signature without Alice's private decryption method. Alice could not later deny sending the message because no one else could have computed $s = D_a(m)$.

2 RSA

RSA [RSA78] was the first set of protocols that satisfied conditions 1-4 from Section 2.

2.1 Key generation

For Alice to send bob an encrypted message m (represented as a number), she first picks two (very) large, random, prime numbers, (p, q), and multiplies them together to yield

$$n = pq$$

n will be used as the modulus to the encryption and decryption algorithms, so it n must be greater than m. She then computes Euler's totient function on n.

$$\phi(n) = \phi(p)\phi(q) = (p-1)(q-1)$$

She then picks another large, random integer d which is relatively prime to $(p-1)(q-1)$

$$\gcd(d, (p-1)(q-1)) = 1$$

Where "gcd" means the "greatest common divisor." Alice then computes the last required number e, from p, q, and d to be the modular multiplicative inverse of $e(modulo(\phi(n)))$

$$d \equiv e^{-1}(modulo(\phi(n)))$$

The public key, pk, consists of the modulus n and the public exponent e. The secret key, sk, consists of the modulus n, and the private exponent d. Formally

$$pk = \{e, n\}$$
$$sk = \{d, n\}$$

2.2 Encryption and Decryption

To encrypt a message m we simply raise the message to the power of the public key, modulo n.

$$m^e mod(n) = c$$

To decrypt the corresponding ciphertext, we raise the message to the power of the private key modulo n.

$$c^d mod(n) = m$$

2.3 Security

The security of the RSA crypto system relies on the difficulty of factoring large numbers. While n is known to any attacker , there is no known efficient algorithm for factoring n into p & q. More generally, the RSA problem is defined as the task of taking the eth roots modulo a composite n: recovering a value m such that $c \equiv m^e(mod(n))$. While no polynomial time algorithm exists for factoring large integers has been found, it has not been proven that none exists, so whether or not the RSA problem is as hard as the integer factorization problem is an open question. The remaining sections of this paper cover novel investigations into attacks against RSA encryption. Section 3 covers how low entropy in random number generators leaves RSA open to attacks that allow rapid integer factorization and construction of private keys. Section 5 covers how one can recover private keys from random key bits. Section 6 covers attacks based on timing.

3 Insufficieant Entropy

Randomness is crucial for modern cryptography. Given its importance, one might expect that today's widely used operating systems and application software generate random numbers securely. The paper this section covers [NHH12] tested this proposition empirically by examining keys in use on the internet.

No one has ever publicly factored a 1024-bit RSA modulus. In contrast, the greatest common divisor (gcd) of two 1024-bit integers can be computed in microseconds on a modern personal computer. This asymmetry leads to a well-known vulnerability of RSA: if an attacker can find two distinct moduli that share a common factor,

$$n_1 = p * q_1$$
$$n_2 = p * q_2$$

he can simply divide both moduli by p to obtain the second factor and compute the secret key. Exploiting this vulnerability then becomes a matter of practically finding RSA keys that have this property.

3.1 Data Collection

Using the open source port scanning software nmap, the authors conducted an exhaustive search of the ipv4 address space to discover hosts that offered TLS or SSH connections on port 443 or 22, respectively The scan ran at 40,566 IPs/second on average and finished in 25 hours. Public keys were obtained via custom software that stopped mid protocol after a public key was given, stored the result in a database, and moved on. 12,828,613 protocol handshakes were completed.

3.2 Factorization

The greatest common divisor of two integers can be computed very efficiently using Euclid's algorithm. With the help of fast integer arithmetic, the time complexity of Euclid's algorithm can be imporoved to $O(n(\lg n)^2)\lg\lg n)$. This is immensely faster than the $O(2^{n^{1/3}(logn)^{2/3}})$ complexity of the fastest known factoring algorithm, the number sieve. Despite the 15 μs runtime of this algorithm on two RSA keys, the size of the data set would have required 30 years of computation to naively run the algorithm on each pair. Instead the authors implemented a custom algorithm based on [Ber04] for factoring a collection of integers into coprimes.

3.3 Vulnerabilities

By taking a macroscopic view of the internet, that authors were able to identify several patterns of vulnerability that would not have been possible to see otherwise.

The most glaring observation was the immense number of shared keys. %61 of TLS hosts, and %65 percent of the SSH hosts served the same key as another host in the scan. Many of the repeated keys were due to shared hosting. 3 of the 10 most common RSA keys were served by the same large hosting providers, and many times a large number of identical TLS certificates were served by the same organization.

While these cases do not result in vulnerabilites, a common case that *did* was the use of manufacturer-default keys. These preconfigured keys wired into the firmware may be accessible through reverse engineering, and in fact many public online databases, like littleblackbox, provide access to private keys for thousands of firmware releases. %5.34 (670,391) of the TLS hosts served manufacturer- default keys.

Another cause of repeated keys results from insufficient entropy during key generation. This leads to two independent hosts choosing a common prime factor in their key. The heuristic algorithm used to compute GCD on the 11,170,883 RSA moduli allowed the researchers to obtain private keys for 23,576 (0.40%) of the TLS certificates, which were served on 64,081 (0.5%) of the TLS hosts , and 1,013(0.02%) of the RSA SSH host keys, which were served on 2459 (0.027%) of th RSA SSH hosts. The vast majority of vulnerable keys were system- generated by headless or embedded network devices (routers, firewalls...).

3.4 Investigation

To better understand the sources of low entropy in key generation, the authors augmented their data analysis with experimental investigations of specific implementations

3.4.1 Linux RNG

The authors took an in depth look at the random number generator for linux (RNG) to investigate the hypothesis that repeated keys were due to insufficient randomness provided by the operating system. The Linux RNG maintains an entropy pool, the *Input pool*, from which the kernal provides random numbers to processes that read from /dev/urandom of /dev/random. Fresh entropy from unpredicatble kernel processes is periodically mixed into the *Input pool.* Sources of this entropy include uninitialized contents of pool buffers, at boot time, disk access timings, and the startup clock time in nanoseconds. an *Input pool.* A common source of entropy used to be IRQ (interrupt request) timings, but modern Linux systems no longer collect entropy from them. As a result, the authors discovered what they called a "boot-time entropy hole" , where a window of vulnerability exists, particularly in embeded systems with no human sources of entropy, and the RNG output may be completely predictable until a threshold of entropy is achieved. Long term keys generated by machines during this period are likely to be vulnerable.

The answer is to make networking devices generate keys only after a threshold entropy has been reached. This can be achieved through a number of means. Defaulting to /dev/random, rather than /dev/urandom will ensure that the process stalls until the kernel has enough randomness in it's pools. Another way to solve this is to have an application wait until a certain amount of time has passed on the system clock before starting to generate keys. On the hardware side, production factories could stock their chips with entropy sequences whose generating process has been tested for security.

3.4.2 OpenSSL

A naive implementation of RSA key generation would simply seed a pseudo random number generator from the OS's entropy pool and then use it to generate p and q. With this approach, factorable keys would be very unlikely. Instead, though, what we see is that some devices are prone to generating keys with common factors. To see why this occured, the authors studied and experimented with the source code of OpenSSL.

OpenSSL relies on an internal entropy pool that it self-maintains. Upon creation, the entropy pool is seeded with bytes read from /dev/urandom, the process ID, the user ID, and the current time. The RSA key generating algorithm extracts entropy from the pool dozens of hundreds of times to produce one key all while, adding entropy from the invocation time of the algorithm. The data set contained many RSA keys with one common prime factor p. Upon inspection of the source code, the authors concluded that the process is hypersensitive to small variations in where the boundary between seconds falls during execution. This means that in two executions where the clock advances to the next second during the calculation of the second prime, the first prime will be equal, but the second will depend on when exactly the second changes. As such vulnerable keys are prone to being produced by embeded or networking devices with low human entropy and whose start up sequences are deterministically similar.

The problem here is not only one of generating keys with lacking entropy, but of deterministic entropy sequences that multiple boxes of the same model share during their boot sequence. We therefore need to add an element of randomness to OpenSSL's key generating process that isnt correlated across these machines. Relying on the kernel's /dev/random, whose sources are more plentiful than OpenSSL's, is the best option. It will stall until the pool is ready, and wont be correlated across machines. Alternatively, we could generate SSL keys upon first use, rather than upon first boot. This would eliminate the similarity across machines in the list of prime factors chosen.

4 Recovering Private Keys from Random Key Bits

4.1 Motivation

For most modern computers, when the power source is removed, the DRAM (Dynamic Random Access Memory) exhibits remnance, that is, the contents are preserved for several seconds, albeit in a degraded form. We begin by assuming that the 0s and 1s in the original representation were present in equal numbers and that the degraded version of the private key demonstrates unidirectional decay, that is for any altered region, either $0 \to 1$ overwhelmingly or $1 \to 0$ overwhelmingly. Halderman et al. in [JAHF] demonstrate the most instances of DRAM degradation are unidirectional. Given then a private key with a ρ fraction of decay, that is, every component of the key has a ρ fraction of its bits degraded, we would know a $\delta = (1-\rho)/2$ fraction of key bits for any given component.

This above scenario differs from other models where one is assigned a probability of correctly guessing a given bit – in our model, we truly do know a $\delta = (1-\rho)/2$ fraction of key bits. We will assume that the acquired private key follows the PKCS#1 standard, meaning it contains the following information:

1. the (n-bit) modulus N
2. the public exponent e
3. the private exponent d
4. $d_p = d \mod (p-1)$
5. $d_q = d \mod (q-1)$
6. $q_p^{-1} = q^{-1} \mod p$

Heninger and Shacham in [HS09] demonstrate that an attacker who takes advantage of this remnance vulnerability will be able to efficiently recover an RSA private key with small public exponent given a random $\delta = 0.27$ fraction of the bits of p, q, d, d_p and d_q. By small, it is meant that e is no longer than 32 bits (which is most often the case).

4.2 k, k_p, and k_q

Before the algorithm can begin, one must recover k_p and k_q. First we compute k. The following analysis makes extensive use of 4 relations. The first,

$$N = pq. \tag{1}$$

Given that $e \equiv d^{-1} \mod \varphi(N)$ where $\varphi(N) = (p-1)(q-1) = N - p - q + 1$, we derive the second relation ,

$$ed = k(N - p - q + 1) + 1 \tag{2}$$

for some nonnegative integer k. Lastly, given that $ed_p \equiv 1 \mod (p-1)$ and $ed_q \equiv 1 \mod (q-1)$, we derive the last two relations,

$$ed_p = k_p(p-1) + 1 \tag{3}$$

$$ed_q = k_q(q-1) + 1 \tag{4}$$

for some nonnegative integers k_p and k_q. These new components, k, k_q, k_p will be pivotal in recovering the original private key.

Boneh, Durfee, and Frankel in [DBF98] note that $0 < k < e$. To see this, assume that $e \leq k$. Given that $d < \varphi(N)$, we would have that $ed < k\varphi(N) + 1$, when in fact, $ed = k\varphi(N) + 1$ by relation (2). Furthermore, $k \neq 0$, since this would contradict relation (2) when reducing modulo e. Thus, $0 < k < e$, and since we assumed e to be small, it is not difficult to enumerate all possible values of k. For each candidate k', define

$$\widetilde{d}(k') := \left\lfloor \frac{k'(N+1)+1}{e} \right\rfloor$$

where $\lfloor \cdot \rfloor$ is the floor function. In [4] it is also observed that

$$0 \leq \widetilde{d}(k) - d < 3\sqrt{N},$$

meaning that $\widetilde{d}(k)$ agrees with d on its $\lfloor n/2 \rfloor - 2$ most significant bits. This implies that a small public exponent will leak half the bits of the private exponent in one of the $\widetilde{d}(k')$. Conversely, Boneh, Durfee, and Frankel prove that a corrupted version of $\widetilde{d}$ of d can be used to find k, provided that $\delta n/2 \gg \lg e$. Heninger and Shacham note that even for 1024-bit N and 32-bit e, there is with "overwhelming probability" enough information to determine k for $\delta \geq 0.27$. This allows us to confidently assume that prior to running the algorithm, the attacker will be able to learn k and will be able to correct the most significant half of the bits of the corrupted $\widetilde{d}$.

We will now need to determine k_p and k_q. Analysis similar to the case of the k demonstrates that $0 < k_p, k_q < e$. In particular, $k_p = (k_p \mod e)$ and $k_q = (k_q \mod e)$. If k is known, through a series of clever manipulations on relations (1)-(4) reduced modulo e, Heninger and Shacham demonstrate that $k_p \equiv -k(k_q^{-1}) \mod e$ and

$$k_p^2 - [k(N-1)+1]k_p - k \equiv 0 \mod e \tag{5}$$

If e has m distinct prime factors, the number of roots (solutions) to (5) is equal to 2^m. By symmetry, k_q must also be a root. The case when e is prime is particularly nice, since there are only 2 roots, thus 2 possible assignments for the pair (k_p, k_q).

Also note that when by reducing relations (3) and (4) modulo e we have that,

$$-k_p^{-1} + 1 = p \mod e$$

$$-k_q^{-1} + 1 = q \mod e$$

so we determine p and q modulo e.

4.3 Reconstruction Algorithm

We may now describe the actual attack which consists of generating partial keys and eliminating those which do not statisfy the relationships established in the previous section. Specifically, if we know bits 1 through $i-1$ of a potential key, we enumerate all possible combinations of values of bit i for p, q, d, d_p, and d_q, and keep a combination if it satisfies (1), (2), (3), (4) mod 2^i.

We assume that we know k_p and k_q. In practice, e is usually prime, in which case we run the algorithm twice (for the two possible assignments described in the last section). Let $x[i]$ denote the ith bit of x, where $x[0]$ denotes the least significant bit. Let $\tau(x)$ denote the exponent on the largest power of 2 which divides x. Since p and q are odd primes (assuming they are large), we immediately know that $p[0] = q[0] = 1$. Hence, $2 \mid p-1$, so $2(2^{\tau(k_p)}) = 2^{1+\tau(k_p)} \mid k_p(p-1)$. By reducing (3) modulo $2^{1+\tau(k_p)}$, we have that

$$d_p \equiv e^{-1} \mod 2^{1+\tau(k_p)}.$$

Thus, we are able to determine/correct the least significant $1+\tau(k_p)$ bits of d_p. Analogous reductions on relations (4) and (2) also allow us to determine/correct the $1+\tau(k_q)$ least significant bits of d_q and the $2+\tau(k)$ least significant bits of d. Furthemore, we can conclude from this analysis that a change in $p[i]$ and $q[i]$ will respectively affect $d_p[i+\tau(k_p)]$ and $d_q[i+\tau(k_q)]$ along with $d[i+\tau(k)]$. Define the ith slice as

$$p[i] \qquad q[i] \qquad d[i+\tau(k)] \qquad d_p[i+\tau(k_p)] \qquad d_q[i+\tau(k_q)]$$

For a given partial solution up to $i-1$ bits, we generate all possible solutions for the ith slice. By doing this for all solutions in the $(i-1)$th slice, we have enumerated all possible solutions for slice i. We already know the only possible solution for the $i=0$ slice, so the algorithm begins there. We recover the factorization of N in one or more possible solutions after having reached $i=\lfloor n/2 \rfloor$.

Given a possible solution (p', q', d', d'_p, d'_q) for slice $i-1$, it would appear that there are $2^5 = 32$ possible solutions for slice i, but there are in fact, at most 2 solutions. The relations (1)-(4) form a set of "constraint polynomials," which in combination with the Multivariate Hensel's Lemma (an important tool in number theory) reveal that the ith slice must satisfy the following conditions,

$$\begin{aligned} p[i] + q[i] &\equiv (n - p'q')[i] \mod 2 \\ d[i+\tau(k)] + p[i] + q[i] &\equiv (k(N+1) + 1 - k(p'+q') - ed')[i+\tau(k)] \mod 2 \\ d_p[i+\tau(k_p)] + p[i] &\equiv (k_p(p'-1) + 1 - ed'_p)[i+\tau(k_p)] \mod 2 \\ d_q[i+\tau(k_q)] + q[i] &\equiv (k_q(q'-1) + 1 - ed'_q)[i+\tau(k_q)] \mod 2. \end{aligned} \tag{6}$$

Or less intimidatingly, we have a system of the following form:

$$\begin{aligned} p[i] + q[i] &\equiv c_1 \mod 2 \\ d[i+\tau(k)] + p[i] + q[i] &\equiv c_2 \mod 2 \\ d_p[i+\tau(k_p)] + p[i] &\equiv c_3 \mod 2 \\ d_q[i+\tau(k_q)] + q[i] &\equiv c_4 \mod 2. \end{aligned} \tag{7}$$

For a given c_1, there are only two possible assignments to $p[i]$ and $q[i]$, which consequently fix the remaining unknowns, thus every solution in slice $i-1$ lifts to two solutions in slice i. In practice, our partial knowledge of the private key would allow us to eliminate a possible solution in a given i slice if it does not agree with known bits. Heninger and Shacham summarize the performance of this algorithm in the following theorem,

Theorem 1 Given the values of a $\delta = .27$ fraction of the bits of p, q, d, d mod p, and d mod q, the algorithm will correctly recover an n-bit RSA key in expected $O(n^2)$ time with probability 1-$\frac{1}{n^2}$.

5 Timing Attacks

The basic goal of a timing attack is to determine some secret value by using precise information about the time taken to execute an algorithm to exploit some flaw in the implementation of that algorithm. This type of attack can be most easily used when there is a conditional statement in the algorithm that can follow two different paths depending on the input. One path is slow and takes a long time to complete, while the other path is fast and takes a short time to complete. For example, assume we have a secret bit x, which can be either 0 or 1. Consider the following pseudocode for an algorithm:

Algorithm 1 A simple conditional statement.

```
if $x = 1$ then
    Perform a slow operation (~400 ms)
else
    Perform a fast operation (~30 ms)
end if
```

An adversary tells us to run this algorithm, and he measures the execution time to be 33 ms. The adversary, then, can conclude that our secret value of x is 0, with high probability.

The main idea of the attack described in [Koc96] is that it is possible to retrieve the entire RSA secret key simply by exploiting the timing characteristics of the decryption operation. By sampling a large enough number of decryption times, we can use statistical techniques to determine a "guess" for the first b bits of the secret key. If we know the first b bits, we can perform a partial execution of the decryption function ourselves, which allows us to determine the next few bits of the key. We continue to iterate this procedure until the entire key has been retrieved.

5.1 Attack on Modular Exponentiation

Recall that RSA decryption works as follows: Given the ciphertext c, the secret key d, and the modulus n, we can determine the original plaintext message m by calculating

$$m = c^d \bmod n$$

But how do we actually compute this value? Naively, we can use an accumulator with an initial value of 1 and perform d multiplications by c to find c^d. Then we can divide by n and take the remainder to find the result m. However, such an approach becomes unwieldly for large values of c and d, which tends to happen in strong cryptographic systems such as RSA. Modern implementations use a square-and-multiply algorithm to perform modular exponentiation. First, note that d can be expressed in the following binary form:

$$d = \sum_{k=0}^{w-1} a_k 2^k$$

where w is the length of d in bits and a_k is the value of bit k in d. Then, the value c^d can be represented as

$$c^d = c^{\left(\sum_{k=0}^{w-1} a_k 2^k\right)} = \prod_{k=0}^{w-1} \left(c^{2^k}\right)^{a_k}$$

which means that the value m can be computed as

$$m \equiv \prod_{k=0}^{w-1} \left(c^{2^k}\right)^{a_k} \pmod{n}$$

One implementation of this algorithm can be seen in the example below.

Algorithm 2 A simple modular exponentiation algorithm.

```
Let $s_0 = 1$
for $k = 0$ upto $w - 1$ do
    if $a_k = 1$ then
        Let $m_k = (s_k \cdot c) \bmod n$
    else
        Let $m_k = s_k$
    end if
    Let $s_{k+1} = m_k^2 \bmod n$
end for
return $m_{w-1}$
```

Note that each iteration k of the main loop may need to perform an extra modular multiplication operation, depending on the kth bit of d. The extra computation time caused by the difference between the two branches is the key to the general timing attack.

Suppose we have j ciphertext messages $c_0, c_1, ..., c_{j-1}$ with corresponding timing measurements $T_0, T_1, ..., T_{j-1}$. Suppose also that we have a guess d_b for the first b exponent bits. Each timing observation consists of $T = e + \sum_{i=0}^{w-1} t_i$, where t_i is the time required for the multiplication and squaring steps for bit i, and e is the measurement error. Given d_b, the attacker can find $\sum_{i=0}^{b-1} t_i$ for each sample c by running b iterations of the loop himself and computing the time taken to run that. Subtracting from T yields $e + \sum_{i=b}^{w-1} t_i$. If m_b was correct, then the variance of $e + \sum_{i=b}^{w-1} t_i$ is expected to be $\operatorname{Var}(e) + (w-b)\operatorname{Var}(t)$. However, if only the first $c < b$ bits of m_b were correct, then the expected variance is $\operatorname{Var}(e) + (w - b + 2c)\operatorname{Var}(t)$. In other words, correctly emulated iterations decrease the expected variance by $\operatorname{Var}(t)$, while iterations following an incorrect exponent bit each increase the variance by $\operatorname{Var}(t)$.

It is possible to approximate t_i using independent standard normal variables. If we assume that $\operatorname{Var}(e)$ is negligible, the expected probability that m_b is correct is:

$$P\left(\sum_{i=0}^{j-1}\left(\sqrt{w-b}X_i + \sqrt{2(b-c)}Y_i\right)^2 > \sum_{i=0}^{j-1}\left(\sqrt{w-b}X_i\right)^2\right)$$

$$= P\left(2\sqrt{2(b-c)(w-b)}\sum_{i=0}^{j-1} X_iY_i + 2(b-c)\sum_{i=0}^{j-1} Y_i^2 > 0\right)$$

where X and Y are standard normal random variables. Because j is relatively large, $\sum_{i=0}^{j-1} Y_i^2 \approx j$, and $\sum_{i=0}^{j-1} X_iY_i$ is approximately normal with $\mu = 0$ and $\sigma = \sqrt{j}$, yielding

$$P\left(2\sqrt{2(b-c)(w-b)}\sqrt{j}Z + 2(b-c)j > 0\right) = P\left(Z > -\frac{\sqrt{j(b-c)}}{\sqrt{2(w-b)}}\right)$$

where Z is a standard normal random variable. Integrating to find the probability of a correct guess yields $\Phi\left(\sqrt{\frac{j(b-c)}{2(w-b)}}\right)$, where $\Phi(d)$ is the area under the standard normal curve from $-\infty$ to d. The required number of samples j is thus proportional to the exponent size w.[Koc96]

5.2 Preventing the Attack

On many processors, the division operation (and subsequently, the modulo operation) is slow. For the conditional statement inside the main loop of the modular exponentiation algorithm shown above, this slowness is largely the reason why there is such a noticeable difference in the execution times betweeen the two branches. Montgomery multiplication eliminates this modulo operation and reduces the size of the timing characteristics, but some variance still remains and can be exploited with more precise timing measurements or an increased number of samples. Some implementations of RSA use the Chinese Remainder Theorem (CRT) to optimize the decryption process. While the attack described above does not allow an adversary to extract the secret key directly, it can be adapted to approximate the value of one of the prime factors of the modulus n, which can then be used to determine the secret key using the other information available to the attacker (i.e. the public exponent).

One way to prevent the attack is to make all operations take the same amount of time. However, fixed-time implementations are also likely to be slow, and unexpected timing variations can still occur due to compiler optimizations and cache misses. Adding random delays to the timing measurements to make them inaccurate is another approach to preventing the attack, but attackers can get around this by adding more samples. The best solution to this problem is to use a technique known as blinding. This can be done by choosing a random pair (v_i, v_f) such that $v_f^{-}1 = v_i^d \bmod n$. Before the modular exponentiation operation, the input message should be multiplied by $v_i \pmod n$, and afterward the result is corrected by multiplying with $v_f \pmod n$. The system should reject messages equal to $0 \pmod n$. If this random pair is kept secret, then the attack does not gain any useful information about the secret key. He is only able to figure out the general timing distribution of the modular exponentiation operation.

References

[Ber04] Daniel J. Bernstein. How to find smooth parts of integers. 2004.

[DBF98] Glenn Durfee Dan Boneh and Yair Frankel. An attack on rsa given a small fraction of the private keys bits. *Advances in Cryptology*, 1514:25–34, 1998.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[HS09] Nadia Heninger and Hovav Shacham. Reconstructing rsa private keys from random key bits. *Advances in Cryptology*, 5677:1–17, 2009.

[JAHF] Nadia Heninger William Clarkson William Paul Joseph Calandrino Ariel Feldman Jacob Appelbaum J. Alex Halderman, Seth Schoen and Edward Felten. Lest we remember: Cold boot attacks on encryption keys. *2008 USENIX Security Symposium.*

[Koc96] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96*, pages 104–113. Springer, 1996.

[NHH12] Zakir Durumeric Eric Wustrow Nadia Heninger, UC San Diego and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. 2012.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb 1978.