Cryptanalysis of RSA: A Survey

Carlos Cid

Cryptanalysis of RSA: A Survey

Carlos Frederico Cid
GSEC – GIAC Security Essentials Certification
Practical Version 1.4b

1 Abstract

The RSA is the most widely deployed public-key cryptosystem and is used for both encryption and digital signature. It is commonly used in securing e-commerce and e-mail, implementing virtual private networks and providing authenticity of electronic documents. It is implemented in most Web servers and browsers, and present in most commercially available security products. In fact, the ubiquity of RSA has placed it at the heart of modern information security. It would not be an overstatement to say that Internet security relies heavily on the security properties of the RSA cryptosystem.

Since its invention in 1977, the RSA cryptosystem has been extensively analyzed for vulnerabilities. While no devastating attack has ever been found, years of cryptanalysis of RSA have given us a broad insight into its properties and provided us with valuable guidelines for proper use and implementation.

In this paper we give a survey of the main methods used in attacks against the RSA cryptosystem. We describe the main factoring methods, attacks on the underlying mathematical function, as well as attacks that exploit details in implementations of the algorithm. While many attacks exist, the system has proven to be very secure, and most problems arise as a result of misuse of the system, bad choice of parameters or flaws in implementations. To conclude, we list a couple of countermeasures that can be used to prevent many of the attacks described.

2 Overview of Public-Key Cryptography

Cryptography can be defined as the study of mathematical techniques related to the security of transmission and storage of information. Cryptography is an important tool in today's information security, and although it has been historically linked to confidentiality, modern cryptography addresses also the issues of integrity, authentication and non-repudiation.

Basically, there are two types of cryptography: symmetric-key cryptography and public-key cryptography. Symmetric-key (or secret-key) cryptography can be seen as an outgrowth of classical cryptography. If users want to securely communicate with each other, they must share a key, which is used to both encrypt and decrypt messages[1]. The security of a symmetric-key scheme should rely on the secrecy of the key, as well as in the “infeasibility” of decryption without knowledge of the same. Examples of symmetric-key encryption algorithms are DES, RC4, Blowfish and AES.

[1] Technically, the encryption and decryption keys may be distinct, but it must be easy to derive one from the other.

Symmetric-key schemes are usually fast. One of the main issues when deploying symmetric-key cryptography is the problem of *key establishment*. The problem is that users must share a secret key to be able to securely communicate, which by nature should be known by no one else. Historically, key distribution was done in advance using a secure channel (e.g. trusted courier). When considering a network of users wishing to communicate securely, each pair of users must share a secret key, which makes it impractical for any medium-size network. A solution could involve a central entity, which would be trusted by all the users (e.g. a trusted third party) and whose job would be the issuance of ephemeral *session keys*. There are a number of other solutions, but it should be clear that key establishment is one of the main problems.

The concept of public-key cryptography was proposed in 1976 by Whitfield Diffie and Martin Hellman in their pioneering paper [7]. As explained in their paper, the main motivation for this new concept was to "minimize the need for secure key distribution channels and supply the equivalent of a written signature". In public-key cryptography each user has a *key pair (e, d),* which consists of a public key e and a private key d. The user *A* can make e publicly available and keep only d secret. Now anyone can encrypt information with e, while only *A* can decrypt it with d. Alternatively, the private key can be used to sign a document, and anyone can use the corresponding public key to verify its authenticity. What makes public-key cryptography work is that it is *computationally infeasible* to derive the secret key d from the corresponding public key e (compare with symmetric-key above). Examples of public-key cryptosystems are Diffie-Hellman key exchange scheme, ElGamal and RSA encryption and signature schemes.

At the heart of public-key cryptography lies the concept of *one-way function*. We say that a function f is "one-way" if it is easy to compute $f(x)$ for every x in the domain of f, but for most randomly chosen y in the image of f, it is computationally infeasible to find x such that $f(x) = y$. A *trapdoor one-way* function is a one-way function for which given an extra information (the *trapdoor*), it becomes feasible to find x such that $f(x) = y$. One can consider the encryption with e as the one-way function, with d being the trapdoor information.

We can think of the situation above analogous to a safe mailbox. Anyone can insert a message into the box (the box is in public domain), while only the owner can open it (with his *private key*). One could still try to break the box to read any message, but depending of the physical strength of the box, this might be infeasible.

Compared with symmetric-key cryptography, public-key cryptosystems are usually slower and require longer keys. On the other hand, it simplifies key distribution[2]. Also conventional symmetric-key algorithms have a limited lifetime: it turns out to be useless once exhaustive key search becomes feasible due to computational progress. Public-key is more flexible, as all one has to do is to select larger keys. In practice, it is common to use hybrid schemes: public-key

[2] There is still the need to authenticate users' public keys, which in practice is done by *public-key certificates*.

techniques are used to establish short-term (symmetric) keys, which are used to secure the communication.

3 The RSA Cryptosystem

The RSA cryptosystem was created in 1977 and named after its inventors Ronald Rivest, Adi Shamir and Leonard Adleman [16]. It is widely used to secure communication in the Internet, ensure confidentiality and authenticity of e-mail, and it has become fundamental to e-commerce. RSA is deployed in the most popular security protocols, including SSL/TLS, SET, SSH, S/MIME, PGP, DNSSEC, as well as most PKI products. Several standards cover the use of RSA for encryption, digital signatures and key establishment. In fact, RSA is usually present wherever security of digital data is a concern.

The underlying mathematical structure of the RSA function is quite simple and this might be another reason for its popularity; people do feel more comfortable on working with an algorithm one can understand. It is based on basic algebraic operations on large integers. Before the description of the algorithm we need to set some conventions though:

- if a, b and n are positive integers, we say that a is equal to b modulo n (denoted as $a \equiv b \bmod n$) if b is the remainder of a divided by n;
- we denote by Z_n the set of integers modulo n. This set can be represented by all non-negative integers less than n. We can define the operations *addition* and *multiplication* in this set by using the usual operations on integers, but taking the result modulo n as defined above. These are called *modular addition* and *modular multiplication.*
- we denote by $gcd(a, b)$ the *greatest common divisor* of a and b. This is defined as the largest positive integer which divides both a and b.

We can now describe the RSA algorithm (for more details see [13]):

- generate two distinct large primes p, q, each roughly the same size;
- compute $n = pq$ and $\phi(n) = (p-1)(q-1)$;
- choose an integer $e < \phi(n)$ such that $\gcd(e, \phi(n))=1$.
- calculate[3] d such that $de \equiv 1 \bmod \phi(n)$.

The pair of integers (e, n) is the public key. The pair (d, n) is the private key (the primes p, q are in principle no longer needed, although they can not be made public). The integer n is the modulus. We will call e and d the public and private exponents, respectively. It is also convention to call the bit length of the modulus n the *size* of the RSA key.

We typically have the modulus n 1024-bit long (each prime is 512-bit long), and the public exponent a relatively small integer (3 and 65537 are commonly used values). In this case the private exponent d will be roughly the same size as n.

[3] using the Extended Euclidean Algorithm.

Now we have:

Encryption:

- represent a message to be encrypted as an integer $M \in Z_n$
- encrypt M as $C \equiv M^e \bmod n$
- the resulting ciphertext C can be decrypted by computing $D \equiv C^d \bmod n$. It follows from $de \equiv 1 \bmod \bullet(n)$ that $D = M$.

The RSA algorithm is also used for generating digital signatures, which can provide authenticity and non-repudiation of electronic legal documents.

Signing:

- represent a message to be signed as an integer $M \in Z_n$. This is usually done by first calculating the hash of the message.
- M can be signed by calculating $S \equiv M^d \bmod n$.
- the signature can be verified by checking if $M \equiv S^e \bmod n$.

In practice messages are encoded (padded) before encryption and signing. The use of the RSA function without prior encoding does not satisfy basic definitions of security and is considered insecure. Also the same private key should not be used to both decrypt and sign messages. Users should have different keys for encryption and signing.

The RSA encryption scheme is a public-key cryptosystem, and the RSA trapdoor function is defined as $f(x) \equiv x^e \bmod n$. The trapdoor is the private exponent d, since $(x^e)^d \equiv x \bmod n$. The problem of how to invert the RSA function without the knowledge of the private exponent d is known as the *RSA problem.*

The security of the RSA cryptosystem relies on the very well known problem of factoring large integers, which is widely believed to be intractable. Factorization of the modulus is devastating for the system. If an adversary can factor n, he can easily calculate the private exponent by solving the congruence $ed \equiv 1 \bmod \bullet(n)$, and thus invert the RSA function.

One interesting point to note is that it has been shown that recovery of the private exponent d is equivalent to factoring the modulus n (see [2]). That means that an adversary who knows (e, d, n) can efficiently factor n. This illustrates a possible misuse of the RSA cryptosystem. To avoid generating new primes for every user, it was thought that a trusted central authority could generate a common modulus and distinct exponent pairs (e_i, d_i) for each user. Although this might seem at first glance a good idea, the fact above shows that it is completely insecure: any user, knowing its own key pair, could factor n and then recover all the other private keys. This shows that an RSA modulus should *never* be used by more than one entity.

Note that the fact above *does not* prove the equivalence between the RSA problem and the integer factorization problem. The RSA problem is, given (e, n) and C, to recover M such that $C \equiv M^e \bmod n$ (i.e. calculate e^{th}-roots modulo n). To

recover the private key d is a much more ambitious target, and above we see that it is as hard as factoring the integer n. Currently it is not known if the RSA and integer factorization problems are computationally equivalent. In fact, there is evidence that, at least for low public exponents, breaking RSA might be easier than factoring [2]. This provides no sign of weakness in the RSA system though; it shows only that the problems might not be equivalent (the RSA problem is still likely to be a hard problem).

4 Cryptanalysis of RSA

While cryptography is the science concerned with the design of ciphers, cryptanalysis is the related study of breaking ciphers. Cryptography and cryptanalysis are somehow complimentary sciences: development in one is usually followed by further development in the other. In particular, cryptanalysis is an important tool for vulnerability assessment of cryptosystems.

In an encryption scheme, the main objective of the adversary is to recover the plaintext M from the related ciphertext[4]. If he is successful, we say he has *broken* the system. In the case of digital signatures, the goal of the adversary is to *forge* signatures. A more ambitious attack is to recover the private key d. If achieved, the adversary can now decrypt all ciphertexts and forge signatures at will. In this case the only solution is the revocation of the key.

Below we give a brief description of the main methods used to attack the RSA cryptosystem. We talk about the main factoring methods, attacks on the underlying mathematical function and attacks which exploit implementation details and flaws. Some of these attacks apply only to the encryption scheme, some result in the key recovery (e.g. factoring). It should be clear from the context to which situation a particular attack applies.

As it is already standard in examples of use of cryptography, we name three entities involved in the system as *Alice*, *Bob* and *Caroline*. Alice and Bob wish to securely communicate with each other, while Caroline is a malicious adversary, trying passively or actively to disturb the communication.

4.1 Factoring

The problem of finding non-trivial factors of a given positive composite integer n is known as the *integer factorization problem*. The integer factorization problem is widely believed to be a hard problem, i.e., there seems to be no polynomial time algorithm that solves the problem for a large proportion of possible inputs. Note that factoring is obviously not always hard, but a hard instance of the problem can be easily created, by simply multiplying two large chosen prime numbers. That is exactly what we do when working with RSA.

As we have seen, the security of the RSA cryptosystem is intimately related to the integer factorization problem. If an adversary can factor the modulus n, he can efficiently calculate the private exponent. In this case we say that he has

[4] Sometimes the adversary may want to recover *any* information about M, even a single bit.

completely broken the encryption scheme: he not only can recover a particular plaintext M, but also decrypt all ciphertexts encrypted with the respective public key.

Therefore, although factoring methods are not used in practice in attacks against RSA, they are important in deriving lower bounds for key sizes and properties of the security parameters. Taking into account the value of data and the threat model, parameters should be chosen such that factoring the modulus is computationally infeasible.

Factoring methods can be divided into special-purpose and general-purpose factoring methods. Special purpose methods depend on special properties of the number to be factored, as the size of the smallest factor p of n, the factorization of $p - 1$, etc. General-purpose methods depend solely on the size of n.

Below we give only an outline of the main factoring methods; apart from advanced mathematical techniques, there are a number of improvements, optimizations and implementation details that are beyond the scope of this article. More details can be found in [10], [11], [12].

4.1.1 Trial Division

Trial division can be considered as an exhaustive search for the RSA private key. The method is to attempt to divide n by all successive primes until one divisor is found. Because a composite number n must have a prime divisor $\leq n^{½}$, one has only to check for all primes up to the square root of n. It then follows from the Prime Number Theorem that the number of attempts is bounded by $\bullet (2n^{½})/(\log(n))$. Although this method is very effective when trying to factor a randomly selected composite integer[5] or relatively small numbers, it is useless against the kind of numbers currently used with RSA.

4.1.2 Pollard's *p – 1* Method

This is a special purpose factoring method, which relies on a special property of a divisor of n. Suppose that p is some (yet unknown) prime divisor of n, for which $p - 1$ is a product of (possibly many) small primes $\leq B$ (we say that $p - 1$ is B-*smooth*). If we make K the product of large enough powers of all primes less than B, then we have that K is a multiple of $p - 1$. If a is a small integer (e.g. $a = 2$), it follows from Fermat's Little Theorem that $a^k \bullet 1 \bmod p$, and therefore p divides the greatest common divisor of $a^k - 1$ and n. Note that K might be very large, but $(a^k - 1)$ can be computed modulo n.

With this method one can find a prime factor of n in time proportional to the largest prime factor of $p - 1$. The problem is that we do not know the factorization of $p - 1$ beforehand (we want to find p !), so we have to choose a bound B and perhaps increase it until we are successful. Thus this method is practical only if B is not too large. For the typical size of RSA primes used today (512 bits), the probability that one can factor n using Pollard's $p - 1$ method is very small. Nevertheless, the very existence of this method is a reason why some

[5] It is known that more than 91% of integers have a factor < 1000.

cryptographic standards (e.g. ANSI X9.31) require RSA moduli with primes for which $p - 1$ has at least a large prime factor (known as *strong primes*).

4.1.3 Pollard 's *rho* Method

This method is based on a fact known as the *birthday paradox*. If you have 23 people in a room, the probability that at least 2 of them share the same birthday is greater than 50%. This fact might seem surprising to many people, thus the name. More generally, if you have a set with N elements and you draw elements at random (with replacement) from this set, then after around $1.2(N)^{1/2}$ draws you would expect to have drawn an element twice.

Pollard's *rho* method works by successively picking at random numbers less than n. If p is an unknown prime divisor of n, then it follows from the above fact that after around $1.2p^{1/2}$ draws we would expect to have drawn x_i, x_j such that $x_i \equiv x_j \bmod p$. Thus we have $p = gcd(x_i - x_j, n)$.

For this method to be effective, one has to choose a function from Z_n into Z_n which behaves "randomly" in Z_n ($f(x) = x^2+1$ will do it), and starting with any x_0 in Z_n, repeatedly calculate $x_j = f(x_{i-1})$. One does not have to calculate $gcd(x_j - x_i, n)$ for all previous numbers x_j (which would require very large memory and would make the method nearly as expensive as exhaustive search). It has been shown that one needs only to compare x_i with x_{2i}. This decreases the requirement for storage and the number of operations, but it can happen that we might miss an early collision, which would only be caught later. The spatial description of the sequence of elements x_i is of the Greek letter r (rho), starting at the tail, iterating until it meets the tail again, and then it cycles on from there until the prime p is found.

The runtime of this algorithm is therefore proportional to the size of the smallest prime dividing n. For the size of today's RSA moduli this method is impractical. But Pollard's *rho* method was used on the factorization of the eighth Fermat number $2^{256} + 1$, which unexpectedly had a small prime factor.

4.1.4 Elliptic Curve Method

The Elliptic Curve Factorization method was introduced by H. W. Lenstra in 1985, and can be seen as a generalization of Pollard's $p - 1$ method. The success of Pollard's method depends on n having a divisor p such that $p - 1$ is smooth; if no such p exists, the method fails. The Elliptic Curve method randomizes the choice, replacing the group Z_p (used in Pollard's method) by a random elliptic curve over Z_p. Because the order of the elliptic curve group behaves roughly as a random integer close to $p + 1$, by repeatedly choosing different curves, one will find with high probability a group with B-smooth order (for a previously selected B), and computation in the group will provide a non-trivial factor of n.

The Elliptic Curve method has a (heuristic) subexponential runtime depending on the size of the prime factors of n, with small factors tending to be found first. The worst case is when p is roughly $(n)^{1/2}$, which applies for RSA moduli. So although

the method cannot be considered a threat against the standard (two-prime) RSA, it must nevertheless be taken into account when implementing the so-called "multi-prime" RSA, where the modulus may have more than two prime factors (as described in [4], [19]).

4.1.5 Quadratic Sieve and Number Field Sieve Methods

The Quadratic Sieve and the Number Field Sieve methods are the most widely used general-purpose factoring methods. Both are based on a method known as "Fermat Factorization": one tries to find integers x, y, such that $x^2 \equiv y^2 \bmod n$ but $x \neq \pm y \bmod n$. In this case we have that n divides $x^2 - y^2 = (x - y)(x + y)$, but it does not divide either term. It then follows that $gcd(x - y, n)$ is a non-trivial factor of n. If $n = pq$, a random solution (x, y) of the congruence $x^2 \equiv y^2 \bmod n$ would give us a factor of n with probability of 50%.

The general approach for finding solutions (x, y) of the congruence above is to choose a set of relatively small primes $S = \{ p_1, p_2, \ldots, p_t \}$ (called *factor base*) and enough integers a_i such that $b_i \equiv a_i^2 \bmod n$ is the product of powers of primes in S. In this case every b_i can be represented as a vector in the t-dimensional vector space over Z_2. If we collect enough b_i's (e. g., $t+1$ of them), then a solution of $x^2 \equiv y^2 \bmod n$ can be found by performing the Gaussian elimination on the matrix $B = [b_i]$, with chance of at least 50% of finding a factor of n.

The first step above is called the "relation collection stage" and is highly parallelizable. The second step is called the "matrix step", in which we work with a huge (sparse) matrix, and will eventually produce a non-trivial factor of n.

It should be clear that the choice of the number of primes of S is very important to the performance of the method: if this number is too small, the relation stage will take very long time, as a very small proportion of numbers will factor over a small set of primes. If we pick too many primes, the matrix will be too large to be efficiently reduced. Also crucial are the methods for choosing the integers a_i's and testing division by primes in S.

The Quadratic Sieve (QS) Method was invented by Carl Pomerance in 1981, and was until recently the fastest general-purpose factoring method. It follows the approach above, introducing an efficient way to determine the integers a_i's, by performing a "sieving process" (recall the "Sieve of Eratosthenes"). It has a subexponential runtime and was used in the factorization of the RSA-129 challenge number in 1994. The effort took around 8 months, with the factor base in this case containing 524,339 primes [11].

The Number Field Sieve (NFS) Method is currently the fastest general-purpose factoring method. The technique is similar to the Quadratic Sieve, but it uses a factor base in the ring of integer of a suitably chosen algebraic number field [10]. From the sieving step on, the QS and NFS methods coincide. For NFS, the matrix is usually larger, but the initial step is more efficient. NFS has also a (heuristic) subexponential runtime, which is (asymptotically) better than the QS. Experiments show that NFS outperforms QS for numbers from 110-120 digits

long, although A. Lenstra states that the crossover point is probably much lower than expected [12].

The Number Field Sieve method was used in the factorization of the RSA-155 challenge number (around 515 bits) in 1999. In total 124,722,179 relations were collected, while the resulting matrix had 6,699,191 rows and 6,711,336 columns. The total elapsed time was 7.4 months [18].

4.2 Attacks on the RSA Function

The attacks below take advantage of special properties of the RSA function. These usually exploit the misuse of the system, bad choice of either the private exponent d or the public exponent e, relation between encrypted messages, bad padding, etc. Many of these attacks use advanced mathematical techniques that are outside the scope of this article; a good reference is [2].

4.2.1 Low Private Exponent Attack

The RSA decryption and signing are very compute-intensive operations, which take time linear to the length of the private exponent d. Thus some low-power devices may want to use a small d instead of a random one, in order to improve performance. However an attack due to M. Wiener shows that the choice of a small d can lead to a total break of the system. More specifically, he showed that if n is the modulus and d is the private exponent, with $d < 1/3(n)^{1/4}$, then given the public key (e, n), an attacker can efficiently recover d (Boneh and Durfee have recently improved the bound to $d < n^{0.292}$).

In practical terms, this means that for a typical 1024-bit RSA modulus, the private exponent d should be at least 300 bits in length. If the public exponent e is chosen to be 65,537 (the most commonly used value), and we calculate d as $de \equiv 1 \bmod \bullet(n)$, then we are guaranteed to have d nearly as long as n, and this attack should not pose a threat.

4.2.2 Partial Key Exposure Attack

A well-known principle in cryptography says that the security of any cryptographic system should rely solely on the secrecy of the private key. An attack on the RSA cryptosystem due to Boneh, Durfee and Frankel shows the importance of protection of the *entire* private exponent d.

They have shown that, if the modulus n is k bits long, given the $(k/4)$ *least significant bits* of d, an attacker can reconstruct all of d in time linear to $(e \log(e))$, where e is the public exponent. This means that if e is small, the exposure of a quarter of bits of d can lead to the recovery of the whole private key d.

The issue of safeguarding of RSA private keys is a crucial one, but it is a problem that is often overlooked. In SSL-enabled servers for example, it is not unusual to have the private key stored in the computer's HD, in *plaintext* form so that the server can be re-started without human interference (an encrypted file would require the encryption key in order to start the server).

Also taken into account the fact that when e is small the RSA system always leaks *half the most significant bits of* d [2], plus a clever technique described in [22], in which one searches for randomness in order to locate private keys in large volumes of data, such as the hard disk filing system (or memory), it should be clear how important the safe storage of the RSA private key is. Perhaps the best solution is the use of tamper-resistant hardware modules or tokens, in which the private key is securely stored and the private operation is performed.

4.2.3 Broadcast and Related Message Attacks

If Bob broadcasts the encryption of the same message M to a sufficient large number of recipients (which have different public keys (e_i, n_i)), Hastad has described an attack in which a malicious eavesdropper can efficiently recover M if all the public exponents are small. This attack can be extended to the case in which instead of sending $(M)^{e_i} \bmod n_i$ to each recipient, Bob sends $(f_i(M))^{e_i} \bmod n_i$, where f_i are known polynomials (for example, some kind of known padding). This is done by solving a system of univariate equations modulo relatively prime composites, which can be efficiently done if sufficiently many equations are given.

There is also an attack when Bob sends to Alice related encrypted messages using the same modulus. If M_1 and M_2 are 2 messages such that $M_1 = f(M_2)$, where f is again a known polynomial function, and Bob sends $C_1 \equiv (M_1)^e \bmod n$ and $C_2 \equiv (M_2)^e \bmod n$, then a malicious eavesdropper can again efficiently recover both messages if e is small. An example of such scenario is one where Bob sends the first encrypted message to Alice, which Caroline intercepts. Because Alice has failed to respond, Bob sends the message again, using a different padding (the function f). Caroline can now recover M using the attack above.

Both of these attacks are more effective if the public exponent e is 3, although they can be prevented if we remove the relation between the messages, usually by adding some kind of *random* padding.

4.2.4 Short Pad Attack

Related to the attacks above, Coppersmith has shown that an adversary can still be successful if the *random* padding Bob adds to the messages is not long enough. For example, if e is 3, the length of the random padding must be at least 1/9 of the message length. A variant of the attack is successful in decrypting a single ciphertext when a large fraction (2/3) of the message is known. For example, PKCS#1 standard ([19]) recommends the use of its simpler padding scheme (v1.5) only for encryption of relatively short messages (typically a 128-bit symmetric key). If a long message is to be encrypted, or if part of a message is known, then the attack above may be a concern, in which case an alternative padding method should be used (e.g. RSA-OAEP also defined in [19]).

4.3 Implementation Attacks

All the attacks against the RSA we have seen so far apply to the underlying cryptographic primitive and parameters. On the other side, implementation

attacks (also called *side-channel attacks*) target specific implementation details. In this case, an attacker typically uses some additional information leaked by the implementation of the RSA function or exploit faults in the implementation. The attacks are usually applied against smart cards and security tokens, and are more effective when the attacker is in possession of the cryptographic module. Defense against side-channel attacks is hard; one usually tries to reduce the amount of information leaked or make it irrelevant to the adversary.

4.3.1 Timing Attack

Timing attacks against RSA were introduced by P. Kocher in 1995 (see [8]). Timing attacks take advantage of the correlation between the private key and the runtime of the cryptographic operation. Recall that the RSA private operation consists of a modular exponentiation, using the private key d as exponent. Modular exponentiations are usually implemented using an algorithm called *repeated squaring algorithm.* If the private key is k bits long, this consists of a loop running through the bits of d, with at most $2k$ modular multiplications. In each step the data is squared, with the execution of a modular multiplication if the current bit of the exponent is 1.

By measuring the runtime of the private operation on a large number of random messages, an attacker can recover bits of d one at a time, beginning with the least significant bit. Note that in view of the *partial key information attack* described earlier, if a low public exponent is used, the attacker needs only to find the first $k/4$ bits using this method; the remaining bits can be found using the previous method.

To defend against timing attacks, one must try to lessen the correlation between the runtime and the private exponent. One solution is to add a delay, so that every modular operation takes the same fixed time. This obviously affects the performance of the operation, which many times may not be a serious issue though (for example, when working with smart cards).

Another solution is called *blinding.* This transforms the data before the private operation using a random value generated by the cryptographic module. Now the operation is performed on a random data unknown to the adversary, which precludes the attack. Recently Boneh and Brumley published a paper [3] describing a timing attack against OpenSSL, which can be done remotely (timing attacks were believed to be effective only when the attacker was in possession of the cryptographic module). They explore OpenSSL's modular exponentiation software implementation, and their attack also recovers the RSA private key. To defend against this attack they recommend performing RSA blinding. Apparently blinding is implemented in OpenSSL, but it is not enabled by default (the performance penalty is of around 2% - 10%).

One important point to note is that timing attacks are not only defined against RSA. A recently announced attack against the SSL protocol took advantage of the difference in the time taken by the current OpenSSL implementation to respond to different errors in the decryption of carefully chosen messages when using a symmetric-key block cipher in the CBC mode [5].

4.3.2 Power Analysis

Following his work on Timing Analysis, P. Kocher, together with researchers from his company Cryptography Research, introduced in 1998 a new form of attack on smart cards and cryptographic tokens called *power analysis.* These attacks are mounted by monitoring the token's power consumption. Based on the fact that the power consumption varies significantly during different steps of the cryptographic operation, an attacker can recover the secret information.

They defined two types of attacks: *Simple Power Analysis* attacks work by directly observing a system's power consumption. *Differential Power Analysis* attacks are more powerful, using statistical analysis and error correction techniques to extract information correlated to private keys.

Although these attacks are quite complex and require a high level of technical skill to implement, Kocher says "they can be performed using a few thousand US dollars of standard equipment and can often break a device in a few hours or less" [6]. These attacks are most effective against smart cards, and there are a number of techniques (both in hardware and in software) that can be used to prevent them.

4.3.3 Fault Analysis

Fault Analysis attacks work by exploiting errors on key-dependent cryptographic operations. These errors can be random, latent (e.g. due to bugs in the implementation) or induced. There are a number of fault analysis attacks against public-key and symmetric-key cryptographic devices. In 1997 Boneh, DeMillo and Lipton introduced an attack against RSA, which exploits possible errors on the RSA private operation in cryptographic devices (see [9]).

As noticed before, the RSA private operation is a very compute-intensive operation, consisting of a modular exponentiation using numbers typically in the range of 300 decimal digits. Many implementations of RSA decryption and signing use a technique known as the Chinese Remainder Theorem (CRT), which by working modulo p and q (instead of module $n = pq$), can give a four-fold improvement in the performance. Boneh, DeMillo and Lipton described a technique, which by exploiting an error occurring during the decryption or signing and analysing the output, an adversary could factor the modulus n and therefore recover the device's private key. Both the output and the input of the operation are necessary for the attack to succeed (making it more effective against signing devices). To perform this kind of attack, one needs only to induce an error into the device during the private operation (for example, by voltage or clock speed variation). We should also note that unlike many of the attacks described before, the difficulty of this one is independent of the key length.

This fault attack against RSA can be easily prevented by requiring the device to verify the operation with the public key before outputting the result. This should not particularly degrade the performance, as the RSA public operation is very fast[6]. Also we note that the attacker needs to have full knowledge of the actual

[6] OpenSSL currently does that, resulting in a 5% slowdown.

string being signed or decrypted; the addition of random padding that is unknown to the attacker by the device prior to the operation would also prevent this type of attack. Nevertheless the existence of fault analysis attacks stresses the need for special care when implementing secure cryptographic applications.

4.3.4 Failure Analysis

Failure analysis exploits feedback from the implementation indicating success or failure of the decryption operation. Attacks using failure analysis are generally adaptive chosen ciphertext attacks, and an application performing the decryption could be seen as an *oracle* that tests the validity of the transmitted ciphertext. An example of this type of attack was introduced by D. Bleichenbacher in 1998 and is known as the *Million Message Attack* (see [1]). This attack exploits the cryptographic message syntax of some implementations.

Prior to RSA encryption, a message is usually padded by adding a few random bytes and some form of redundancy, which can be used later to check correctness of decryption. A widely used encoding format, defined in [19], is known as PKCS#1 v1.5. This is the encoding used by a number of cryptographic protocols, including SSLv3. When using this method, a message M after being encoded would have the following format:

$$\mathrm{PM} = 00\ 02 \,||\, \mathrm{PS} \,||\, 00 \,||\, \mathrm{M},$$

where 00 and 02 are bytes of value 0 and 2, PS is a padding string of non-zero random bytes, and || denotes concatenation. Now the string PM would be encrypted with the RSA public key, and the corresponding ciphertext transmitted.

At the other end, the application (usually a web server) decrypts the ciphertext with its private key, checks if the result has the expected format, and if so, removes the padding and continues with the processing of M.

A problem with the PKCS#1 v1.5 encoding format is that it does not have a property known as *plaintext-awareness*. Informally speaking, this means that it is quite feasible to generate a ciphertext, of which the decryption has a valid format without actually knowing the corresponding plaintext. Using this fact Bleichenbacher proposed an attack in which one could decrypt a ciphertext of choice based on the responses he would get from the application following the decryption of carefully chosen ciphertexts (a typical example of *adaptive chosen ciphertext attack*). For example, if the adversary wanted to decrypt a ciphertext C, he would generate and transmit a related ciphertext C'. If the decryption of C' did not have the expected format, some SSL implementations would return an error indicating “decryption failed”. On the other hand, if the decryption did have the expected format, the implementation would then continue processing the message M', which one expected to be the SSL session’s pre-master secret, but it is probably just a random string. This would probably cause a failure at the end of the SSL handshake, and the error returned by the server would be different. Depending on what response he got, the attacker could choose the next ciphertext to send and eventually after around 2^{20} queries (for a typical 1024-bit modulus), he would decrypt the original ciphertext C.

To prevent this attack, one could make sure that implementations returned the same kind of error in both of the situations above, preferably at the same stage during the negotiation (to preclude timing attacks as well). Another solution is to use another encoding system, which has stronger security properties. PKCS#1 v2.1 defines another encoding format known as OAEP [19], which is plaintext-aware and has some other security attributes [7].

5 Prevention and Countermeasures

Years of cryptanalysis of RSA have provided us some very clever attacks, and although no devastating attack has ever been found, there are a number of issues users and developers alike must be aware when working with RSA. Points that deserve special attention are: key size, properties of parameters (primes, exponents), encoding and implementation details.

Key Size – As a rule, RSA keys should be long enough so as to make attacks against the system infeasible. The choice should take into consideration a number of factors such as the value of data being protected, the expected lifetime of the data, the threat model, and the best possible attacks. Many of the current standards require a minimum length of 1024 bits for RSA keys. Since the solution of the RSA-155 challenge, there is a general consensus that 512-bit keys are too short and should not be used.

The Number Field Sieve method is the most effective general-purpose factorization method and is typically used to derive lower bounds for RSA key sizes. The NFS has two distinct phases, the sieving phase and the matrix phase. The sieving phase requires thousands of computers running in parallel. The space required for the matrix phase seems to be the biggest practical problem to scale current attacks.

In [21] Silverman concludes that 1024-bit keys should be secure for at least another 20 years. His estimate follows a cost-based analysis, which takes into account not only time complexity, but also the cost of memory (space) and the difficulty of scaling hardware. The National Institute of Standard and Technology (NIST) Key Management Guideline [15] recommends the use of RSA keys with length of at least 1024 bits for data that needs to be protected no later than the year 2015. For data that needs to be protected beyond that, the key size should be at least 2048 bits.

In 2002 D. Bernstein published a paper in which he proposed some improvements to NFS, mostly to do with implementation issues (parallelization). By also following a cost-based analysis, he estimated that the cost of mounting attacks against RSA *for very large key sizes* might not be as great as previously thought. His paper did cause concern to some, followed even by announcements of revocation of 1024-bit keys [14]. Still most experts do not believe the practical key sizes (e.g. 1024-bit) are impacted by his new methods. RSA Laboratories considers NIST guidelines to be reasonable, while stressing that the value of data must always be taken into account [20].

[7] although J. Manger has also described a side-channel attack on RSA-OAEP.

One should however note that cryptanalysis is a science that is steadily advancing. Silverman defends a flexible, pro-active security policy, where key sizes are not statically defined and therefore can be increased with any improvement in factoring techniques.

Strong Primes - Apart from a minimal length, some cryptographic standards also require that the primes used for RSA have some special properties. In particular, that $p-1$ needs to have a large prime factor. These so-called *strong primes*[8] are required in order to prevent Pollard's $p-1$ factorization method (see 4.1.2). In spite of that, Rivest and Silverman find these requirements unnecessary [17]. In view of the Elliptic Curve and Number Field Sieve methods (which have better runtime than Pollard's $p-1$ method), they believe that strong primes offer a negligible increase in security when compared with random primes. The real security comes from actually choosing large enough primes p and q.

Multi-prime RSA – To speed up the RSA private operation, there are proposals of using more than two primes to generate the modulus. Using CRT, the speedup of an RSA system using k primes over the standard RSA is of around $k^2/4$. For example, a system deployed with 1024-bit modulus n, which is the product of 4 distinct (256-bit) primes, would perform around 4 times faster than the standard two-prime system. While the Number Field Sieve method cannot take advantage of this special form of n, 256-bit primes are currently considered within the bounds of the Elliptic Curve method (which is quite effective in finding small primes). Therefore, it is not recommended that 1024-bit moduli use more than three factors [4].

Public Exponent – In most RSA implementations, the public exponent e is usually chosen to be either 3 or $2^{16} + 1$ (= 65,537), with which the public operation takes 2 and 17 modular multiplications, respectively (instead of ~ 1000 multiplications expected for a randomly chosen e). In view of some of the attacks presented in section 4.2, we believe 65,537 is more secure and should be used.

Private Exponent – As described in section 4.2.1, Wiener's low private exponent attack is quite effective, and if successful, can result in total recovery of the private exponent d. For the typical 1024-bit key, the private exponent should be at least 300 bits long. One should also pay special attention to the safeguarding of the private key. We saw that the knowledge of a fraction of the key might allow the recovery of the entire key. The hardware solution is the most secure and should be considered whenever possible.

Encoding – The RSA algorithm applied to messages without any kind of preprocessing (known as *raw RSA*) offers a very weak level of security and should never be used. Messages should always be encoded prior to encryption or signing. The study of properties of the encoding method to be used is an area of active research, as it has great impact in the overall security of the resulting cryptosystem. There are a number of different encoding methods defined, with all

[8] Other definitions of *strong primes* may require further properties.

of them adding enough randomness as well as some redundancy. Special care is needed when using small public exponent and short padding. PKCS#1 [19] defines two encoding (padding) methods for encryption: PKCS#1v1.5 is a simple, ad-hoc design, which is widely deployed (SSL/TLS). Optimal Asymmetric Encryption Padding (OAEP) method is more complex, but has much better security properties.

Implementation – Given the amount of research done on the underlying mathematical structure of RSA, one can probably say that currently the greatest threats to the security of the RSA cryptosystem are flawed implementations. In fact, the existence of side-channel attacks shows that extensive study of the mathematical structure of the RSA algorithm is not enough. Timing attacks are particularly effective, especially ones that exploit feedback from the implementation. Therefore developers and designers of cryptographic applications and protocols need to pay special care and attention to implementation details, in particular leakage of information and error handling.

6 Conclusions

The RSA cryptosystem is the *de facto* standard for public-key encryption and signature worldwide. It is implemented in the most popular security products and protocols in use today, and can be seen as one of the basis for secure communication in the Internet. Its underlying function and properties have been extensively studied by mathematicians and security professionals for more than a quarter of a century. While a number of attacks have been devised during this period, exploiting special properties of the RSA function as well as details in particular implementations, it has stood up well over the years and its security has never been put in doubt. No devastating attack has ever been found and most problems appear to be the result of misuse of the system, bad choice of parameters or flaws in implementations. In fact, years of research have probably increased the trust the security community has on RSA, and we have every reason to believe that it will remain the most used public-key algorithm for years to come.

7 References

[1]. D. Bleichenbacher, B. Kaliski and J. Staddon. *Recent Results on PKCS #1: RSA Encryption Standard.* RSA Laboratories' Bulletin No. 7, June 1998. ftp://ftp.rsasecurity.com/pub/pdfs/bulletn7.pdf

[2]. D. Boneh. Twenty Years of Attacks on the RSA Cryptosystem. *Notices of the American Mathematical Society, 46(2):203--213, 1999.* http://crypto.stanford.edu/~dabo/abstracts/RSAattack-survey.html

[3]. D.Boneh and D. Brumley. *Remote Timing Attacks are Practical.* 2003 http://crypto.stanford.edu/~dabo/abstracts/ssl-timing.html

[4]. D. Boneh and H. Shacham. *Fast Variants of RSA.* CryptoBytes. Volume 5, No. 1 – RSA Laboratories. http://www.rsasecurity.com/rsalabs/cryptobytes/CryptoBytes_January_2002_final.pdf

[5]. B. Canvel. *Password Interception in a SSL/TLS Channel*. http://lasecwww.epfl.ch/memo_ssl.shtml
[6]. Cryptography Research. *Differential Power Analysis (DPA)* http://www.cryptography.com/resources/whitepapers/DPA.html
[7]. W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Trans. Inform. Theory IT-22* (1976), 644-654
[8]. B. Kaliski. *Timing Attacks on Cryptosystems.* RSA Laboratories' Bulletin No. 2, January 1996. ftp://ftp.rsasecurity.com/pub/pdfs/bull-2.pdf
[9]. B. Kaliski and M. Robshaw. *Comments on Some New Attacks on Cryptographic Devices.* RSA Laboratories' Bulletin No. 5, July 1997. ftp://ftp.rsasecurity.com/pub/pdfs/bulletn5.pdf
[10]. N. Koblitz. *A Course in Number Theory and Cryptography*. 2nd Edition. Springer-Verlag, 1994
[11]. E. Landquist. *The Quadratic Sieve Factoring Algorithm.* http://www.math.uiuc.edu/~landquis/quadsieve.pdf
[12]. A. Lenstra. *Computational Methods in Public Key Cryptology* Available at http://citeseer.nj.nec.com/lenstra01computational.html
[13]. A. Menezes, P. van Oorschot and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, October 1996 Available for free download at http://www.cacr.math.uwaterloo.ca/hac/
[14]. News Group Discussion. *1024-bit RSA keys in Danger of Compromise.* BugTraq Archive, March 2002 http://www.securityfocus.com/archive/1/263924
[15]. NIST. *Key Management Guideline - Workshop Document*. Draft, October 2001. http://csrc.nist.gov/encryption/kms/key-management-guideline-(workshop).pdf
[16]. R. Rivest, A. Shamir and L. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems*. Communications of ACM 21 (1978) 120-125* http://theory.lcs.mit.edu/~cis/pubs/rivest/rsapaper.ps
[17]. R. Rivest and R. Silverman. *Are Strong Primes Needed for RSA?* RSA Laboratories Technical Notes and Reports ftp://ftp.rsasecurity.com/pub/pdfs/sp2.pdf
[18]. RSA Laboratories. *Factorization of RSA-155*. http://www.rsasecurity.com/rsalabs/challenges/factoring/rsa155.html
[19]. RSA Laboratories. *PKCS #1 v2.1 - RSA Encryption Standard*. June 2002. http://www.rsasecurity.com/rsalabs/pkcs/pkcs-1/index.html
[20]. RSA Laboratories. *Has the RSA Algorithm been compromised as result of Bernstein's paper?* RSA Laboratories Technical Notes. April 2002 http://www.rsasecurity.com/rsalabs/technotes/bernstein.html
[21]. R. Silverman. *A Cost-Based Security Analysis of Symmetric and Asymmetric Key Lengths*. RSA Laboratories' Bulletin No. 13, April 2000. http://www.rsasecurity.com/rsalabs/bulletins/bulletin13.html
[22]. A. Shamir and N. van Someren. *Playing hide and seek with stored keys.* Lectures in Computer Science, 1998 http://www.simovits.com/archive/keyhide2.pdf

Last Updated: July 2nd, 2020

Upcoming SANS Training

Click here to view a list of all SANS Courses

SANS Amsterdam August 2020 Part 1	**Amsterdam, NL**	**Aug 03, 2020 - Aug 08, 2020**	**Live Event**
SANS Reboot - NOVA 2020	**Arlington, VAUS**	**Aug 10, 2020 - Aug 15, 2020**	**Live Event**
SANS FOR508 Canberra August 2020	**Canberra, AU**	**Aug 17, 2020 - Aug 22, 2020**	**Live Event**
SANS Amsterdam August 2020 Part 2	**Amsterdam, NL**	**Aug 17, 2020 - Aug 22, 2020**	**Live Event**
SANS Virginia Beach 2020	**Virginia Beach, VAUS**	**Aug 30, 2020 - Sep 04, 2020**	**Live Event**
SANS Philippines 2020	**Manila, PH**	**Sep 07, 2020 - Sep 19, 2020**	**Live Event**
SANS London September 2020	**London, GB**	**Sep 07, 2020 - Sep 12, 2020**	**Live Event**
SANS Baltimore Fall 2020	**Baltimore, MDUS**	**Sep 08, 2020 - Sep 13, 2020**	**Live Event**
SANS Munich September 2020	**Munich, DE**	**Sep 14, 2020 - Sep 19, 2020**	**Live Event**
SANS Network Security 2020	**Las Vegas, NVUS**	**Sep 20, 2020 - Sep 25, 2020**	**Live Event**
SANS Northern VA - Reston Fall 2020	**Reston, VAUS**	**Sep 28, 2020 - Oct 03, 2020**	**Live Event**
SANS San Antonio Fall 2020	**San Antonio, TXUS**	**Sep 28, 2020 - Oct 03, 2020**	**Live Event**
SANS OnDemand	**OnlineUS**	**Anytime**	**Self Paced**
SANS SelfStudy	**Books & MP3s OnlyUS**	**Anytime**	**Self Paced**