9. The RSA Trapdoor Permutation (cont'd)

9.3.2 Improving Performance of RSA

Using Small Exponents e. A common way to speed up the computation of the RSA trapdoor permutation is to use small values of e, which leads to a fast evaluation of the RSA function. The minimal value e possibly satisfying $\gcd(e, \varphi(N)) = 1$ is obviously $e = 3$ since $\gcd(2, \varphi(N)) = 2$. However, relying on values e that small should be avoided since sending the same message encrypted with e different public keys (which all rely on the same public exponent e but different moduli N_i) allows an attacker already to retrieve the message from the ciphertexts. The recommended value of e is $e = 65537 = 2^{16} + 1$, as this allows for computing the RSA trapdoor permutation using only 17 modular multiplications.

Exploiting the Chinese Remainder Theorem One can exploit the Chinese Remainder Theorem to compute RSA inverses separately modulo p and q. For computing $c^d \bmod N$, one does the following:

- Compute u and v such that $up + vq = 1$.
- Compute $m_p := c^{d_p} \bmod p$ and $m_q := c^{d_q} \bmod q$ where $d_p = d \bmod p - 1$ and $d_q = d \bmod q - 1$.
- Set $m := upm_q + vqm_p \bmod N$.

The advantage is that all computations are carried out in significantly smaller groups, using exponents of only half the size, thus gaining roughly a factor of 4 in speed. The computational overhead is only two multiplications and one addition and thus can be ignored. Note that the values u and v can be computed once and for all and then used for inverting different values c.

Using Small Exponents d – Bad idea! A still pretty common efficiency improvement is to make the value d small in order to speed up the computation of the inverse of the RSA trapdoor permutation. However, Wiener proved in 1989 that, if $d < 1/3N^{0.25}$, then one can fully recover d given (N, e). We will describe this very nice attack in the next section. Boneh, Durfee, and Frankel improved this bound in 1998 to $d < N^{0.292}$. For $d < N^{0.5}$, it is expected that d can be recovered as well given (N, e), but it is still an open problem how this can be achieved.

9.4 Wiener's Attack Against Small Exponents d

In this section we describe Wiener's attack against RSA if the exponent d is small, more precisely, if we have $d < \frac{N^{0.25}}{3}$. Let $N = p \cdot q$ as usual and assume that $q < p < 2q$. As we have argued before, it is tempting for Bob to choose a small d, as this speeds up the inversion of RSA.

Since $ed = 1 \bmod \varphi(n)$, there is an integer k such that $ed - k\varphi(N) = 1$. Thus we have

$$\left|\frac{e}{\varphi(N)} - \frac{k}{d}\right| = \frac{1}{d\varphi(N)}.$$

From $N = pq > q^2$ it follows $q < \sqrt{n}$, hence

$$0 < N - \varphi(N) = p + q - 1 < 2q + q - 1 < 3q < 3\sqrt{n}.$$

Now we obtain

$$\begin{aligned}\left|\frac{e}{N} - \frac{k}{d}\right| &= \left|\frac{ed - kN}{dN}\right| \\ &= \left|\frac{1 + k(\varphi(N) - N)}{dN}\right| \\ &< \frac{3k\sqrt{N}}{dN} \\ &= \frac{3k}{d\sqrt{N}}.\end{aligned}$$

Since $k < d$ and $d < \frac{N^{0.25}}{3}$, we have that $3k < 3d < N^{0.25}$, and hence

$$\left|\frac{e}{N} - \frac{k}{d}\right| < \frac{1}{dN^{0.25}} < \frac{1}{3d^2}. \tag{9.1}$$

Note that the fraction $\frac{e}{N}$ can be computed based on public information. Thus, this means that the secret fraction $\frac{k}{d}$ is very close to the public fraction $\frac{e}{N}$ the adversary can compute. It turns out that in this case, the adversary can even compute $\frac{k}{d}$ efficiently using the so-called continued fraction expansion.

9.4.1 Continued Fractions

We give a brief introduction on continued fractions, only as far as we will need to show how Wiener's attack works. A *(finite) continued fraction (expansion)* is an m-tuple of non-negative integers

$$[q_1, \ldots, q_m],$$

which is an abbreviation of the following rational number:

$$q_1 + \cfrac{1}{q_2 + \cfrac{1}{q_3 + \ldots + \cfrac{1}{q_m}}}.$$

For $1 \leq j \leq m$, the tuple $[q_1, \ldots, q_j]$ is said to be the j-th convergent of $[q_1, \ldots, q_m]$.

Suppose a and b are positive integers such that $\gcd(a, b) = 1$, then the continued fraction expansion $[q_1, \ldots, q_m]$ of $\frac{a}{b}$ can be computed using the Euclidean algorithm. Here the quotients arising in the algorithm when applied to a and b yield the desired m-tuple. This is best demonstrated by means of an example. Assume we would like to compute the continued fraction expansion of $\frac{30}{53}$. Then applying the Euclidean Algorithm to 30 and 53 yields:

$$\begin{aligned} 30 &= 0 \cdot 53 + 30 \\ 53 &= 1 \cdot 30 + 23 \\ 30 &= 1 \cdot 23 + 7 \\ 23 &= 3 \cdot 7 + 2 \\ 7 &= 3 \cdot 2 + 1 \\ 2 &= 2 \cdot 1 + 0 \end{aligned}$$

Then the continued fraction expansion of $\frac{30}{53}$ is the tuple $[0, 1, 1, 3, 3, 2]$. Correctness of this method of computing the tuple can be checked by a simple calculation:

$$[0,1,1,3,3,2] = 0 + \frac{1}{1+\frac{1}{1+\frac{1}{3+\frac{1}{3+\frac{1}{2}}}}} = \frac{1}{1+\frac{1}{1+\frac{1}{3+\frac{2}{7}}}} = \frac{1}{1+\frac{1}{1+\frac{7}{23}}} = \frac{1}{1+\frac{23}{30}} = \frac{30}{53}$$

The following famous theorem, which we give without proof, will later allow us to break RSA for small values of d.

Theorem 9.1 *Let $a, b, c, d \in \mathbb{N}$. Assume that $\gcd(a, b) = \gcd(c, d) = 1$ and*

$$\left|\frac{a}{b} - \frac{c}{d}\right| < \frac{1}{2d^2}.$$

Then $\frac{c}{d}$ equals one of the convergents of the continued fraction expansion of $\frac{a}{b}$. □

9.4.2 Wiener's Attack

Now we exploit Theorem 9.1 to mount a successful attack against RSA if d is sufficiently small. Equation 9.1 immediately allows us to apply Theorem 9.1 since $\gcd(k, d) = 1$ by assumption and since $\gcd(e, N) \neq 1$ would immediately allow us to factor N. Thus, one of the convergents of the continued fraction expansion of the known fraction $\frac{e}{N}$ equals the unknown and searched fraction $\frac{k}{d}$. These prefixes can easily be computed, but it remains to define a suitable check to identify which convergent is the desired one.

For every convergent $\frac{x}{y}$, we compute $M := (ey-1)/x$. Note that if $\frac{x}{y} = \frac{k}{d}$, then $M = \varphi(N)$. We then try to factor N given N and M using the method for factoring N and $\varphi(N)$ which we currently explore on the exercise sheet. If we picked the correct convergent, i.e., if we indeed have $M = \varphi(N)$, this procedure will give us the correct prime factors p and q of N; for wrong convergents, no solution will exist.

Let us conclude with an illustration of this attack for artificially small parameters: Suppose $N = 160523347$ and $e = 60728973$. The continued fraction expansion of $\frac{e}{N}$ is given by $[0, 2, 1, 1, 1, 4, 12, 102, 1, 1, 2, 3, 2, 2, 36]$; it's first convergents are 0, $\frac{1}{2}$, $\frac{1}{3}$, $\frac{2}{5}$, $\frac{3}{8}$, $\frac{14}{37}$, and so on. For $\frac{14}{37}$, we obtain $M = \frac{60728973 \cdot 37 - 1}{14}$. This yields $p = 12347$, $q = 13001$, and indeed $p \cdot q = N$. Thus $d = 37$.

9.5 How to Securely Encrypt using RSA (or any OWF)

We have already seen that using RSA naively results in an insecure encryption scheme, as it is vulnerable to several attacks. The central idea to avoid these attacks is to pre-process the messages

before applying the RSA function to them. This is often called *padding* and is used for two main reasons. First, it adds randomness to the encryption, as otherwise distinguishing ciphertexts is obviously easy. Secondly, its partially fixed structure prevents blinding attacks, e.g., the ones we have seen for naive RSA and for ElGamal.

9.5.1 PKCS#1 V1.5

PKCS#1 V1.5 is a standard from 1991 which is widely deployed in practice. It uses an ad-hoc padding scheme, whose security was never deeply analyzed, and which unfortunately turned out to be insecure. Bleichenbacher came up with a successful attack against PKCS#1 V1.5 in 1998, which caused V1.5 to be replaced with V2.0 that relies on a more suitable padding.

Let F be a (family of) keyed trapdoor permutations with key generation Gen and domains $\mathcal{M}_{pk}$, e.g, the RSA trapdoor permutation. Let n be a natural number and let efficient and invertible embeddings $\nu_{pk} : \{0,1\}^n \to \mathcal{M}_{pk}$ be given for all $(pk, *) \in [\mathsf{Gen}(n)]$. We use these embeddings implicitly when applying $\mathsf{F}(pk, \cdot)$ to bitstrings of length n. Let a message $m \in \{0,1\}^{n_0}$ be given, where $n_0 < n - 96$. Then the scheme works as follows:

- Generate public and secret keys $(pk, sk) \leftarrow \mathsf{Gen}(k)$.
- To compute $c \leftarrow \mathsf{E}(pk, m)$ for $m \in \{0,1\}^{n_0}$, choose a string r of length 64 bits at random, and compute
$$c := \mathsf{F}(pk,\ 0002 \parallel r \parallel 0000 \parallel m),$$
where numbers are in hexadecimal notation (i.e., both 0002 and 0000 are of length 16 bits).
- To compute $\mathsf{D}(sk, c)$, compute $\mathsf{F}^{-1}(sk, c) =: x \parallel r \parallel y \parallel m$, where $|x| = |y| = 16$ and $|m| = n_0$. If $x \neq 0002$ output $\downarrow$, else output m.

Correctness is verified as follows:

$$\begin{aligned}
\mathsf{D}(sk, \mathsf{E}(pk, m)) &= \mathsf{D}(sk, \mathsf{F}(pk, 0002 \parallel r \parallel 0000 \parallel m)) \\
&= \begin{cases} m & \text{, if } x = 0002 \\ \downarrow & \text{, else} \end{cases} \\
&\quad \text{where } x \parallel r \parallel m := \mathsf{F}^{-1}(sk, \mathsf{F}(pk, 0002 \parallel R \parallel 0000 \parallel m)) \text{ for } |x| = 16, |m| = n_0 \\
&= m.
\end{aligned}$$

The Bleichenbacher Attack Bleichenbacher's attack (which is specific to SSL which uses PKCS#1 V1.5) exploits the way the above padding interacts with the SSL implementation: After some handshake the browser randomly chooses a secret key K, pads the key as explained above and applies the RSA trapdoor permutation to this padded value. This ciphertext is then sent to the server. The server decrypts the ciphertext, and if it notices that the resulting plaintext is misformed, i.e., if its 16 most significant bits do not equal 0002, then it returns $\downarrow$. Else it answers with a different message, indicating that the key exchange was successfully completed.

Consider the following attacker: Given an arbitrary ciphertext c that it would like to decrypt, the attacker computes $c' := r^e \cdot c \bmod N$ for a randomly chosen value r and the public encryption key (e, N) of the server, and sends c' to the server yielding either an error or a confirmation message. Thus the answer of the server implicitly reveals if the underlying plaintext was of the correct format. The attacker repeats this step roughly four million times for different values r and collects all the answers. Every such answer gives the attacker one bit of information about the various plaintexts corresponding to the ciphertexts constructed in the above manner, and it turns out that using the

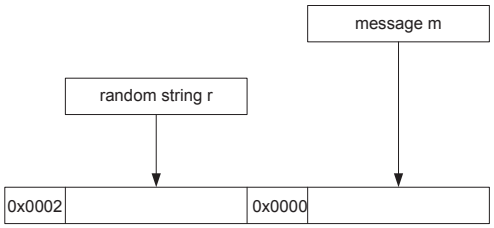

Figure 9.1: PKCS#1 V1.5

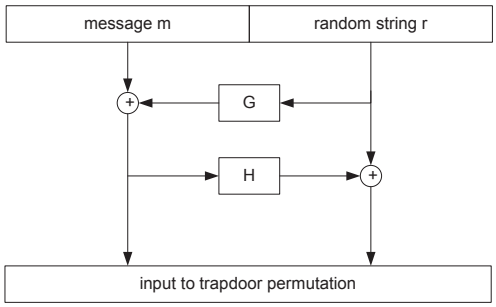

Figure 9.2: OAEP

homomorphic properties of RSA, an attacker can use this leaked information to fully reconstruct the plaintext corresponding to ciphertext c. Showing this calculation in detail is lengthy though, so that we refer to Bleichenbacher's original paper for details.

9.5.2 PKCS#1 V2.0 – OAEP

OAEP constitutes a generic method for constructing encryption schemes based on arbitrary keyed trapdoor permutations. Let a family of keyed trapdoor permutations F with key generation Gen and domains $\mathcal{M}_{pk}$ be given. Furthermore, let l, n be integers with $l < n$, and let $n_0 := n - l$. Let efficient and invertible embeddings $\nu_{pk}\colon \{0,1\}^n \to \mathcal{M}_{pk}$ be given for all $(pk, *) \in [\mathsf{Gen}(n)]$, which we again use implicitly when applying $\mathsf{F}(pk, \cdot)$ to bitstrings of length n. Let $G\colon \{0,1\}^{n_0} \to \{0,1\}^l$ and $H\colon \{0,1\}^l \to \{0,1\}^{n_0}$ be hash functions. The OAEP encryption scheme then works as follows:

- Generate public and secret keys $(pk, sk) \leftarrow \mathsf{Gen}(n)$.
- To compute $c \leftarrow \mathsf{E}(pk, m)$ for $m \in \{0,1\}^l$ first choose a random $r \leftarrow_{\mathcal{R}} \{0,1\}^{n_0}$ and let

$$c := \mathsf{F}(pk, m \oplus G(r) \parallel r \oplus H(m \oplus G(r))).$$

  This is illustrated in Figure 9.2.
- To compute $\mathsf{D}(sk, c)$ one computes $\mathsf{F}^{-1}(sk, c) =: x_1 \parallel x_2$, where $|x_1| = l$, and $|x_2| = n_0$. Then one calculates $m := x_1 \oplus G(x_2 \oplus H(x_1))$.

To see that decryption is correct one calculates

$$\begin{aligned}
\mathsf{D}(sk, \mathsf{E}(pk, m)) &= \mathsf{D}(sk, \mathsf{F}(pk, m \oplus G(r) \parallel r \oplus H(m \oplus G(r)))) \\
&= x_1 \oplus G(x_2 \oplus H(x_1)) \text{ for } x_1 \parallel x_2 = \mathsf{F}^{-1}(sk, \mathsf{F}(pk, m \oplus G(r) \parallel r \oplus H(m \oplus G(r)))) \\
&= (m \oplus G(r)) \oplus G((r \oplus H(m \oplus G(r))) \oplus H(m \oplus G(r))) \\
&= (m \oplus G(r)) \oplus G(r) \\
&= m.
\end{aligned}$$

The Random Oracle Model A proof of OAEP was only attempted in the so-called *Random Oracle model*, which constitutes an idealization of hash functions. Intuitively, instead of working with the correct properties of hash functions, e.g., one-wayness or collision-resistance, one assumed that the hash function is replaced by a truly random function. This random function is then written as a so-called *random oracle* as follows: (i) If a value is hashed for the first time, its hash value is

chosen randomly and independently of previous values, and (ii) if a value should be hashed that was already hashed before, then the (previously stored) same hash value is returned. The random oracle model thus constitutes an idealization of hash functions in order to allow for much simpler proofs. However, there is no theorem stating that these proofs remain valid if the random oracle is implemented with an actual hash function in reality, and there even exist results showing that in certain cases, a random oracle cannot be securely realized by any hash function. Thus proofs in the random oracle model should essentially be considered heuristics. (Recall that the Cramer-Shoup encryption scheme did not have to abstract its hash function into a random oracle which was one of the main reasons why this encryption scheme got a lot of attention.)

On the Security of OAEP There traditionally was a security proof of OAEP in the random oracle model. However, Shoup found a gap in this proof in 2000, and showed that this gap cannot be closed in general, i.e., that CCA2-security of OAEP cannot be proved for arbitrary trapdoor-permutations.[1] After discovering this gap, Shoup also proposed an improved construction OAEP+, which we will see below, and proved it CCA2-secure in the random oracle model. After Shoup's work, Okamoto et al. showed that OAEP based on the RSA trapdoor permutation is CCA2-secure. The resulting encryption scheme is often called RSA-OAEP, and its security could only be proven by exploiting properties similar to random self-reducibility that are specific to RSA.

9.5.3 OAEP+

OAEP+ constitutes a generic method for constructing encryption schemes based on arbitrary keyed trapdoor permutations, similar to OAEP. Let a family of keyed trapdoor permutations F with key generation Gen and domains $\mathcal{M}_{pk}$ be given. Let k, l, n be integers with $k + l < n$ such that both 2^{-k} and 2^{-l} are negligible in n. Let efficient and invertible embeddings $\nu_{pk} : \{0,1\}^n \to \mathcal{M}_{pk}$ be given for all $(pk, *) \in [\mathsf{Gen}(n)]$ as before. Let $G\colon \{0,1\}^k \to \{0,1\}^{n-k-l}$, $H'\colon \{0,1\}^{n-l} \to \{0,1\}^l$, and $H\colon \{0,1\}^{n-k} \to \{0,1\}^k$ be hash functions. The OAEP+ encryption scheme then works as follows:

- Generate public and secret keys $(pk, sk) \leftarrow \mathsf{Gen}(n)$.
- To compute $c \leftarrow \mathsf{E}(pk, m)$ for $m \in \{0,1\}^{n-k-l}$, choose a random $r \leftarrow_{\mathcal{R}} \{0,1\}^k$ and let

$$\begin{aligned} s &:= (G(r) \oplus m) \parallel H'(r \parallel m), \text{ and} \\ c &:= \mathsf{F}(pk, s \parallel (H(s) \oplus r)). \end{aligned}$$

  This is illustrated in Figure 9.3.
- To compute $\mathsf{D}(sk, c)$ one computes $\mathsf{F}^{-1}(sk, c) =: x_1 \parallel x_2 \parallel x_3$ where $|x_1| = n - k - l$, $|x_2| = l$, and $|x_3| = k$. Then one calculates

$$m = G(H(x_1 \parallel x_2) \oplus x_3) \oplus x_1.$$

  If $x_2 = H'(x_3 \oplus H(x_1 \parallel x_2) \parallel m)$ then output m as the decryption, otherwise reject the ciphertext.

Correctness of decryption can be shown as usual. Shoup showed that if F is a family of keyed trapdoor permutations, then OAEP+ is CCA2-secure in the random oracle model.

[1] This gap essentially arose from confusions with different security definitions that will not matter in the sequel. We only point out that OAEP is CCA1-secure, which is a strictly weaker notion of security than CCA2-security.

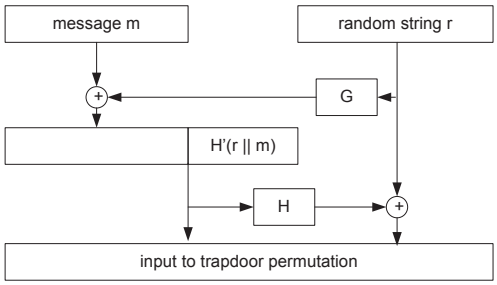

Figure 9.3: OAEP+

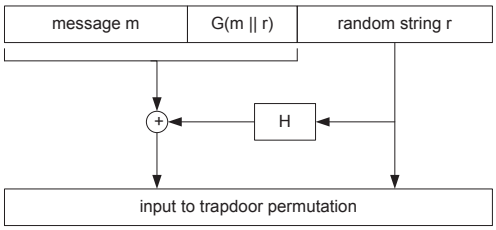

Figure 9.4: SAEP+

9.5.4 RSA-SAEP+

SAEP+ was introduced by Boneh in 2001 as a construction that is simpler than OAEP+ while additionally providing tighter security proofs. As a minor drawback, it can only be used with the RSA and the so-called Rabin trapdoor permutations. We will concentrate on RSA-SAEP+ in the sequel, i.e., SAEP+ based on the RSA trapdoor permutation.

Let a family of keyed trapdoor permutations F with key generation Gen and domains $\mathcal{M}_{pk}$ be given. Let j, k, l, n be integers with $n = j + k + l$ and $j > n/2$, and let efficient and invertible embeddings $\nu_{pk} : \{0,1\}^n \to \mathcal{M}_{pk}$ be given for all $(pk, *) \in [\mathsf{Gen}(n)]$ as before. Let $G\colon \{0,1\}^{k+l} \to \{0,1\}^j$, and $H\colon \{0,1\}^l \to \{0,1\}^{j+k}$ be hash functions. The RSA-OAEP+ encryption scheme then works as follows:

- Generate public and secret keys $(pk, sk) \leftarrow \mathsf{Gen}(n)$.
- To compute $c \leftarrow \mathsf{E}(pk, m)$ for $m \in \{0,1\}^k$, choose a random $r \leftarrow_{\mathcal{R}} \{0,1\}^l$ and let

$$c := \mathsf{F}(pk, H(r) \oplus (m \parallel G(m \parallel r)) \parallel r).$$

  This is illustrated in Figure 9.3.
- To compute $\mathsf{D}(sk, c)$ one computes $\mathsf{F}^{-1}(sk, c) =: x_1 \parallel x_2$, where $|x_1| = j + k$ and $|x_2| = l$. Then one calculates

$$m \parallel y_2 := x_1 \oplus H(x_2) \text{ where } |m| = k \text{ and } |y_2| = j.$$

  If $y_2 = G(m \parallel x_2)$ then output m as the decryption, otherwise reject the ciphertext.

Correctness can be shown as usual. Similar to OAEP+, RSA-SAEP+ is CCA2-secure provided that RSA is indeed a trapdoor permutation.

9.5.5 Implementation Attacks on Encryption Schemes

The attacks we have investigated so far all exploited the input/output behavior of the encryption scheme. In addition to this, measuring the time a decryption algorithm needs to perform a specific computation might leak information on the secret decryption key. Attacks that measure timing characteristics of cryptographic algorithms are called *timing attacks*. Exponentiations $c^d \bmod N$ are usually carried out by the repeated squaring technique, i.e., for each bit of the secret exponent d the algorithm either performs a squaring operation, or a squaring operation and a multiplication. Thus in the first case, the algorithm will clearly proceed faster so that simply measuring the execution time reveals the hamming-weight of the secret exponent d. Statistical analysis of a large number

of such timings then allows for recovering the complete value of d. Note that timing attacks can be carried out easily if one has access to a computer performing cryptographic operations, but sometimes even remote attacks are possible by measuring reaction times of servers provided that the network latency is sufficiently low.

A more sophisticated type of attacks are *power* and *emanation attacks*, where an attacker can measure the current energy a microprocessor is consuming while performing operations involving a secret key. For naive implementations of an algorithm one can sometimes directly see secret bits in the measured power consumption curves, e.g., when a register is shifted cyclically. This attack type is a severe threat against smartcards used, e.g., as signature cards, as the manipulation of the reader might remain undetected.

Fault attacks are slightly more esoteric in practice but very interesting from a conceptional point of view. By injecting errors into computations involving a secret key, an attacker might be able to provoke a different result that might be helpful for recovering secret information. These faults often are inserted by radiation such as X-ray, but also heating of microprocessors or memory can lead to errors that can be successfully exploited.