CS5431 – Computer Security Practicum Spring 2017

Lecture 4: RSA in Practice

February 17, 2017 *Instructor: Eleanor Birrell*

1 Intro to RSA

Asymmetric encryption is an integral part of the modern world; it is used to bootstrap secure communication channels and secure storage solutions. The security of encrypted data is built on a sophisticated mathematical theory developed over the last forty years. But building a secure cryptosystem depends on more than just theory. In this class we will take a detailed look at the theory and practice of RSA, an asymmetric encryption algorithm first developed in the 1970s and still in widespread use today.

2 Theoretical Foundations

The theory behind RSA depends on classic results from number theory. To understand how RSA encryption and decryption are performed, and why these operations produce correct plaintext, we will begin by reviewing some necessary mathematical background.

For RSA, messages are encoded as integers, and operations are performed with respect to a finite subset of the integers. This is an example of *modular arithmetic*.

Modular Arithmetic. Modular arithmetic is like standard arithmetic (multiplication or addition) on integers except it is performed on a finite subset of non-negative integers $\{0, \ldots, n-1\}$, that is *modulo* n. If the product (or sum) of two integers a, b is greater than n, then the product (or sum) modulo n is defined to be the remainder of ab/n (respectively $(a+b)/n$).

More specifically, an RSA modulus is chosen to be the product of two (large) prime numbers $n = pq$, and a pair of numbers is chosen satisfying the property that $ed = 1 \bmod (p-1)(q-1)$. The private key is defined to be the triple (p, q, d) and the public key is the pair (n, e).

To encrypt a message using RSA, you encode the message as an integer $m < n$ and compute the ciphertext

$$c = m^e \bmod n.$$

To decrypt a message, the receiver computes

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

Note that RSA can also be used to generate digital signatures; the decryption function is used to generate signatures and the encryption function is used to verify signatures.

Signing and verification keys are generated the same way as decryption and encryption keys, but the same keys should never be used for both signing and encryption

In order to understand why RSA is correct, it helps to assume some classic results from number theory.

Chinese Remainder Theorem. The Chinese Remainder Theorem—so called because it was first proven by the Chinese mathematician Sunzi in the third century A.D.—implies that for any two primes p and q and any two numbers a, b if $a \equiv b \bmod p$ and $a \equiv b \bmod q$ then $a \equiv b \bmod pq$. Consider, for example, $p = 2, q = 3$:

$1 \bmod 2 = 7 \bmod 2 = 1$	$1 \bmod 3 = 7 \bmod 3 = 1$	$1 \bmod 6 = 7 \bmod 6 = 1$
$2 \bmod 2 = 8 \bmod 2 = 0$	$2 \bmod 3 = 8 \bmod 3 = 2$	$2 \bmod 6 = 8 \bmod 6 = 2$
$3 \bmod 2 = 9 \bmod 2 = 1$	$3 \bmod 3 = 9 \bmod 3 = 0$	$3 \bmod 6 = 9 \bmod 6 = 3$

Fermat's Little Theorem. Fermat's little theorem—first stated by Pierre de Fermat in 1640—implies that for any prime p and any $a < p$, $a^{p-1} \bmod p \equiv 1$. Consider, for example, the value $p = 5$:

$$1^4 = 1 = 1 \bmod 5$$
$$2^4 = 16 = 1 \bmod 5$$
$$3^4 = 81 = 1 \bmod 5$$
$$4^4 = 256 = 1 \bmod 5$$

Using these two theorems, we can prove that RSA will correctly decrypt the original plaintext message:

$$c^d \bmod p = (m^e \bmod n)^d \bmod p = m \cdot m^{ed-1} \bmod p = m \cdot m^{\ell(p-1)} \bmod p = m \bmod p$$

and likewise

$$c^d \bmod q = (m^e \bmod n)^d \bmod q = m \cdot m^{ed-1} \bmod q = m \cdot m^{\ell'(q-1)} \bmod q = m \bmod q$$

therefore the Chinese Remainder Theorem implies that

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

All this math only proves that the receiver (who knows the private key) will be able to correctly retrieve the original messages; it does not prove that an encrypted ciphertext is secure. In fact there is no proof of security for RSA. However, the algorithm was first proposed forty years ago and no one has yet announced a way to break the theory.

3 Performance Optimizations

In order for RSA ciphertexts to remain secure, the adversary must not be able to factor the public modulus n. If an adversary can calculate $pq = n$ then they can compute the secret exponent d from the public exponent e and thereby decrypt the ciphertext. In order to be confident in the infeasibility of factoring n, people tend to choose very large numbers. The National Institute of Science and Technology (NIST) currently recommends a minimum of 2048-bit keys. But performing the necessary arithmetic functions with numbers that size is expensive, so most implementations have adopted one or both of two proposed optimizations.

Chinese Remainder Algorithm. This algorithm, based on the Chinese remainder theorem, improves the performance of RSA decryptions by storing five (precomputed) values along with the private modulus d: $p, q, d \bmod p-1, d \bmod q-1, q^{-1} \bmod p$. Using these stored values, a ciphertext can be decrypted by computing

$$\begin{aligned} m_1 &= c^{d \bmod p-1} \bmod p \\ m_2 &= c^{d \bmod q-1} \bmod q \\ h &= q^{-1}(m_1 - m_2) \bmod p \\ m &= m_2 + hq \end{aligned}$$

Without getting into the details of why this is correct, we can observe that this algorithm is faster than a naive decryption dispite requiring two modular exponentiations because the moduli p, q are significantly smaller than $n = pq$. Many popular crypto libraries, including OpenSSL and Java's crypto libraries have adopted this algorithm to improve decryption performance.

Square-and-Multiply If you have the exponent stored as a binary number, which of course a computer does, then you can improve over a naive modular exponentiation using a technique known as square-and-multiply, summarized by the pseudocode below

```
res = 1;
while(exp > 0){
    if(exp % 2 == 1){
        res = (res*base) % p;
    }
    base = base^2 % p;
    exp >> 1;
}
return res;
```

Implementing this algorithm for computing modular exponentiation significantly improves performance and has been adopted by all major crypto libraries.

4 RSA in Practice

Although there have been no successful attacks to date that have compromised the underlying theory, practical implemenations are another matter entirely. In fact, there have been numerous successful attacks both in the research literature and in the wild. In some cases, attacks leveraged a vulnerability unique to a particular implementation. In other cases, attacks took advantage of an inherent vulnerability in the RSA protocol. r

4.1 Choosing Good Keys

As discussed earlier, the security of RSA relies on the assumption that the modulus n is hard do factor. However, not all numbers are hard to factor in practice. Consider the algorithms taught in grade school for recognizing factors of 2,3,9,11, etc. In fact, it turns out that selecting $n = pq$ to be the product of two large prime numbers does not necessarily make n hard to factor.

If p and q are "too close" (approximately $pq < 2n^{1/4}$), n can be factored using a technique called Fermat factorization. Checking that the selected key avoids this vulnerability can be efficiently checked during key generation.

If either $p - 1$ or $q - 1$ has only small factors, then n can be factored using a technique called Pollard's algorithm. Strong primes (those for which $p \pm 1$ have at least one factor with greater than 100 bits) are immune to Pollard's algorithm, so some authorities (including the ANSI X9.31 standard released by the American National Standards Institute) require that RSA keys use only strong primes. However, the difficulty of identifying strong primes and the existence of other approaches to factorization (e.g., Lenstra elliptic curve factorization and sieve-based approaches) has caused other authorities (including RSA Security) to reject this recommendation.

Another consideration is choosing the value of the public modulus e. If e is small, then messages encoded as low-value integers (specifically those for which $m^e < n$) can be easily decrypted simply by computing the eth root of the observed ciphertext. This vulnerability is often eliminated by fixing the public exponent to be a suitable value, generally 65537. In fact, NIST issued a report in 2007 prohibiting public moduli less than 65537.

4.2 Randomness Quality

Thus far, we have focused the importance of selecting prime numbers p, q such that n can't be easily factored. It is equally important to select primes that can't be easily *guessed*.

The standard approach to generating a key is to find primes p, q by picking random numbers and then checking (using Miller-Rabin randomized primality testing) whether the randomly selected numbers are prime. This proceeds until you find a number that is prime (and satisfies any other prime properties you want to test for).

The problem with this approach is that true randomness is hard to find. Physical sources of entropy, including dice, coins, radioactive compounds, and interstellar radiation are good, but they are generally hard for a computer to come by. Most computers do their best to collect randomness sources and then use computational techniques to extract uniform randomness (using a randomness extractor function) and extend their randomness (using a pseudorandom generator function).

If the RSA key generation algorithm runs with insufficient randomness, then the keys it generates will be more predictable. When this is observed in the real world, it has caused real problems.

Early versions of the Netscape browser used the time of day, process id, and parent process id as the "randomness" used to seed the pseudorandom generator; this resulted in vulnerable SSL keys. The vulnerability was reported to Netscape by researchers at CERN in 1994 but was not fixed until Netscape 2.

In May 2008, the Debian project announced that it had patched a vulnerability in the OpenSSL package they were distributing. The bug turned out to have been introduced two years earlier when a well-intentioned developer had commented out a couple of lines of code that had been causing Valgrind warnings. The commented lines were in fact responsible for copying fresh randomness into the seed for the pseudorandom generator used to create keys. The practical upshot of which was that the only "randomness" going into the encryption keys was the processid. The resulting keys were insecure and predictable, leading to widespread key (and certificate) revocation.

In 2012, researchers from UCSD and the University of Michigan published the results of a large-scale student of public keys on the Internet. Using a simple greatest common divisor (GCD) calculation, they were about to recover RSA private keys for .5% of TLS hosts (they were also able to recover 1.03% of signing keys from TLS hosts). The vulnerability arose from machines that used low-quality randomness to generate their keys; whenever two machines shared one (but not both) of their chosen primes p and q, then the common factor could be easily recovered, resulting in the compromise of both private keys. In many cases, the hosts with vulnerable machines turned out to be IoT or embedded devices with no good source of high-quality randomness. In other cases, the vulnerable keys were due to faulty implementations like that distributed by Debian four years earlier.

4.3 Padding

The discussion thus far has considered a simple, deterministic specification of RSA. However, this version has several problems in practice. One problem is that if an attacker knows that the message is drawn from a small space (e.g., the message is either going to be "attack at dawn" or "attack at dusk"), then the attacker can just try encrypting each of the possible messages and check which one matches the observed ciphertext. This is called a *chosen plaintext attack*.

Another problem occurs if a principal sends the same message to multiple receivers

using the same public exponent but different moduli n (a situation made more likely by the adoption of standard values of e). When this occurs, an attacker can leverage the Chinese Remainder Theorem to recover the original message. Variants of this attack have also been developed for non-equal but related messages.

Both of these attacks can be overcome by introducing *padding*. Padding introduces structured, randomized values into the plaintext encoding of a message before the message is encrypted. When correctly done, padding can ensure that messages are not vulnerable to guessing attacks or related-message attacks. Naive padding schemes, however, can introduce vulnerabilities The current recommended padding scheme is called Optimal Asymmetric Encryption Padding (OAEP) uses a random value together with two hash functions to produce randomized plaintexts that are secure against all known attacks. An overview of OAEP is given in Figure 1.

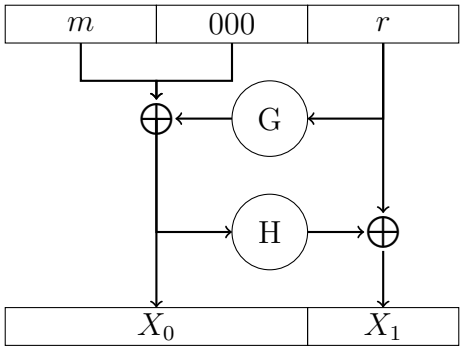

Figure 1: Optimal Asymmetric Encryption Padding (OAEP)

4.4 Blinding

Recall from our discussion earlier that real-world implementations use a variant of square-and-multiply to improve the performance of their decryption functions. The optimized algorithm loops through the bits of the secret exponent and branches on the value of each bit. This means that the behavior of the program—the time it takes to execute, the power drawn, the pattern of accesses to cache—depend strongly on the value of the secret exponent. In the mid-nineties, several lines of research demonstrated that these features (timing, power) could be measured accurately enough to learn the secret key, a class of attacks known as side-channel attacks.

One immediate reaction to the discovery of side-channel attacks was to propose that all differences between behavior depending on secret values should be eliminated. However, it turned out that not only did this negatively affect performance, but it was very difficult to achieve in practice. Attempts to eliminate distinctions only made the attacker's job slightly harder (by introducing more noise to filter out) or caused the attacker to identify a new, unprotected side-channel.

However, it was soon observed that successful side-channel attacks generally relied on the attacker's ability to observe decryptions of ciphertexts corresponding to known plaintexts controlled by the attacker. Exploiting this knowledge, standard crypto libraries started introducing *blinding*. Blinding is a general defense against side-channel attacks that works by encoding the plaintext in a randomized way prior to encryption. For RSA, this is done by modifying the encryption and decryption functions so that $c = (r, (mr)^e \bmod n)$ where r is a randomly chosen value and decryption is performed by computing $m = r^{-1}c^d \bmod n$. Blinding ensures that an attacker cannot control the value passed to the modular exponentiation function and thus serves as an effective defense against most side-channel attacks. Blinding is now standardly used for RSA implementations in standard crypto libraries, and blinding techniques have also been proposed for most other cryptographic protocols.