RSA Signing is Not RSA Decryption

Signing a document with pen and ink (the obvious equivalent of a digital signature) is quite different than opening padlocked box with a key (the perhaps less obvious equivalent of decryption). Nevertheless, both involve using a secret: how to write your own distinctive signature, and the shape of a distinctive key.

There are algorithms for digital signatures, and algorithms for encryption schemes. Sometimes it turns out that the key idea underlying an algorithm can be used for both purposes. For example, the key idea behind the El Gamal encryption algorithm can also be used to build a digital signature algorithm. The same is true for the well-known RSA algorithm.

Unfortunately, there's a tendency to oversimplify by asserting that digital signature algorithms are the **same** as the corresponding encryption scheme algorithms. They aren't. Nonetheless, you will sometimes find claims that (for example) RSA signing is the same as RSA decryption. That kind of claim is partially true, but also partially false.

The RSA Function

Let's look carefully at RSA to see what the relationship between signatures and encryption/decryption really is. But let's leave some of the mathematical details abstract, so that we don't have to get into any number theory.

In 1977, Rivest, Shamir, and Adelman discovered that the following function could be used for building cryptographic algorithms. I'll call it the *RSA function*:

```
R(x,k,n) = x^k (mod n)
```

Arguments x, k, and n are all integers, potentially very large integers. But n won't be important in the rest of our discussion, so from now on, we'll leave it out and simply write R(x,k).

Along with the RSA function, Rivest, Shamir, and Adelman discovered a way of choosing two keys, K and k, such that

```
R(R(x,K),k) = x, and
R(R(x,k),K) = x.
```

That is, applying R with K "undoes" applying R with k, and vice versa.

RSA Encryption

We could use R to attempt to build an encryption scheme using public encryption key K and private decryption key k:

```
Enc(m; K) = R(m,K)
Dec(c; k) = R(c,k)
```

To encrypt a plaintext m, just apply the RSA function with the public key; to decrypt, apply it with the private key.

That's where many textbook descriptions of RSA encryption stop. **But it's not the whole story.**

In practice, using the "textbook" version of RSA encryption is actually insecure. There are many attacks on it. Here's just one:

> Suppose Alice sends two messages, m1 and m2, encrypted with her public key K_A.
>
> ```
> c1 = Enc(m1; K) = R(m1,K) = m1^K (mod n)
> c2 = Enc(m2; K) = R(m2,K) = m2^K (mod n)
> ```
>
> What happens if m1 and m2 happen to be the same plaintext m?
>
> ```
> c1 = m^K mod n = c2
> ```
>
> Therefore a Dolev-Yao adversary observing the network will be able to learn something about the plaintexts m1 and m2--- namely, that they are the same---by observing ciphertexts c1 and c2! A secure encryption scheme shouldn't allow the adversary to learn that.

For more attacks, see [D. Boneh, A. Joux, and P. Nguyen. Why Textbook ElGamal and RSA Encryption are Insecure. In *Proc. AsiaCrypt*, 2000.] and [J. Katz and Y. Lindell. *Introduction to Modern Cryptography*, section 10.4. Chapman & Hall/CRC, 2008.].

To make "textbook" RSA encryption secure, we preprocess the plaintext m before applying the RSA function. We correspondingly do some postprocessing during decryption after applying the RSA function:

```
Enc(m; K) = R(pre(m),K)
Dec(c; k) = post(R(c,k))
```

There are several pre- and post-processing schemes. They are usually called *padding* schemes, though that's a slight misnomer: they do more than just padding. One of the best is OAEP: optimal asymmetric encryption padding, invented by Bellare and Rogaway in 1994. Amongst other things, OAEP pre-processing prevents the attack we observed above by XORing a cryptographic hash of an unpredictable nonce to the plaintext.

In more (though not quite full) detail, OAEP pre-processing works as follows:

```
OAEP-pre(m):
r = random nonce
X = (m || 00...0) XOR H(r) // pad m with zeros
Y = r XOR H(X)
output X || Y
```

Notation || denotes bit concatenation, and H is a cryptographic hash function. OAEP post-processing undoes the pre-processing:

```
OAEP-post(m'):
split m' into X || Y
r = Y XOR H(X)
(m || 00...0) = X XOR G(R)
output m
```

Putting this all together, we get the RSA-OAEP encryption scheme:

```
Enc(m; K) = R(OAEP-pre(m),K)
Dec(c; k) = OAEP-post(R(c,k))
```

RSA-OAEP is provably secure for some very strong, well-accepted definitions of security of encryption schemes. "Textbook" RSA, of course, is not secure in that sense.

RSA Digital Signatures

We could use R to attempt to build a digital signature scheme using public verification key K and private signing key k:

```
Sign(m; k) = R(m,k)
Ver(m; s; K) = R(s,K) == m
```

To sign a message m, just apply the RSA function with the private key to produce a signature s; to verify, apply the RSA function with the public key to the signature, and check that the result equals the expected message.

The main problem with the simple scheme just suggested is that messages might be too long---roughly speaking, the RSA function can't accomodate messages that are longer than the key. With encryption schemes, we solve that problem with block cipher modes. With digital signatures schemes, we instead solve that problem with cryptographic hashes:

```
Sign(m; k) = R(H(m),k)
Ver(m; s; K) = R(s,K) == H(m)
```

That's the textbook description of RSA signatures. And it's more-or-less the whole story. You can think of the hash function H as being the equivalent of both the pre- and post-processing used for RSA encryption.

(There is a more complex pre- and post-processing scheme for signatures called PSS (probabilistic signature scheme) that is provably secure. It's not as widely implemented, nor do I know of any attacks on the simpler hashing scheme above.)

RSA Encryption vs. RSA Digital Signatures

Repeated from above, here is our RSA encryption scheme and our RSA digital signature scheme in their "textbook" form:

```
Enc(m; K) = R(m,K)
Dec(c; k) = R(c,k)
Sign(m; k) = R(m,k)
Ver(m; s; K) = R(s,K) == m
```

And here are the same algorithms in their practical form, as used in real implementations:

```
Enc(m; K) = R(OAEP-pre(m),K)
Dec(c; k) = OAEP-post(R(c,k))
Sign(m; k) = R(H(m),k)
Ver(m; s; K) = R(s,K) == H(m)
```

Given what we know now, let's consider the claim that RSA signing is the same as RSA decryption: is the Sign function the same as the Dec function? In the real, practical world, clearly not. Sign involves a hash function H, whereas Dec involves a post-processing function OAEP-post. With Sign, H is applied directly to the message, then the RSA function is applied later. With Dec, the RSA function is applied first, and OAEP-post is applied later. Likewise, RSA signature verification is clearly different from RSA encryption.

But there is one way in which RSA signing is similar to RSA decryption: both involve a call to the RSA function with private key k as an argument. Likewise, RSA signature verification and RSA encryption both involve calling the RSA function with public key K as an argument. You can see that in the "textbook" formulations of the algorithms.

Conclusion

In the abstract world of textbooks, RSA signing and RSA decryption do turn out to be the same thing. In the real world of implementations, they are not. So don't ever use a real-world implementation of RSA decryption to compute RSA signatures. In the best case, your implementation will break in a way that you notice. In the worst case, you will introduce a vulnerability that an attacker could exploit.

Furthermore, don't make the mistake of generalizing from RSA to conclude that any encryption scheme can be adapted as a digital signature algorithm. That kind of adaptation works for RSA and El Gamal, but not in general.