Chapter 12

Digital signatures

In the public key setting, the primitive used to provide data integrity is a digital signature scheme. In this chapter we look at security notions and constructions for this primitive.

12.1 Digital signature schemes

A digital signature scheme is just like a message authentication scheme except for an asymmetry in the key structure. The key sk used to generate signatures (in this setting the tags are often called signatures) is different from the key pk used to verify signatures. Furthermore pk is public, in the sense that the adversary knows it too. So while only a signer in possession of the secret key can generate signatures, anyone in possession of the corresponding public key can verify the signatures.

Definition 12.1.1 A *digital signature scheme* $\mathcal{DS} = (\mathcal{K}, \mathrm{Sign}, \mathrm{VF})$ consists of three algorithms, as follows:

- The randomized *key generation* algorithm $\mathcal{K}$ (takes no inputs and) returns a pair (pk, sk) of keys, the public key and matching secret key, respectively. We write $(pk, sk) \overset{\$}{\leftarrow} \mathcal{K}$ for the operation of executing $\mathcal{K}$ and letting (pk, sk) be the pair of keys returned.
- The *signing* algorithm Sign takes the secret key sk and a message M to return a *signature* (also sometimes called a *tag*) $\sigma \in \{0,1\}^* \cup \{\perp\}$. The algorithm may be randomized or stateful. We write $\sigma \overset{\$}{\leftarrow} \mathrm{Sign}_{sk}(M)$ or $\sigma \overset{\$}{\leftarrow} \mathrm{Sign}(sk, M)$ for the operation of running Sign on inputs sk, M and letting σ be the signature returned.
- The deterministic *verification* algorithm VF takes a public key pk, a message M, and a candidate signature σ for M to return a bit. We write $d \leftarrow \mathrm{VF}_{pk}(M, \sigma)$ or $d \leftarrow \mathrm{VF}(pk, M, \sigma)$ to denote the operation of running VF on inputs pk, M, σ and letting d be the bit returned.

We require that $\mathrm{VF}_{pk}(M, \sigma) = 1$ for any key-pair (pk, sk) that might be output by $\mathcal{K}$, any message M, and any $\sigma \neq \perp$ that might be output by $\mathrm{Sign}_{sk}(M)$. If Sign is stateless then we associate to each public key a *message space* $\mathsf{Messages}(pk)$ which is the set of all M for which $\mathrm{Sign}_{sk}(M)$ never returns $\perp$. ▮

Let S be an entity that wants to have a digital signature capability. The first step is key generation: S runs $\mathcal{K}$ to generate a pair of keys (pk, sk) for itself. Note the key generation algorithm is run locally by S. Now, S can produce a digital signature on some document $M \in \mathsf{Messages}(pk)$ by running $\mathrm{Sign}_{sk}(M)$ to return a signature σ. The pair (M, σ) is then the authenticated version of the document. Upon receiving a document M' and tag σ' purporting to be from S, a receiver B in possession of pk verifies the authenticity of the signature by using the specified verification procedure, which depends on the message, signature, and public key. Namely he computes $\mathrm{VF}_{pk}(M', \sigma')$, whose value is a bit. If this value is 1, it is read as saying the data is authentic, and so B accepts it as coming from S. Else it discards the data as unauthentic.

Note that an entity wishing to verify S's signatures must be in possession of S's public key pk, and must be assured that the public key is authentic, meaning really is S's key and not someone else's key. We will look later into mechanisms for assuring this state of knowledge. But the key management processes are not part of the digital signature scheme itself. In constructing and analyzing the security of digital signature schemes, we make the assumption that any prospective verifier is in possession of an authentic copy of the public key of the signer. This assumption is made in what follows.

A viable scheme of course requires some security properties. But these are not our concern now. First we want to pin down what constitutes a specification of a scheme, so that we know what are the kinds of objects whose security we want to assess.

The key usage is the "mirror-image" of the key usage in an asymmetric encryption scheme. In a digital signature scheme, the holder of the secret key is a sender, using the secret key to tag its own messages so that the tags can be verified by others. In an asymmetric encryption scheme, the holder of the secret key is a receiver, using the secret key to decrypt ciphertexts sent to it by others.

The signature algorithm might be randomized, meaning internally flip coins and use these coins to determine its output. In this case, there may be many correct tags associated to a single message M. The algorithm might also be stateful, for example making use of a counter that is maintained by the sender. In that case the signature algorithm will access the counter as a global variable, updating it as necessary. The algorithm might even be both randomized and stateful. However, unlike encryption schemes, whose encryption algorithms must be either randomized or stateful for the scheme to be secure, a deterministic, stateless signature algorithm is not only possible, but common.

The signing algorithm might only be willing to sign certain messages and not others. It indicates its unwillingness to sign a message by returning $\perp$. If the scheme is stateless, the message space, which can depend on the public key, is the set of all messages for which the probability that the signing algorithm returns $\perp$ is zero. If the scheme is stateful we do not talk of such a space since whether or not the signing algorithm returns $\perp$ can depend not only on the message but on its state.

The last part of the definition says that signatures that were correctly generated will pass the verification test. This simply ensures that authentic data will be accepted by the receiver. In the case of a sateful scheme, the requirement holds for any state of the signing algorithm.

12.2 A notion of security

Digital signatures aim to provide the same security property as message authentication schemes; the only change is the more flexible key structure. Accordingly, we can build on our past work in understanding and pinning down a notion of security for message authentication; the one for digital signatures differs only in that the adversary has access to the public key.

The goal of the adversary F is forgery: It wants to produce document M and tag σ such that $\mathrm{VF}_{pk}(M, \sigma) = 1$, but M did not originate with the sender S. The adversary is allowed a chosen-message attack in the process of trying to produce forgeries, and the scheme is secure if even after such an attack the adversary has low probability of producing forgeries.

Let $\mathcal{DS} = (\mathcal{K}, \mathrm{Sign}, \mathrm{VF})$ be an arbitrary digital signature scheme. Our goal is to formalize a measure of insecurity against forgery under chosen-message attack for this scheme. The adversary's actions are viewed as divided into two phases. The first is a "learning" phase in which it is given oracle access to $\mathrm{Sign}_{sk}(\cdot)$, where (pk, sk) was a priori chosen at random according to $\mathcal{K}$. It can query this oracle up to q times, in any manner it pleases, as long as all the queries are messages in the underlying message space $\mathsf{Messages}(pk)$ associated to this key. Once this phase is over, it enters a "forgery" phases, in which it outputs a pair (M, σ). The adversary is declared successful if $M \in \mathsf{Messages}(pk)$, $\mathrm{VF}_{pk}(M, \sigma) = 1$ and M was not a query made by the adversary to the signing oracle. Associated to any adversary F is thus a success probability called its advantage. (The probability is over the choice of keys, any probabilistic choices that Sign might make, and the probabilistic choices, if any, that F makes.) The advantage of the scheme is the success probability of the "cleverest" possible adversary, amongst all adversaries restricted in their resources to some fixed amount. We choose as resources the running time of the adversary, the number of queries it makes, and the total bit-length of all queries combined plus the bit-length of the output message M in the forgery.

Definition 12.2.1 Let $\mathcal{DS} = (\mathcal{K}, \mathrm{Sign}, \mathrm{VF})$ be a digital signature scheme, and let A be an algorithm that has access to an oracle and returns a pair of strings. We consider the following experiment:

```
Experiment $\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(A)$
    $(pk, sk) \xleftarrow{\$} \mathcal{K}$
    $(M, \sigma) \leftarrow A^{\mathrm{Sign}_{sk}(\cdot)}(pk)$
    If the following are true return 1 else return 0:
    –  $\mathrm{VF}_{pk}(M, \sigma) = 1$
    –  $M \in \mathsf{Messages}(pk)$
    –  $M$ was not a query of $A$ to its oracle
```

The *uf-cma-advantage* of A is defined as

$$\mathbf{Adv}^{\text{uf-cma}}_{\mathcal{DS}}(A) \quad = \quad \Pr\left[\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(A) = 1\right] \; . \quad \blacksquare$$

In the case of message authentication schemes, we provided the adversary not only with an oracle for producing tags, but also with an oracle for verifying them. Above, there is no verification oracle. This is because verification of a digital signature does not depend on any quantity that is secret from the adversary. Since the adversary has the public key and knows the algorithm VF, it can verify as much as it pleases by running the latter.

When we talk of the time-complexity of an adversary, we mean the worst case total execution time of the entire experiment. This means the adversary complexity, defined as the worst case execution time of A plus the size of the code of the adversary A, in some fixed RAM model of computation (worst case means the maximum over A's coins or the answers returned in response to A's oracle queries), plus the time for other operations in the experiment, including the time for key generation and the computation of answers to oracle queries via execution of the encryption algorithm.

As adversary resources, we will consider this time complexity, the message length μ, and the number of queries q to the sign oracle. We define μ as the sum of the lengths of the oracle queries plus the length of the message in the forgery output by the adversary. In practice, the queries correspond to messages signed by the legitimate sender, and it would make sense that getting these examples is more expensive than just computing on one's own. That is, we would expect q to be smaller than t. That is why q, μ are resources separate from t.

12.3 RSA based signatures

The RSA trapdoor permutation is widely used as the basis for digital signature schemes. Let us see how.

12.3.1 Key generation for RSA systems

We will consider various methods for generating digital signatures using the RSA functions. While these methods differ in how the signature and verification algorithms operate, they share a common key-setup. Namely the public key of a user is a modulus N and an encryption exponent e, where $N = pq$ is the product of two distinct primes, and $e \in \mathbf{Z}^*_{\varphi(N)}$. The corresponding secret contains the decryption exponent $d \in \mathbf{Z}^*_{\varphi(N)}$ (and possibly other stuff too) where $ed \equiv 1 \pmod{\varphi(N)}$.

How are these parameters generated? We refer back to Definition 10.9 where we had introduced the notion of an RSA generator. This is a randomized algorithm having an associated security parameter and returning a pair $((N, e), (N, p, q, d))$ satisfying the various conditions listed in the definition. The key-generation algorithm of the digital signature scheme is simply such a generator, meaning the user's public key is (N, e) and its secret key is (N, p, q, d).

Note N is not really secret. Still, it turns out to be convenient to put it in the secret key. Also, the descriptions we provide of the signing process will usually depend only on N, d and not p, q, so it may not be clear why p, q are in the secret key. But in practice it is good to keep them there because their use speeds up signing via the Chinese Remainder theorem and algorithm.

Recall that the map $\mathsf{RSA}_{N,e}(\cdot) = (\cdot)^e \bmod N$ is a permutation on Z^*_N with inverse $\mathsf{RSA}_{N,d}(\cdot) = (\cdot)^d \bmod N$.

Below we will consider various signature schemes all of which use the above key generation algorithm and try to build in different ways on the one-wayness of RSA in order to securely sign.

12.3.2 Trapdoor signatures

Trapdoor signatures represent the most direct way in which to attempt to build on the one-wayness of RSA in order to sign. We believe that the signer, being in possession of the secret key N, d, is the only one who can compute the inverse RSA function $\mathsf{RSA}^{-1}_{N,e} = \mathsf{RSA}_{N,d}$. For anyone else, knowing only the public key N, e, this task is computationally infeasible. Accordingly, the signer signs a message by performing on it this "hard" operation. This requires that the message be a member of Z^*_N, which, for convenience, is assumed. It is possible to verify a signature by performing the "easy" operation of computing $\mathsf{RSA}_{N,e}$ on the claimed signature and seeing if we get back the message.

More precisely, let $\mathcal{K}_{\rm rsa}$ be an RSA generator with associated security parameter k, as per Definition 10.9. We consider the digital signature scheme $\mathcal{DS} = (\mathcal{K}_{\rm rsa}, \mathrm{Sign}, \mathrm{VF})$ whose signing and verifying algorithms are as follows:

```
Algorithm $\mathrm{Sign}_{N,p,q,d}(M)$
    If $M \notin \mathbf{Z}^*_N$ then return $\perp$
    $x \leftarrow M^d \bmod N$
    Return $x$
```

```
Algorithm $\mathrm{VF}_{N,e}(M, x)$
    If $(M \notin \mathbf{Z}^*_N$ or $x \notin \mathbf{Z}^*_N)$ then return 0
    If $M = x^e \bmod N$ then return 1 else return 0
```

This is a deterministic stateless scheme, and the message space for public key (N, e) is $\mathsf{Messages}(N, e) = Z^*_N$, meaning the only messages that the signer signs are those which are elements of the group Z^*_N.

In this scheme we have denoted the signature of M by x. The signing algorithm simply applies $\mathsf{RSA}_{N,d}$ to the message to get the signature, and the verifying algorithm applies $\mathsf{RSA}_{N,e}$ to the signature and tests whether the result equals the message.

The first thing to check is that signatures generated by the signing algorithm pass the verification test. This is true because of Proposition 10.7 which tells us that if $x = M^d \bmod N$ then $x^e = M \bmod N$.

Now, how secure is this scheme? As we said above, the intuition behind it is that the signing operation should be something only the signer can perform, since computing $\mathsf{RSA}_{N,e}^{-1}(M)$ is hard without knowledge of d. However, what one should remember is that the formal assumed hardness property of RSA, namely one-wayness under known-exponent attack (we call it just one-wayness henceforth) as specified in Definition 10.10, is under a very different model and setting than that of security for signatures. One-wayness tells us that if we select M at random and then feed it to an adversary (who knows N, e but not d) and ask the latter to find $x = \mathsf{RSA}_{N,e}^{-1}(M)$, then the adversary will have a hard time succeeding. But the adversary in a signature scheme is not given a random message M on which to forge a signature. Rather, its goal is to create a pair (M, x) such that $\mathrm{VF}_{N,e}(M, x) = 1$. It does not have to try to imitate the signing algorithm; it must only do something that satisfies the verification algorithm. In particular it is allowed to choose M rather than having to sign a given or random M. It is also allowed to obtain a valid signature on any message other than the M it eventually outputs, via the signing oracle, corresponding in this case to having an oracle for $\mathsf{RSA}_{N,e}^{-1}(\cdot)$. These features make it easy for an adversary to forge signatures.

A couple of simple forging strategies are illustrated below. The first is to simply output the forgery in which the message and signature are both set to 1. The second is to first pick at random a value that will play the role of the signature, and then compute the message based on it:

```
Forger $F_1^{\mathrm{Sign}_{N,p,q,d}(\cdot)}(N,e)$
    Return $(1,1)$
```

```
Forger $F_2^{\mathrm{Sign}_{N,p,q,d}(\cdot)}(N,e)$
    $x \xleftarrow{\$} Z_N^*$ ; $M \leftarrow x^e \bmod N$
    Return $(M,x)$
```

These forgers makes no queries to their signing oracles. We note that $1^e \equiv 1 \pmod{N}$, and hence the uf-cma-advantage of F_1 is 1. Similarly, the value (M, x) returned by the second forger satisfies $x^e \bmod N = M$ and hence it has uf-cma-advantage 1 too. The time-complexity in both cases is very low. (In the second case, the forger uses the $O(k^3)$ time to do its exponentiation modulo N.) So these attacks indicate the scheme is totally insecure.

The message M whose signature the above forger managed to forge is random. This is enough to break the scheme as per our definition of security, because we made a very strong definition of security. Actually for this scheme it is possible to even forge the signature of a given message M, but this time one has to use the signing oracle. The attack relies on the multiplicativity of the RSA function.

```
Forger $F^{\mathrm{Sign}_{N,e}(\cdot)}(N,e)$
    $M_1 \xleftarrow{\$} Z_N^* - \{1, M\}$ ; $M_2 \leftarrow M M_1^{-1} \bmod N$
    $x_1 \leftarrow \mathrm{Sign}_{N,e}(M_1)$ ; $x_2 \leftarrow \mathrm{Sign}_{N,e}(M_2)$
    $x \leftarrow x_1 x_2 \bmod N$
    Return $(M,x)$
```

Given M the forger wants to compute a valid signature x for M. It creates M_1, M_2 as shown, and obtains their signatures x_1, x_2. It then sets $x = x_1 x_2 \bmod N$. Now the verification algorithm will check whether $x^e \bmod N = M$. But note that

$$x^e \equiv (x_1 x_2)^e \equiv x_1^e x_2^e \equiv M_1 M_2 \equiv M \pmod{N} .$$

Here we used the multiplicativity of the RSA function and the fact that x_i is a valid signature of M_i for $i = 1, 2$. This means that x is a valid signature of M. Since M_1 is chosen to not be 1 or M, the same is true of M_2, and thus M was not an oracle query of F. So F succeeds with probability one.

These attacks indicate that there is more to signatures than one-wayness of the underlying function.

12.3.3 The hash-then-invert paradigm

Real-world RSA based signature schemes need to surmount the above attacks, and also attend to other impracticalities of the trapdoor setting. In particular, messages are not usually group elements; they are possibly long files, meaning bit strings of arbitrary lengths. Both issues are typically dealt with by pre-processing the given message M via a hash function to yield a point y in the range of $\mathsf{RSA}_{N,e}$, and then applying $\mathsf{RSA}_{N,e}^{-1}$ to y to obtain the signature. The hash function is public, meaning its description is known, and anyone can compute it.

To make this more precise, let $\mathcal{K}_{\rm rsa}$ be an RSA generator with associated security parameter k and let Keys be the set of all modulli N that have positive probability to be output by $\mathcal{K}_{\rm rsa}$. Let Hash be a family of functions whose key-space is Keys and such that $\mathsf{Hash}_N\colon \{0,1\}^* \to \mathbf{Z}_N^*$ for every $N \in \mathsf{Keys}$. Let $\mathcal{DS} = (\mathcal{K}_{\rm rsa}, \text{Sign}, \text{VF})$ be the digital signature scheme whose signing and verifying algorithms are as follows:

Algorithm $\text{Sign}_{N,p,q,d}(M)$	Algorithm $\text{VF}_{N,e}(M,x)$
$y \leftarrow \mathsf{Hash}_N(M)$	$y \leftarrow \mathsf{Hash}_N(M)$
$x \leftarrow y^d \bmod N$	$y' \leftarrow x^e \bmod N$
Return x	If $y = y'$ then return 1 else return 0

Let us see why this might help resolve the weaknesses of trapdoor signatures, and what requirements security imposes on the hash function.

Let us return to the attacks presented on the trapdoor signature scheme above. Begin with the first forger we presented, who simply output $(1,1)$. Is this an attack on our new scheme? To tell, we see what happens when the above verification algorithm is invoked on input $1,1$. We see that it returns 1 only if $\mathsf{Hash}_N(1) \equiv 1^e \pmod{N}$. Thus, to prevent this attack it suffices to ensure that $\mathsf{Hash}_N(1) \neq 1$. The second forger we had previously set M to $x^e \bmod N$ for some random $x \in Z_N^*$. What is the success probability of this strategy under the hash-then-invert scheme? The forger wins if $x^e \bmod N = \mathsf{Hash}(M)$ (rather than merely $x^e \bmod N = M$ as before). The hope is that with a "good" hash function, it is very unlikely that $x^e \bmod N = \mathsf{Hash}_N(M)$. Consider now the third attack we presented above, which relied on the multiplicativity of the RSA function. For this attack to work under the hash-then-invert scheme, it would have to be true that

$$\mathsf{Hash}_N(M_1) \cdot \mathsf{Hash}_N(M_2) \equiv \mathsf{Hash}_N(M) \pmod{N} . \tag{12.1}$$

Again, with a "good" hash function, we would hope that this is unlikely to be true.

The hash function is thus supposed to "destroy" the algebraic structure that makes attacks like the above possible. How we might find one that does this is something we have not addressed.

While the hash function might prevent some attacks that worked on the trapdoor scheme, its use leads to a new line of attack, based on collisions in the hash function. If an adversary can find two distinct messages M_1, M_2 that hash to the same value, meaning $\mathsf{Hash}_N(M_1) = \mathsf{Hash}_N(M_2)$, then it can easily forge signatures, as follows:

```
Forger $F^{\mathrm{Sign}_{N,p,q,d}(\cdot)}(N,e)$
    $x_1 \leftarrow \mathrm{Sign}_{N,p,q,d}(M_1)$
    Return $(M_2, x_1)$
```

This works because M_1, M_2 have the same signature. Namely because x_1 is a valid signature of M_1, and because M_1, M_2 have the same hash value, we have

$$x_1^e \equiv \mathsf{Hash}_N(M_1) \equiv \mathsf{Hash}_N(M_2) \pmod{N} ,$$

and this means the verification procedure will accept x_1 as a signature of M_2. Thus, a necessary requirement on the hash function Hash is that it be CR2-KK, meaning given N it should be computationally infeasible to find distinct values M, M' such that $\mathsf{Hash}_N(M) = \mathsf{Hash}_N(M')$.

Below we will go on to more concrete instantiations of the hash-then-invert paradigm. But before we do that, it is important to try to assess what we have done so far. Above, we have pin-pointed some features of the hash function that are necessary for the security of the signature scheme. Collision-resistance is one. The other requirement is not so well formulated, but roughly we want to destroy algebraic structure in such a way that Equation (12.1), for example, should fail with high probability. Classical design focuses on these attacks and associated features of the hash function, and aims to implement suitable hash functions. But if you have been understanding the approaches and viewpoints we have been endeavoring to develop in this class and notes, you should have a more critical perspective. The key point to note is that what we need is not really to pin-point necessary features of the hash function to prevent certain attacks, but rather to pin-point *sufficient* features of the hash function, namely features sufficient to prevent *all* attacks, even ones that have not yet been conceived. And we have not done this. Of course, pinning down necessary features of the hash function is useful to gather intuition about what sufficient features might be, but it is only that, and we must be careful to not be seduced into thinking that it is enough, that we have identified all the concerns. Practice proves this complacence wrong again and again.

How can we hope to do better? Return to the basic philosophy of provable security. We want assurance that the signature scheme is secure under the assumption that its underlying primitives are secure. Thus we must try to tie the security of the signature scheme to the security of RSA as a one-way function, and some security condition on the hash function. With this in mind, let us proceed to examine some suggested solutions.

12.3.4 The PKCS #1 scheme

RSA corporation has been one of the main sources of software and standards for RSA based cryptography. RSA Labs (now a part of Security Dynamics Corporation) has created a set of standards called PKCS (Public Key Cryptography Standards). PKCS #1 is about signature (and encryption) schemes based on the RSA function. This standard is in wide use, and accordingly it will be illustrative to see what they do.

The standard uses the hash-then-invert paradigm, instantiating Hash via a particular hash function $\mathsf{PKCS\text{-}Hash}$ which we now describe. Recall we have already discussed collision-resistant hash functions. Let us fix a function $h: \{0,1\}^* \rightarrow \{0,1\}^l$ where $l \geq 128$ and which is "collision-resistant" in the sense that nobody knows how to find any pair of distinct points M, M' such that $h(M) = h(M')$. Currently the role tends to be played by SHA-1, so that $l = 160$. Prior to that it was MD5, which has $l = 128$. The RSA PKCS #1 standard defines

$$\mathsf{PKCS\text{-}Hash}_N(M) \quad = \quad \mathsf{00\,01\,FF\,FF} \cdots \mathsf{FF\,FF\,00} \,\|\, h(M) \ .$$

Here $\|$ denotes concatenation, and enough FF-bytes are inserted that the length of $\mathsf{PKCS\text{-}Hash}_N(M)$ is equal to k bits. Note the the first four bits of the hash output are zero, meaning as an integer it is certainly at most N, and thus most likely in $\mathbf{Z}_N^*$, since most numbers between 1 and N are in $\mathbf{Z}_N^*$. Also note that finding collisions in PKCS-Hash is no easier than finding collisions in h, so if the latter is collision-resistant then so is the former.

Recall that the signature scheme is exactly that of the hash-then-invert paradigm. For concreteness, let us rewrite the signing and verifying algorithms:

```
Algorithm Sign$_{N,p,q,d}(M)$
    $y \leftarrow \textsf{PKCS-Hash}_N(M)$
    $x \leftarrow y^d \bmod N$
    Return $x$
```

```
Algorithm VF$_{N,e}(M,x)$
    $y \leftarrow \textsf{PKCS-Hash}_N(M)$
    $y' \leftarrow x^e \bmod N$
    If $y = y'$ then return 1 else return 0
```

Now what about the security of this signature scheme? Our first concern is the kinds of algebraic attacks we saw on trapdoor signatures. As discussed in Section 12.3.3, we would like that relations like Equation (12.1) fail. This we appear to get; it is hard to imagine how $\textsf{PKCS-Hash}_N(M_1) \cdot \textsf{PKCS-Hash}_N(M_2) \bmod N$ could have the specific structure required to make it look like the PKCS-hash of some message. This isn't a proof that the attack is impossible, of course, but at least it is not evident.

This is the point where our approach departs from the classical attack-based design one. Under the latter, the above scheme is acceptable because known attacks fail. But looking deeper there is cause for concern. The approach we want to take is to see how the desired security of the signature scheme relates to the assumed or understood security of the underlying primitive, in this case the RSA function.

We are assuming RSA is one-way, meaning it is computationally infeasible to compute $\textsf{RSA}_{N,e}^{-1}(y)$ for a randomly chosen point $y \in Z_N^*$. On the other hand, the points to which $\textsf{RSA}_{N,e}^{-1}$ is applied in the signature scheme are those in the set $S_N = \{ \textsf{PKCS-Hash}_N(M) \,:\, M \in \{0,1\}^* \}$. The size of S_N is at most 2^l since h outputs l bits and the other bits of $\textsf{PKCS-Hash}_N(\cdot)$ are fixed. With SHA-1 this means $|S_N| \leq 2^{160}$. This may seem like quite a big set, but within the RSA domain Z_N^* it is tiny. For example when $k = 1024$, which is a recommended value of the security parameter these days, we have

$$\frac{|S_N|}{|Z_N^*|} \leq \frac{2^{160}}{2^{1023}} = \frac{1}{2^{863}} .$$

This is the probability with which a point chosen randomly from Z_N^* lands in S_N. For all practical purposes, it is zero. So RSA could very well be one-way and still be easy to invert on S_N, since the chance of a random point landing in S_N is so tiny. So the security of the PKCS scheme cannot be guaranteed solely under the standard one-wayness assumption on RSA. Note this is true no matter how "good" is the underlying hash function h (in this case SHA-1) which forms the basis for PKCS-Hash. The problem is the design of PKCS-Hash itself, in particular the padding.

The security of the PKCS signature scheme would require the assumption that RSA is hard to invert on the set S_N, a miniscule fraction of its full range. (And even this would be only a necessary, but not sufficient condition for the security of the signature scheme.)

Let us try to clarify and emphasize the view taken here. We are not saying that we know how to attack the PKCS scheme. But we are saying that an absence of known attacks should not be deemed a good reason to be satisfied with the scheme. We can identify "design flaws," such as the way the scheme uses RSA, which is not in accordance with our understanding of the security of RSA as a one-way function. And this is cause for concern.

12.3.5 The FDH scheme

From the above we see that if the hash-then-invert paradigm is to yield a signature scheme whose security can be based on the one-wayness of the RSA function, it must be that the points y on which $\mathsf{RSA}_{N,e}^{-1}$ is applied in the scheme are random ones. In other words, the output of the hash function must always "look random". Yet, even this only highlights a necessary condition, not (as far as we know) a sufficient one.

We now ask ourselves the following question. Suppose we had a "perfect" hash function Hash. In that case, at least, is the hash-then-invert signature scheme secure? To address this we must first decide what is a "perfect" hash function. The answer is quite natural: one that is random, namely returns a random answer to any query except for being consistent with respect to past queries. (We will explain more how this "random oracle" works later, but for the moment let us continue.) So our question becomes: in a model where Hash is perfect, can we *prove* that the signature scheme is secure if RSA is one-way?

This is a basic question indeed. If the hash-then-invert paradigm is in any way viable, we really must be able to prove security in the case the hash function is perfect. Were it not possible to prove security in this model it would be extremely inadvisable to adopt the hash-then-invert paradigm; if it doesn't work for a perfect hash function, how can we expect it to work in any real world setting?

Accordingly, we now focus on this "thought experiment" involving the use of the signature scheme with a perfect hash function. It is a thought experiment because no specific hash function is perfect. Our "hash function" is no longer fixed, it is just a box that flips coins. Yet, this thought experiment has something important to say about the security of our signing paradigm. It is not only a key step in our understanding but will lead us to better concrete schemes as we will see later.

Now let us say more about perfect hash functions. We assume that Hash returns a random member of Z_N^* every time it is invoked, except that if twice invoked on the same message, it returns the same thing both times. In other words, it is an instance of a random function with domain $\{0,1\}^*$ and range Z_N^*. We have seen such objects before, when we studied pseudorandomness: remember that we defined pseudorandom functions by considering experiments involving random functions. So the concept is not new. We call Hash a random oracle, and denote it by H in this context. It is accessible to all parties, signer, verifiers and adversary, but as an oracle. This means it is only accessible across a specified interface. To compute $H(M)$ a party must make an oracle call. This means it outputs M together with some indication that it wants $H(M)$ back, and an appropriate value is returned. Specifically it can output a pair (hash, M), the first component being merely a formal symbol used to indicate that this is a hash-oracle query. Having output this, the calling algorithm waits for the answer. Once the value $H(M)$ is returned, it continues its execution.

The best way to think about H is as a dynamic process which maintains a table of input-output pairs. Every time a query (hash, M) is made, the process first checks if its table contains a pair of the form (M, y) for some y, and if so, returns y. Else it picks a random y in Z_N^*, puts (M, y) into the table, and returns y as the answer to the oracle query.

We consider the above hash-then-invert signature scheme in the model where the hash function Hash is a random oracle H. This is called the Full Domain Hash (FDH) scheme. More precisely, let $\mathcal{K}_{\text{rsa}}$ be an RSA generator with associated security parameter k. The *FDH-RSA signature scheme associated to* $\mathcal{K}_{\text{rsa}}$ is the digital signature scheme $\mathcal{DS} = (\mathcal{K}_{\text{rsa}}, \text{Sign}, \text{VF})$ whose signing and verifying algorithms are as follows:

```
Algorithm $\text{Sign}^{H(\cdot)}_{N,p,q,d}(M)$
    $y \leftarrow H(M)$
    $x \leftarrow y^d \bmod N$
    Return $x$
```

```
Algorithm $\text{VF}^{H(\cdot)}_{N,e}(M, x)$
    $y \leftarrow H(M)$
    $y' \leftarrow x^e \bmod N$
    If $y = y'$ then return 1 else return 0
```

The only change with respect to the way we wrote the algorithms for the generic hash-then-invert scheme of Section 12.3.3 is notational: we write H as a superscript to indicate that it is an oracle accessible only via the specified oracle interface. The instruction $y \leftarrow H(M)$ is implemented by making the query (hash, M) and letting y denote the answer returned, as discussed above.

We now ask ourselves whether the above signature scheme is secure under the assumption that RSA is one-way. To consider this question we first need to extend our definitions to encompass the new model. The key difference is that the success probability of an adversary is taken over the random choice of H in addition to the random choices previously considered. The forger F as before has access to a signing oracle, but now also has access to H. Furthermore, Sign and VF now have access to H. Let us first write the experiment that measures the success of forger F and then discuss it more.

```
Experiment $\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(F)$
    $((N, e), (N, p, q, d)) \xleftarrow{\$} \mathcal{K}_{\text{rsa}}$
    $H \xleftarrow{\$} \mathsf{Func}(\{0,1\}^*, \mathbf{Z}_N^*)$
    $(M, x) \xleftarrow{\$} F^{H(\cdot), \text{Sign}^{H(\cdot)}_{N,p,q,d}(\cdot)}(N, e)$
    If the following are true return 1 else return 0:
    – $\text{VF}^{H}_{pk}(M, \sigma) = 1$
    – $M$ was not a query of $A$ to its oracle
```

Note that the forger is given oracle access to H in addition to the usual access to the sign oracle that models a chosen-message attack. After querying its oracles some number of times the forger outputs a message M and candidate signature x for it. We say that F is successful if the verification process would accept M, x, but F never asked the signing oracle to sign M. (F is certainly allowed to make hash query M, and indeed it is hard to imagine how it might hope to succeed in forgery otherwise, but it is not allowed to make sign query M.) The *uf-cma-advantage* of A is defined as

$$\mathbf{Adv}^{\text{uf-cma}}_{\mathcal{DS}}(A) \quad = \quad \Pr\left[\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(A) = 1\right] \ .$$

We will want to consider adversaries with time-complexity at most t, making at most q_{sig} sign oracle queries and at most q_{hash} hash oracle queries, and with total query message length μ. Resources refer again to those of the entire experiment. We first define the *execution time* as the time taken by the entire experiment $\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(F)$. This means it includes the time to compute answers to oracle queries, to generate the keys, and even to verify the forgery. Then the time-complexity t is supposed to upper bound the execution time plus the size of the code of F. In counting hash queries we again look at the entire experiment and ask that the total number of queries to H here be at most q_{hash}. Included in the count are the direct hash queries of F, the indirect hash queries made by the signing oracle, and even the hash query made by the verification algorithm in the last step. This latter means that q_{hash} is always at least the number of hash queries required for a verification, which for FDH-RSA is one. In fact for FDH-RSA we will have $q_{\text{hash}} \geq q_{\text{sig}} + 1$, something to be kept in mind when interpreting later results. Finally μ is the sum of the lengths of all messages in sign queries plus the length of the final output message M.

However, there is one point that needs to be clarified here, namely that if time-complexity refers to that of the entire experiment, how do we measure the time to pick H at random? It is an infinite object and thus cannot be actually chosen in finite time. The answer is that although we write H as being chosen at random upfront in the experiment, this is not how it is implemented. Instead, imagine H as being chosen dynamically. Think of the process implementing the table we described, so that random choices are made only at the time the H oracle is called, and the cost is that of maintaining and updating a table that holds the values of H on inputs queried so far. Namely when a query M is made to H, we charge the cost of looking up the table, checking whether $H(M)$ was already defined and returning it if so, else picking a random point from $\mathbf{Z}_N^*$, putting it in the table with index M, and returning it as well.

In this setting we claim that the FDH-RSA scheme is secure. The following theorem upper bounds its uf-cma-advantage solely in terms of the ow-kea advantage of the underlying RSA generator.

Theorem 12.3.1 Let $\mathcal{K}_{\rm rsa}$ be an RSA generator with associated security parameter k, and let $\mathcal{DS}$ be the FDH-RSA scheme associated to $\mathcal{K}_{\rm rsa}$. Let F be an adversary making at most $q_{\rm hash}$ queries to its hash oracle and at most $q_{\rm sig}$ queries to its signing oracle where $q_{\rm hash} \geq 1 + q_{\rm sig}$. Then there exists an adversary I such that

$$\mathbf{Adv}^{\rm uf\text{-}cma}_{\mathcal{DS}}(F) \;\leq\; q_{\rm hash} \cdot \mathbf{Adv}^{\rm ow\text{-}kea}_{\mathcal{K}_{\rm rsa}}(I)\;. \tag{12.2}$$

and I, F are of comparable resources. ▮

The theorem says that the only way to forge signatures in the FDH-RSA scheme is to try to invert the RSA function on random points. There is some loss in security: it might be that the chance of breaking the signature scheme is larger than that of inverting RSA in comparable time, by a factor of the number of hash queries made in the forging experiment. But we can make $\mathbf{Adv}^{\rm ow\text{-}kea}_{\mathcal{K}_{\rm rsa}}(t')$ small enough that even $q_{\rm hash} \cdot \mathbf{Adv}^{\rm ow\text{-}kea}_{\mathcal{K}_{\rm rsa}}(t')$ is small, by choosing a larger modulus size k.

One must remember the caveat: this is in a model where the hash function is random. Yet, even this tells us something, namely that the hash-then-invert paradigm itself is sound, at least for "perfect" hash functions. This puts us in a better position to explore concrete instantiations of the paradigm.

Let us now proceed to the proof of Theorem 12.3.1. Remember that inverter I takes as input (N, e), describing $\mathsf{RSA}_{N,e}$, and also a point $y \in Z_N^*$. Its job is to try to output $\mathsf{RSA}_{N,e}^{-1}(y) = y^d \bmod N$, where d is the decryption exponent corresponding to encryption exponent e. Of course, neither d nor the factorization of N are available to I. The success of I is measured under a random choice of $((N, e), (N, p, q, d))$ as given by $\mathcal{K}_{\rm rsa}$, and also a random choice of y from Z_N^*. In order to accomplish its task, I will run F as a subroutine, on input public key (N, e), hoping somehow to use F's ability to forge signatures to find $\mathsf{RSA}_{N,e}^{-1}(y)$. Before we discuss how I might hope to use the forger to determine the inverse of point y, we need to take a closer look at what it means to run F as a subroutine.

Recall that F has access to two oracles, and makes calls to them. At any point in its execution it might output (hash, M). It will then wait for a return value, which it interprets as $H(M)$. Once this is received, it continues its execution. Similarly it might output (sign, M) and then wait to receive a value it interprets as $\mathrm{Sign}^{H(\cdot)}_{N,p,q,d}(M)$. Having got this value, it continues. The important thing to understand is that F, as an algorithm, merely communicates with oracles via an interface. It does not control what these oracles return. You might think of an oracle query like a system call. Think of F as writing an oracle query M at some specific prescribed place in memory. Some process is expected to put in another prescribed place a value that F will take as the answer. F reads what is there, and goes on.

When I executes F, no oracles are actually present. F does not know that. It will at some point make an oracle query, assuming the oracles are present, say query (hash, M). It then waits for an answer. If I wants to run F to completion, it is up to I to provide some answer to F as the answer to this oracle query. F will take whatever it is given and go on executing. If I cannot provide an answer, F will not continue running; it will just sit there, waiting. We have seen this idea of "simulation" before in several proofs: I is creating a "virtual reality" under which F can believe itself to be in its usual environment.

The strategy of I will be to take advantage of its control over responses to oracle queries. It will choose them in strange ways, not quite the way they were chosen in Experiment $\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(F)$. Since F is just an algorithm, it processes whatever it receives, and eventually will halt with some output, a claimed forgery (M, x). By clever choices of replies to oracle queries, I will ensure that F is fooled into not knowing that it is not really in $\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(F)$, and furthermore x will be the desired inverse of y. Not always, though; I has to be lucky. But it will be lucky often enough.

We begin by consider the case of a very simple forger F. It makes no sign queries and exactly one hash query (hash, M). It then outputs a pair (M, x) as the claimed forgery, the message M being the same in the hash query and the forgery. (In this case we have $q_{\text{sig}} = 0$ and $q_{\text{hash}} = 2$, the last due to the hash query of F and the final verification query in the experiment.) Now if F is successful then x is a valid signature of M, meaning $x^e \equiv H(M) \bmod N$, or, equivalently, $x \equiv H(M)^d \bmod N$. Somehow, F has found the inverse of $H(M)$, the value returned to it as the response to oracle query M. Now remember that I's goal had been to compute $y^d \bmod N$ where y was its given input. A natural thought suggests itself: If F can invert $\mathsf{RSA}_{N,e}$ at $H(M)$, then I will "set" $H(M)$ to y, and thereby obtain the inverse of y under $\mathsf{RSA}_{N,e}$. I can set $H(M)$ in this way because it controls the answers to oracle queries. When F makes query (hash, M), the inverter I will simply return y as the response. If F then outputs a valid forgery (M, x), we have $x = y^d \bmod N$, and I can output x, its job done.

But why would F return a valid forgery when it got y as its response to hash query M? Maybe it will refuse this, saying it will not work on points supplied by an inverter I. But this will not happen. F is simply an algorithm and works on whatever it is given. What is important is solely the distribution of the response. In Experiment $\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(F)$ the response to (hash, M) is a random element of Z_N^*. But y has exactly the same distribution, because that is how it is chosen in the experiment defining the success of I in breaking RSA as a one-way function. So F cannot behave any differently in this virtual reality than it could in its real world; its probability of returning a valid forgery is still $\mathbf{Adv}^{\text{uf-cma}}_{\mathcal{DS}}(F)$. Thus for this simple F the success probability of the inverter in finding $y^d \bmod N$ is exactly the same as the success probability of F in forging signatures. Equation (12.2) claims less, so we certainly satisfy it.

However, most forgers will not be so obliging as to make no sign queries, and just one hash query consisting of the very message in their forgery. I must be able to handle any forger.

Inverter I will define a pair of subroutines, H-*Sim* (called the hash oracle simulator) and *Sign-Sim* (called the sign oracle simulator) to play the role of the hash and sign oracles respectively. Namely, whenever F makes a query (hash, M) the inverter I will return H-*Sim*(M) to F as the answer, and whenever F makes a query (sign, M) the inverter I will return *Sign-Sim*(M) to F as the answer. (The *Sign-Sim* routine will additionally invoke H-*Sim*.) As it executes, I will build up various tables (arrays) that "define" H. For $j = 1, \ldots, q_{\text{hash}}$, the j-th string on which H is called in the experiment (either directly due to a hash query by F, indirectly due to a sign query by F, or due to the final verification query) will be recorded as $Msg[j]$; the response returned by the hash oracle simulator to $Msg[j]$ is stored as $Y[j]$; and if $Msg[j]$ is a sign query then the response returned to F as the "signature" is $X[j]$. Now the question is how I defines all these values.

Suppose the j-th hash query in the experiment arises indirectly, as a result of a sign query $(\mathsf{sign}, \mathit{Msg}[j])$ by F. In Experiment $\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(F)$ the forger will be returned $H(\mathit{Msg}[j])^d \bmod N$. If I wants to keep F running it must return something plausible. What could I do? It could attempt to directly mimic the signing process, setting $Y[j]$ to a random value (remember $Y[j]$ plays the role of $H(\mathit{Msg}[j])$) and returning $(Y[j])^d \bmod N$. But it won't be able to compute the latter since it is not in possesion of the secret signing exponent d. The trick, instead, is that I first picks a value $X[j]$ at random in Z_N^* and sets $Y[j] = (X[j])^e \bmod N$. Now it can return $X[j]$ as the answer to the sign query, and this answer is accurate in the sense that the verification relation (which F might check) holds: we have $Y[j] \equiv (X[j])^e \bmod N$.

This leaves a couple of loose ends. One is that we assumed above that I has the liberty of defining $Y[j]$ at the point the sign query was made. But perhaps $\mathit{Msg}[j] = \mathit{Msg}[l]$ for some $l < j$ due to there having been a hash query involving this same message in the past. Then the hash value $Y[j]$ is already defined, as $Y[l]$, and cannot be changed. This can be addressed quite simply however: for any hash query $\mathit{Msg}[l]$, the hash simulator can follow the above strategy of setting the reply $Y[l] = (X[l])^e \bmod N$ at the time the hash query is made, meaning it prepares itself ahead of time for the possibility that $\mathit{Msg}[l]$ is later a sign query. Maybe it will not be, but nothing is lost.

Well, almost. Something is lost, actually. A reader who has managed to stay awake so far may notice that we have solved two problems: how to use F to find $y^d \bmod N$ where y is the input to I, and how to simulate answers to sign and hash queries of F, but that these processes are in conflict. The way we got $y^d \bmod N$ was by returning y as the answer to query (hash, M) where M is the message in the forgery. However, we do not know beforehand which message in a hash query will be the one in the forgery. So it is difficult to know how to answer a hash query $\mathit{Msg}[j]$; do we return y, or do we return $(X[j])^e \bmod N$ for some $X[j]$? If we do the first, we will not be able to answer a sign query with message $\mathit{Msg}[j]$; if we do the second, and if $\mathit{Msg}[j]$ equals the message in the forgery, we will not find the inverse of y. The answer is to take a guess as to which to do. There is some chance that this guess is right, and I succeeds in that case.

Specifically, notice that $\mathit{Msg}[q_{\text{hash}}] = M$ is the message in the forgery by definition since $\mathit{Msg}[q_{\text{hash}}]$ is the message in the final verification query. The message M might occur more than once in the list, but it occurs at least once. Now I will choose a random i in the range $1 \le i \le q_{\text{hash}}$ and respond by y to hash query $(\mathsf{hash}, \mathit{Msg}[i])$. To all other queries j it will respond by first picking $X[j]$ at random in Z_N^* and setting $H(\mathit{Msg}[j]) = (X[j])^e \bmod N$. The forged message M will equal $\mathit{Msg}[i]$ with probability at least $1/q_{\text{hash}}$ and this will imply Equation (12.2). Below we summarize these ideas as a proof of Theorem 12.3.1.

It is tempting from the above description to suggest that we always choose $i = q_{\text{hash}}$, since $\mathit{Msg}[q_{\text{hash}}] = M$ by definition. Why won't that work? Because M might also have been equal to $\mathit{Msg}[j]$ for some $j < q_{\text{hash}}$, and if we had set $i = q_{\text{hash}}$ then at the time we want to return y as the answer to M we find we have already defined $H(M)$ as something else and it is too late to change our minds.

Proof of Theorem 12.3.1: We first decribe I in terms of two subroutines: a hash oracle simulator $H\text{-}\mathit{Sim}(\cdot)$ and a sign oracle simulator $\mathit{Sign}\text{-}\mathit{Sim}(\cdot)$. It takes inputs N, e, y where $y \in \mathbf{Z}_N^*$ and maintains three tables, Msg, X and Y, each an array with index in the range from 1 to q_{hash}. It picks a random index i. All these are global variables which will be used also be the subroutines. The intended meaning of the array entries is the following, for $j = 1, \ldots, q_{\text{hash}}$–

$Msg[j]$ – The j-th hash query in the experiment

$Y[j]$ – The reply of the hash oracle simulator to the above, meaning the value playing the role of $H(Msg[j])$. For $j = i$ it is y.

$X[j]$ – For $j \neq i$, the response to sign query $Msg[j]$, meaning it satisfies $(X[j])^e \equiv Y[j] \pmod{N}$. For $j = i$ it is undefined.

The code for the inverter is below.

```
Inverter $I(N, e, y)$
    Initialize arrays $Msg[1 \ldots q_{\rm hash}]$, $X[1 \ldots q_{\rm hash}]$, $Y[1 \ldots q_{\rm hash}]$ to empty
    $j \leftarrow 0$ ; $i \stackrel{\$}{\leftarrow} \{1, \ldots, q_{\rm hash}\}$
    Run $F$ on input $(N, e)$
    If $F$ makes oracle query $(\mathsf{hash}, M)$
        then $h \leftarrow H\text{-}Sim(M)$ ; return $h$ to $F$ as the answer
    If $F$ makes oracle query $(\mathsf{sign}, M)$
        then $x \leftarrow Sign\text{-}Sim(M)$ ; return $x$ to $F$ as the answer
    Until $F$ halts with output $(M, x)$
    $y' \leftarrow H\text{-}Sim(M)$
    Return $x$
```

The inverter responds to oracle queries by using the appropriate subroutines. Once it has the claimed forgery, it makes the corresponding hash query and then returns the signature x.

We now describe the hash oracle simulator. It makes reference to the global variables instantiated in in the main code of I. It takes as argument a value v which is simply some message whose hash is requested either directly by F or by the sign simulator below when the latter is invoked by F.

We will make use of a subroutine *Find* that given an array A, a value v and index m, returns 0 if $v \notin \{A[1], \ldots, A[m]\}$, and else returns the smallest index l such that $v = A[l]$.

```
Subroutine $H\text{-}Sim(v)$
    $l \leftarrow Find(Msg, v, j)$ ; $j \leftarrow j + 1$ ; $Msg[j] \leftarrow v$
    If $l = 0$ then
        If $j = i$ then $Y[j] \leftarrow y$
        Else $X[j] \stackrel{\$}{\leftarrow} Z_N^*$ ; $Y[j] \leftarrow (X[j])^e \bmod N$
        EndIf
        Return $Y[j]$
    Else
        If $j = i$ then abort
        Else $X[j] \leftarrow X[l]$ ; $Y[j] \leftarrow Y[l]$ ; Return $Y[j]$
        EndIf
    EndIf
```

The manner in which the hash queries are answered enables the following sign simulator.

```
Subroutine $Sign\text{-}Sim(M)$
    $h \leftarrow H\text{-}Sim(M)$
    If $j = i$ then abort
    Else return $X[j]$
    EndIf
```

Inverter I might abort execution due to the "abort" instruction in either subroutine. The first such situation is that the hash oracle simulator is unable to return y as the response to the i-th hash query because this query equals a previously replied to query. The second case is that F asks for the signature of the message which is the i-th hash query, and I cannot provide that since it is hoping the i-th message is the one in the forgery and has returned y as the hash oracle response.

Now we need to lower bound the ow-kea-advantage of I with respect to $\mathcal{K}_{\mathrm{rsa}}$. There are a few observations involved in verifying the bound claimed in Equation (12.2). First that the "view" of F at any time at which I has not aborted is the "same" as in Experiment $\mathbf{Exp}^{\mathrm{uf\text{-}cma}}_{\mathcal{DS}}(F)$. This means that the answers being returned to F by I are distributed exactly as they would be in the real experiment. Second, F gets no information about the value i that I chooses at random. Now remember that the last hash simulator query made by I is the message M in the forgery, so M is certainly in the array Msg at the end of the execution of I. Let $l = Find(Msg, M, q_{\mathrm{hash}})$ be the first index at which M occurs, meaning $Msg[l] = M$ but no previous message is M. The random choice of i then means that there is a $1/q_{\mathrm{hash}}$ chance that $i = l$, which in turn means that $Y[i] = y$ and the hash oracle simulator won't abort. If x is a correct signature of M we will have $x^e \equiv Y[i] \pmod N$ because $Y[i]$ is $H(M)$ from the point of view of F. So I is successful whenever this happens. ■

12.3.6 PSS0: A security improvement

The FDH-RSA signature scheme has the attractive security attribute of possessing a proof of security under the assumption that RSA is a one-way function, albeit in the random oracle model. However the quantitative security as given by Theorem 12.3.1 could be better. The theorem leaves open the possibility that one could forge signatures with a probability that is q_{hash} times the probability of being able to invert the RSA function at a random point, the two actions being measured with regard to adversaries with comparable execution time. Since q_{hash} could be quite large, say 2^{60}, there is an appreciable loss in security here. We now present a scheme in which the security relation is much tighter: the probability of signature forgery is not appreciably higher than that of being able to invert RSA in comparable time.

The scheme is called PSS0, for "probabilistic signature scheme, version 0", to emphasize a key aspect of it, namely that it is randomized: the signing algorithm picks a new random value each time it is invoked and uses that to compute signatures. The scheme $\mathcal{DS} = (\mathcal{K}_{\mathrm{rsa}}, \mathrm{Sign}, \mathrm{VF})$, like FDH-RSA, makes use of a public hash function $H\colon \{0,1\}^* \to Z_N^*$ which is modeled as a random oracle. Additonally it has a parameter s which is the length of the random value chosen by the signing algorithm. We write the signing and verifying algorithms as follows:

```
Algorithm $\mathrm{Sign}^{H(\cdot)}_{N,p,q,d}(M)$
    $r \xleftarrow{\$} \{0,1\}^s$
    $y \leftarrow H(r \,\|\, M)$
    $x \leftarrow y^d \bmod N$
    Return $(r, x)$
```

```
Algorithm $\mathrm{VF}^{H(\cdot)}_{N,e}(M, \sigma)$
    Parse $\sigma$ as $(r, x)$ where $|r| = s$
    $y \leftarrow H(r \,\|\, M)$
    If $x^e \bmod N = y$
        Then return 1 else return 0
```

Obvious "range checks" are for simplicity not written explicitly in the verification code; for example in a real implementation the latter should check that $1 \le x < N$ and $\gcd(x, N) = 1$.

This scheme may still be viewed as being in the "hash-then-invert" paradigm, except that the hash is randomized via a value chosen by the signing algorithm. If you twice sign the same message, you are likely to get different signatures. Notice that random value r must be included in the signature since otherwise it would not be possible to verify the signature. Thus unlike the previous schemes, the signature is not a member of Z_N^*; it is a pair one of whose components is an s-bit string and the other is a member of Z_N^*. The length of the signature is $s+k$ bits, somewhat longer than signatures for deterministic hash-then-invert signature schemes. It will usually suffice to set l to, say, 160, and given that k could be 1024, the length increase may be tolerable.

The success probability of a forger F attacking $\mathcal{DS}$ is measured in the random oracle model, via experiment $\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(F)$. Namely the experiment is the same experiment as in the FDH-RSA case; only the scheme $\mathcal{DS}$ we plug in is now the one above. Accordingly we have the insecurity function associated to the scheme. Now we can summarize the security property of the PSS0 scheme.

Theorem 12.3.2 Let $\mathcal{DS}$ be the PSS0 scheme with security parameters k and s. Let F be an adversary making q_{sig} signing queries and $q_{\text{hash}} \geq 1 + q_{\text{sig}}$ hash oracle queries. Then there exists an adversary I such that

$$\mathbf{Adv}^{\text{uf-cma}}_{\mathcal{DS}}(F) \quad \leq \quad \mathbf{Adv}^{\text{ow-kea}}_{\mathcal{K}_{\text{rsa}}}(I) + \frac{(q_{\text{hash}}-1)\cdot q_{\text{sig}}}{2^s} \,. \;\blacksquare \tag{12.3}$$

Say $q_{\text{hash}} = 2^{60}$ and $q_{\text{sig}} = 2^{40}$. With $l = 160$ the additive term above is about 2^{-60}, which is very small. So for all practical purposes the additive term can be neglected and the security of the PSS0 signature scheme is tightly related to that of RSA.

We proceed to the proof of Theorem 12.3.2. The design of I follows the same framework used in the proof of Theorem 12.3.1. Namely I, on input N, e, y, will execute F on input N, e, and answer F's oracle queries so that F can complete its execution. From the forgery, I will somehow find $y^d \bmod N$. I will respond to hash oracle queries of F via a subroutine *H-Sim* called the hash oracle simulator, and will respond to sign queries of F via a subroutine *Sign-Sim* called the sign oracle simulator. A large part of the design is the design of these subroutines. To get some intuition it is helpful to step back to the proof of Theorem 12.3.1.

We see that in that proof, the multiplicative factor of q_{hash} in Equation (12.2) came from I's guessing at random a value $i \in \{1, \ldots, q_{\text{hash}}\}$, and hoping that $i = \mathit{Find}(\mathit{Msg}, M, q_{\text{hash}})$ where M is the message in the forgery. That is, it must guess the time at which the message in the forgery is first queried of the hash oracle. The best we can say about the chance of getting this guess right is that it is at least $1/q_{\text{hash}}$. However if we now want I's probability of success to be as in Equation (12.3), we cannot afford to guess the time at which the forgery message is queried of the hash oracle. Yet, we certainly don't know this time in advance. Somehow, I has to be able to take advantage of the forgery to return $y^d \bmod N$ nonetheless.

A simple idea that comes to mind is to return y as the answer to all hash queries. Then certainly a forgery on a queried message yields the desired value $y^d \bmod N$. Consider this strategy for FDH. In that case, two problems arise. First, these answers would then not be random and indpendent, as required for answers to hash queries. Second, if a message in a hash query is later a sign query, I would have no way to answer the sign query. (Remember that I computed its reply to hash query $\mathit{Msg}[j]$ for $j \neq i$ as $(X[j])^e \bmod N$ exactly in order to be able to later return $X[j]$ if $\mathit{Msg}[j]$ showed up as a sign query. But there is a conflict here: I can either do this, or return y, but not both. It has to choose, and in FDH case it chooses at random.)

The first problem is actually easily settled by a small algebraic trick, exploiting what is called the *self-reducibility* of RSA. When I wants to return y as an answer to a hash oracle query $\mathit{Msg}[j]$, it picks a random $X[j]$ in Z_N^* and returns $Y[j] = y \cdot (X[j])^e \bmod N$. The value $X[j]$ is chosen randomly and independently each time. Now the fact that $\mathsf{RSA}_{N,e}$ is a permutation means that all the different $Y[j]$ values are randomly and independently distributed. Furthermore, suppose $(M, (r, x))$ is a forgery for which hash oracle query $r \parallel M$ has been made and got the reponse $Y[l] = y \cdot (X[l])^e \bmod N$. Then we have $(x \cdot X[l]^{-1})^e \equiv y \pmod N$, and thus the inverse of y is $x \cdot X[l]^{-1} \bmod N$.

The second problem however, cannot be resolved for FDH. That is exactly why PSS0 pre-pends the random value r to the message before hashing. This effectively "separates" the two kinds of hash queries: the direct queries of F to the hash oracle, and the indirect queries to the hash oracle arising from the sign oracle. The direct hash oracle queries have the form $r \parallel M$ for some l-bit string r and some message M. The sign query is just a message M. To answer it, a value r is first chosen at random. But then the value $r \parallel M$ has low probability of having been a previous hash query. So at the time any new direct hash query is made, I can assume it will never be an indirect hash query, and thus reply via the above trick.

Here now is the full proof.

Proof of Theorem 12.3.2: We first decribe I in terms of two subroutines: a hash oracle simulator $H\text{-}Sim(\cdot)$ and a sign oracle simulator $Sign\text{-}Sim(\cdot)$. It takes input N, e, y where $y \in \mathbf{Z}_N^*$, and maintains four tables, R, V, X and Y, each an array with index in the range from 1 to q_{hash}. All these are global variables which will be used also be the subroutines. The intended meaning of the array entries is the following, for $j = 1, \ldots, q_{\text{hash}}$–

$V[j]$ – The j-th hash query in the experiment, having the form $R[j] \parallel Msg[j]$

$R[j]$ – The first l-bits of $V[j]$

$Y[j]$ – The value playing the role of $H(V[j])$, chosen either by the hash simulator or the sign simulator

$X[j]$ – If $V[j]$ is a direct hash oracle query of F this satisfies $Y[j] \cdot X[j]^{-e} \equiv y \pmod N$. If $V[j]$ is an indirect hash oracle query this satisfies $X[j]^e \equiv Y[j] \pmod N$, meaning it is a signature of $Msg[j]$.

Note that we don't actually need to store the array Msg; it is only referred to above in the explanation of terms.

We will make use of a subroutine $Find$ that given an array A, a value v and index m, returns 0 if $v \notin \{A[1], \ldots, A[m]\}$, and else returns the smallest index l such that $v = A[l]$.

```
Inverter $I(N, e, y)$
    Initialize arrays $R[1 \ldots q_{\text{hash}}]$, $V[1 \ldots q_{\text{hash}}]$, $X[1 \ldots q_{\text{hash}}]$, $Y[1 \ldots q_{\text{hash}}]$, to empty
    $j \leftarrow 0$
    Run $F$ on input $N, e$
    If $F$ makes oracle query $(\mathsf{hash}, v)$
        then $h \leftarrow H\text{-}Sim(v)$ ; return $h$ to $F$ as the answer
    If $F$ makes oracle query $(\mathsf{sign}, M)$
        then $\sigma \leftarrow Sign\text{-}Sim(M)$ ; return $\sigma$ to $F$ as the answer
    Until $F$ halts with output $(M, (r, x))$
    $y \leftarrow H\text{-}Sim(r \parallel M)$ ; $l \leftarrow Find(V, r \parallel M, q_{\text{hash}})$
    $w \leftarrow x \cdot X[l]^{-1} \bmod N$ ; Return $w$
```

We now describe the hash oracle simulator. It makes reference to the global variables instantiated in in the main code of I. It takes as argument a value v which is assumed to be at least s bits long, meaning of the form $r \parallel M$ for some s bit strong r. (There is no need to consider hash queries not of this form since they are not relevant to the signature scheme.)

```
Subroutine $H\text{-}Sim(v)$
    Parse $v$ as $r \parallel M$ where $|r| = s$
    $l \leftarrow Find(V, v, j)$ ; $j \leftarrow j+1$ ; $R[j] \leftarrow r$ ; $V[j] \leftarrow v$
    If $l = 0$ then
        $X[j] \xleftarrow{\$} Z_N^*$ ; $Y[j] \leftarrow y \cdot (X[j])^e \bmod N$ ; Return $Y[j]$
    Else
        $X[j] \leftarrow X[l]$ ; $Y[j] \leftarrow Y[l]$ ; Return $Y[j]$
    EndIf
```

Every string v queried of the hash oracle is put by this routine into a table V, so that $V[j]$ is the j-th hash oracle query in the execution of F. The following sign simulator does not invoke the hash simulator, but if necessary fills in the necessary tables itself.

```
Subroutine $Sign\text{-}Sim(M)$
    $r \xleftarrow{\$} \{0,1\}^s$
    $l \leftarrow Find(R, r, j)$
    If $l \neq 0$ then abort
    Else
        $j \leftarrow j+1$ ; $R[j] \leftarrow r$ ; $V[j] \leftarrow r \parallel M$ ; $X[j] \xleftarrow{\$} Z_N^*$ ; $Y[j] \leftarrow (X[j])^e \bmod N$
        Return $X[j]$
    EndIf
```

Now we need to establish Equation (12.3).

First consider $\mathbf{Exp}^{\text{ow-kea}}_{\mathcal{K}_{\text{rsa}}}(I)$ and let $\Pr_1[\cdot]$ denote the probability function in this experiment. Let BAD_1 be the event that I aborts due to the "abort" instruction in the sign-oracle simulator.

Now consider $\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(F)$, and let $\Pr_2[\cdot]$ denote the probability function in this experiment. Let BAD_2 be the event that the sign oracle picks a value r such that F had previously made a hash query $r \parallel M$ for some M.

Let SUCC be the event (in either experiment) that F succeeds in forgery. Now we have

$$\begin{aligned}
\mathbf{Adv}^{\text{uf-cma}}_{\mathcal{DS}}(F) &= \Pr_2[\text{SUCC}] \\
&= \Pr_2[\text{SUCC} \wedge \overline{\text{BAD}_2}] + \Pr_2[\text{SUCC} \wedge \text{BAD}_2] \\
&\leq \Pr_2[\text{SUCC} \wedge \overline{\text{BAD}_2}] + \Pr_2[\text{BAD}_2] \\
&= \Pr_1[\text{SUCC} \wedge \overline{\text{BAD}_1}] + \Pr_1[\text{BAD}_1] && (12.4) \\
&= \mathbf{Adv}^{\text{ow-kea}}_{\mathcal{K}_{\text{rsa}}}(I) + \Pr_1[\text{BAD}_1] && (12.5) \\
&\leq \mathbf{Adv}^{\text{ow-kea}}_{\mathcal{K}_{\text{rsa}}}(I) + \frac{(q_{\text{hash}}-1)q_{\text{sig}}}{2^s} \,. && (12.6)
\end{aligned}$$

This establishes Equation (12.3). Let us now provide some explanations for the above.

First, Equation (12.6) is justified as follows. The event in question happens if the random value r chosen in the sign oracle simulator is already present in the set $\{R[1], \ldots, R[j]\}$. This set has size at most $q_{\text{hash}} - 1$ at the time of a sign query, so the probability that r falls in it is at most $(q_{\text{hash}} - 1)/2^s$. The sign oracle simulator is invoked at most q_{sig} times, so the bound follows.

It is tempting to think that the "view" of F at any time at which I has not aborted is the "same" as the view of F in Experiment $\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(F)$. This is not true, because it can test whether or not BAD occured. That's why we consider bad events in both games, and note that

$$\mathbf{Adv}^{\text{ow-kea}}_{\mathcal{K}_{\text{rsa}}}(I) \;=\; \Pr_1\left[\text{SUCC} \wedge \overline{\text{BAD}_1}\right] \;=\; \Pr_2\left[\text{SUCC} \wedge \overline{\text{BAD}_2}\right] \;.$$

This is justified as follows. Remember that the last hash simulator query made by I is $r \,\|\, M$ where M is the message in the forgery, so $r \,\|\, M$ is certainly in the array V at the end of the execution of I. So $l = \mathit{Find}(V, r \,\|\, M, q_{\text{hash}}) \neq 0$. We know that $r \,\|\, M$ was not put in V by the sign simulator, because F is not allowed to have made sign query M. This means the hash oracle simulator has been invoked on $r \,\|\, M$. This means that $Y[l] = y \cdot (X[l])^e \bmod N$ because that is the way the hash oracle simulator chooses its replies. The correctness of the forgery means that $x^e \equiv H(r \,\|\, M) \pmod{N}$, and the role of the H value here is played by $Y[l]$, so we get $x^e \equiv Y[l] \equiv y \cdot X[l] \pmod{N}$. Solving this gives $(x \cdot X[l]^{-1})^e \bmod N = y$, and thus the inverter is correct in returning $x \cdot X[l]^{-1} \bmod N$. ▮

It may be worth adding some words of caution about the above. It is tempting to think that

$$\mathbf{Adv}^{\text{ow-kea}}_{\mathcal{K}_{\text{rsa}}}(I) \;\geq\; \left[1 - \frac{(q_{\text{hash}} - 1) \cdot q_{\text{sig}}}{2^s}\right] \cdot \mathbf{Adv}^{\text{uf-cma}}_{\mathcal{DS}}(F) \;,$$

which would imply Equation (12.3) but is actually stronger. This however is not true, because the bad events and success events as defined above are not independent.