

Chapter 2

Cryptographic Hardness Assumptions

As noted in the previous chapter, it is impossible to construct a digital signature scheme that is secure against an all-powerful adversary. Instead, the best we can hope for is to construct schemes that are secure against *computationally bounded* adversaries (that, for our purposes, means adversaries running in probabilistic polynomial time). Even for this “limited” class of adversaries, however, we do not currently have any constructions that can be proven, unconditionally, to be secure. In fact, it is not too difficult to see that the existence of a secure signature scheme would imply¹ $P \neq NP$, a breakthrough in complexity theory. (While there is general belief that $P \neq NP$ is true, we seem very far away from being able to prove this.) Actually, as we will see below, the existence of a secure signature scheme implies the existence of one-way functions, something not known to follow from $P \neq NP$ and thus an even stronger result. (Informally, the issue is that $P \neq NP$ only guarantees the existence of problems that are hard *in the worst case*. But a secure signature scheme is required to be “hard to break” *on the average* — in particular, for “average” public keys generated by signers.)

Given this state of affairs, all existing constructions of signature schemes are proven secure relative to some *assumption* regarding the hardness of solving some (cryptographic) problem. We introduce some longstanding and widely used assumptions in this chapter; other, more recent cryptographic assumptions are introduced as needed throughout the rest of the book.

2.1 “Generic” Cryptographic Assumptions

We begin by discussing “generic” cryptographic assumptions before turning to various concrete, number-theoretic constructions conjectured to satisfy these assump-

¹ See any book on complexity theory for definitions of these classes, which are not essential to the rest of the book.

tions. While any scheme used in practice must be based on some concrete “hard” problem, there are several advantages of studying generic assumptions:

- A signature scheme based on generic assumptions is not tied to any *particular* “hard” problem, and therefore offers greater flexibility to the implementor. As but one illustration of this flexibility, note that a signature scheme based on a *specific* assumption must be scrapped if the assumption is found to be false, whereas a signature scheme based on generic assumptions (that was instantiated with the particular assumption found to be false) can simply be instantiated using a different concrete problem.
- Constructions based on generic assumptions are often simpler to analyze and understand, since abstracting away the “unnecessary” details has the effect of highlighting those details that are important.
- Generic constructions are interesting from a theoretical point of view insofar as they indicate what is feasible, and what are the minimal assumptions that are necessary. These are often useful steps toward developing practical schemes.

On the other hand, tailoring a signature scheme to a specific assumption can often lead to a much more efficient scheme than simply “plugging in” that same assumption to a generic template. Indeed, constructions based on specific assumptions are generally orders of magnitude more efficient than schemes based on generic assumptions, regardless of how the latter are instantiated.

2.1.1 One-Way Functions and Permutations

The most basic building block in cryptography — in fact, as we will see, the minimal assumption needed for constructing secure signature schemes — is a *one-way function*. Informally, a one-way function f is a function that is “easy” to compute but “hard” to invert *on the average*, in a way made precise below. We give two definitions of a one-way function: the first is easier to work with, while the second is easier to instantiate using known number-theoretic primitives (and can also yield more efficient constructions). Fortunately, the two definitions are equivalent in the sense that one exists if and only if the other does. We also define two notions of one-way *permutations*, though equivalence in this case is not known to hold.

Definition 2.1. A function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ is a **one-way function** if:

1. (Easy to compute:) There is a deterministic, polynomial-time algorithm Eval_f such that for all k and all $x \in \{0, 1\}^k$ we have $\text{Eval}_f(x) = f(x)$. (From now on, we do not explicitly mention Eval_f and only refer to f itself, keeping in mind that f can be computed in polynomial time.)
2. (Hard to invert:) The following is negligible for all PPT algorithms A :

$$\Pr \left[x \leftarrow \{0, 1\}^k; y := f(x); x' \leftarrow A(1^k, y) : f(x') = y \right].$$

Note that it is not required that $x = x'$ in the above.

A one-way function f is a **one-way permutation** if f is bijective and length-preserving (i.e., $|f(x)| = |x|$ for all x).

It is not difficult to show that the existence of a one-way function implies $P \neq NP$. On the other hand, we do not currently know whether $P \neq NP$ necessarily implies the existence of one-way functions.

While the above definition is convenient when one-way functions or permutations are used to construct other objects, it does not provide a natural model for the concrete examples of one-way functions/permutations that we currently know. Instead, it is often simpler to work with *families* of one-way functions/permutations, defined next.

Definition 2.2. A tuple $\Pi = (\text{Gen}, \text{Samp}, f)$ of PPT algorithms is a **function family** if the following hold:

1. Gen , the **parameter-generation algorithm**, is a probabilistic algorithm that takes as input a security parameter 1^k and outputs parameters I with $|I| \geq k$. Each value of I output by Gen defines sets D_I and R_I that constitute the domain and range, respectively, of the function f_I defined below.
2. Samp , the **sampling algorithm**, is a probabilistic algorithm that takes as input parameters I and outputs an element of D_I (except possibly with negligible probability).
3. f , the **evaluation algorithm**, is a deterministic algorithm that takes as input parameters I and an element $x \in D_I$, and outputs an element $y \in R_I$. We write this as $y := f_I(x)$. (That is, the function f_I is defined by the behavior of f on parameters I .)

Π is a **permutation family** if the following additionally hold:

1. For all I output by Gen , the distribution defined by the output of $\text{Samp}(I)$ is (statistically close to) the uniform distribution on D_I .
2. For all I output by Gen it holds that $D_I = R_I$ and the function f_I is a bijection.

If Π is a permutation family and there exists a polynomial p such that $D_I = \{0, 1\}^{p(k)}$ for all I output by $\text{Gen}(1^k)$, then we say that Π is a **permutation family over bit-strings**. In this case we will assume the trivial sampling algorithm (that simply outputs its random coins).

Definition 2.3. A function/permutation family $\Pi = (\text{Gen}, \text{Samp}, \text{Eval})$ is **one-way** if for all PPT algorithms A , the following is negligible (in k):

$$\Pr[I \leftarrow \text{Gen}(1^k); x \leftarrow \text{Samp}(I); y := f_I(x); x' \leftarrow A(I, y) : f_I(x') = y].$$

Any one-way permutation family satisfying some mild additional conditions can be transformed into a one-way permutation family over bit-strings, and we now sketch this transformation. Let Π be a one-way permutation family with $D_I \subseteq \{0, 1\}^{p(k)}$ (for some polynomial p) for all I output by $\text{Gen}(1^k)$. We additionally require that:

- Given I , the set D_I is efficiently recognizable. (I.e., there is a polynomial-time algorithm A that takes as input I and a string $x \in \{0, 1\}^{p(k)}$ and correctly decides whether $I \in D_I$.)
- For all I , the set D_I is *dense* in $\{0, 1\}^{p(k)}$. That is, $|D_I|/2^{p(k)} = 1/\text{poly}(k)$.

Construct a permutation family $\Pi' = (\text{Gen}', \text{Samp}', f')$ as follows: Gen' is identical to Gen . The sampling algorithm Samp' is the trivial one that outputs a random string of length $p(k)$ (we assume that k can be determined from I). Finally, define function $f'_I : \{0, 1\}^{p(k)} \rightarrow \{0, 1\}^{p(k)}$ as:

$$f'_I(x) = \begin{cases} f_I(x) & x \in D_I \\ x & \text{otherwise} \end{cases}.$$

Note that Π' is not necessarily one-way, since f'_I is trivial to invert on any point $y \notin D_I$. Nevertheless, it is hard to invert f'_I on a noticeable fraction of its range. This hardness can be “amplified” by running many copies of Π' in parallel. That is, define $\Pi'' = (\text{Gen}'', \text{Samp}'', f'')$ as follows: Gen'' is the same as Gen . The sampling algorithm Samp'' outputs a random string of length $\ell(k) \cdot p(k)$ for an appropriate polynomial ℓ . Finally,

$$f''_I(x_1 \| \cdots \| x_{\ell(k)}) \stackrel{\text{def}}{=} f'_I(x_1) \| \cdots \| f'_I(x_{\ell(k)}).$$

Intuitively, it is clear that inversion is difficult as long as *any* of the x_i are in D_I , and this is true for some x_i with all but negligible probability (for ℓ chosen appropriately). A formal proof that Π'' is a one-way permutation family over bit-strings is not much more difficult.

We have defined both one-way functions (cf. Definition 2.1) and one-way function families (cf. Definition 2.3). We now show that these definitions are equivalent.

Lemma 2.1. *A one-way function f (in the sense of Definition 2.1) exists iff a one-way function family (in the sense of Definition 2.3) exists.*

Proof (sketch). It is immediate that a one-way function f implies the existence of a one-way function family: simply let Gen be the trivial algorithm that on input 1^k outputs $I = 1^k$; take Samp to be the algorithm that on input $I = 1^k$ outputs a uniformly distributed string $x \in \{0, 1\}^k$; and define $f_I(x) = f(x)$.

The other direction is also conceptually simple, just more technical. Let $\Pi = (\text{Gen}, \text{Samp}, f)$ be a one-way function family such that the running time of Gen is bounded by p_1 and the running time of Samp is bounded by p_2 , and let $p \stackrel{\text{def}}{=} p_1 + p_2$; note that p is a polynomial and furthermore that the combined length of the random tapes used by Gen and Samp for security parameter k is bounded by $p(k)$. Define f as follows: on input $r \in \{0, 1\}^k$ find the largest integer \bar{k} such that $p(\bar{k}) \leq k$. Parse r as $r_1 | r_2$ with $|r_1| = p_1(\bar{k})$ and $|r_2| \geq p_2(\bar{k})$. Set $I := \text{Gen}(1^{\bar{k}}; r_1)$ and $x := \text{Samp}(I; r_2)$ (note that we fix the random tapes of Gen and Samp , so this step is deterministic), and compute $y := f_I(x)$. The output of f is the pair (I, y) . The proof that f is a one-way function is tedious, but straightforward.

The above shows that one-way functions are equivalent to one-way function families. In contrast, while the existence of one-way permutations is easily seen to imply the existence of one-way permutation families, the converse is not known. Moreover, the specific number-theoretic assumptions discussed below yield one-way permutation families (indeed, one-way permutation families over bit-strings) much more naturally than they do one-way permutations. We will therefore work exclusively with the notion of one-way permutation families over bit-strings.

This is a good place to record the following observation.

Theorem 2.1. *The existence of a signature scheme that is existentially unforgeable under a one-time random-message attack implies the existence of a one-way function.*

Proof (sketch). In fact even security against a *no*-message attack suffices to prove the claim. Let $\Pi = (\text{Gen}, \text{Sign}, \text{Vrfy})$ be a signature scheme that is existentially unforgeable under a no-message attack, where an adversary is given only the public key pk and succeeds if it outputs (m, σ) with $\text{Vrfy}_{pk}(m, \sigma) = 1$. Let $p(k)$ be a polynomial bounding the length of the random tape used by Gen on security parameter 1^k . Define a one-way function f as follows: on input $r \in \{0, 1\}^k$, compute the largest integer \bar{k} such that $p(\bar{k}) \leq k$. Then run $\text{Gen}(1^{\bar{k}}; r)$ to obtain (pk, sk) , and output pk .

Observe that any PPT algorithm A inverting f can be used to forge signatures in Π as follows: given pk , run A to obtain a string r . If $f(r) = pk$, then this means that running $\text{Gen}(1^{\bar{k}}; r)$ yields a pair (pk, sk') . It is then trivial to output a forgery on any message m by computing the signature $\sigma \leftarrow \text{Sign}_{sk'}(m)$. (Note that sk' need not be equal to the “real” secret key sk used by the signer; i.e., there may be multiple valid secret keys associated with the single public key pk . But correctness of Π implies that this does not matter, since valid signatures with respect to pk can be produced using *any* secret key associated with pk .)

2.1.2 Trapdoor Permutations

A stronger notion than that of one-way functions is obtained by introducing an “asymmetry” of sorts whereby one party *can* efficiently accomplish some task that is infeasible for anyone else. This leads to the idea of *trapdoor permutations* that may be viewed, informally, as one-way permutations that can be efficiently inverted given some additional “trapdoor” information. (One can also consider *trapdoor functions* but these turn out to be much less useful.) A definition follows.

Definition 2.4. A tuple $\Pi = (\text{Gen}, \text{Samp}, f, f^{-1})$ of PPT algorithms is a **trapdoor permutation family** if the following hold:

- Gen , the **parameter-generation algorithm**, is a probabilistic algorithm that takes as input a security parameter 1^k and outputs parameters I (with $|I| \geq k$) along with an associated trapdoor td .

Each value of I output by Gen defines a set D_I that constitutes the domain and range of a permutation f_I defined below.

- Samp , the **sampling algorithm**, is a probabilistic algorithm that takes as input parameters I and outputs an element $x \in D_I$ whose distribution is statistically close to the uniform distribution over D_I . We will sometimes leave Samp implicit and just write² $x \leftarrow D_I$.
- f , the **evaluation algorithm**, is a deterministic algorithm that takes as input parameters I and an element $x \in D_I$, and outputs an element $y \in D_I$. We write this as $y := f_I(x)$.
- f^{-1} , the **inversion algorithm**, is a deterministic algorithm that takes as input parameters I , a trapdoor td , and an element $y \in D_I$. It outputs an element $x \in D_I$. We leave I implicit, and write this as $x := f_{\text{td}}^{-1}(y)$.
- For all k , all (I, td) output by $\text{Gen}(1^k)$, and all $x \in D_I$ we have $f_{\text{td}}^{-1}(f_I(x)) = x$, and hence $f_{\text{td}}^{-1}(\cdot)$ and $f_I(\cdot)$ are both permutations over D_I , and inverses of each other.
- The following is negligible for all PPT algorithms A :

$$\Pr \left[(I, \text{td}) \leftarrow \text{Gen}(1^k); y \leftarrow D_I; x \leftarrow A(I, y) : f_I(x) = y \right].$$

For brevity, and since it will not cause confusion, we simply refer to a “trapdoor permutation” rather than a “trapdoor permutation family”.

Because f_I is a permutation, choosing x uniformly from D_I and then setting $y := f_I(x)$ results in a value y that is uniformly distributed in D_I . We note also that it is possible for f_I to be defined over some set that (strictly) contains D_I , but the function is only guaranteed to be a *bijection* when its inputs are taken from D_I . The final condition of the definition, however, requires that it be “hard” to find *any* x mapping to y (i.e., even an $x \notin D_I$).

Occasionally, when we do not care about the particular index I or trapdoor td , we will write $(f, f^{-1}) \leftarrow \text{Gen}(1^k)$ and write $f(\cdot)$ in place of $f_I(\cdot)$ and $f^{-1}(\cdot)$ in place of f_{td}^{-1} . Of course, it is important to keep in mind that I is required in order to evaluate f , and that f^{-1} can only be evaluated efficiently with knowledge of td .

Trapdoor permutations, in the sense defined above, do not suffice for most of the applications we will see in this book. Instead, we need the following strengthening:

Definition 2.5. A trapdoor permutation family $\Pi = (\text{Gen}, \text{Samp}, f, f^{-1})$ is called **doubly enhanced**³ if the following conditions hold:

1. The following is negligible for all PPT algorithms A :

$$\Pr \left[(I, \text{td}) \leftarrow \text{Gen}(1^k); r \leftarrow \{0, 1\}^*; y := \text{Samp}(I; r); x \leftarrow A(I, y, r) : f_I(x) = y \right].$$

² Technically, $x \leftarrow D_I$ refers to selecting x uniformly from D_I . Since the distribution produced by Samp is statistically close to uniform, the difference is unimportant.

³ We use this terminology to distinguish our definition from that of enhanced trapdoor permutations, which satisfy only the first condition.

That is, it should be difficult to find a pre-image of y *even when given the random coins used to sample* y .

2. Let $p(k)$ denote the length of the random tape used by Samp on security parameter 1^k . There exists a PPT algorithm Samp' that takes as input I and outputs a tuple (x, y, r) with $x \in D_I$ and $r \in \{0, 1\}^{p(k)}$ and such that:

- $f_I(x) = y$ and $y = \text{Samp}(I; r)$;
- The distribution on r is statistically close to uniform.

We can also define a trapdoor permutation *over bit-strings* in the natural way (cf. Definition 2.2). It is not hard to see that any trapdoor permutation over bit-strings is also a doubly enhanced trapdoor permutation: the first condition of Definition 2.5 holds by virtue of the fact that Samp is trivial (since $y = \text{Samp}(I; y)$), and the second condition holds by letting Samp' be the algorithm that chooses x uniformly, sets $y := f_I(x)$, and sets $r := y$. All the concrete examples of trapdoor permutations that we will see in this book can be suitably “massaged” to be trapdoor permutations over bit-strings.

2.1.3 Clawfree (Trapdoor) Permutations

A pair of *clawfree permutations* is, informally, a pair of efficiently computable permutations f_0, f_1 defined over the same domain for which it is hard to find a *claw*: namely, a pair x_0, x_1 with $f_0(x_0) = f_1(x_1)$. A pair of clawfree *trapdoor* permutations additionally has an associated *trapdoor* td that allows for efficient inversion of f_0 and f_1 . Observe that given such trapdoor information, it is easy to find a claw: simply choose arbitrary y and compute $x_0 := f_0^{-1}(y)$ and $x_1 := f_1^{-1}(y)$; thus, hardness of finding a claw holds only for algorithms that do not have access to the trapdoor.

Definition 2.6. A tuple $\Pi = (\text{Gen}, \text{Samp}, f_0, f_1)$ of PPT algorithms is a **clawfree permutation family** if the following hold:

- Gen , the **parameter-generation algorithm**, is a probabilistic algorithm that takes as input a security parameter 1^k and outputs parameters I (with $|I| \geq k$) along with an associated trapdoor td .
Each value of I output by Gen defines a set D_I that constitutes the domain and range of permutations $f_{I,0}, f_{I,1}$ defined below.
- Samp , the **sampling algorithm**, is a probabilistic algorithm that takes as input parameters I and outputs an element $x \in D_I$ whose distribution is statistically close to the uniform distribution over D_I . We usually leave Samp implicit, and just write $x \leftarrow D_I$.
- f_0 and f_1 , the **evaluation algorithms**, are deterministic algorithms that take as input parameters I and an element $x \in D_I$, and output an element $y \in D_I$. We write this as $y := f_{I,0}(x)$ or $y := f_{I,1}(x)$.

- The following is negligible for all PPT algorithms A :

$$\Pr \left[(I, \text{td}) \leftarrow \text{Gen}(1^k); (x_0, x_1) \leftarrow A(I) : f_{I,0}(x_0) = f_{I,1}(x_1) \right].$$

$\Pi = (\text{Gen}, \text{Samp}, f_0, f_1, f_0^{-1}, f_1^{-1})$ is a **clawfree trapdoor permutation family** if $(\text{Gen}, \text{Samp}, f_0, f_1)$ is a clawfree permutation family and the following additionally hold:

- f_0^{-1} and f_1^{-1} , the **inversion algorithms**, are deterministic algorithms that take as input parameters I , a trapdoor td , and an element $y \in D_I$. They output an element $x \in D_I$. We leave I implicit, and write this as $x := f_{\text{td},0}^{-1}(y)$ or $x := f_{\text{td},1}^{-1}(y)$.
- For all k , all (I, td) output by $\text{Gen}(1^k)$, all $x \in D_I$, and $b \in \{0, 1\}$ we have $f_{\text{td},b}^{-1}(f_{I,b}(x)) = x$. Thus, $f_{\text{td},b}^{-1}(\cdot)$ and $f_{I,b}(\cdot)$ are permutations over D_I and inverses of each other.

As in the case of trapdoor permutations, we often refer to “clawfree (trapdoor) permutations” rather than “clawfree (trapdoor) permutation families.” We may also switch to a less cumbersome notation and write $(f_0, f_1, f_0^{-1}, f_1^{-1}) \leftarrow \text{Gen}(1^k)$ for the output of Gen , and then use $f_0(\cdot), f_1(\cdot)$ in place of $f_{I,0}(\cdot), f_{I,1}(\cdot)$ and, similarly, use $f_0^{-1}(\cdot), f_1^{-1}(\cdot)$ in place of $f_{\text{td},0}^{-1}(\cdot), f_{\text{td},1}^{-1}(\cdot)$. As before, it is important to keep in mind that f_0^{-1}, f_1^{-1} cannot be efficiently evaluated without knowledge of td .

We also note, once again, that it is possible for f_0, f_1 to be defined over some set (strictly) containing the domain D over which these functions are guaranteed to be permutations. The final condition of the definition, however, requires that it be “hard” to find *any* x_0, x_1 for which $f_0(x_0) = f_1(x_1)$ (i.e., even $x_0, x_1 \notin D$).

The existence of clawfree trapdoor permutations represents a (possibly) stronger assumption than the existence of trapdoor permutations:

Lemma 2.2. *If $\Pi = (\text{Gen}, \text{Samp}, f_0, f_1, f_0^{-1}, f_1^{-1})$ is a clawfree trapdoor permutation family, then $\Pi' = (\text{Gen}, \text{Samp}, f_0, f_0^{-1})$ is a trapdoor permutation family. Thus, the existence of clawfree permutations implies the existence of trapdoor permutations.*

Proof (sketch). The syntactic requirements are easily seen to match up, and so all we need to prove is hardness of inversion. Fix any PPT algorithm A' and define:

$$\delta_{A'}(k) \stackrel{\text{def}}{=} \Pr \left[(I, \text{td}) \leftarrow \text{Gen}(1^k); y \leftarrow D_I; x \leftarrow A'(I, y) : f_0(x) = y \right].$$

This is exactly the probability with which A' inverts Π' , and so we prove that Π' is a trapdoor permutation family by showing that $\delta_{A'}(k)$ is negligible.

Consider the following algorithm A for finding a claw in Π , using A' as a subroutine:

Algorithm A:

The algorithm is given parameters I , generated using $\text{Gen}(1^k)$.

Its goal is to find a claw.

- Choose $x_1 \leftarrow D_I$ and compute $y := f_1(x_1)$.
- Run $A'(I, y)$ to obtain x_0 .
- If $f_0(x_0) = y$, then output the claw (x_0, x_1) .

Clearly, A runs in polynomial time. Furthermore, A succeeds in outputting a claw whenever A' succeeds in inverting y with respect to f_0 . Since A chooses x_1 uniformly at random from D_I and f_1 is a permutation over this set, the value y given by A to A' is also uniformly distributed in D_I . Thus, the probability that A' succeeds in inverting y is exactly $\delta_{A'}(k)$, and this is exactly the probability with which A outputs a claw. The fact that Π is clawfree thus implies that $\delta_{A'}(k)$ is negligible, as desired.

By analogy with the case of trapdoor permutations, we may also define a notion of doubly enhanced clawfree trapdoor permutations:

Definition 2.7. Let $\Pi = (\text{Gen}, \text{Samp}, f_0, f_1, f_0^{-1}, f_1^{-1})$ be a clawfree trapdoor permutation family. We say Π is **doubly enhanced** if both $\Pi_0 = (\text{Gen}, \text{Samp}, f_0, f_0^{-1})$ and $\Pi_1 = (\text{Gen}, \text{Samp}, f_1, f_1^{-1})$ are doubly enhanced trapdoor permutation families. That is:

- For $b \in \{0, 1\}$ and any PPT algorithm A the following is negligible:

$$\Pr \left[(I, \text{td}) \leftarrow \text{Gen}(1^k); r \leftarrow \{0, 1\}^*; y := \text{Samp}(I; r); x \leftarrow A(I, y, r) : f_{I,b}(x) = y \right].$$

- If we let $p(k)$ denote the length of the random tape used by Samp on security parameter 1^k , there exist PPT algorithms $\text{Samp}_0, \text{Samp}_1$ where Samp_b takes as input I and outputs a tuple (x, y, r) with $x \in D_I$ and $r \in \{0, 1\}^{p(k)}$ and such that:
 1. $f_{I,b}(x) = y$ and $y = \text{Samp}(I; r)$;
 2. the distribution on r is statistically close to uniform.

We may also define a *clawfree trapdoor permutation family over bit-strings* in the obvious way, and it is easy to see that any such family is also a doubly enhanced clawfree trapdoor permutation family.

2.2 Specific Assumptions

The discussion thus far has been very general. We now show some concrete examples of number-theoretic problems conjectured to be hard, and demonstrate how these can be used to instantiate the generic assumptions described thus far. We assume in this section some familiarity with basic number theory; see the notes at the end of this chapter for pointers to existing references covering this material.

In this chapter we have chosen to focus on the most well known and long-standing cryptographic assumptions; some more recent assumptions are introduced and discussed in Chapters 4 and 5.

2.2.1 Hardness of Factoring

The factoring problem is probably the longest-studied “hard” problem in algorithmic number theory. It is also one of the easiest one-way functions to explain, at least informally, since multiplication is clearly “easy” (i.e., polynomial time) yet finding the prime factorization of a (large) number is widely believed⁴ to be “hard”. But does the conjectured “hardness of factoring” trivially imply a one-way function? A natural first candidate for a one-way function is the function $f_{\text{mult}}(x, y) = xy$. A little thought, however, shows that f_{mult} is decidedly *not* one-way: with probability $3/4$ at least one of x or y will be even, making it trivial to find a factor of xy (recall that one-wayness is defined in terms of the inability to find *any* preimage of a randomly generated point). To avoid problems of this sort, we simply need to restrict the inputs of f_{mult} to (large) *primes* of equal length. Formally, we construct a function family $(\text{Gen}, \text{Samp}, f)$ as follows (cf. Definition 2.3):

- $\text{Gen}(1^k)$ simply outputs $I = 1^k$. We let D_I denote the set of all pairs of k -bit primes.
- $\text{Samp}(1^k)$ is a randomized algorithm that outputs two random (and independently chosen) k -bit primes.
- $f(p, q)$ outputs the product pq .

One way to state the factoring conjecture is as the assumption that the family $(\text{Gen}, \text{Samp}, f)$ defined above is one-way.

Of course, we have omitted what is perhaps the most important detail in the above: how to generate random primes in polynomial time. An algorithm computing Samp follows fairly readily from the following two facts:

1. Prime numbers are sufficiently dense that a random integer is prime with “sufficiently high” probability.
2. There exist (probabilistic) polynomial-time algorithms that can determine (except with negligibly small error) whether a given integer is prime.

We refer the reader to the references listed in the notes at the end of this chapter for further information.

For our purposes, it will be convenient to let GenModulus denote an (unspecified, but polynomial-time) algorithm that, on input 1^k , outputs (N, p, q) such that $N = pq$, and p and q are k -bit primes (with all but negligible probability in k). We can then express the factoring assumption relative to a particular algorithm GenModulus :

Definition 2.8. We say that **factoring is hard relative to** GenModulus if for all PPT algorithms A , the following is negligible:

$$\Pr[(N, p, q) \leftarrow \text{GenModulus}(1^k); (p, q) \leftarrow A(N) : pq = N].$$

⁴ It is crucial to keep in mind here that running time is measured in terms of the *length(s) of the input(s)* and not their magnitude. It is easy to factor a number N in time linear in N using trial division by all numbers less than N . But a polynomial-time algorithm for factoring N is required to work in time polynomial in $|N| = \Theta(\log N)$.

The factoring assumption is that there exists a GenModulus relative to which factoring is hard.

We stress that we do not require that GenModulus choose p and q to be *random* k -bit primes; though that is certainly one possibility (that is also used frequently in practice), we allow for other means of choosing the primes p and q so long as the factoring assumption (relative to GenModulus) is still believed to hold.

Interestingly, the factoring assumption — that, at first glance, seems only to guarantee the existence of a one-way function — can be used to construct a much stronger cryptographic primitive: a (doubly enhanced) clawfree trapdoor permutation family. We first show how to use the factoring assumption to construct a trapdoor permutation family, and then describe the extension to give the result claimed.

We begin with a small amount of (standard) number-theoretic background. Given any integer $N > 1$, let $\mathbb{Z}_N \stackrel{\text{def}}{=} \{0, \dots, N-1\}$. It is a well-known fact that this is a group under addition modulo N . We also define

$$\mathbb{Z}_N^* \stackrel{\text{def}}{=} \{x \in \{1, \dots, N-1\} \mid \gcd(x, N) = 1\}.$$

It is not too difficult to prove that \mathbb{Z}_N^* is a group under multiplication modulo N ; this follows from the fact that \mathbb{Z}_N^* contains exactly those elements of \mathbb{Z}_N that have a multiplicative inverse modulo N .

The squaring function modulo N is the function that maps $x \in \mathbb{Z}_N^*$ to $x^2 \bmod N$. Elements of \mathbb{Z}_N^* that have a square root are called *quadratic residues* modulo N , and we denote the set of quadratic residues modulo N by QR_N . If N is a product of two distinct, odd primes, then squaring modulo N is a four-to-one function; i.e., each quadratic residue modulo N has exactly four square roots. We use this fact in the proof of the following:

Lemma 2.3. *Let $N = pq$ with p, q distinct, odd primes. Given x, \hat{x} such that $x^2 = y = \hat{x}^2 \bmod N$ but $x \not\equiv \pm \hat{x} \bmod N$, it is possible to factor N in polynomial time.*

Proof. We claim that either $\gcd(N, x + \hat{x})$ or $\gcd(N, x - \hat{x})$ is equal to one of the prime factors of N . Since gcd computations can be carried out in polynomial time, this proves the lemma.

If $x^2 = \hat{x}^2 \bmod N$ then

$$0 = x^2 - \hat{x}^2 = (x - \hat{x}) \cdot (x + \hat{x}) \bmod N,$$

and so $N \mid (x - \hat{x})(x + \hat{x})$. Then $p \mid (x - \hat{x})(x + \hat{x})$ and so p divides one of these terms. Say $p \mid (x + \hat{x})$ (the proof proceeds similarly if $p \mid (x - \hat{x})$). If $q \mid (x + \hat{x})$ then $N \mid (x + \hat{x})$, but this cannot be the case since $x \not\equiv -\hat{x} \bmod N$. So $q \nmid x + \hat{x}$ and $\gcd(N, x + \hat{x}) = p$.

The following important result shows (roughly) that if N is hard to factor then squaring modulo N is one-way. Formally, define a function family $\Pi_{\text{squaring}} = (\text{Gen}, \text{Samp}, f)$ as follows:

- $\text{Gen}(1^k)$ computes $(N, p, q) \leftarrow \text{GenModulus}(1^k)$, and outputs parameters N . Let $D_N = \mathbb{Z}_N^*$ and $R_N = \text{QR}_N$.

- $\text{Samp}(N)$ chooses a uniform element of \mathbb{Z}_N^* . (This can be done easily by choosing random elements of \mathbb{Z}_N until one is found that is relatively prime to N .)
- $f_N(x)$ outputs $x^2 \bmod N$.

Theorem 2.2. *If factoring is hard relative to GenModulus , then Π_{squaring} is a one-way function family.*

Proof. Let A be a probabilistic polynomial-time algorithm, and define

$$\epsilon_A(k) \stackrel{\text{def}}{=} \Pr \left[N \leftarrow \text{Gen}(1^k); y \leftarrow \text{QR}_N; x \leftarrow A(N, y) : x^2 = y \bmod N \right].$$

Since setting $y := x^2 \bmod N$ for a uniformly random $x \in \mathbb{Z}_N^*$ is equivalent to choosing y uniformly from QR_N (because squaring is four-to-one), the above exactly represents A 's success probability in inverting the squaring function modulo N . Showing that $\epsilon_A(k)$ is negligible thus proves the theorem.

Consider the following probabilistic polynomial-time algorithm A_{fact} that attempts to factor moduli output by GenModulus :

Algorithm A_{fact} :

The algorithm is given a modulus N as input.

- Choose random $x \leftarrow \mathbb{Z}_N^*$ and compute $y := x^2 \bmod N$.
- Run $A(N, y)$ to obtain output \hat{x} .
- If $\hat{x}^2 = y \bmod N$ and $\hat{x} \neq \pm x \bmod N$, then factor N using Lemma 2.3.

By Lemma 2.3, we know that A_{fact} succeeds in factoring N exactly when $\hat{x} \neq \pm x \bmod N$ and $\hat{x}^2 = y \bmod N$. Since the modulus N given as input to A_{fact} is generated by $\text{GenModulus}(1^k)$, and y is a random quadratic residue modulo N (since x was chosen uniformly at random from \mathbb{Z}_N^*), the probability that A outputs \hat{x} satisfying $\hat{x}^2 = y \bmod N$ is exactly $\epsilon_A(k)$. Moreover, conditioned on the value of the quadratic residue y given to A , the value x used by A_{fact} is equally likely to be any of the four possible square roots of y . This means that, conditioned on A outputting *some* square root \hat{x} of y , the probability that $\hat{x} \neq \pm x \bmod N$ is exactly $1/2$. Putting this together, we have:

$$\begin{aligned} & \Pr[(N, p, q) \leftarrow \text{GenModulus}(1^k) : A_{\text{fact}} \text{ factors } N] \\ &= \Pr \left[\begin{array}{l} (N, p, q) \leftarrow \text{GenModulus}(1^k); x \leftarrow \mathbb{Z}_N^*; \\ y := x^2 \bmod N; \hat{x} \leftarrow A(N, y) \end{array} : \hat{x} \neq \pm x \bmod N \wedge \hat{x}^2 = y \bmod N \right] \\ &= \frac{1}{2} \cdot \Pr \left[\begin{array}{l} (N, p, q) \leftarrow \text{GenModulus}(1^k); x \leftarrow \mathbb{Z}_N^*; \\ y := x^2 \bmod N; \hat{x} \leftarrow A(N, y) \end{array} : \hat{x}^2 = y \bmod N \right] \\ &= \epsilon_A(k)/2. \end{aligned}$$

Since factoring is hard relative to GenModulus , we conclude that $\epsilon_A(k)$ must be negligible, completing the proof.

One approach to making Π_{squaring} a permutation family is to consider specific moduli N and restrict the domain of the function. For $N = pq$ a product of two distinct primes p and q , we say that N is a *Blum integer* if $p = q = 3 \pmod{4}$. It is a fact that if N is a Blum integer, then any quadratic residue modulo N has exactly one square root that is *also* a quadratic residue. Thus, the squaring function for a Blum integer N is a permutation over QR_N .

It is also known that computing square roots modulo N is “easy” (i.e., can be done in polynomial time) given the factorization of N . Combining this with the previous observation, we obtain a trapdoor permutation family based on factoring:

- $\text{Gen}(1^k)$ computes $(N, p, q) \leftarrow \text{GenModulus}(1^k)$, where GenModulus is such that $p = q = 3 \pmod{4}$. It then outputs parameters N and trapdoor (p, q) . Let $D_N = \text{QR}_N$.
- $\text{Samp}(N)$ chooses a uniform element of $y \in \text{QR}_N$. (This can be done easily by choosing a random element $r \in \mathbb{Z}_N^*$ and setting $y := r^2 \pmod{N}$.)
- $f_N(x)$ outputs $x^2 \pmod{N}$.
- $f_{(p,q)}^{-1}(y)$ computes the unique square root of y modulo N that is itself a quadratic residue.

Theorem 2.2 implies that the above is a trapdoor permutation family as long as factoring is hard relative to GenModulus . Notice, however, that (as described) it is not a *doubly enhanced* trapdoor permutation family: given the random coins r used by Samp , which we view⁵ as an element of \mathbb{Z}_N^* , it is trivial to compute a square root of the output value $y = r^2 \pmod{N}$. We will see below how this can be addressed.

Extending the above gives a construction of a *clawfree* trapdoor permutation family:

- $\text{Gen}(1^k)$ computes $(N, p, q) \leftarrow \text{GenModulus}(1^k)$, where $p = q = 3 \pmod{4}$, and chooses random $z \leftarrow \text{QR}_N$. It then outputs parameters (N, z) and trapdoor (p, q) . Let $D_N = \text{QR}_N$.
- $\text{Samp}(N)$ chooses a uniform element of QR_N as above.
- Given (N, z) , we define f_0 and f_1 as follows:

$$f_0(x) = x^2 \pmod{N} \text{ and } f_1(x) = z \cdot x^2 \pmod{N}.$$

- Given (N, z) and the trapdoor information (p, q) , the inverses of f_0 and f_1 can be computed as follows: To compute $f_0^{-1}(y)$, find the unique square root of y modulo N that is itself a quadratic residue. To compute $f_1^{-1}(y)$, find the unique square root of $y/z \pmod{N}$ that is itself a quadratic residue.

Theorem 2.3. *If factoring is hard relative to GenModulus , then the above constitutes a clawfree trapdoor permutation family.*

⁵ This is justified since it is easy to map a sufficiently long bit-string to an element of \mathbb{Z}_N^* such that a random bit-string yields an element of \mathbb{Z}_N^* whose distribution is statistically close to uniform, and the mapping is invertible in the sense required.

Proof. The only condition difficult to verify is that it is computationally infeasible to find a claw. We show that any “claw-finding” algorithm A can be used to compute square roots modulo N . Theorem 2.2 thus implies that finding a claw is computationally infeasible.

Fix any PPT algorithm A and define

$$\varepsilon_A(k) \stackrel{\text{def}}{=} \Pr[(N, z, p, q) \leftarrow \text{Gen}(1^k); (x_0, x_1) \leftarrow A(N, z) : x_0^2 = z \cdot x_1^2]. \quad (2.1)$$

Since this is exactly the probability with which A succeeds in finding a claw, we need to show that $\varepsilon_A(k)$ is negligible.

Construct a PPT algorithm A' computing modular square roots as follows:

Algorithm A' :

The algorithm is given a modulus N and an element $z \in \text{QR}_N$ as input.

- Run $A(N, z)$ to obtain output (x_0, x_1) .
- If $x_0^2 = z \cdot x_1^2 \bmod N$, then output $x_0/x_1 \bmod N$.

It is easy to see that if the input z given to A' is chosen uniformly from QR_N , then the input (N, z) given to A is distributed identically to the experiment of Equation (2.1). Thus, the probability that A outputs (x_0, x_1) with $x_0^2 = z \cdot x_1^2 \bmod N$ is exactly $\varepsilon_A(k)$. Furthermore, whenever this occurs A' outputs a square root of its input z . But if factoring is hard relative to GenModulus , we know from Theorem 2.2 that this can happen with only negligible probability.

As in the case of the trapdoor permutation family presented earlier, the construction just given is not doubly enhanced. We will fix this below. To do so, we need to introduce some brief facts about the *Jacobi function* $\mathcal{J}_N : \mathbb{Z}_N^* \rightarrow \{-1, +1\}$. (We introduce here all the facts that are needed for the construction that follows. For further information about the Jacobi function, consult the references at the end of the chapter.) An element $x \in \mathbb{Z}_N^*$ with $\mathcal{J}_N(x) = +1$ is said to have *Jacobi symbol* $+1$, and similarly if $\mathcal{J}_N(x) = -1$ then we say that x has *Jacobi symbol* -1 . The relevant facts are:

1. Exactly half the elements of \mathbb{Z}_N^* have Jacobi symbol $+1$, and half have Jacobi symbol -1 .
2. Given N, x , it is possible to compute $\mathcal{J}_N(x)$ in polynomial time *without knowledge of the factorization of N* .
3. For N a Blum integer, we have seen that every quadratic residue z has exactly one square root x that is also a quadratic residue. It is furthermore the case that z has exactly two square roots with Jacobi symbol $+1$, and these are given by $\pm x \bmod N$.

Let \mathcal{J}_N^{+1} denote the set of elements of \mathbb{Z}_N^* with Jacobi symbol $+1$. We now present the construction of a doubly enhanced clawfree trapdoor permutation:

- $\text{Gen}(1^k)$ computes $(N, p, q) \leftarrow \text{GenModulus}(1^k)$, where $p = q = 3 \bmod 4$, and chooses random $z \leftarrow \text{QR}_N$. It then outputs parameters (N, z) and trapdoor (p, q) . Let $D_N = \text{QR}_N$.
- $\text{Samp}(N)$ chooses a uniform element $y \in \text{QR}_N$ by choosing a random $r \in \mathcal{J}_N^{+1}$ and setting $y := r^2 \bmod N$. (The random $r \in \mathcal{J}_N^{+1}$ is chosen by taking random bit-strings r_1, \dots of the appropriate length, and letting r be the first of these that is in \mathbb{Z}_N^* and has Jacobi symbol $+1$.)
- Given (N, z) , we define f_0 and f_1 as follows:

$$f_0(x) = x^4 \bmod N \text{ and } f_1(x) = z^2 \cdot x^4 \bmod N.$$

- Given (N, z) and the trapdoor information (p, q) , the inverses of f_0 and f_1 can be computed as follows: To compute $f_0^{-1}(y)$, find the unique fourth root of y modulo N that is itself a quadratic residue. To compute $f_1^{-1}(y)$, find the unique fourth root of $y/z^2 \bmod N$ that is itself a quadratic residue.

Theorem 2.4. *If factoring is hard relative to GenModulus , then the above constitutes a doubly enhanced clawfree trapdoor permutation family.*

Proof (sketch). We first show that finding a claw implies the ability to compute square roots modulo N ; it follows from Theorem 2.2 that finding a claw is infeasible. Say we are given N and a quadratic residue $z \in \text{QR}_N$. Given a claw (x_0, x_1) such that $x_0^4 = z^2 \cdot x_1^4 \bmod N$, we have $(x_0^2/x_1^2)^2 = z^2 \bmod N$. Since the quadratic residue $z^2 \bmod N$ has a *unique* square root that is also a quadratic residue (namely, z itself), it must be the case that $x_0^2/x_1^2 = z \bmod N$ and so $x_0/x_1 \bmod N$ is a square root of z as desired.

A similar proof shows that it is hard to find a pre-image of y with respect to f_1 even when given the randomness used to generate y . Let A be a PPT algorithm inverting f_1 , and consider the following probabilistic polynomial-time algorithm A' for computing square roots:

Algorithm A' :

The algorithm is given a modulus N and $\hat{r} \in \text{QR}_N$ as inputs.

- Choose random $\hat{z} \in \mathbb{Z}_N^*$ and set $z := \hat{z}^2 \bmod N$.
- Choose random $b \in \{0, 1\}$ and set $r := (-1)^b \hat{r} \bmod N$.
- Compute $y := r^2 \bmod N$.
- Run $A(N, z, y, r)$ to obtain output x .
- Output $\hat{z} \cdot x$.

The input to A is distributed correctly, since r is uniformly distributed in \mathcal{J}_N^{+1} (this follows because $\mathcal{J}_N(\hat{r}) = +1$ since \hat{r} is a quadratic residue) and z is uniform in QR_N . Furthermore, we have

$$z^2 x^4 = y \bmod N \Rightarrow (\hat{z}^2 x^2)^2 = \hat{r}^2 \bmod N \Rightarrow \hat{z}^2 x^2 = \hat{r} \bmod N$$

(the final implication uses the fact that quadratic residues have a unique square root that is also a quadratic residue), and so $\hat{z}x$ is a square root of \hat{r} . Lemma 2.3 thus implies that A inverts with only negligible probability. A proof for the case of inverting f_0 follows analogously.

To conclude, we show algorithms $\text{Samp}_0, \text{Samp}_1$ as required⁶ by Definition 2.7. Samp_0 proceeds as follows: Given (N, z) , choose random $x \in \mathbb{Z}_N^*$ and compute $y := x^4 \bmod N$. Choose a random bit b and compute $r := (-1)^b \cdot x^2 \bmod N$. Output (x, y, r) . It is clear that (x, y, r) satisfy the functional requirement of Definition 2.7. Furthermore, y is uniform in QR_N and r is a random square root of y having Jacobi symbol $+1$. From this it follows that r is uniform in \mathcal{J}_N^{+1} .

Algorithm Samp_1 as required by Definition 2.7 can be defined analogously as follows: Given (N, z) , choose random $x \in \mathbb{Z}_N^*$ and compute $y := z^2 x^4 \bmod N$. Choose a random bit b and compute $r := (-1)^b \cdot z \cdot x^2 \bmod N$; output (x, y, r) . Once again, (x, y, r) satisfy the functional requirement of Definition 2.7. Also, y is uniform in QR_N and r is a random square root of y having Jacobi symbol $+1$. From this it follows that r is uniform in \mathcal{J}_N^{+1} .

2.2.2 The RSA Assumption

Another popular assumption related to factoring is the *RSA assumption*, named after R. Rivest, A. Shamir, and L. Adleman who proposed this assumption in 1978. The RSA assumption implies that factoring is hard, while the converse it not known; thus, the RSA assumption is potentially stronger than the assumption that factoring is hard. (In other words, it is possible that an efficient algorithm for the RSA problem might be developed even while the factoring problem remains infeasible.) Nevertheless, the RSA assumption has stood the test of time for over 30 years.

We begin with a little background. Let $N = pq$ be a product of two odd primes p and q . Then we have seen that \mathbb{Z}_N^* is a group with respect to multiplication modulo N . An easy computation shows that the number of elements in \mathbb{Z}_N^* is given by $\phi(N) \stackrel{\text{def}}{=} (p-1) \cdot (q-1)$; in other words, $\phi(N)$ is the order of the group \mathbb{Z}_N^* . From this it follows that if e is an integer that is relatively prime to $\phi(N)$, then the function $f_{N,e} : \mathbb{Z}_N^* \rightarrow \mathbb{Z}_N^*$ given by $f_{N,e}(x) = x^e \bmod N$ is in fact a *permutation*. In other words, for every $y \in \mathbb{Z}_N^*$ there exists a unique $x \in \mathbb{Z}_N^*$ satisfying $x^e = y \bmod N$. For x, y satisfying this relation, we write $x = y^{1/e} \bmod N$.

The RSA problem, roughly speaking, is to compute the e th root of y modulo N ; the RSA assumption is that computing e th roots modulo an integer N of unknown factorization is hard. Formally, let GenRSA be a probabilistic polynomial-time algorithm that, on input 1^k , outputs a modulus N that is the product of two k -bit primes (except possibly with negligible probability), along with an integer $e > 0$

⁶ We concentrate on showing how a uniform $r \in \mathcal{J}_N^{+1}$ can be sampled, since going from this to the random coins needed to generate r is easy (due to the fact that the Jacobi symbol is efficiently computable).

with $\gcd(e, \phi(N)) = 1$ and an integer $d > 0$ satisfying $ed = 1 \bmod \phi(N)$. (We will see below the role played by d . Note that such a d is guaranteed to exist since e is relatively prime to $\phi(N)$.) Then:

Definition 2.9. We say that **the RSA problem is hard relative to** GenRSA if for all PPT algorithms A , the following is negligible:

$$\Pr[(N, e, d) \leftarrow \text{GenRSA}(1^k); y \leftarrow \mathbb{Z}_N^*; x \leftarrow A(N, e, y) : x^e = y \bmod N].$$

The RSA assumption can be formalized as the assumption that there exists a GenRSA relative to which the RSA problem is hard. For certain applications, however, additional assumptions regarding the output of GenRSA are required.

The RSA assumption implies the existence of a one-way permutation family. Moreover, for any (N, e, d) output by GenRSA and any $y \in \mathbb{Z}_N^*$ we have

$$(y^d)^e = y^{de} = y^{1 \bmod \phi(N)} = y^1 = y \bmod N,$$

and so computing e th roots is equivalent to raising to the d th power. Thus, viewing d as a trapdoor we see that the RSA assumption implies the existence of trapdoor permutations. Moreover, since sampling uniformly from \mathbb{Z}_N^* is trivial, we actually obtain a doubly enhanced trapdoor permutation “for free”.

As with GenModulus in the previous section, here too we are agnostic regarding the exact way GenRSA is implemented. One way to implement GenRSA based on any algorithm GenModulus (not necessarily outputting Blum integers) is as follows:

1. On input 1^k , compute $(N, p, q) \leftarrow \text{GenModulus}(1^k)$. Then compute

$$\phi(N) := (p-1)(q-1).$$

2. Choose $e > 0$ such that $\gcd(e, \phi(N)) = 1$. (We leave the exact manner in which e is chosen unspecified, though in practice certain values of e may be preferred.) Compute $d := e^{-1} \bmod \phi(N)$ and output (N, e, d) .

Since $\phi(N)$ can be computed given the factorization of N , and $d = e^{-1} \bmod \phi(N)$ can be computed in polynomial time given $\phi(N)$, it is clear that hardness of the RSA problem relative to GenRSA as constructed above implies hardness of factoring relative to GenModulus. As mentioned earlier, the converse is not known to be true. On the other hand, the only techniques currently known for attacking the RSA problem rely on first factoring N . In addition, it is known that recovering the trapdoor d , given N and e , is as hard as factoring (nevertheless, it is not clear that recovery of d is *necessary* in order to compute e th roots). With respect to existing technology, then, the RSA and factoring problems may be viewed as “equally hard”.

As in the case of the factoring assumption, the RSA assumption may also be used to construct a (doubly enhanced) clawfree trapdoor permutation family in a very similar fashion:

- Gen(1^k) computes $(N, e, d) \leftarrow \text{GenRSA}(1^k)$ and chooses random $z \leftarrow \mathbb{Z}_N^*$. It then outputs parameters (N, e, z) and trapdoor d . Let $D_N = \mathbb{Z}_N^*$.

- $\text{Samp}(N)$ chooses a uniform element of \mathbb{Z}_N^* in the trivial way.
- Given (N, e, z) , define f_0 and f_1 as follows:

$$f_0(x) = x^e \bmod N \text{ and } f_1(x) = z \cdot x^e \bmod N.$$

- Given (N, e, z) and the trapdoor information d , the inverses of f_0 and f_1 can be computed as follows: To compute $f_0^{-1}(y)$, simply compute $y^d \bmod N$. To compute $f_1^{-1}(y)$, simply compute $(y/z)^d \bmod N$.

The proof that the above is a clawfree trapdoor permutation follows the proof of Theorem 2.3, and is omitted. Observe that this construction is also doubly enhanced (once again, this comes “for free” due to the triviality of sampling from \mathbb{Z}_N^*).

In Chapter 4, we will introduce the (more recent) *strong* RSA assumption that offers additional flexibility as compared to the RSA assumption described here.

2.2.3 The Discrete Logarithm Assumption

An assumption of a different flavor is obtained by considering the *discrete logarithm problem*, which may be defined in any finite cyclic group \mathbb{G} . For the most part, we will consider only groups \mathbb{G} of prime order q (though this is not required in any sense); such groups have the feature that every element in \mathbb{G} other than the identity is a generator, and have several other advantages as well. Letting g be a generator of the group, and $h \in \mathbb{G}$ be arbitrary, define the discrete logarithm of h with respect to g (denoted $\log_g h$) as the smallest non-negative integer x such that $g^x = h$. (Note that we always have $\log_g h < q$.) The discrete logarithm problem is to compute x given g and a random group element h . We remark that it is easy to sample a random element $h \in \mathbb{G}$ by choosing a uniform integer $y \in \{0, \dots, q-1\}$ and setting $h := g^y$.

For *certain* classes of groups, the discrete logarithm problem is believed to be hard. The problem is certainly not hard in *all* cyclic groups, and hardness depends to a great extent on the way elements of the group are represented. (This must be so, since all cyclic groups of the same order q are isomorphic yet the discrete logarithm problem is easy in the additive group \mathbb{Z}_q .)

To formally state the discrete logarithm assumption, we must consider an infinite sequence of groups as defined by an appropriate *group-generation algorithm*. Let \mathcal{G} be a polynomial-time algorithm that, on input 1^k , outputs a (description of a) cyclic group \mathbb{G} , its order q (with q a k -bit integer), and a generator $g \in \mathbb{G}$. (Unless stated otherwise, we will also assume that q is prime except with negligible probability.) We also require that membership in \mathbb{G} can be tested efficiently, and that the group operation in \mathbb{G} can be computed efficiently (namely, in time polynomial in k). This implies that exponentiation in \mathbb{G} can be performed efficiently as well.

Definition 2.10. The **discrete logarithm problem is hard relative to \mathcal{G}** if for all PPT algorithms A , the following is negligible:

$$\Pr[(\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^k); h \leftarrow \mathbb{G}; x \leftarrow A(1^k, \mathbb{G}, q, g, h) : g^x = h].$$

The discrete logarithm assumption is that there exists a \mathcal{G} relative to which the discrete logarithm problem is hard.

We often abuse terminology and say that the discrete logarithm problem is hard for \mathbb{G} when \mathbb{G} is a group that is output by \mathcal{G} .

It is immediate that the discrete logarithm assumption implies the existence of a one-way function family; take $f_{\mathbb{G},q,g}(x)$ that outputs g^x . The functions in this family are one-to-one if we take the domain of $f_{\mathbb{G},q,g}(x)$ to be \mathbb{Z}_q . For certain groups \mathbb{G} (for which the discrete logarithm problem is assumed to be hard), we can in fact obtain a one-way *permutation* family. One example, commonly used in cryptography, is given by groups of the form \mathbb{Z}_p^* for p prime. (The order of this group is $q = p - 1$, and is not prime. The fact that groups of this form are cyclic is not obvious, but can be proved using basic field theory.) In this case, the “natural” mapping $f_{\mathbb{G},q,g} : \mathbb{Z}_{p-1} \rightarrow \mathbb{Z}_p^*$ is the one given above, where $f_{\mathbb{G},q,g}(x) = g^x \bmod p$. But by simply “shifting” the domain we get the function $f'_{\mathbb{G},q,g} : \mathbb{Z}_p^* \rightarrow \mathbb{Z}_p^*$ given by

$$f'_{\mathbb{G},q,g}(x) = g^{x-1} \bmod p.$$

We do not discuss other examples of groups for which the discrete logarithm problem is assumed to be hard. For the purposes of this book, abstracting the choice of group will be more convenient and suffices for our purposes.

The reader may be familiar with other assumptions related to the discrete logarithm assumption, most prominently the *computational* and *decisional* Diffie-Hellman assumptions. Interestingly, although these assumptions are extremely useful for the construction of efficient public-key encryption schemes, they have thus far had little application to the construction of efficient signature schemes. An exception is in the context of bilinear maps, discussed in Chapter 5, and we defer any discussion to there.

2.3 Hash Functions

Hash functions play a central role in the construction of secure signature schemes. Constructions of hash functions based on various assumptions are known, and we also have extremely efficient constructions (not based on any particular hard cryptographic problem) that one can reasonably assume to be secure. We will return to this point after defining the basic types of hash functions we will consider.

2.3.1 Definitions

A hash function is simply a function that *compresses* its input. Hash functions are used throughout computer science, but we will be interested in two types of *cryptographic* hash functions. Our hash functions will be *keyed*: that is, a function H is

defined (this is sometimes also called a hash *family*); a key s is sampled uniformly at random and published; and we then look at the security of the keyed function H_s . A hash function H is said to be *collision-resistant* if, roughly speaking, it is hard to find distinct x, x' that “collide”, that is, for which $H_s(x) = H_s(x')$. A hash function is called *universal one-way* if, informally, it is hard to find a collision for a *pre-specified* input x , chosen independently of the key s ;

Definition 2.11. A **hash function** is a pair of probabilistic polynomial-time algorithms (Gen, H) such that:

- Gen is a probabilistic algorithm that on input 1^k outputs a key s . (We assume that 1^k is implicit in s .)
- There exists a polynomial ℓ such that H takes as input a key s and $x \in \{0, 1\}^*$, and outputs a string $H_s(x) \in \{0, 1\}^{\ell(k)}$. (Note that the running time of H is allowed to depend on $|x|$.)

If H_s is defined only for inputs $x \in \{0, 1\}^{\ell'(k)}$, where $\ell'(k) > \ell(k)$ for all k , then we say that (Gen, H) is a **fixed-length hash function for inputs of length ℓ'** .

We now define the security properties stated informally earlier.

Definition 2.12. Hash function (Gen, H) is **collision-resistant** if the following is negligible for all PPT algorithms A :

$$\Pr \left[s \leftarrow \text{Gen}(1^k); (x, x') \leftarrow A(1^k, s) : x \neq x' \wedge H_s(x) = H_s(x') \right].$$

Definition 2.13. (Gen, H) is **universal one-way** if the following is negligible for all stateful PPT algorithms A :

$$\Pr \left[x \leftarrow A(1^k); s \leftarrow \text{Gen}(1^k); x' \leftarrow A(1^k, s) : x \neq x' \wedge H_s(x) = H_s(x') \right].$$

This is sometimes also called **second pre-image resistance** in the literature.

It is easy to see that collision-resistance implies universal one-wayness.

2.3.2 The Merkle-Damgård Transform

The Merkle-Damgård (MD) transform can be used to convert a fixed-length hash function (Gen, H') to a hash function (Gen, H) taking inputs of arbitrary length, while preserving collision-resistance. Besides being useful in its own right, the existence of the MD transform means that we can focus our attention on designing collision-resistant *compression functions* operating on short, fixed-length inputs, and then automatically convert such compression functions into full-fledged hash functions. The MD transform is also used extensively in practical constructions of hash functions.

Construction 2.1: The Merkle-Damgård transform

Let (Gen, H') be a fixed-length hash function for inputs of length $2\ell(k)$. (This assumption is for simplicity only.) Construct a hash function (Gen, H) as follows:

- The key-generation algorithm Gen is unchanged.
 - H_s is defined for inputs of length at most $2^\ell - 1$. To compute $H_s(x)$ do:
 1. Set $L := |x|$ and $B := \lceil \frac{L}{\ell} \rceil$ (i.e., B is the number of blocks in x). Pad the input x with zeroes until its length is an integer multiple of ℓ , and parse the result as the sequence of ℓ -bit blocks x_1, \dots, x_B . Set $x_{B+1} := L$, where L is encoded using exactly ℓ bits.
 2. Set $z_0 := 0^\ell$.
 3. For $i = 1, \dots, B + 1$, compute $z_i := H'_s(z_{i-1} \| x_i)$.
 4. Output z_{B+1} .
-

Theorem 2.5. *If (Gen, H') is collision-resistant, then so is (Gen, H) .*

Proof (sketch). The proof follows easily from the observation that any collision in H_s yields a collision in H'_s . Let x and x' be two different strings, of lengths L and L' respectively, with $H_s(x) = H_s(x')$. Let x_1, \dots, x_B denote the padded version of x , and let $x'_1, \dots, x'_{B'}$ be the result of padding x' . Recall further that $x_{B+1} = L$ and $x'_{B'+1} = L'$. Let z_i (resp. z'_i) be as in Construction 2.1. There are two cases to consider:

1. *Case 1: $L \neq L'$.* We have

$$H_s(x) = H'_s(z_B \| L) = H'_s(z'_{B'} \| L') = H_s(x').$$

Since $z_B \| L \neq z'_{B'} \| L'$, this is a collision in H'_s .

2. *Case 2: $L = L'$.* In this case, note that $B = B'$ and $x_{B+1} = x'_{B+1}$. Since $x \neq x'$, there must exist an index i with $1 \leq i \leq B$ such that $x_i \neq x'_i$. Let $i^* \leq B + 1$ be the *largest* index for which $z_{i^*-1} \| x_{i^*} \neq z'_{i^*-1} \| x'_{i^*}$. If $i^* = B + 1$ then $z_B \| x_{B+1}$ and $z'_B \| x'_{B+1}$ are a collision in H'_s exactly as in the previous case. If $i^* \leq B$, then maximality of i^* implies $z_{i^*} = z'_{i^*}$, in which case $z_{i^*-1} \| x_{i^*}$ and $z'_{i^*-1} \| x'_{i^*}$ are a collision in H'_s .

The MD transform is only guaranteed to preserve collision-resistance; it is *not* guaranteed to preserve universal one-wayness. (A variant of the MD transform where an independent key is chosen for each iteration *does* preserve universal one-wayness, although this approach yields a hash function with a very long key. See further discussion at the end of Section 2.3.4.)

2.3.3 Constructing Collision-Resistant Hash Functions

In this section we describe constructions of (fixed-length) collision-resistant hash functions based on the number-theoretic assumptions introduced earlier in this chapter. We conclude with a brief discussion of hash functions used in practice.

Collision-resistant hash functions from clawfree permutations. We begin with a construction of collision-resistant hashing from any clawfree permutation family. Although not very practical, the construction provide a good illustration of how “strong” cryptographic primitives can be constructed from “weaker” building blocks. It also shows that the factoring and RSA assumptions imply the existence of collision-resistant hash functions.

Construction 2.2: Collision resistance from clawfree permutations

Let $(\text{Gen}, \text{Samp}, f_0, f_1)$ be a clawfree permutation family, and assume that if I is output by $\text{Gen}(1^k)$ then elements of D_I can be described using ℓ bits. Define the fixed-length hash function (Gen_H, H) as follows:

- $\text{Gen}_H(1^k)$ computes $I \leftarrow \text{Gen}(1^k)$, chooses $r \leftarrow D_I$, and outputs the key $s = (I, r)$. In what follows, let $\ell = \ell(k)$ and write f_0, f_1 in place of $f_{I,0}, f_{I,1}$.
 - $H_s(x)$, where $x \in \{0, 1\}^{2\ell(k)}$, parses s as (I, r) and parses $x = x_1 \cdots x_{2\ell}$ where each x_i is a single bit. It then outputs $f_{x_{2\ell}}(f_{x_{2\ell-1}}(\cdots(f_{x_1}(r))\cdots))$.
-

Theorem 2.6. *If $(\text{Gen}, \text{Samp}, f_0, f_1)$ is a clawfree permutation family, then hash function (Gen_H, H) from Construction 2.2 is collision-resistant.*

Proof (sketch). The proof follows easily from the observation that any collision in H_s yields a claw. For $x \in \{0, 1\}^{2\ell}$ and $1 \leq i \leq 2\ell$ define

$$H_s^i = f_{x_i}(f_{x_{i-1}}(\cdots(f_{x_1}(r))\cdots));$$

note that $H_s(x) = H_s^{2\ell}(x)$. Let $x = x_1 \cdots x_{2\ell}$ and $x' = x'_1 \cdots x'_{2\ell}$ be two different strings with $H_s(x) = H_s(x')$, and let i denote the largest index such that $x_i \neq x'_i$ (so $x_j = x'_j$ for all $j > i$). Since f_0, f_1 are permutations,

$$H_s^{2\ell}(x) = H_s^{2\ell}(x') \Rightarrow H_s^i(x) = H_s^i(x') \Rightarrow f_{x_i}(H_s^{i-1}(x)) = f_{x'_i}(H_s^{i-1}(x'));$$

but then $H_s^{i-1}(x)$ and $H_s^{i-1}(x')$ are a claw.

Collision-resistant hash functions from the discrete logarithm assumption. In certain groups \mathbb{G} (namely, where there is an efficiently computable bijection from \mathbb{G} to $\mathbb{Z}_{|\mathbb{G}|}$), the discrete logarithm assumption implies a clawfree permutation family

and thus a construction of a collision-resistant hash function as discussed above. We give a more direct and more efficient construction here, that additionally has the advantage of not working with arbitrary \mathbb{G} .

Let \mathcal{G} be a group-generation algorithm, and assume that if $(\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^k)$ then elements of \mathbb{G} can be described using $k + O(1)$ bits. (This assumption is for simplicity only, and a generalization of the following construction works if this assumption does not hold.) Define a fixed-length hash function (Gen, H) as follows:

- $\text{Gen}(1^k)$ computes $(\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^k)$ and chooses random $h \in \mathbb{G}$. It outputs the key $s = (\mathbb{G}, q, g, h)$.
- Let $s = (\mathbb{G}, q, g, h)$, and define $H_s : \mathbb{Z}_q \times \mathbb{Z}_q \rightarrow \mathbb{G}$ as

$$H_s(x, y) = g^x h^y.$$

If we want to view H_s as mapping bit-strings to bit-strings, note that H_s can handle inputs of length $2 \cdot (k - 1)$ (since any $(k - 1)$ -bit integer can be viewed naturally as an element of \mathbb{Z}_q , using the fact that q is a k -bit integer), and the output of H_s can be encoded using $k + O(1)$ bits. Thus, for k large enough we have compression.

Theorem 2.7. *If the discrete logarithm problem is hard relative to \mathcal{G} , the above construction is collision resistant.*

Proof. Let A be a PPT collision-finding algorithm, and let

$$\epsilon_A(k) \stackrel{\text{def}}{=} \Pr \left[\begin{array}{l} (\mathbb{G}, q, g, h) \leftarrow \text{Gen}(1^k); \\ (x, y, x', y') \leftarrow A(\mathbb{G}, q, g, h) : (x, y) \neq (x', y') \wedge H_s(x, y) = H_s(x', y') \end{array} \right]$$

(where $s = (\mathbb{G}, q, g, h)$) denote the probability with which A finds a collision. Construct the following PPT algorithm B solving the discrete logarithm problem relative to \mathcal{G} :

Algorithm B :

The algorithm is given (\mathbb{G}, q, g, h) as input.

Its goal is to compute $\log_g h$.

- Run $A(\mathbb{G}, q, g, h)$ to obtain $x, y, x', y' \in \mathbb{Z}_q$.
- If $y \neq y'$, output $(x - x') \cdot (y' - y)^{-1} \bmod q$. (Any $(y' - y) \neq 0$ has an inverse modulo q since q is prime.)

First note that A 's input when run as a sub-routine by B is distributed identically to the keys output by Gen (this is true because B 's input has h chosen uniformly from \mathbb{G}). Thus, A returns a collision with probability exactly $\epsilon_A(k)$. We complete the proof of the theorem by showing that B outputs the correct answer $\log_g h$ whenever A finds a collision. To see this, note that

$$g^x h^y = g^{x'} h^{y'} \Rightarrow g^{x-x'} = h^{y'-y},$$

and if the above holds and furthermore $(x, y) \neq (x', y')$ then it must be the case that $y' - y \neq 0$. So $g^{(x-x')/(y'-y)} = h$, and this is exactly what is output by B .

Dedicated collision-resistant hash functions. We have seen that collision-resistant hash functions can be constructed based on a variety of number-theoretic assumptions. Yet these constructions are rather inefficient. In practice, dedicated constructions of (conjectured) collision-resistant hash functions are used that are orders of magnitude faster. These functions are generally *unkeyed* and have fixed length outputs. For these reasons, they cannot be said to satisfy asymptotic notions of security; nevertheless, appropriate *concrete* notions of security can be defined. Notable examples of hash functions in widespread use as of the time of this writing include SHA-1 (which hashes arbitrary-length inputs to 160-bit outputs) and SHA-256 (which hashes arbitrary-length inputs to 256-bit outputs).

2.3.4 Constructing Universal One-Way Hash Functions

Collision-resistance is a strong requirement. From a theoretical point of view, we currently know how to construct collision-resistant hash functions *only* from concrete, number-theoretic assumptions; constructions based on generic assumptions such as trapdoor permutations are not known. (Moreover, there is evidence that such constructions are impossible.) From a practical point of view, recent years have seen tremendous progress developing methods to attack hash functions. A prime example is the hash function MD5. For well over a decade MD5 was considered to be collision-resistant for all practical purposes. In 2005, however, Chinese cryptanalysts discovered a new technique for finding collisions in MD5. The attacks only got better, to the point where collisions in MD5 can now be found in minutes, and *structured* collisions in MD5 (i.e., colliding inputs x, x' that each satisfy certain formatting requirements) can now easily be found as well.

It is thus useful, when possible, to rely on the weaker assumption of universal one-wayness. Doing so potentially allows constructions based on weaker assumptions, and only makes it harder for an adversary to attack a deployed scheme. (In particular, MD5 is still considered to be universal one-way for the time being.) As an example of the former, we show here a construction of a (fixed-length) universal one-way hash function from any one-way permutation. (The construction can be easily modified to work with families of one-way permutations over bit-strings as well.) It is known that universal one-way hash functions can be constructed based on the (minimal) assumption of one-way *functions*; this construction is quite complex, unfortunately, and is beyond the scope of this book.

The construction of a universal one-way hash function from one-way permutations will use a particular pairwise-independent function family we now introduce. Let \mathbb{F}_{2^k} denote the field with 2^k elements, and note that there is a natural correspondence between elements in \mathbb{F}_{2^k} and k -bit strings. Define the function family

$$\mathcal{H}_k \stackrel{\text{def}}{=} \{h_{a,b} : \mathbb{F}_{2^k} \rightarrow \{0, 1\}^{k-1} \mid b \in \mathbb{F}_{2^k}, a \in \mathbb{F}_{2^k} \setminus \{0\}\},$$

as

$$h_{a,b}(x) = \text{chop}(ax + b),$$

where chop simply removes the final bit of its input. We will use the following properties of \mathcal{H}_k :

Lemma 2.4. *For every $b \in \mathbb{F}_{2^k}$, $a \in \mathbb{F}_{2^k} \setminus \{0\}$, the function $h_{a,b}$ is two-to-one.*

Proof. Fix a, b and any $z \in \{0, 1\}^{k-1}$. Let $z_0 = z\|0$ and $z_1 = z\|1$. The equation $ax + b = z_0$ has the unique solution $x = a^{-1} \cdot (z_0 - b)$ in \mathbb{F}_{2^k} (using $a \neq 0$), and similarly for the equation $ax + b = z_1$.

Lemma 2.5. *Fix arbitrary $y \in \mathbb{F}_{2^k}$, and consider choosing a, b in the following way:*

1. *Choose uniform $y' \in \mathbb{F}_{2^k} \setminus \{y\}$, uniform $z \in \{0, 1\}^{k-1}$, and uniform $c \in \{0, 1\}$. Set $z' = z\|c$ and $\bar{z}' = z\|\bar{c}$, and view z', \bar{z}' as elements of \mathbb{F}_{2^k} .*
2. *Solve for a, b in the following system of equations:*

$$ay + b = z' \tag{2.2}$$

$$ay' + b = \bar{z}'. \tag{2.3}$$

Then the distribution induced on (a, b) is uniform over $(\mathbb{F}_{2^k} \setminus \{0\}) \times \mathbb{F}_{2^k}$.

Proof. Note first that y', z, c determine a, b uniquely, and that $a \neq 0$ always. Now fix $a \in \mathbb{F}_{2^k} \setminus \{0\}$ and $b \in \mathbb{F}_{2^k}$ and let us see how many choices of y', z, c result in this (a, b) being chosen. Since a, y, b are now fixed, there is a unique choice of z, c satisfying Equation (2.2). Given this, $y' = (\bar{z}' - b) \cdot a^{-1} \neq y$ is the unique value satisfying Equation (2.3). We thus see that each pair (a, b) is selected with probability

$$\frac{1}{|\mathbb{F}_{2^k} \setminus \{0\}| \times |\{0, 1\}^{k-1}| \times |\{0, 1\}|} = \frac{1}{|\mathbb{F}_{2^k} \setminus \{0\}| \times |\mathbb{F}_{2^k}|},$$

and so the distribution of (a, b) is uniform over the indicated sets.

Let $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ be a length-preserving bijection. Define (Gen, H) as follows:

- $\text{Gen}(1^k)$ chooses uniform $a \in \mathbb{F}_{2^k} \setminus \{0\}$ and $b \in \mathbb{F}_{2^k}$, and outputs the key (a, b) .
- $H_{a,b}(x)$ outputs $h_{a,b}(f(x))$.

Note that $H_{a,b}$ is two-to-one for every a, b ; this follows from Lemma 2.4 and the fact that f is one-to-one.

Theorem 2.8. *If f is a one-way permutation, then the above hash function is universal one-way.*

Proof. Let A be a PPT collision-finding algorithm in the sense of Definition 2.13. Define

$$\varepsilon_A(k) \stackrel{\text{def}}{=} \Pr \left[x \leftarrow A(1^k); (a, b) \leftarrow \text{Gen}(1^k); x' \leftarrow A(a, b) : x \neq x' \wedge H_{a,b}(x) = H_{a,b}(x') \right].$$

Construct the following PPT algorithm B inverting f :

Algorithm B :

The algorithm is given $y' \in \{0, 1\}^k$ as input.

Its goal is to compute $x' \in \{0, 1\}^k$ with $f(x') = y'$.

- Run $A(1^k)$ to obtain x ; set $y = f(x)$. If $y' = y$ then output x and stop.
- Otherwise choose random z, c and then compute a, b as in Lemma 2.5. Run $A(a, b)$ to obtain x' . Output x' .

If $y' = y$ then B clearly outputs an inverse of y . Conditioned on the event that this does not occur, y' is uniform in $\mathbb{F}_{2^k} \setminus \{y\}$. (This follows because B 's input y' is computed as $y' = f(x')$ for uniform x' , and f is a permutation.) It follows from Lemma 2.5 that a, b are distributed identically to the output of Gen , and so A finds a collision with probability exactly $\varepsilon_A(k)$ in this case.

By construction of B ,

$$H_{a,b}(x) = \text{chop}(a \cdot f(x) + b) = \text{chop}(a \cdot y + b) = \text{chop}(z \| c) = z.$$

Since $H_{a,b}$ is two-to-one and

$$H_{a,b}(f^{-1}(y')) = \text{chop}(a \cdot y' + b) = \text{chop}(z \| \bar{c}) = z$$

collides with x , it follows that $f^{-1}(y')$ is the *only* input that collides with x , and so B outputs the correct result $f^{-1}(y')$ whenever A finds a collision. The theorem follows.

Improving the compression. The construction above only compresses its input by a single bit. Nevertheless, this suffices for constructing a universal one-way hash function mapping $p(k)$ -bit inputs to $(k - 1)$ -bit outputs (for any desired polynomial p) using a variant of the Merkle-Damgård transform, with the difference being that *independent* keys must be used in each iteration. This gives a universal one-way hash function where the key size grows linearly in the input length and, in fact, is longer than the input. (This is no problem as far as the definition of universal one-wayness is concerned, but causes difficulty in some applications as will become clear in Chapter 3.) Transformations that improve the compression using smaller keys are also known; these can be used to construct universal one-way hash functions handling inputs of unbounded length. The details are beyond the scope of this book. Furthermore, as noted earlier, universal one-way hash functions can be constructed from one-way *functions*. For completeness we record the following:

Theorem 2.9. *Assuming the existence of one-way functions, there exist universal one-way hash functions (for arbitrary-length inputs).*

2.4 Applications of Hash Functions to Signature Schemes

We wrap up this chapter by showing how to use cryptographic hash functions to improve the parameters of digital signature schemes. We first show how to increase the message length of a signature scheme: specifically, we show how to convert a signature scheme capable of signing k -bit messages into one that can sign messages of arbitrary length. (We have already shown such a construction in Section 1.9, but the one given here is much more efficient.) This technique is used extensively in practice to enable signing large files. We then show how to decrease the size of public keys (at the expense of increasing the signature length), constructing in particular a one-time signature scheme that can sign messages twice as long as its own public key. This will form a crucial ingredient in our construction (in the following chapter) of an existentially unforgeable signature scheme from any one-way function.

2.4.1 Increasing the Message Length

Given a signature scheme for “short” messages, a natural way of handling longer messages is to *hash* all messages before signing them. We show how this can be implemented using both collision-resistant and universal one-way hash functions.

Using collision-resistant hash functions. When using collision-resistant hash functions, the above idea is simple to implement.

Construction 2.3: Increasing the message length using collision resistance

Let $\Pi = (\text{Gen}, \text{Sign}, \text{Vrfy})$ be a signature scheme for k -bit messages, and let (Gen_H, H) be a hash function mapping $p(k)$ -bit inputs to k -bit outputs. Construct signature scheme $\Pi' = (\text{Gen}', \text{Sign}', \text{Vrfy}')$ for $p(k)$ -bit messages as follows:

Key generation: $\text{Gen}'(1^k)$ computes $(pk, sk) \leftarrow \text{Gen}(1^k)$ and $s \leftarrow \text{Gen}_H(1^k)$. The public key is (pk, s) and the secret key is (sk, s) .

Signature generation: Algorithm $\text{Sign}'_{sk,s}(m)$ outputs $\text{Sign}_{sk}(H_s(m))$.

Signature verification: Algorithm $\text{Vrfy}'_{pk,s}(m, \sigma)$ outputs $\text{Vrfy}_{pk}(H_s(m), \sigma)$.

Theorem 2.10. *If Π is existentially unforgeable (resp., strongly unforgeable) under an adaptive chosen-message attack and (Gen_H, H) is collision-resistant, then Π' is existentially unforgeable (resp., strongly unforgeable) under an adaptive chosen-message attack.*

Proof (sketch). The proof is quite straightforward, and so we merely provide a sketch for the case of existential unforgeability. Let m_1, \dots, m_ℓ denote the messages

submitted by an adversary A to the signing oracle $\text{Sign}'_{sk,s}(\cdot)$, and let (m, σ) denote a purported forgery output by A . There are two possibilities: either $H_s(m) = H_s(m_i)$ for some $i \in \{1, \dots, \ell\}$ or not. If so, then A has found a collision in H_s (something that, by assumption on (Gen_H, H) , occurs with only negligible probability). If not, then $H_s(m)$ is a k -bit string different from all k -bit strings $\{H_s(m_1), \dots, H_s(m_\ell)\}$ that were signed using scheme Π . But then A has in fact output a valid forgery for Π (something that, by assumption on Π , occurs with only negligible probability).

Using universal one-way hash functions. In order to use universal one-way hash functions, we have to work a little harder. Note first that Theorem 2.10 is no longer guaranteed to hold if (Gen_H, H) is only universal one-way: In that case an adversary who observes s — which is included in the public key — might be able to find two different messages m, m' hashing to the same value, and then forge a signature on m' after requesting a signature on m . However, we *can* claim the following weaker version of Theorem 2.10:

Theorem 2.11. *If Π is existentially unforgeable (resp., strongly unforgeable) under a known-message attack and (Gen_H, H) is universal one-way, then Π' (as in Construction 2.3) is existentially unforgeable (resp., strongly unforgeable) under a known-message attack.*

Proof (sketch). The proof uses exactly the same ideas as in the proof of Theorem 2.10, the key difference being that in a known-message attack on Π' the adversary must “commit” to its messages m_1, \dots, m_ℓ *before* it sees the public key (and so, in particular, before it sees the key s used for the hash function). Thus, if the adversary were able to output a forgery (m, σ) with $H_s(m) = H_s(m_i)$ for some i , this would violate the assumed universal one-wayness of (Gen_H, H) .

The above already suffices to increase the message length for signature schemes secure against an adaptive chosen-message attack: given an existentially unforgeable signature scheme Π (which is in particular also secure against known-message attacks) for k -bit messages, apply the above theorem to obtain scheme Π' for arbitrary-length messages that is secure against *known* message attacks (and hence also against random-message attacks); then apply either of Theorems 1.1 or 1.2. As we shall see, a direct construction with better efficiency is possible.

The problem with Construction 2.3 (when a universal one-way hash function is used in place of a collision-resistant hash function) is that the adversary may select messages to be signed in a manner that depends on the hash key s included as part of the public key. We would like to prevent this, and “force” the adversary to choose the messages submitted to the signing oracle independently of the hash key. We can accomplish this by choosing the hash key “on the fly” as part of signature generation. Specifically, consider the following construction starting with an existentially unforgeable signature scheme $\Pi = (\text{Gen}, \text{Sign}, \text{Vrfy})$ and universal one-way hash function (Gen_H, H) :

- Key generation is unchanged; i.e., compute $(pk, sk) \leftarrow \text{Gen}(1^k)$.

- To sign a message m using secret key sk , compute $s \leftarrow \text{Gen}_H(1^k)$ and output the signature

$$(s, \text{Sign}_{sk}(s|H_s(m))).$$

We stress that a *fresh* key s is computed for every message signed.

- To verify the signature (s, σ) on a message m with respect to a public key pk , output 1 iff $\text{Vrfy}_{pk}(s|H_s(m), \sigma) \stackrel{?}{=} 1$.

The above construction is existentially unforgeable. To see this, let m_1, \dots, m_ℓ denote the messages submitted by the adversary A to its signing oracle, and let $(s_1, \sigma_1), \dots, (s_\ell, \sigma_\ell)$ be the signatures returned. Say A outputs forgery $(m, (s, \sigma))$ with $m \notin \{m_1, \dots, m_\ell\}$. Arguing as in the proof of Theorem 2.10, if $(s, H_s(m)) \neq (s_i, H_{s_i}(m_i))$ for all i then A has, in fact, generated a forgery in the original scheme Π (something that is assumed to occur with only negligible probability). On the other hand, if $(s, H_s(m)) = (s_i, H_{s_i}(m_i))$ but $m \neq m_i$ (making the simplifying, but inessential, assumption that the $\{s_i\}$ are distinct), then A has violated the assumed universal one-wayness of (Gen_H, H) . (The key point being that the adversary chose m_i *before* it knew the hash key s_i .)

The main problem with this transformation is that the hash key *itself* is signed (along with the hashed message) by the underlying scheme, yet many theoretical constructions of universal one-way hash functions use rather long keys. In particular, when the hash key is longer than the input length of the hash function (as was the case for the construction described at the end of Section 2.3.4) the transformation is of no use. Instead, Construction 2.4 — which can be viewed as following the same paradigm used in Construction 1.2 — can be utilized.

Construction 2.4: Increasing the message length using universal one-wayness

Let $\Pi = (\text{Gen}, \text{Sign}, \text{Vrfy})$ be a signature scheme for $2k$ -bit messages, (Gen_H, H) be a hash function mapping $p(k)$ -bit inputs to k -bit outputs and having keys of length $h(k)$, and (Gen'_H, H') be a hash function mapping $h(k)$ -bit inputs to k -bit outputs. Construct signature scheme $\Pi' = (\text{Gen}', \text{Sign}', \text{Vrfy}')$ for $p(k)$ -bit messages as follows:

Key generation: $\text{Gen}'(1^k)$ computes $(pk, sk) \leftarrow \text{Gen}(1^k)$ and $s' \leftarrow \text{Gen}'_H(1^k)$. The public key is (pk, s') and the secret key is (sk, s') .

Signature generation: $\text{Sign}'_{sk, s'}(m)$ computes $s \leftarrow \text{Gen}_H(1^k)$ and outputs

$$(s, \text{Sign}_{sk}(H'_{s'}(s) | H_s(m))).$$

Once again, we stress that a fresh key s is chosen for each message signed.

Signature verification: $\text{Vrfy}'_{pk, s'}(m, (s, \sigma))$ outputs $\text{Vrfy}_{pk}(H'_{s'}(s) | H_s(m), \sigma)$.

The reader is referred to [11] for a proof of the following:

Theorem 2.12. *If Π is existentially unforgeable (resp., strongly unforgeable) under an adaptive chosen-message attack and (Gen_H, H) and (Gen'_H, H') are both universal one-way, then Π' is existentially unforgeable (resp., strongly unforgeable) under an adaptive chosen-message attack.*

2.4.2 Reducing the Public-Key Length

As our next application of hash functions to signature schemes, we consider ways of shortening the public key. While these techniques are generally useful — in particular, they show that schemes with optimal public-key size are possible — our goal here is to use these techniques to construct a (one-time) signature scheme capable of signing messages *twice as long as* its own public key; this will be used when we construct existentially unforgeable signature schemes based on general assumptions in the next chapter.

The obvious way to decrease the public-key size is to simply hash the original public key. Formally, let $\Pi = (\text{Gen}, \text{Sign}, \text{Vrfy})$ be an existentially unforgeable signature scheme having $q(k)$ -bit public keys, and let (Gen_H, H) be a hash function mapping $q(k)$ -bit inputs to k -bit outputs. Then the following scheme $\Pi' = (\text{Gen}', \text{Sign}', \text{Vrfy}')$ has public keys of length k :

- $\text{Gen}'(1^k)$ computes $(pk, sk) \leftarrow \text{Gen}(1^k)$ and $s \leftarrow \text{Gen}_H(1^k)$, and sets $pk' := H_s(pk)$. The public key is (s, pk') and the secret key is (pk, sk) .
- $\text{Sign}'_{pk, sk}(m)$ outputs $(pk, \text{Sign}_{sk}(m))$.
- $\text{Vrfy}'_{s, pk'}(m, (pk, \sigma))$ outputs 1 iff (1) $H_s(pk) \stackrel{?}{=} pk'$ and (2) $\text{Vrfy}_{pk}(m, \sigma) \stackrel{?}{=} 1$.

It is not difficult to verify that Π' is existentially unforgeable if (Gen_H, H) is universal one-way. (Note that the hashed input pk is chosen independently of the hash key s .) But public keys in Π' have length $|pk'| + |s| = k + |s|$ bits, an improvement only if $|s| < q(k) - k$. While this bound on the length of the hash key s can be achieved fairly easily if we are willing to assume collision-resistance, the bound is more difficult to ensure based on universal one-wayness alone (cf. the discussion at the end of Section 2.3.4).

Fortunately, it is possible to guarantee a public key shorter than the message by running sufficiently many copies of the original signature scheme in parallel.

Let us first verify the claim regarding the lengths of the public key and the messages. Π' has public keys of size $h(k) + \ell \cdot k$ and can sign messages of length $3k \cdot \ell$. By our choice of ℓ we have

$$3k\ell > 2k\ell + k \cdot (2h(k)/k) = 2k\ell + 2h(k),$$

and so the messages that can be signed have length at least twice that of the public key. As for the security of the construction, we have:

Construction 2.5: A signature scheme for messages twice as long as public keys

Let $\Pi = (\text{Gen}, \text{Sign}, \text{Vrfy})$ be a signature scheme for $3k$ -bit messages having $q(k)$ -bit public keys, and let (Gen_H, H) be a hash function mapping $q(k)$ -bit inputs to k -bit outputs and having keys of length $h(k)$. Choose $\ell(k) > 2h(k)/k$, and construct signature scheme $\Pi' = (\text{Gen}', \text{Sign}', \text{Vrfy}')$ as follows:

Key generation: $\text{Gen}'(1^k)$ does as follows:

1. Compute $s \leftarrow \text{Gen}_H(1^k)$.
2. For $i = 1$ to ℓ , compute $(pk_i, sk_i) \leftarrow \text{Gen}(1^k)$ and set $pk'_i := H_s(pk_i)$.

The public key is $(s, pk'_1, \dots, pk'_\ell)$, and the secret key is $(pk_1, sk_1, \dots, pk_\ell, sk_\ell)$.

Signature generation: $\text{Sign}'_{s, pk'_1, \dots, pk'_\ell}(m)$ parses m as m_1, \dots, m_ℓ with $|m_i| = 3k$ for all i . It then outputs the signature $(pk_1, \text{Sign}_{sk_1}(m_1), \dots, pk_\ell, \text{Sign}_{sk_\ell}(m_\ell))$.

Signature verification: $\text{Vrfy}'_{s, pk'_1, \dots, pk'_\ell}(m, (pk_1, \sigma_1, \dots, pk_\ell, \sigma_\ell))$ parses the message m as m_1, \dots, m_ℓ with $|m_i| = 3k$ for all i . It then outputs 1 iff for all i : (1) $H_s(pk_i) \stackrel{?}{=} pk'_i$, and (2) $\text{Vrfy}_{pk_i}(m_i, \sigma_i) \stackrel{?}{=} 1$.

Theorem 2.13. *If Π is existentially unforgeable (resp., strongly unforgeable) under a one-time chosen-message attack and (Gen_H, H) is universal one-way, then Π' is existentially unforgeable (resp., strongly unforgeable) under a one-time chosen-message attack.*

Proof (sketch). We treat the case of existential unforgeability; strong unforgeability can be proven similarly. Consider a PPT adversary A attacking Π' in a one-time chosen-message attack. Let $m' = m'_1, \dots, m'_\ell$ be the message whose signature is requested by A , and say A outputs the forged signature $(\widehat{pk}_1, \sigma_1, \dots, \widehat{pk}_\ell, \sigma_\ell)$ on the message $m = m_1, \dots, m_\ell \neq m'$. If $\widehat{pk}_i \neq pk_i$ for some i , then A has violated the assumed universal one-wayness of (Gen_H, H) . Letting j be any index with $m_j \neq m'_j$, we thus have that σ_j is a forged signature on the message m_j with respect to scheme Π (and public key pk_j).

An alternate, somewhat easier proof of the above theorem relies on the construction of universal one-way hash functions with high compression and short keys. We have given the above proof in order to keep the exposition self-contained.

Construction 2.5 is not existentially unforgeable when an adversary can request signatures on more than one message (even if Π is). However, a variant of the construction — in which each block of a signed message is pre-pended with a random, message-specific identifier — is secure in that sense (when Π is). See Construction 1.4 for the general idea.

2.5 Further Reading

Goldreich's book [56] is a good source for further information regarding generic assumptions, while more details regarding the number-theoretic assumptions discussed here can be found in [72]. The notions of a one-way function and a trapdoor permutation originate in the work of Diffie and Hellman [40], though formal definitions appeared only much later. Clawfree trapdoor permutations were introduced by Goldwasser, Micali, and Rivest [61] in the course of constructing the first secure digital signature scheme, and that work also contains a construction of clawfree trapdoor permutations based on the hardness of factoring (that is slightly different from the one given here).

Rabin [97] was the first to propose a trapdoor function based on the hardness of factoring, and Williams [110] and Blum [16] suggested restricting N to a special form to obtain a trapdoor *permutation*. As mentioned previously, the RSA assumption is due to Rivest, Shamir, and Adleman [99]. A recent survey by Boneh [17] discusses various attacks on RSA and also covers known results on the relationship between the RSA and factoring assumptions. The discrete logarithm assumption (without the restriction to prime order groups) is due to Diffie and Hellman [40].

Collision-resistant hash functions were first formally defined by Damgård [37], and the construction of collision-resistant hash functions from clawfree permutations is from that work as well. The Merkle-Damgård transformation was introduced independently in [38, 81], and our treatment in Section 2.3.2 is adapted from [72, Section 4.6.4]. Universal one-way hash functions originated in the work of Naor and Yung [88], where the construction of universal one-way hash functions based on one-way permutations was given. Rompel [100] (see also [71]) showed that universal one-way hash functions could be built from any one-way function. See [72] for extensive further discussion about hash functions and their applications.

Techniques for increasing the compression of universal one-way hash functions with reduced key expansion can be found in [11, 105]. Construction 2.4 is due to [11].



<http://www.springer.com/978-0-387-27711-0>

Digital Signatures

Katz, J.

2010, XIII, 192 p., Hardcover

ISBN: 978-0-387-27711-0