Work in progress

This is an early draft of a chapter of a book on the foundations of cryptographic authentication being coauthored by Francisco Corella, Sukhi Chuhan and Veronica Wojnas. Please send comments to the authors.

2. Cryptographic primitives

2.1 Preliminaries

2.1.1 Cryptographic assumptions and security reductions

The security of the cryptographic primitives described in this chapter is based on cryptographic assumptions stating that it is difficult to perform certain computations, e.g. factoring an RSA modulus, computing a discrete logarithm in a particular cyclic group, or finding colliding outputs of a particular hash function. A *security reduction* is a mathematical proof that, it there is an "efficient algorithm" that breaches the security of a primitive with "non-negligible probability", there is also an efficient algorithm that performs the difficult computation with non-negligible probability.

The terms "efficient algorithm" and "negligible probability" are defined in the context of a thought experiment where one imagines that, instead of a single cryptographic system where each key, each modulus and each group order has its own particular bit length, there is a sequence of systems indexed by a security parameter, where the bit lengths of keys, moduli and group orders increase with the security parameter.

2.1.2 System parameterization

It is easy to imagine such a sequence for systems where primitives have keys of arbitrary length, but more difficult for systems that include primitives such as AES that are only defined for a small number of key lengths, or primitives such as hash functions that take no keys at all.

Boneh and Shoup [1] overcome this difficulty by adding to the thought experiment an efficient probabilistic algorithm that takes as input the security parameter and produces a *system parameter*, which can be viewed as a randomized configuration of the system for a particular value of the security parameter. In the thought experiment with *system parameterization*, algorithms and events take the security parameter and the system parameter as additional inputs. A deterministic or probabilistic algorithm can then be defined as *efficient* if its running time is polynomial in the security parameter, and the probability of an event as *negligible* if it is asymptotically smaller than the inverse of any polynomial function of the security parameter. Significantly, in these definitions the randomness includes the coin tosses of the algorithm that produces the system parameter on input the security parameter.

The above definitions are specific to cryptography. In complexity theory an efficient algorithm is an algorithm that runs in polynomial time on the number of bits of its inputs. The two definitions are compatible if the security parameter is provided to the algorithm as the length of a bit string where all the bits are equal to 1, and the total bit length of the other inputs is polynomial in the security parameter.

2.1.3 Cyclic groups and discrete logarithms

An often-used cryptographic assumption is the difficulty of computing discrete logarithms in a particular cyclic group. In this section we first go over mathematical background related to cyclic groups in Section 2.1.3.1, before discussing the security strengths provided by discrete logarithm assumptions in Section 2.1.3.2.

2.1.3.1 Mathematical background on cyclic groups

A *group* is a set equipped with a binary operation that has the following properties: the operation is associative; the operation has an *identity element*, such that application of the operation to an element and the identity element results in the element; and every element of the set has an *inverse element*, such that application of the operation to the element and its inverse element results in the identity element. Precisely speaking, the group is an ordered pair comprising the set and the binary operation. But we will gloss over the distinction between a set such as, for example, the set $\mathbb{Z}$ of the relative integers, and that set equipped with one or more operations, such as, for example, $(\mathbb{Z}, +)$ or $(\mathbb{Z}, +, \cdot)$, when the operations in question are clear from the context.

A *subgroup* of a group is a subset of the elements of the group such that (i) application of the operation of the group to two elements of the subset results in an element of the subset; (ii) the identity element of the group is in the subset; and (iii) the inverse of every element of the subset is in the subset.

An *Abelian group* is a group whose operation is commutative.

A *ring* is a set equipped with two binary operations, called addition and multiplication, which is an Abelian group for addition, and where multiplication is associative, has an identity element, and is distributive with respect to addition.

A *field* is a ring where multiplication is commutative and every element other than the additive identity element has a multiplicative inverse. A field can also be equivalently defined as a set equipped with two Abelian groups, called the additive group and the multiplicative group, where the additive group spans the entire set, the multiplicative group comprises the elements of the set other than the additive identity element, and multiplication is distributive over addition.

The operation of a group that is not part of a field may be called addition or multiplication and notated accordingly, using + for addition and · or juxtaposition for multiplication, with the identity element written 0 or 1 and the inverse of x written $-x$ or x^{-1} respectively. The group is then called an "additive group" or a "multiplicative group", but such appellation just refers to how the operation is written. In cryptography, additive notation is used for the group of points of an elliptic curve, and multiplicative notation for most other groups.

The operation of a group is also called the *group law*. Two other operations are derived from the group law and written in accordance with the notation used for the group law:

- Application of the group operation to an element x and the inverse of an element y is a binary operation called subtraction and written $\mathrm{x} - \mathrm{y}$ in an additive group, or division and written $\frac{x}{y}$ or x/y in a multiplicative group.
- Repeated application of the group law to a group element and an accumulator initialized as the identity element is an external operation that takes as operands the group element and an integer specifying the number of repetitions. In an additive group, the external operation is called *scalar multiplication* and written with the relative integer operand to the left of the group element, as in $3\mathrm{P} = \mathrm{P} + \mathrm{P} + \mathrm{P}$. In a multiplicative group, the external operation is called exponentiation and written with the relative integer as a superscript, as in $g^3 = g \cdot g \cdot g$ or $g^3 = ggg$ Negative integers are used to specify that the group law is to be applied to the inverse of the group element rather than to the element itself.

We shall be using multiplicative notation in the rest of this section.

Definitions. Let G be a group and g an element of G. The set $\{g^i\}_{i \in Z}$ of powers of g forms a subgroup of G written $\langle g \rangle$ said to be *generated* by g. The element g is called a *generator* of the subgroup.

Definitions. A group that has a generator is said to be *monogenous*, and a finite monogenous group is said to be *cyclic*.

Theorem 1. A subgroup of a monogenous group is monogenous.

Proof. Let G be a monogenous group, H a subgroup of G, and g a generator of G. Let m be the smallest positive integer such that g^m is in H. Since H is a subgroup of the group G generated by g, every element h of H is of the form g^k for some $\mathrm{k} \in Z$. Divide k by m to obtain q and r such that $\mathrm{k} = \mathrm{mq} + \mathrm{r},\ 0 \le r < \mathrm{m}$. Since g^k and g^m are both in H, $g^r = g^k(g^m)^{-q}$ is also in H, and since m is the smallest positive integer such that g^m is in H, but $\mathrm{r} < \mathrm{m}$, it must be that $\mathrm{r} = 0$. Hence $g^k = (g^m)^q$ and $h = g^k \in \langle g^m \rangle$. Since this is the case for every $h \in H$, we have $H \subseteq \langle g^m \rangle$. And since $g^m \in H$, $\langle g^m \rangle \subseteq H$. Thus $\langle g^m \rangle = \mathrm{H}$ and $\langle g^m \rangle$ is a generator of H. ■

Corollary 2. A subgroup of a cyclic group is cyclic.

Proof. Follows from Theorem 1 and the definition of a cyclic group. ■

Proposition 3. Every cyclic group is Abelian.

Proof. For every pair of elements (g^i, g^j) of a cyclic group generated by g, we have $g^i g^j = g^{ij} = g^{ji} = g^j g^i$. ■

Definitions. The *order of a finite group* G, written $|G|$, is the number of elements of the group. A *proper subgroup* of a group G is a subgroup of G other than G itself. The *trivial subgroup* of G is the subgroup of order 1, containing the identity element.

Definition. The *order of an element* of a (finite or infinite) group that generates a finite subgroup, written $|g|$, is the number of elements of the subgroup. Thus, $|g| = |\langle g\rangle|$.

Theorem 4 (Lagrange). If G is a finite group and H is a subgroup of G, the order of H divides the order of G.

Proof. The binary relation $x \sim y \Leftrightarrow xy^{-1} \in H$ is an equivalence relation that partitions G into equivalence classes all having the same number of elements as H. Hence the order of G is the product of the order of H by the number of classes. ■

Corollary 5. A group of prime order has no non-trivial proper subgroups.

Proof. Since a prime number p has no divisors other than 1 and p, a group of prime order has no subgroups other than the trivial group, of order 1, and itself, of order p.

Theorem 6. A group of prime order is cyclic, and every element of the group other than the identity element is a generator of the group.

Proof. Let g be an element of a group G other than the identity element. Since $\langle g\rangle$ is a subgroup of G, by Lagrange's theorem (Theorem 4), $|g|$ divides $|G|$. Since g is not the identity element of G, $|g| \neq 1$. And since a prime number has no divisors other than 1 and itself, if $|G|$ is prime, $|g| = |G|$, hence $\langle g\rangle = G$, and G is cyclic. ■

Theorem 7. Let G be a group, and g an element of G of finite order $n \neq 1$. Then n is the smallest positive integer m such that $g^m = 1$; the integers k such that $g^k = 1$ are the multiples of n; and if integers i and j are such that $g^i = g^j$, they are congruent modulo n.

Proof. Let G be a group, and g an element of G of finite order. We shall refer to the sequence $(g^i)_{i\in\mathbb{Z}}$ as the sequence of powers of g. Each entry in the sequence is a pair (i, g^i) of an index $i \in \mathbb{Z}$ and a value g^i. The sequence can be viewed as a surjective map $i \mapsto g^i$ from $\mathbb{Z}$ onto $\langle g\rangle = \{g^i\}_{i\in\mathbb{Z}}$, which is finite by hypothesis.

Since the sequence is infinite while the set if finite, there must be a positive integer m such that $\mathrm{g}^{\mathrm{s+m}} = \mathrm{g}^{\mathrm{s}}$ for some $s \in \mathbb{Z}$. From $g^{s+m} = g^s$ it follows that $\mathrm{g}^{\mathrm{m}} = 1$ and hence that $g^{s+m} = g^s$ holds for *every* s in Z. Let *n* be the smallest such *m*. Then the sequence is periodic with period *n*, and, furthermore, there are no repeated values in any interval of the sequence of length *n*. From this it follows that two entries have the same value if and only if their indices are congruent modulo *n*. It also follows that all the elements of $\langle g \rangle$ occur without repetition in any interval of length *n*, and hence that *n* is the order of *g*.

The conclusions of the theorem then follow immediately. Since $\mathrm{g}^{\mathrm{n}} = 1$ and there are no repeated values in the interval (1, *n*], the order *n* of *g* is the smallest integer *m* such that $\mathrm{g}^{\mathrm{m}} = 1$. Since two entries have the same value if and only if their indices are congruent modulo *n*, two integers *i* and *j* are such that $\mathrm{g}^{\mathrm{i}} = \mathrm{g}^{\mathrm{j}}$ if and only if they are congruent modulo *n*, and as $\mathrm{g}^0 = 1$, the integers *k* such that $\mathrm{g}^{\mathrm{k}} = 1$ are the multiples of *n*. ∎

Corollary 8. Let *G* be a group, *g* an element of *G* of finite order $|g| \neq 1$, and *q* a prime number. Then $g^q = 1$ if and only if $|g| = \mathrm{q}$.

Proof. By Theorem 7, $g^q = 1$ if and only if q is a multiple of $|g|$. But since a prime number has no divisors other 1 and itself, and $|g| \neq 1$ by hypothesis, q is a multiple of $|g|$ if and only if $\mathrm{q} = |\mathrm{g}|$. ∎

Corollary 9. Let *G* be a group, *g* an element of *G* of finite order $n > 1$, and *h* an element of $\langle g \rangle$. Then there exists a unique nonnegative integer $i < n$ such that $g^i = h$.

Proof. Let *G*, *g* and *h* be as stated. Since $\langle g \rangle$ is the set of powers of *g*, there exists $k \in \mathbb{Z}$ such that $g^k = h$. By Theorem 7, the set of such *k* is a congruence class modulo *n*, which has exactly one representative *i* in the range $0 \le i < n$.

Definition. If *G* is a group, *g* an element of *G* of finite order $n > 1$, and *h* an element of $\langle g \rangle$, the *discrete log* of *h* to the base *g* if the unique nonnegative integer $i < n$ such that $g^i = h$.

Theorem 10. Let *G* be a cyclic group of order *n*, *g* a generator of *G*, and *H* a subgroup of *G* of order *h*. Then *h* is a divisor of *n* and *H* is generated by $g^{n/h}$.

Remark. We already knew that *h* is a divisor of *n*, by Lagrange's theorem (Theorem 4). The proof of this theorem rediscovers that fact in the narrower context of cyclic groups.

Proof. If $\mathrm{h} = 1$, then $g^{n/h} = g^n = 1$ and the conclusions are vacuously true. Assume now that $\mathrm{h} \neq 1$. Since *G* is cyclic, by Corollary 2, *H* is cyclic. Let g' be a generator of *H*. Since *H* is a subgroup of $\mathrm{G} = \langle g \rangle$ of order greater than 1, $g' = g^m$ for some positive integer m.

Let d = gcd(m, n) Since d divides m every power of g^m is a power of g^d, hence $\langle g^m \rangle \subseteq \langle g^d \rangle$. On the other hand, the extended Euclidean algorithm on inputs m and n outputs relative integers r and s such that rm + sn = d. And, by Theorem 7, $g^n = 1$. Therefore $g^d = (g^m)^r(g^n)^s = (g^m)^r 1^s = (g^m)^r$. Thus, every power of g^d is a power of g^m, and $\langle g^d \rangle \subseteq \langle g^m \rangle$. Hence $\langle g^d \rangle = \langle g^m \rangle =$ H.

Now we have a group G of order n generated by g and a group H of order h generated by g^d, with $(g^d)^{n/d} = g^n = 1$. By Theorem 7 applied to the group G, n is the smallest positive integer k such that $g^k = 1$. But then n/d must be the smallest positive integer j such that $(g^d)^j = 1$, for otherwise, if there is a positive integer j < n/d such that $(g^d)^j = 1$, then the positive integer k = dj < d(n/d) = n is a positive integer smaller than n such that $g^k = 1$. Therefore, by Theorem 7 now applied in reverse to $\langle g^d \rangle$ = H, n/d must be the order h of H. Hence h = n/d, hd = n, h is a divisor of n, d = n/h, and H is generated by $g^{n/h}$. ■

Theorem 11. Let G be a cyclic group of order n, g a generator of G, and h a divisor of n. Then the subgroup H of G generated by $g^{n/h}$ has order h.

Proof. By Theorem 7 applied to group G, $g^n = 1$. Hence $g^{(n/h)h} = 1$ and, by Theorem 7 applied to group H, h must be a multiple ij of the order i of H, and $g^{(n/h)i} = 1$. But since n = $(n/h)^{\mathrm{ij}}$, unless j = 1, this contradicts the fact that n must be the smallest k such that $g^k = 1$. Hence j = 1 and h = ij = i is the order of H. ■

Theorem 12 (Fundamental theorem of cyclic groups). If G is a cyclic group of order n, for every divisor h of n there is a unique subgroup H of G of order h.

Remarks.

1. Recall that we define a cyclic group as finite monogenous group. Authors who refer to $(\mathbb{Z}, +)$ as an infinite cyclic group refer to Theorem 12 as the fundamental theorem of *finite* cyclic groups.
2. Some authors add to the statement of the fundamental theorem the fact that every subgroup of G is cyclic (which follows from Corollary 2), the fact that the order a subgroup of G is a divisor of the order of G (which follows from Lagrange's theorem (Theorem 4) or from Theorem 10), and/or statements derived from Theorem 10 or Theorem 11.
3.

Proof. Let G be a cyclic group of order n and h a divisor of n. Since G is cyclic, it has a generator g, and by Theorem 11, the subgroup H of G generated by $g^{n/h}$ has order h. And by Theorem 10, any other subgroup H' of G of order h is also generated by $g^{n/h}$ and is therefore the same subgroup as H. ■

Corollary 13. If G is a cyclic group of order n, the function that maps each subgroup of G to its order is a bijection onto the set of divisors of n.

Proof. Follows immediately from the fundamental theorem (Theorem 12) and Lagrange's theorem (Theorem 4). ■

Theorem 14 (Subgroup membership test). Let G be a cyclic group, H a subgroup of G of prime order q, and g an element of G. Then g is in H if and only if $g^q = 1$.

Proof. Let G, H, and g be as stated. First, assume that g is in H. If g is the identity element of G, then $g^q = 1$. Otherwise, by Theorem 6, g is a generator of H, and therefore $|g| = \mathrm{q}$. Then, by Theorem 7, we have again that $g^q = 1$. Conversely, assume that $g^q = 1$. If g is the identity element of the group G, then it is also in the subgroup H as its identity element. Otherwise $|g| \neq 1$ and, by Corollary 8, $|g| = \mathrm{q}$. Hence H and $\langle g$ have same order and, by the Fundamental Theorem, they are the same subgroup of G. Therefore g is in H.

Definition (Cofactor). Let G be a finite group of order n, and H a subgroup of G of order q. By Lagrange's theorem (Theorem 4) there exists a positive integer r such that n = rq. If G is cyclic and q is prime, r is called the *cofactor* of H in G. Notice that the ratio r = n/q is defined whether or not G is cyclic and whether or not q is prime. But the term *cofactor* is used in contexts where the following theorem is applicable.

Theorem 15 (Cofactor clearing). Let G be a cyclic group of order n, H a subgroup of G of prime order q, r the cofactor of H in G, and x any element of G such that $x^r \neq 1$. Then x^r is a generator of H.

Proof. Let G, H, r and x be as stated. Let g be a generator of G, with $\mathrm{x} = g^k$. Then $x^n = x^{rq} = g^{krq} = g^{rqk} = g^{nk} = (g^n)^k = 1$ by Theorem 7. From $x^{rq} = 1$ if follows by Corollary 8 that $|x^r| = \mathrm{q} = |H|$. Hence by the fundamental theorem of cyclic groups (Theorem 12), $\langle x^r \rangle = \mathrm{H}$. ■

2.1.3.2 Security strengths provided by discrete logarithm assumptions

The security strength of the discrete logarithm assumption in a cyclic group depends, of course, on the size of the group, but also on its subgroup structure and its mathematical structure.

By subgroup structure we mean whether the group has any non-trivial proper subgroups, and if so if it has a large proper subgroup. The security implications of subgroup structure are discussed in Section 2.1.3.2.1.

By mathematical structure we mean the representation of the group elements as mathematical entities, which we categorize using NIST's classification of cryptographic primitives. The security implications of mathematical structure are discussed in Sections 2.1.3.2.3-6.

2.1.3.2.1 Group order and subgroup structure

By Corollary 13, the subgroup structure of a cyclic group G of order n is determined by the divisor structure of n.

If n is composite and only has small prime divisors, then an adversary can use the Pohlig-Hellman algorithm of Section 2.1.3.2.1.1 to reduce the problem of computing a discrete log in G to the problem of computing a discrete log in each of the prime order subgroups of G.

If n is composite but has a large prime divisor q, G has a unique subgroup of order q. We shall see how several primitives are implemented in such a subgroup, taking advantage of Theorem 15 (cofactor clearing) to generate elements of the subgroup and Theorem 14 (subgroup membership test) to check if an element of G is an element of the subgroup. However, G is also likely to have small subgroups, and care must be taken to prevent small subgroup attacks, as discussed in Section 2.1.3.2.1.2.

If n is prime, by contrast, neither Pohlig-Hellman nor small subgroup attacks are available to an adversary. As we shall see in Section 2.1.5, the group of points of a NIST elliptic curve is an example of a cyclic group of prime order.

2.1.3.2.1.1 Pohlig-Hellman algorithm

Pohlig-Hellman is an algorithm for computing discrete logarithms in a cyclic group G of composite order $n = q_1^{e-1} \cdots q_r^{e_r}$, where $q_1 \cdots q_r$ are the primes dividing n. It was originally formulated for the multiplicative group of a finite field [2], but variants for general cyclic groups have since been specified [1, §16.1.2], [3, §3.6.4].

In [1, §16.1.2], Boneh and Shoup compute the discrete log α of an element u of G to base g, where g is a generator of G. The computation has two steps. First, a recursive algorithm is used to compute the discrete log α_i of $u^{q_i^*}$ to the base $g^{q_i^*}$, where $q_i^* = n/q_i^{e_i}$ for $1 \leq i \leq r$. Then the Chinese Remainder Theorem [3, §2.1.2.0-1] is used to compute α from the α_i.

Boneh and Shoup estimate that the computation of each α_i is only as hard as computing a discrete log in the subgroup of G of prime order q_i, and conclude that the difficulty of computing α is determined by the size of the largest prime dividing n.

2.1.3.2.1.2 Small subgroup attacks

In a small subgroup attack, the attacker tricks the victim into believing that an element g of a cyclic group G is an element of a subgroup of G of large prime order, while in fact it is an element of a small subgroup. The victim may then compute $y = g^x$ where x is a secret such as a private key and use y in a manner that provides information about x to the attacker. Well-known examples are attacks against Diffie-Hellman key exchange [4] [5] [6].

In a Diffie-Hellman key exchange, two parties Alice and Bob with private keys x_A, x_B and public keys $y_A = g^{x_A}, y_B = g^{x_B}$ compute a shared secret $s = y_A^{x_B} = g^{x_A x_B} = y_B^{x_A}$ after exchanging their public keys. The parameters of the Diffie-Hellman primitive specify the group G, a large prime q that divides the order of G, and the base g of the exponentiations, which must be an element of the unique subgroup H of order q.

In a small subgroup attack where party A is the victim, the attacker, Mallory, causes Alice to compute $s' = y_M^{x_A}$, where y_M lies in a small subgroup H' instead of $S = y_B^{x_A}$, where y_B lies in H. In different attack variants, Mallory could to that, for example, by playing the role of Bob, or as an man-in-the-middle between Alice and Bob (who may legitimately exchange their public keys in the clear), or by controlling a certificate authority that issues a certificate to Bob with the wrong public key.

If Mallory has access to s', he can compute x_A as the discrete log of s' to base y_M. Typically, however, that will not the case, because Alice and Bob don't need to tell each other the results of their shared secret computations, which should equal. But Mallory will typically have access to data disclosed by A that has been computed using cryptographic keys derived from S', such as a symmetric signature computed with an HMAC key, or a plaintext decrypted using an AES key. And since s' lies in the small subgroup H', Mallory can enumerate the elements of H', and see which one of them would produce the data disclosed by A when used as s'. He can then compute x_A as the discrete log of that element to base y_M.

RFC 2785 [6] describes several attack variants in cases where the group G is the multiplicative group of a finite field, and countermeasures against the attacks. The public key validation countermeasure of [6, §3.1] is based on the subgroup membership test of Theorem 14 above.

2.1.3.2.2 NIST classification of cryptographic primitives

NIST defines "security strength" as "a number associated with the amount of work (i.e., the number of operations) that is required to break a cryptographic algorithm or system" [7] and assigns security strengths to four classes of cryptographic primitives in a table of "comparable security strengths" [8, Table 2]. Two of these classes comprise primitives that are based on discrete log assumptions in cyclic groups: FFC (Finite Field Cryptography, column 3) and ECC (Elliptic Curve Cryptography, column 5). As we shall see in Section 2.1.5, the group of points in some elliptic curves is not cyclic, but it is cyclic in the elliptic curves that are used in cryptography today.

2.1.3.2.3 Strength of the discrete log assumption in FFC primitives

FFC primitives include Diffie-Hellman, DSA, and the FFC variant of MQV, specified in Section 5.7.2.1 of [9]. Their security relies on the difficulty of computing discrete logs in a prime order subgroup of the multiplicative group Z_p^* of the field Z_p of integers modulo a prime p, called a *Schnorr group* because such a group was used by Schnorr for his identification protocol and his signature scheme, as we shall see in Chapter 3.

A Schnorr group is constructed by choosing the prime number p, and a prime number q that divides $\mathrm{p} - 1$. The multiplicative group Z_p^* is cyclic [10, Theorem 62] and has order $\mathrm{p} - 1$. Therefore, by the fundamental theorem of cyclic groups (Theorem 12), it has a unique subgroup H of order q, which is cyclic by Corollary 2 and has cofactor $\mathrm{r} = (p-1)/\mathrm{q}$ in Z_p^*. And by the cofactor clearing theorem (Theorem 15), if $1 < \mathrm{x} < \mathrm{p}$ and $\mathrm{g} = x^r \bmod \ \neq 1$, then g is a generator of H. The subgroup H is a Schnorr group, and FFC primitives can be based on the difficulty of computing the discrete log of an element u of H to the base g.

At this time, the fastest algorithm available to an attacker for computing the discrete log of u to base g in the subgroup H of Z_p^* is the General Number Field Sieve (GNFS) [11] [12], which performs the computation in time $L_p[\alpha, c]$, with $\alpha = 1/3$ and $\mathrm{c} = (64/9)^{1/3}$, where L_p refers to the asymptotic “L-notation” [13], defined by

$$L_p[\alpha, c] = \exp\Big(\big(\mathrm{c}+\mathrm{o}(1)\big)(\log \mathrm{p})^{\alpha}(\log\log \mathrm{p})^{1\text{-}\alpha}\Big),$$

or equivalently by

$$L_p[\alpha, c] = \exp\Big(\big(\mathrm{c}+\mathrm{o}(1)\big)\mathrm{b}^{\alpha}(\log \mathrm{b})^{1\text{-}\alpha}\Big),$$

where $\mathrm{b} = \log\ $ is the bitlength of the prime number p. The term $\mathrm{o}(1)$ in the L-notation is one of the better-known asymptotic notations $\mathrm{O}(\,)$, $\mathrm{o}(\,)$, $\Omega(\,)$, $\omega(\,)$, and $\Theta(\,)$ [14].

L-notation is useful for comparing the asymptotic performance of algorithms that have time complexities ranging from polynomial to exponential. The value $\alpha = 0$ corresponds to polynomial complexity, as

$$L_p[0, c] = \exp\Big(\big(\mathrm{c}+\mathrm{o}(\mathrm{i})\big)\log\ \Big) = b^{c+o(i)},$$

while $\alpha = 1$ corresponds to exponential complexity, as

$$L_p[1, c] = \exp\Big(\big(\mathrm{c}+\mathrm{o}(\mathrm{i})\big)\mathrm{b}\Big).$$

Intermediate values $0 < \alpha < 1$ correspond to algorithms that are sub-exponential but super-polynomial.

Since GNFS computes discrete logs in Z_p^* in time $L_p\big[1/3, (64/9)^{1/3}\big]$, it is classified as having sub-exponential time complexity. By contrast, we shall see in Section 2.1.3.2.4 that the discrete log algorithms currently available for attacking ECC primitives on most curves have exponential time complexity. Hence FCC primitives are asymptotically weaker than ECC primitives.

Furthermore, it turns out that most of the computations that GNFS does depend only on the prime p and can be performed in a precomputation phase, enabling an attack dubbed “Logjam” where the attacker performs the precomputation once and uses the result to attack any number of discrete logs.

When Logjam was discovered in 2015 [15] [16], several Internet protocols recommended or required the use of a small number of specific primes *p* deemed to be "safe". Those primes were thus being built into the Internet and were being used universally for many different purposes, providing a huge incentive for attackers with large resources to invest on performing the precomputation phase on those primes. Since the vulnerability was discovered shortly after the Snowden revelations, it was feared that the NSA might have been aware of it and might have been using it to perform widespread eavesdropping on the internet. Suspicions that leaked NSA claims of cryptanalytic prowess referred to Logjam were disputed [17], but the need to take precautions against Logjam attacks remains to this day.

Together, the speed of the GNFS and the Logjam vulnerability have motivated a gradual shift from FFC primitives to ECC primitives in Internet protocols and cryptographic authentication solutions. DSA is no longer included in the Digital Signature Standard [18, §4].

2.1.3.2.4 Strength of the discrete log assumption in ECC primitives

ECC primitives include ECDSA, EdDSA, ECDH and the ECC variant of MQV described in [9, §5.7.2.3]. Their security relies on the difficulty of computing discrete logs in a cyclic subgroup of prime order of an ECC group, where by ECC group we mean the group of points on an elliptic curve.

As we shall see in Section 2.1.5, an ECC group is Abelian, but it may or may not be cyclic. Furthermore, discrete logs are easy to compute in some curves. In spite of these drawbacks, ECC primitives are regarded as providing strong security, and they are the primitives of choice for digital signature and key agreement in new deployments of cryptographic authentication solutions.

The reason for their popularity is that there are currently no discrete-log algorithms available to an attacker that are fast because they exploit the general structure of an ECC group, i.e. the mere fact that the elements of the group are the points of an elliptic curve. There are algorithms that are fast because they exploit the particular structures of the ECC groups of several categories of *insecure curves* [1, §15.3]. But those algorithms can be denied to the attacker by not using curves in those categories, leaving only to the attacker relatively slow generic algorithms such as *baby-step/giant-step* [1, §16.1.1], or *Pollard rho* [19], [1, §11.2.5] that work on all cyclic groups.

Elliptic curve technology is complicated, and requiring implementors to figure out if a particular curve belongs to one of the known categories of insecure curves would impose an undue burden on the implementors and entail a security risk. This is one of the drawbacks presented by ECC groups, along with the fact that some of them are not cyclic. The strategy for avoiding these drawbacks is to use curves vetted by experts that have cyclic ECC groups and are not in any of the known categories of insecure curves. A list of such *safe curves* is currently available at [20].

This strategy begs two questions: (i) what if there are categories of insecure curves that are not yet known? and (ii) what if there are generic algorithms faster than baby-step/giant-step and Pollard rho that are not yet known?

An answer to question (i) is that categories of insecure curves consist of curves that are not "normal"; they have special features, and those special features make them insecure. One way of avoiding curves that may someday be found to be insecure is to use "normal" curves, resisting the urge to be clever and design curves with special features for particular purposes.

Surprisingly, there is an easy answer to question (ii): there is no need to worry about it because there is a high lower bound on the performance of generic algorithms.

2.1.3.2.5 Speed of generic discrete-log algorithms

As discussed in the previous section, there are no known algorithms for computing discrete logs in all ECC groups other than generic algorithms that work on all cyclic groups.

The fastest generic algorithm known today is the baby-step/giant-step algorithm. It runs in square-root time, i.e. it performs roughly $q^{1/2}$ group operations to compute a discrete log in a cyclic group of prime order q. In L-notation, square-root time corresponds to the parameters $\alpha = 1$ and $c = 1/2$, since

$$L_q[1,1/2] = \exp\left(\left(1/2+\mathrm{o}(1)\right)\log \mathrm{q}\right) = q^{1/2+o(1)}.$$

It is thus classified as exponential in the L-notation scale (since $\alpha = 1$) and is much slower than GNFS which is sub-exponential with $\alpha = 1/3$.

It may be possible to discover faster generic algorithms than baby-step/giant-step in the future, but not much faster.

An algorithm that computes discrete logs in a cyclic group of prime order q is informally said to be *generic* if it does not take advantage of the structure of the group; but it could more formally be defined as generic if it could be specified in the generic group model [1, §16.3], where group operations are performed by an oracle that hides the structure of the group. Baby-step/giant-step is a generic algorithm according to this more formal definition.

To compute a discrete log in a group of prime order q, baby-step/giant-step as described in [1, §16.1.1] would require between $q^{1/2}$ and $2q^{1/2}$ group operations. No generic algorithm can be much faster than that. It is shown in Theorem 16.3 of [1, §16.3] that an adversary trying to guess a discrete log in a generic group of prime order q after making at most T queries to the oracle would have a probability not greater than $((3T + 2)^2 + 1)/\mathrm{q}$ of succeeding. Succeeding with probability greater than ½ would thus require a number of queries T greater than $\left((q - 2)^{1/2} - 2\right)/3 \approx q^{1/2}/3$, which is only six times less than the number of group operations performed by baby-step/giant-step in the worst case.

2.1.3.2.6 NIST's comparison of security strengths

The difference in security strength provided by the discrete log assumptions used by FFC and ECC primitives is reflected in the minimum bitlength requirements of the parameters in columns 3 and 5 of the NIST table of comparable security strengths [3, Table 2].

FFC primitives rely on the difficulty of computing discrete logs in a Schnorr group constructed as described above in Section 2.1.3.2.3, and the parameters L and N in column 3 are to the bitlength of the prime p and the order of the group respectively. ECC primitives rely on the difficulty of computing discrete logs in a cyclic subgroup of prime order of an ECC group, and the parameter f is the order of the subgroup. As the size of the security strength specified in column 1 doubles from 128 to 256, the required bitlengths of N and f double from 256 to 512, but the required bitlength of L grows much faster, by a factor of 5, from 3072 to 15360.

This is because the GNFS makes it possible to compute discrete logs in Z_p^* in time $L_p\left(1/3, (64/9)^{1/3}\right)$, which is approximately equal, neglecting the $o(1)$ term, to

$$\exp\left((64/9)^{1/3}(3072)^{1/3}\left(\log(3072)\right)^{2/3}\right) \approx 2^{143}$$

when $b = \log \ = 3072$, and

$$\exp\left((64/9)^{1/3}(15360)^{1/3}\left(\log(15360)\right)^{2/3}\right) \approx 2^{276}$$

when $b = \log \ = 15360$. Thus, increasing the bitlength of p by a factor of 5 only increases the amount of effort that the adversary must make to compute discrete logs from 2^{143} to 2^{276}, as if the number of bits had approximately doubled from 143 to 276, and only a brute farce attack was available to the adversary.

Two additional remarks about the [3, Table 2] may be useful:

1. Column 4 refers to primitives such as RSA whose security is based on the difficulty of factoring a modulus n, and the parameter k is the sized of n. Integer factorization was the original application of the GNFS; hence it is not surprising that the values of k in column 4 are the same as the values of L in column 3.
2. The values of N, k and f in columns 3 and 4 are twice the security strength because the primitives in those columns must provide collision resistance.

2.1.4 Galois fields

We have seen the definition of "field" in Section 2.1.3.1. A "Galois field" is a field that has a finite number of elements. Finite fields are called a Galois fields in honour of the French mathematician Evariste Galois, and because finite fields are a core concept of Galois theory.

Galois theory is an extensive area of mathematics, but we only need a small part of it as background for cryptographic authentication. In this section we state the facts we need about Galois fields without formal proofs or references to those proofs. Readers who want to know more are referred to the large number of textbooks about Galois theory.

The order, or size, of a Galois field is the number of elements of the field, just like the order of a group is the number of elements of the group. If a Galois field has order q, its multiplicative group has order $q - 1$.

The order of a Galois field is a prime power $q = p^k$, where p is a prime and k is a positive integer, or just a prime p in the special case where $k = 1$. In either case, the prime p is the characteristic of the field, defined as the smallest positive integer n such that $\underbrace{1 + \cdots + 1}_{\text{n}} = 0$. A binary field is a field with characteristic 2. In this book, fields will be finite and non-binary unless otherwise stated.

All fields of same order q are isomorphic, and generically referred to as $\mathrm{GF}(q)$ or $\mathbb{F}_p$. The field $\mathbb{Z}_p$ of integers modulo a prime p is a particular representation of $\mathbb{F}_p$.
A representation of $\mathbb{F}_{p^k}$, $k > 1$ can be constructed using a monic irreducible polynomial P of degree k with coefficients in $\mathbb{Z}_p$. The elements of the representation are then the polynomials of degree less than k, addition is polynomial addition, and multiplication is polynomial multiplication followed by taking the remainder of the Euclidian division of the product by P.

A field of order p^k has a subfield of order p^j if and only if j divides k. When that is the case, there is only one such subfield. That makes it possible to refer generically to the subfield $\mathbb{F}_{p^j}$ of $\mathbb{F}_{p^k}$. If a field F is a subfield of a field F', F' is called an *extension field* of F.

The non-zero elements of a field of order q form a group of order $q - 1$. We saw in Section 2.1.3.2.3 that the group $\mathbb{Z}_p^*$ is cyclic, referring to [9, Theorem 62]. But the cited theorem actually states that every finite subgroup of the multiplicative group of any field is cyclic. Thus, the multiplicative group of any Galois field is cyclic.

2.1.5 Elliptic curves

2.1.5.1 The projective plane

Let F be a field, let $F^* = F \setminus \{0\}$, and let $F^{3*} = F^3 \setminus \{(0,0,0)\}$ be the three-dimensional affine space F^3 deprived of its origin. If $P = (X, Y, Z)$ and $P' = (X', Y', Z')$ are elements of F^{3*}, the homothecy relation $P \sim P' \Leftrightarrow (\exists k \in F^*)\big((X', Y', Z') = (kX, kY, kZ)\big)$ is an equivalence relation that partitions F^{3*} into classes called *projective points*. Each projective point consists of the affine points other than $(0,0,0)$ of a line of F^3 that goes through the origin. The quotient set $F^{3*}/\sim$ is *the projective plane* over F, written $\mathbf{P}2(F)$.

A point of the projective plane has *projective coordinates* (X, Y, Z) defined up to homothecy. If $Z \neq 0$ it also has *affine coordinates* $(x, y) = (X/Z, Y/Z)$. A projective point that has affine coordinates is called a *regular projective point*, while a point where $Z = 0$ is called a *point at infinity*. The points at infinity are lines of F^3 (minus the origin) that lie in the plane with equation $Z = 0$. Each regular projective point is a line of F^3 (minus the origin) that intersects the plane with equation $Z = 1$, at an affine point whose coordinates in that plane are the affine coordinates of the projective point. The plane with equation $Z = 1$ is the image of the translation by the vector $(0,0,1)$ of the plane with equation $Z = 0$. We shall refer to the latter as the *base affine plane*, and to the former as the *translated affine plane*.

2.1.5.2 Algebraic plane curves

A projective algebraic equation $\mathrm{H}(X, Y, Z) = 0$ where H is an irreducible homogeneous polynomial over F is compatible with the homothecy relation. Hence its solutions define a set of projective points, called a *projective algebraic plane curve*. (If H can be factored, each factor defines a separate curve.)

Assigning $(x, y, 1)$ to (X, Y, Z) dehomogenizes H, producing an affine algebraic equation $h(x, y) = 0$ that defines an *affine algebraic plane curve*. The affine curve is the intersection of the projective curve with the base affine plane.

Conversely, H can be derived from h by adding power of Z factors to each term as needed to homogenize it. Thus, the projective curve and the affine curve can be viewed as two aspects of a single algebraic plane curve.

The points of infinity of the projective curve correspond to asymptotes of the affine curve. Their coordinates can be determined up to homothecy by (i) setting Z to 0 in the equation $H(X, Y, Z) = 0$; (ii) setting one of the remaining variables X, Y to 1; and (iii) solving the equation for the last remaining variable.

In the special case where the polynomial H is of degree 1, the algebraic plane curve is a *projective line*, consisting of the projective points that lie in a plane of F^3 containing the origin. In the further special case where the projective equation is $\mathrm{Z} = 0$, the plane of F^3 is the base plane, and the projective, called the *line at infinity*, contains all the points at infinity.

2.1.5.3 Weierstrass equation

A projective Weierstrass equation E over a field F, sometimes written E/F, is a homogeneous algebraic equation of the form

$$Y^2 Z = X^3 + aXZ^2 + bZ^3,$$

where $a, b \in \mathrm{F}$ and the discriminant $4a^3 + 27b^2$ is not equal to 0.

If F' is an extension field of F, possibly equal to F, the equation E/F defines an algebraic plane curve $E(F')$, called an *elliptic curve*, in the projective plane $\mathbf{P}2(F')$. The point at infinity of $E(F')$, which we shall call $\mathcal{O}$, has coordinates $(0,1,0)$ defined up to homothecy in $\mathbf{P}2(F')$, as determined by setting $Z = 0$ and $Y = 1$ in the equation, then trivially solving for X to get $X = 0$. The point at infinity thus lies in the base plane, where it comprises the affine points of the y axis other than the origin.

The elliptic curve $E(F')$ can equivalently be defined by the affine algebraic equations

$$y^2 = x^3 + ax + b$$

obtained by dehomogenizing the projective equation as described in Section 2.1.5.2.

Besides referring to the Weierstrass equation, the notation E, or E/F, is also used to refer generically to all the elliptic curves $E(F')$ for all extension fields F' of F. A point of $E(F')$ for a particular extension F' is then said to be an *L-rational point* of E or E/F.

The above projective and affine equations are sometimes called *simplified,* or *short Weierstrass equations*, to distinguish them from longer forms of the equations from which they can be derived using *admissible changes of variables* [21, §3.1.1].

2.1.5.4 Jacobian coordinates

Besides projective and affine coordinates, an elliptic curve may also be specified using Jacobian coordinates, which are more efficient for certain computations. Jacobian coordinates are defined so that a point with Jacobian coordinates (X, Y, Z) has affine coordinates $x = X/Z^2$ and $y = Y/Z^3$. Replacing x and y with these values in the affine Weierstrass equation and multiplying by Z^6 yields the equivalent equation in Jacobian coordinates

$$Y^2 = X^3 + aXZ^4 + bZ^6.$$

Setting $Z = 0$ in that equation gives $Y^2 = X^3$. Hence the Jacobian coordinates of the point at infinity are $\{k^2, k^3, 0\}_{k \in F}$ while the Jacobian coordinates of a regular point with affine coordinates (x, y) are $\{(xk^2, yk^3, k)\}_{k \in F}$.

2.1.5.5 Group law

A binary operation on pairs of points of an elliptic curve may be defined in such a way that the resulting structure is an Abelian group whose identity element is the point at infinity. The group may or may not be cyclic, but it is cyclic for the safe curves specified in [20].

Since the concept of elliptic curve is defined in projective space, the rules defining the group law are most naturally formulated in projective coordinates. However, it may be advantageous in some cases to represent the points of an elliptic curve in Jacobian coordinates, or in affine coordinates with the point at infinity treated as a special case in computations. Some computations with Jacobian coordinates may be more efficient, while computations with affine coordinates may be viewed as simpler because a regular point has a single tuple of coordinates.

Formulas defining the group law using projective coordinates can be found in [22, §2.6.1] and [23]. Formulas for Jacobian coordinates are provided in [22, §2.6.2], [24], and for affine coordinates in [22, §2.2], [25].

A geometric interpretation of the affine formulas can be found in [22, §2.2] and a geometric interpretation in projective space for groups or points of general cubic plane curves, not just elliptic curves can be found in [26, §3.2].

2.1.5.6 Montgomery and Edwards curves

Montgomery and Edwards curves are algebraic plane curves defined by equations that can be converted to the Weierstrass equation of Section 2.1.5.3 by changes of variables. They are viewed as different "shapes" or "models" of elliptic curves that provide the same functionality as the *short Weierstrass model* while offering computational benefits..

They were separately introduced by Montgomery [27] and Edwards [28] but the work of Bernstein and his colleagues [29] [30] [31] eventually brought them together into the Ed25519 signature scheme, an instance of the EdDSA scheme described below in Section 2.2.3.3 that uses an Edwards curve birationally equivalent to the Montgomery Curve25519 [29].

2.1.5.6.1 Montgomery curves

A Montgomery curve [32] is an algebraic plane curve defined by an affine equation $E/\mathbb{F}_p$ of the form

$$By^2 = x(x^2 + Ax + 1),$$

or a projective equation of the form

$$BY^2Z = X(X^2 + AXZ + Z^2)$$

with $B \neq 0$ and $A^2 \neq 4$.

A computational benefit of Montgomery curves is that they provide a fast implementation of the *Montgomery Ladder* [32, Algorithm 4], an algorithm that computes a scalar multiplication using only affine x-coordinates (or projective (X, Z) coordinates).

The Montgomery Ladder takes as inputs a scalar $k \in \mathbb{Z}$ and coordinates $\big(X(P), Z(P)\big)$ of a point P and computes the coordinates $((X(Q), Z(Q))$ of the point Q resulting from the scalar multiplication of P by k, written $Q = [k]P$. It can be implemented in an elliptic curve defined by a Weierstrass equation, but the Montgomery equation yields a more efficient implementation, as shown in [32, §3.4].

The Montgomery Ladder can be used to implement a simplified variant of ECDH, where the base point, the public keys, and the shared secret are all defined, up to the sign of their *y* coordinates, by their *x* coordinates. This simplification was pointed out as early as 1987 by Miller in the paper where he proposed the use of elliptic curves for cryptography [33, last paragraph].

2.1.5.6.2 Curve25519

Bernstein paper "Curve25519: new Diffie-Hellman speed records" [29] made Curve25519 a famous elliptic curves and $2^{255} - 19$ a famous prime number. Paradoxically, however, Curve25519 is not the name of a curve in that paper, and the paper does not credit $2^{255} - 19$ with any speed record breaking properties.

Originally, Bernstein used the name Curve25519 to refer to an implementation of the Diffie-Hellman key agreement primitive simplified to use only x-coordinates as explained above in Section 2.1.5.6.2. He later changed the terminology, using "X25519" to refer to the simplified Diffie-Hellman implementation and "Curve25519" to refer to the curve used in X25519 [34].

To achieve the speed records achieved by what is now called X25519, Bernstein optimized many choices affecting speed. Thirteen optimized choices are listed at the end of [29, Section 1], one of them being "Use a prime extremely close to2^b for some b". But no doubt the use of the Montgomery Ladder to implement Diffie-Hellman use only *x* coordinates deserves a log of the credit.

2.1.5.6.3 Edwards curves

An Edwards curve, as defined by Bernstein and Lange in [30], is an algebraic plane curve defined by an affine equation $E/\mathbb{F}_p$ of the form

$$x^2 + y^2 = 1 + dx^2y^2,$$

with $d \notin \{0,1\}$, or a projective equation of the form

$$(X^2 + Y^2)Z^2 = Z^4 + dX^2Y^2.$$

Formulas for group operations presented in [30, §4] are faster than formulas for NIST curves defined by Weierstrass equations with Jacobian coordinates, optimized by the choice of coefficient $a = -3$ as explained in [22, §2.6.5].

But the main benefit of Edwards curves is that they support *unified addition* formulas, i.e. formulas that can be used for point addition and point doubling. This is important because unified addition formulas [35] can provide protection against private key recovery using side-channel attacks based on timing analysis [36] or power consumption analysis [37].

2.1.5.6.4 Twisted Edwards curves

In [31], Bernstein and his colleagues introduced "twisted Edwards curves" as curves defined by the equation

$$ax^2 + y^2 = 1 + \mathrm{d}x^2y^2.$$

It should be noted, however, that such curves are not an alternative to ordinary Edwards curves, but rather a generalization thereof, since as stated in [31, Definition 2.1], "an Edwards curve is a twisted Edwards curve with $a = 1$". What matters is that every curve defined by the generalized equation is "birationally equivalent" to a Montgomery curve as stated in [31, Theorem 32] and vice-versa, and thus that "the generalization brings the speed of the Edwards addition law to every Montgomery curve".

2.2 Primitives used in cryptographic authentication

2.2.1 Hash functions and hash trees

Cryptographic hash functions provide essential functionality for cryptographic authentication. They have traditionally been used for purposes such as compressing a string before signing it, or transforming an interactive zero-knowledge proof into a non-interactive one by means of the Fiat-Shamir heuristic. And as we shall in Chapter 3, however, they have now started to be used for implementing selective disclosure credentials.

2.2.1.1 Cryptographic properties of hash functions

A cryptographic hash function is a function $y = f(x)$ that accepts very long bit strings *x* as inputs (long enough to be of unlimited size for practical purposes), produces bit strings *y* of fixed length as outputs, and has the following properties:

1. Preimage resistance: if $y = f(x)$ an adversary who knows *f* (e.g. who can inspect the program or the Turing machine that implements *f*) and is given *y* has a negligible probability of finding *x.*
2. Second preimage resistance: if $y = f(x)$, an adversary who knows *f*, *y* and *x* has a negligible probability of finding an x' such that $y = f(x')$.
3. Collision resistance: an adversary who knows *f* has a negligible probability of finding *x* and x' such that $f(x) = f(x')$.

As we saw in Section 2.1.5, the concept of negligible probability is difficult to use in connection with hash functions, which do not have a key that would increase in length with the security parameter. We also saw that Boneh and Shoup [1] solve this problem by introducing a system parameter that can be viewed as a randomized configuration of the cryptographic system for each particular value of the security parameter. In their approach, a hash function does not have a key that grows in length with the security parameter, but the system parameter defines a configuration of the hash function that grows in complexity.

A formal definition of system parameterization can be found in [1, §2.3.2], and a definition of collision resistance using system parameterization in [1, §8.1.1]. A proof of security of an omission-tolerant digest using a simplified treatment of hash functions with system parameterization can be found in [38].

2.2.1.2 Choosing a hash function

Many cryptographic hash functions are available today. A catalog can be found in [39].

But one must be careful when choosing a hash function. The cryptographic properties of hash functions can be formally defined using system parameterization, and used in formal proofs of security of cryptographic primitives that use hash functions. But there is absolutely no formal proof that any hash function actually provides its purported cryptographic properties. Design of hash functions is a black art. Techniques such as the "sponge construction" [40] are heuristically motivated, but heuristic motivation is not formal proof.

Two criteria can be used to choose a hash function: the amount of trust in the designers of the algorithm, and the amount of scrutiny that the algorithm has received.

NIST has standardized three families of cryptographic hash functions SHA-1, SHA-2 and SHA-3, where SHA means Secure Hash Algorithm [41]. SHA-1 consists of a single function, which has now been deprecated. SHA-2 [42] includes SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256. It was published in 2002, but it has not been deprecated and SHA-256 is still widely used today. SHA-3 [43] includes SHA3-224, SHA3-256, SHA3-384, and SHA3-512.

Can NIST be trusted? There were reasons not to trust it after Snowden revealed the presence of a backdoor in Dual-EC-DRBG, an algorithm for deterministic random bit generation that the NSA has supplied to NIST and NIST had added to the original version of SP 800-90A. A description of the backdoor and references to the incident can be found in [44].

A deterministic random bit generator, like a hash function, is something that cannot be formally verified. And SHA-2 was designed by the NSA. SHA-3, on the other hand, is part of Keccak [45], which won the "SHA-3" Cryptographic Hash Algorithm Competition in 2012. Therefore, by the trust criterion, SHA-3 should clearly be preferred over SHA-2. On the other hand, SHA-2 has been used and scrutinized since 2002, and Keccak is more recent, having published in 2008. But then, as the winner of the competition, it has certainly received a fair share of scrutiny. So, either one is a reasonable choice.

2.2.1.3 Structured cryptographic digests: lists, chains, trees

A hash function can compute digests of very large data structures encoded as bit strings. But very large data structures do not come into life as bit strings all at once. They grow over time as structured data, and it is desirable to maintain a digest that is incrementally updated with as the data grows. Structured digests such as hash lists, hash chains and hash trees make that possible.

For example, a bitcoin blockchain is a chain of blocks [46]. When a new block is mined, it has a link to the previous one and a hash that covers the transactions in the block and the link to the previous block. The hash of the last block thus serves as a digest of entire blockchain.

As another example, Ethereum keeps all its transactions in a kind of hash tree called a Merkle Patricia Trie (sic) [47], which is very large but can be updated efficiently when a new transaction is created.

Structured digests also make it possible to implement selective disclosure credentials by hiding data. For example, a credential can contain a list of user attributes and a parallel list of digests of those attributes. The issuer signs over the digests, which the user presents to the verifier along with the attributes that it wants to disclose. We shall see in Chapter 3 how this technique is used in SD-JWT credentials.

The earliest kind of structured digest was the Merkle tree, invented and patented by Ralph Merkle in 1979 [48]. A Merkle tree is a binary tree where each internal node is labelled by the hash of the concatenation of the hashes of its two children, while each leaf node is labelled by the hash of a data block. The purpose of the tree is to allow a prover to demonstrate to a verifier that the data blocks are linked to the tree, by the presence of their hashes as labels of the leaf nodes.

The tree is balanced, so the number of data blocks is $n = 2^d$, where *d* is the depth of the tree, defined as the number of nodes in the path from the root to a leaf node, not including the root. To prove that a data block is linked to a leaf node, the prover sends to the verifier the label of the root and the labels of the siblings of the nodes in the path. Besides computing the hash of the data block, the verifier only needs to compute the hashes of the $d = \log_2 n$ nodes in the path with its siblings. By contrast, if the nodes were chained together as in the bitcoin blockchain the verifier would have to compute *n* hashes.

A Merkle tree has no indication of its depth, so shortening a tree of depth greater than *d* by removing the nodes at depth greater than *d* results in a different tree with the same root label. The label of a node at depth *d* is the hash of the concatenation of the labels its children in the original tree, which have been removed in the shorter tree. But the concatenation of two labels is a valid data block, so the node at depth *d* is now a valid leaf node. Shortening the tree is sometimes referred to as a “second preimage” attack, but this is incorrect terminology, since the hash function that computes the label of the node at depth *d* does not have two preimages, it has one preimage which is viewed differently in the two trees.

In any case, the only practical consequence of a shortening attack could be to remove evidence that the victim has recorded documents by linking them to the tree. This could be a concern in Certificate Transparency, where Merkle trees are used as append-only certificate logs. The Certificate Transparency standard solves the problem by prepending a byte to the input to the hash function and using different bytes in leaf nodes and internal nodes [49, §2.1.1].

In Chapter 11 we shall see how a very different kind of hash tree, called a *typed hash tree*, can be used to construct *fusion credentials* that support authentication with multiple authentication factors, with selective disclosure of attributes and selective presentation of authentication factors.

2.2.2 Symmetric signatures and HMAC

A *hash-based message authentication code* (HMAC) [50] is a kind of *message authentication code* (MAC) [51] commonly used as a symmetric signature. As we shall see in Chapter 13, an HMAC is also being used, in combination with key agreement using a static ECDH key, to produce an "ECDH-agreed MAC" usable as a *repudiable* asymmetric signature.

The simplest method of combining a secret key *K* with a message *M* to produce a MAC using a hash function *H* is to hash the concatenation of the key and the message, i.e. to compute

$$Mac(K, M) = H(K|M),$$

where the vertical bar denotes concatenation. But this method is vulnerable to the *length-extension attack* described in Section 2.2.2.2 if the hash function is based on the *Merkle–Damgård construction* described in Section 2.2.2.1 and omits the optional "finalization step" of the construction. As we shall see, the hash functions most used today *are* based on the Merkle–Damgård construction and *do omit* the finalization step. The method used by HMAC to combine *K* and *M*, described below in Section 2.2.2.3, is not vulnerable to the attack.

2.2.2.1 The Merkle–Damgård construction

As described and illustrated in [52], a hash function based on the Merkle–Damgård construction adds length padding to the message, breaks the padded message into equal-sized blocks, and executes a loop that uses a *compression function* to merge each block into a hash.

The initial value of the hash is an initialization vector (IV) specified by the definition of the hash function. The final value is output as the value of the function or fed into an optional finalization step. In the SHA-2 family of hash functions [42], the finalization step is used by the functions that truncate their output, i.e. SHA-224, SHA-384, SHA-

512/224 and SHA-512/256; but it is omitted by the main functions, SHA-256 and SHA-512.

2.2.2.2 The length extension attack

The length extension attack can allow an attacker to use a signature computed with a symmetric key *K* by the victim of the attack to forge a signature on a different message verifiable with the key *K*, potentially allowing the attacker to impersonate the victim vis-à-vis a relying party if the relying uses possession of the key as an authentication factor. The attack is potentially usable against authentication with a static ECDH credential by using an ECDH-agreed MAC.

It is a complicated attack, but the essence of it can be easier to understood if we simplify the Merkle–Damgård construction by assuming that the initial vector *IV* and the length padding *LP* are blocks of same size as the message blocks. The loop of calls to the compression function can then be modelled as a function *Loop* that takes as inputs a block *i* to be used as initialization vector and a sequence of blocks *m*, and has the following "loop continuation" property: for every block *i* and sequences of blocks *m*, *m'*,

$$Loop(i, m|m') = Loop(Loop(i, m), m')$$

where | denotes concatenation of block sequences. We can also define: a function *LP* that takes as input sequence of blocks *m* and returns a single block

$$LP(m)$$

that encodes the length of *m*; a function *Hash* that takes as input a sequence *m* and returns

$$\mathrm{Hash}(m) = \mathrm{Loop}\big(IV, m\big|LP(m)\big),$$

where *IV* is the initialization vector specified by the definition of the hash function; and a function *Mac* that takes as inputs key *k* and a sequence of blocks *m* and returns

$$Mac(k, m) = Hash(k|m).$$

Suppose the party in legitimate possession of a key *K* has used it to compute a symmetric signature

$$S = Mac(K, M)$$

and we (the attackers) have obtained the signature *S* and the message *M*, but not *K*.

To forge a signature S' we are going to take advantage of the loop continuation property of the compression loop by using the legitimate signature *S* instead of *IV* as initialization vector. Let M' be a sequence of blocks that we intend to construct. Then

$$\begin{aligned} S' &= Loop(S, M') \\ &= Loop(Mac(K, M), M') \\ &= Loop(Hash(K \mid M), M') \\ &= Loop(Loop(IV, K \mid M \mid LP(K|M)), M') \\ &= Loop(IV, K \mid M \mid LP(K \mid M) \mid M') \end{aligned}$$

Now we can use any sequence of blocks M'' to construct M' as

$$M' = M" \mid LP(K \mid M \mid LP(K \mid M) \mid M").$$

Then

$$\begin{aligned} S' &= Loop(IV, K \mid M \mid LP(K \mid M) \mid M'' \mid LP(K \mid M \mid LP(K \mid M) \mid M'')) \\ &= Hash(K \mid M \mid LP(K \mid M) \mid M'') \\ &= Mac(K, M \mid LP(K \mid M) \mid M'') \end{aligned}$$

And if we let

$$M''' = M \mid LP(K \mid M) \mid M''$$

we have

$$S' = Mac(K, M'''),$$

which is a verifiable signature.

2.2.2.3 How HMAC avoids the length extension attack

The MAC algorithm that we have shown to be vulnerable to the length extension attack is the very simple

$$MAC(K, M) = Hash(K|M)$$

As specified in [50], HMAC computes instead

$$HMAC(K, M) = Hash(K\ XOR\ opad \mid Hash(K\ XOR\ ipad) \mid M))$$

where the length of K is adjusted by adding zeros to be equal to the byte-length B of message blocks, and *ipad* and *opad* are the bytes 0x36 and 0x5C repeated B times.

Clearly, this construction is not vulnerable to the length extension attack, as the output of the compression loop of the inner hash function is hashed by the outer hash function.

Besides avoiding the length extension attack, the HMAC design also addresses weakness of earlier constructions such as

$$MAC(K, M) = Hash(K \mid M \mid K).$$

The purpose of the *ipad* and the *opad* is to use different keys in the inner and outer hashes.

A rationale for the design can be found in [53].

2.2.2.4 Should HMAC be used with SHA-3?

HMAC is designed to be usable with any hash function. However, its purpose is to address weaknesses of hash functions that could affect the security of MAC constructions, such as the loop continuation property of the Merkle–Damgård construction.

As we saw in Section 2.2.1.2, SHA-3 is part of Keccak, which uses the sponge construction instead of the Merkle-Damgård construction. The sponge construction has no known vulnerabilities at this time, so it may not be necessary to use HMAC with SHA-3. It may instead by OK to compute a MAC as

$$MAC(K, M) = Hash(K \mid M),$$

which would be more efficient than the nested hashing of HMAC.

The design and security section of the Keccak.team web site states that “Unlike SHA-1 and SHA-2, Keccak does not have the length-extension weakness, hence does not need the HMAC nested construction. Instead, MAC computation can be performed by simply prepending the message with the key.” This is a fair statement. But the fact that the sponge construction does not have known weaknesses at this time does not mean that weakness will not be found in the future. The HMAC construction adds a strong layer to of security to the security provided by the underlying has function. And defense-in-depth is a good security principle. So, if performance is not an overriding concern, it may be a good idea to use HMAC with SHA-3.

2.2.3 Asymmetric signature schemes

An asymmetric signature scheme consists of a key generation algorithm, a signing algorithm, and a verification algorithm. The key generation algorithm produces a private key and a public key. (In cryptography, the private key is called a secret key, but in technology the term private key is more commonly used.) The signing algorithm takes as inputs the private key and a message, and outputs a signature. The verification algorithm takes as inputs the public key, a message and a purported signature and produces a Boolean output, either “true” to accept the signature, or “false” to reject it.

A formal definition of an asymmetric signature scheme using the system parameterization methodology discussed in Section 2.1.5 can be found in [1, §13.1].

An asymmetric signature scheme is deemed secure against chosen message attack if an adversary that uses an efficient algorithm has a negligible probability of outputting a forgery after repeatedly asking a challenger who knows the private key for signatures of messages chosen by the adversary. Definitions of “efficient algorithm” and “negligible probability” in the context of system parameterization can be found in Section 2.1.5. There is no explicit limit on the number of queries that the adversary can make, only an implicit limit due to the fact that the running time of the algorithm used by the adversary is polynomial in the security parameter, and making queries takes time.

2.2.3.1 EUF-CMA vs SUF-CMA security

Two definitions of “forgery” can be used in conjunction with the above definition of security [54]. In the most commonly used definition, a forgery is a signature on a message that has not been submitted in a query to the challenger. A signature on a message submitted in a query that is different from the signature in the response to the query is not deemed a forgery. Security with this definition of forgery is called existential unforgeability under chosen message attack (EUF-CMA). A malleable signature scheme where an adversary can modify a signature without knowing the private key may qualify as providing EUF-CMA security.

In the alternative definition, a forgery is a message-signature pair that has not been returned by the challenger in response to a query. Security with this definition of forgery is called strong existential unforgeability under chosen message attack (SUF-CMA). A malleable signature scheme where an adversary can modify a signature without knowing the private key does not qualify as providing EUF-CMA security. Formal definitions of EUF-CMA and SUF-CMA in the context of system parameterization can be found in [1, §13.1.1].

We shall see in Section 2.2.3.3.3 that ECDSA is not SUF-CMA secure.

2.2.3.2 Discrete-log signature schemes in cyclic groups

The use of the discrete log assumption as the basis for the construction of asymmetric signatures originated with Schnorr's signature scheme [55], which was followed by NIST's Digital Signature Algorithm (DSA), now deprecated [18, §4]. Boneh and Shoup [1, §19.3] attribute the choice by NIST of the "more ad-hoc" DSA as the US federal signature standard to the fact that Schnorr signatures were patent-protected. Schnorr and DSA signatures both rely on the difficulty of computing discrete logs in a Schnorr group, i.e. in a prime order subgroup of the multiplicate group of the field $\mathbb{Z}_p^*$. The security strength provided by the discrete log assumption in a Schnorr group is discussed above in Section 2.1.3.2.3.

NIST later ported DSA from Schnorr groups to groups of points of elliptic curves and referred to the resulting signature scheme as ECDSA. The original version of ECDSA in FIPS 186-4 [42, §6] required the use of NIST-recommended curves, listed in Appendix D, that included Koblitz curves over binary fields and Weierstrass curves over prime fields with cofactor 1, i.e. with groups of points of prime order. Curves over binary fields have now been deprecated generally in cryptography at large, but NIST's curves P-256, P-384 and P-521, a.k.a. secp256r1, secp384r1 and secp521r1, are commonly used today. The security strength provided by the discrete log assumption in elliptic curve groups is discussed above in Section 2.1.3.2.4.

More recently, Bernstein and colleagues have designed a higher performance signature scheme called EdDSA [56] [57] that has been published by the IETF in the informational RFC 8032 [58] and standardized by NIST in FIPS 186-5 [18, §7]. EdDSA has two variants, Ed25519 and Ed448, which use two twisted Edwards curves, called Edwards25519 [59, §3.2.3.1] and Edwards448 [59, §3.2.3.2], that are birationally equivalent to the Montgomery curves Curve25519 (discussed above in Section 2.1.5.6.2) and Curve 448.

In the latest version of ECDSA [18, §6], the list of recommended curves, now published in SP 800-186 [59], includes the Montgomery curves Curve25519 and Curve448, the twisted Edwards curves Edwards25519 and Edwards448, and mappings of Curve25519 and Curve448 to the short Weierstrass model. ECDSA has also incorporated as an option a feature found in EdDSA, viz. the deterministic generation of the per-message secret.

2.2.3.2.1 Unified notation for facilitating comparison of different discrete log signature schemes

To facilitate comparisons between schemes, the following notations and conventions are used throughout Sections 2.2.3.2.2-4:

- **G** is a cyclic group and **H** is a subgroup of **G** of prime order q.
- Departing from tradition, but following [1, §19.3], we use multiplicative notation for all groups, even for groups of points of elliptic curves. Elements of $\mathbb{Z}_q$ are still called scalars, and operation of a scalar on a group element is called scalar exponentiation.
- **h** is a function that takes multiple arguments, converts them to bit strings, concatenates the bit strings, and applies a cryptographic hash function to the concatenation.
- In DSA and ECDSA, **s** is a function takes as argument an element of **G** and returns a scalar.
- Upper-case variables denote group elements; in particular, G always denotes a generator of **H**.
- The lower-case variable m always denotes the message to be signed, and other lower-case variables denote scalars or other integers.
- All operations on lower case variables take place in $\mathbb{Z}_q$, i.e. they are performed modulo q.

2.2.3.2.2 The Schnorr signature scheme

Parameters

- **G** is $\mathbb{Z}_p^*$, the multiplicative group of integers modulo a prime p.
- **H** is the unique subgroup of **G** of order q, where q is a large prime that divides $p-1$; $i.e.$, **H** is what is now called a *Schnorr group*. But Schnorr's paper [55] mentions the possibility of using elliptic curve groups.
- G is an agreed-upon generator of **H**. The cofactor clearing of Theorem 15 can be used to construct a generator, and the membership test of Theorem 14 can be used to verify that a purported generator of **H** is indeed in **H**.
- **h** is an unspecified cryptographic hash function, which must be a one-way function. Schnorr argues that it may not need to be collision resistant.

Key pair generation

1. Generate a random scalar d $\in \mathbb{Z}_q$ to be used as the private key.
2. Compute the public key $Q = G^{-d}$.

Signature generation

1. Let m be the message to be signed.
2. Generate a random scalar $r \in \mathbb{Z}_q$.

3. Compute $R = G^r$.
4. Compute $e = \mathbf{h}(R, m)$.
5. Compute $\mathrm{s = r + de}$.
6. The signature is (e, s).

Signature verification

1. Validate the verification equation

Verification equation

$$\mathrm{e} = \mathrm{h}(G^s Q^e, m)$$

Proof of correctness

(Reasoning backwards from the verification equation)

	$e \overset{?}{=} \mathbf{h}(G^s Q^e, m)$
Expanding *e*:	$\boldsymbol{h}(R, m) \overset{?}{=} \boldsymbol{h}(G^s Q^e, m)$
	$R \overset{?}{=} G^s Q^e$
Expanding *R*:	$G^r \overset{?}{=} G^s Q^e$
Expanding *s*:	$G^r \overset{?}{=} G^{r+de} Q^e$
Expanding *Q*:	$G^r \overset{?}{=} G^{r+de} G^{-de}$
Simplifying:	$G^r = G^r$

2.2.3.2.3 DSA and ECDSA

DSA and ECDSA have different parameters but, in the unified notation, the specifications of their algorithms for key pair generation, signature generation and signature verification are identical.

DSA parameters

- As in the Schnorr signature scheme, **G** is $\mathbb{Z}_p^*$, the multiplicative group of integers modulo a prime p, and **H** is the unique subgroup of **G** of order q, where q is a large prime that divides $p - 1$. **H** is thus a Schnorr group.
- As in the Schnorr scheme, G is an agreed-upon generator of **H**. The cofactor clearing of Theorem 15 can be used to construct a generator, and the membership test of Theorem 14 can be used to verify that a purported generator of **H** is indeed in **H**.
- **h** is a NIST-approved hash function.
- The function **s** takes as input an element of $\mathbb{Z}_p^*$, i.e. a congruence class of integers modulo p, obtains the representative of the class in the interval $[0, p)$ by reducing any other representative modulo p, and reduces the result modulo q.

ECDSA parameters

- **G** is the group of points of one of the NIST-recommended curves listed in [59]. Curves over binary fields are listed but deprecated.
- **H** is a subgroup of **G** of prime order q. The cofactor of **H** in **G** is 1 in the original prime-field curves P-192 (now deprecated), P-224, P-256, P-384 and P-521. The cofactor is 8 in Edwards25519, Curve25519 and W-25519, and 4 in Edwards448, Curve448 andW-448. Although Curve25519, W-25519, Curve448 and W-448 are listed in [59], they are qualified as "alternative representations included for implementation flexibility" that are "not to be used for ECDSA or EdDSA directly". The curves Edwards25519 and Edwards411, also used in EdDSA, do not have any qualifications or restrictions.
- G is an agreed-upon base point of the curve, stipulated by the curve specification.
- **h** is a NIST-approved hash function.
- The function **s** takes as input an element of **G**, which is a point of an elliptic curve over a prime field $\mathbb{F}_p$, obtains the x coordinate of that point, which is an element of $\mathbb{F}_p$, i.e. of a congruence class of integers modulo p, obtains the representative of that class in the interval $[0, p)$ by reducing modulo p any other representative, and reduces the result modulo q. We shall see below how the use of the x coordinate to map a point to a scalar causes ECDSA to lack SUF-CMA security.

DSA and ECDSA key pair generation

1. Generate a random scalar d $\in \mathbb{Z}_q$ to be used as the private key.
2. Compute the public key $Q = G^d$.

DSA and ECDSA signature generation

1. Let m be the message to be signed.
2. Generate a per-message secret k $\in \mathbb{Z}_q$, either at random as specified in [18, §A.3.1] or [18, §A.3.2], or deterministically by deriving it from the private key and the message as specified in [18, §A.3.3].
3. Compute $R = G^k$.
4. Compute $\mathrm{r} = \mathrm{s}(R)$
5. Compute $\mathrm{s} = k^{-1}(h(m) + dr)$
6. The signature is (r, s)

DSA and ECDSA signature verification

1. Compute $\mathrm{u} = \mathrm{h}(m)s^{-1}$
2. Compute $v = rs^{-1}$
3. Compute $R' = G^u Q^v$
4. Compute $r' = \mathrm{s}(R')$
5. Validate the verification equation

Verification equation

$$r = r'$$

Proof of correctness

(Reasoning backwards from the verification equation)

	$r \overset{?}{=} r'$
Expanding *r* and *r'*:	$\boldsymbol{s}(R) \overset{?}{=} \boldsymbol{s}(R')$
	$R \overset{?}{=} R'$
Expanding R and R':	$G^k \overset{?}{=} G^u Q^v$
Expanding Q:	$G^k \overset{?}{=} G^u G^{dv}$
Expanding *u* and *v*:	$G^k \overset{?}{=} G^{\boldsymbol{h}(m)s^{-1}} G^{drs^{-1}}$
Grouping:	$G^k \overset{?}{=} G^{(h(m)+dr)s^{-1}}$
Expanding *s*:	$G^k \overset{?}{=} G^{(\boldsymbol{h}(m)+dr)\left(k^{-1}(\boldsymbol{h}(m)+dr)\right)^{-1}}$
Simplifying:	$G^k = G^k$

Remark: ECDSA lacks SUF-CMA

The distinction between EUF-CMA and SUF-CMA is explained above in Section 2.2.3.1. ECDSA lacks SUF-CMA security because $-s$ can be substituted for s in the signature verification process without causing the process to fail. The substitution causes step 1 to produce $-u$, step 2 to produce $-v$, and step 3 to produce $(R')^{-1}$. And $s(R') = s((R')^{-1})$ because R' and $(R')^{-1}$ are symmetric across the x axis (remember we are using multiplicative notation for the group of points), and thus have the same *x* coordinate.

2.2.3.4 EdDSA

EdDSA has a variant called Ed25519 that uses the curve Edwards25519 and a variant called Ed448 that uses the curve Edwards448. Here we describe the Ed25519 variant. Information about the Ed448 variant can be found in the EdDSA specification at [18, §7] and inRFC 8032 [58].

2.2.3.4.1 Cofactor clearing and clamping

Edwards25519 is not an elliptic curve of prime order. Its group has a subgroup of prime order with cofactor 8. This motivates two controversial features of the EdDSA specification:

1. In the verification equation as specified in [56], the cofactor is cleared from the group elements (by multiplying them by 8 in additive notation). However, this is not necessary because the equation with cofactor clearing is implied by the by the equation without cofactor clearing. That cofactor clearing is unnecessary is acknowledged in RFC 8302 [58, §8.3] and the NIST specification [18, §7.7], so we do not include cofactor clearing in the signature verification algorithm specified below
2. The private key, represented as a 32-octet little-endian integer, is multiplied by 8 by setting the first three bits of the first octet to zero, as part of a bitwise

operation called clamping that also includes setting the last bit of the last octet to zero and the second to last bit of the last octet to one. The purpose of clamping is not explained in the specification, and it is argued in [60, §4.2.3] that it may not be necessary. We do include the clamping step in the signature generation algorithm below.

2.2.3.4.2 Generation of the private key and the per-message secret

In Ed25519, the per-message secret is generated deterministically by calling SHA-512 on the concatenation of a 32-octed string called *hdigest2* and the message itself. The string *hdigest2* is the right half of a 64-octet string that is the result of calling SHA-512 on a 32-octet root secret. The left half of the 64-octed string, called hdigest1, is clamped, and the result of the clamping, treated as a little-endian integer, is used as the exponent of a scalar exponentiation on the base point of the curve to obtain the public key.

In [56], Bernstein et al. refer to the root secret as the "secret key", and RFC 8032 [58] and NIST [18, §7] equivalently refer to it as the "private key". This is an unfortunate choice of terminology. The root secret is not what is clamped, is not the discrete log of the public key, and is not the analog of the private key of ECDSA and the Schnorr signature scheme; hence it should not be called the secret or private key. When such terminology is used, EdDSA seems radically different from ECDSA and the Schnorr signature scheme, when in fact it is derived from Schnorr and very similar to ECDSA. Following Brendel et al. [60], we use instead the term "private key" and the variable name *d* to refer to the result of the clamping.

2.2.3.4.3 The Ed25519 signature scheme

Parameters

- **G** is the group of points of the curve Edwards25519 specified in [59, §3.2.3.1]
- **H** is the subgroup of **G** of prime order
  ```
        2252  x14def9de a2f79cd6 5812631a 5cf5d3ed,
  ```
  whose cofactor in **G** is 8.
- *G* is the base point with *x* coordinate
```
0x216936d3 cd6e53fe c0a4e231 fdd6dc5c 692cc760 9525a7b2 c9562d60 8f25d51a
```
  and *y* coordinate
```
0x66666666 66666666 66666666 66666666 66666666 66666666 66666666 66666658.
```
- **h** is SHA-512.

Key pair generation

1. Generate a random 32-octet string to be used as the root secret w.
2. Compute the private key $d = clamp\left(lefthalf\big(h(w)\big)\right)$.
3. Compute the public key $Q = G^d$.

Deterministic generation of the per-message secret

1. Compute the per-message secret $r = h\big(righthalf\big(h(w)\big), m\big)$.

Signature generation

1. Compute $R = G^r$.
2. Compute $s = r + h(R, Q, m)d$.
3. The signature is (R, s).

Signature verification

1. Validate the verification equation.

Verification equation

$$G^s = RQ^{h(R,Q,m)}$$

Proof of correctness

(Reasoning backwards from the verification equation)

	$G^s \overset{?}{=} RQ^{\mathbf{h}(R,Q,m)}$
Expanding R and Q:	$G^s \overset{?}{=} G^r G^{d\mathbf{h}(R,Q,m)}$
Expanding s:	$G^{r+h(R,Q,m)d} \overset{?}{=} G^r G^{d\mathbf{h}(R,Q,m)}$
Rearranging:	$G^{r+h(R,Q,m)d} = G^{r+h(R,Q,m)d}$

References

[1] D. Boneh and V. Shoup, “A Graduate Course in Applied Cryptography,” 2024. [Online]. Available: https://toc.cryptobook.us/

[2] S. C. Pohlig and M. E. Hellman, "An Improved Algorithm for Computing Discrete Logarithms in GF(p) and Its Cryptographic Significance," IEEE Transactions on Information Theory, vol. 24, no. 1, pp. 106-110, 1978.

[3] A. J. Menezes et al, Handbook of Applied Cryptography, CRC Press, 1997.

[4] C. H. Lim and P. J. Lee, "A key recovery attack on discrete log-based schemes using a prime order subgroup," in Advances in Cryptology --- Crypto'97, 1997.

[5] Law, L et al., "An Efficient Protocol for Authenticated Key Agreement," Designs, Codes and Cryptography, vol. 28, pp. 119-134, March 2003.

[6] R. Zucherato, "RFC 2785: Methods for Avoiding the "Small-Subgroup" Attacks on the Diffie-Hellman Key Agreement Method for S/MIME," 2000.

[7] NIST Information Technology Laboratory, CSRC, "Glossary: Security Strength," [Online]. Available: https://csrc.nist.gov/glossary/term/security_strength

[8] E. Barker, "NIST SP 800-57 Part 1 Rev. 5: Recommendation for Key Management," [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf

[9] E. Barker et al, "NIST SP 800-56A Rev. 3: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography," [Online]. Available: https://csrc.nist.gov/pubs/sp/800/56/a/r3/final

[10] J. Rotman, Galois theory, New York: Springer, 1998.

[11] H. R. Haarberg, "The Number Field Sieve for Discrete Logarithms," Norwegian University of Science and Technology, Department of Mathematical Sciences, 2016. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2394427/14190_FULLTEXT.pdf

[12] A. Joux et al, "The Number Field Sieve in the Medium Prime Case," 2006. [Online]. Available: https://www.iacr.org/archive/crypto2006/41170323/41170323.pdf

[13] Wikipedia, "L-notation," [Online]. Available: https://en.wikipedia.org/wiki/L-notation

[14] McCann, "Asymptotic Notation: O(), o(), Ω(), ω(), and Θ()," 2009. [Online]. Available: https://www2.cs.arizona.edu/classes/cs345/summer14/files/bigO.pdf

[15] D. Adrian et al, "Weak Diffie-Hellman and the Logjam Attack," [Online]. Available: https://weakdh.org/

[16] D. Adrian and others, "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice," in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015.

[17] E. Ronen and A. Shamir, "Critical Review of Imperfect Forward Secrecy," [Online]. Available: https://web.archive.org/web/20211211100114/https:/www.wisdom.weizmann.ac.il/~eyalro/RonenShamirDhReview.pdf

[18] NIST Information Technology Laboratory, "FIPS 186-5: Digital Signature Standard (DSS)," [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf

[19] J. M. Pollard, "Monte Carlo methods for index computation (mod p)," in Math. Comp. 32 (1978), 918-924

[20] “Safe Curves: choosing safe curves for elliptic-curve cryptography,” [Online]. Available: https://safecurves.cr.yp.to/.

[21] D. Hankerson et al, Guide to Elliptic Curve Cryptography, Springer-Verlag, 2004.

[22] L. C. Washington, Elliptic Curves, Number Theory and Cryptography, CRC Press, 2008.

[23] Wikibooks, "Cryptography/Prime Curve/Standard Projective Coordinates," [Online]. Available: https://en.wikibooks.org/wiki/Cryptography/Prime_Curve/Standard_Projective_Coordinates.

[24] Wikibooks, "Cryptography/Prime Curve/Jacobian Coordinates," [Online]. Available: https://en.wikibooks.org/wiki/Cryptography/Prime_Curve/Jacobian_Coordinates.

[25] Wikibooks, "Cryptography/Prime Curve/Affine Coordinates," [Online]. Available: https://en.wikibooks.org/wiki/Cryptography/Prime_Curve/Affine_Coordinates.

[26] I. R. Shafarevich, Basic Algebraic Geometry, Book 1: Varieties in Projective Space, Third Edition, Springer, 2013.

[27] P. L. Montgomery, "Speeding the Pollard and Elliptic Curve Methods," Mathematics of Computation, vol. 48, no. 177, pp. 243-264, 1987.

[28] H. M. Edwards, "A normal form for ellliptic curves," Bulletin of the American Mathematical Society, vol. 44, pp. 393-422, 2007.

[29] D. J. Bernstein, "Curve25519: New Diffie-Hellman Speed Records," in Public Key Cryptography - PKC 2006.

[30] D. J. Bernstein and T. Lange, "Faster Addition and Doubling on Elliptic Curves," in Advances in Cryptology – ASIACRYPT 2007.

[31] D. J. Bernstein et al, "Twisted Edwards Curves," in Progress in Cryptology – AFRICACRYPT 2008.

[32] C. Costello and B. Smith, "Montgomery curves and their arithmetic," J Cryptogr Eng, vol. 8, pp. 227-240, September 2018.

[33] V. S. Miller, "Use of Elliptic Curves in Cryptography," in CRYPTO '85: Advances in Cryptology, 1985.

[34] D. J. Bernstein, "25519 naming," 26 August 2014. [Online]. Available: https://mailarchive.ietf.org/arch/msg/cfrg/-9LEdnzVrE5RORux3Oo_oDDRksU/.

[35] D. Stebila and N. Thériault, "Unified Point Addition Formulæ and Side-Channel Attacks," in Cryptographic Hardware and Embedded Systems - CHES 2006.

[36] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in Advances in Cryptology - CRYPTO '96.

[37] Kocher, P. et al, "Differential Power Analysis," in Advances in Cryptology — CRYPTO' 99.

[38] F. Corella and K. Lewison, "An Omission-Tolerant Cryptographic Checksum," 2019. [Online]. Available: https://eprint.iacr.org/2019/192.

[39] Wikipedia, "Comparison of cryptographic hash functions," [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_cryptographic_hash_functions.

[40] Guido Bertoni et al, "The sponge and duplex constructions," [Online]. Available: https://keccak.team/sponge_duplex.html.

[41] NIST Information Technology Laboratory, "Hash Functions," [Online]. Available: https://csrc.nist.gov/projects/hash-functions.

[42] NIST Information Technology Laboratory, "FIPS 180-4 Secure Hash Standard (SHS)," 2015. [Online]. Available: https://csrc.nist.gov/pubs/fips/180-4/upd1/final.

[43] NIST Information Technology Laboratory, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," 2015. [Online]. Available: https://csrc.nist.gov/pubs/fips/202/final.

[44] F. Corella, "Cryptographic Module Standards at a Crossroads after Snowden's Revelations - Repeating the Dual EC DRBG mistake," 12 November 2015. [Online].

[45] Guido Bertoni et al, "Keccak," [Online]. Available: https://keccak.team/index.html.

[46] A. M. Antonopoulos, Mastering Bitcoin: Unlocking Digital Crypto-Currencies., O'Reilly Media, 2014.

[47] Ethereum, "Merkle Patricia Trie (sic)," [Online]. Available: https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/.

[48] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in Advances in Cryptology – CRYPTO '87, 1988.

[49] B. Laurie et al, "RFC 9162: Certificate Transparency Version 2.0," 2021. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc9162.

[50] H. Krawczyk et al, "RFC 2104: HMAC: Keyed-Hashing for Message Authentication," [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2104.

[51] NIST Information Technology Laboratory, "Glossary - Message Authentication Code (MAC) algorithm," [Online]. Available: https://csrc.nist.gov/glossary/term/message_authentication_code_algorithm.

[52] Wikipedia, "Merkle–Damgård construction," [Online]. Available: htps://en.wikipedia.org/wiki/Merkle%E2%80%93Damg%C3%A5rd_construction.

[53] M. Bellare et al, "Message Authentication using Hash Functions— The HMAC Construction," CryptoBytes, vol. 2, no. 1, 1996.

[54] M. Green, "EUF-CMA and SUF-CMA," [Online]. Available: https://blog.cryptographyengineering.com/euf-cma-and-suf-cma/.

[55] C. P. Schnorr, "Efficient Signature Generation by Smart Cards," J. Cryptology, vol. 4, p. 161–174, 1991.

[56] Bernstein, D. J. et al, "High-speed high-security signatures," Journal of Cryptographic Engineering, vol. 2, no. 2, pp. 77-89, 2012.

[57] D. J. Bernstein, "EdDSA for more curves," 2016. [Online]. Available: https://ed25519.cr.yp.to/eddsa-20150704.pdf.

[58] S. Josefsson and I. Liusvaara, "RFC 8032: Edwards-Curve Digital Signature Algorithm (EdDSA)," January 2017. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8032.

[59] Chen, L. et al, "NIST SP 800-186: Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters," February 2023. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/186/final.

[60] J. Brendel et al, "The Provable Security of Ed25519: Theory and Practice," in IEEE Symposium on Security and Privacy, 2021.