

Cryptographic Engineering

working draft

Matthew D. Green
Johns Hopkins University

Excerpt: Chapter Chapter 4, Mathematical basics

§4.4.3 Cyclic groups (book pages 4-8)

Posted for 601.445/601.645 Practical Cryptographic Systems, Fall 2026
Lecture 7, Wed Sep 23: public-key cryptography I (groups, Diffie–Hellman)

<https://practicalcrypto.org>

This is an excerpt from an unfinished book draft: expect gaps, rough edges, and errors, and note that section numbers and page numbers may change in later drafts. Cross-references to other chapters point to material that is not included here. Copyright © 2026 Matthew D. Green. All rights reserved. Posted for the students of the course above; **please do not redistribute** — link to the course site instead. Built 2026-09-06.

The downside of this approach is that it can be less efficient: since there is approximately a 37% chance that we will discard a given value N , the number of random bits we must sample will also increase proportionally. Moreover, while we can calculate the *expected* number of values we will need to sample, the exact number we need can vary each time we perform the experiment. This can cause problems for algorithms that need exactly predictable runtimes or numbers of input bits.

Oversampling. While rejection sampling is guaranteed to eliminate the bias, an alternative approach is to simply *reduce* the bias to the point where it is small enough that it doesn’t trouble us. Instead of sampling N from a small set such as $[0, 15]$, one might sample a much larger number of bits, *e.g.*, $N \in \{0, 1\}^{32}$. We now compute $D = N \bmod 10$ as before. This still yields a bias: if N is uniform in the range $[0, 4294967289]$, then D will produce any digit in $[0, 9]$ with equal probability. If $N \in [4294967290, 4294967295]$ then the resulting D will be in the range $[0, 5]$, leading to an increased probability of those digits occurring. However, note that this second outcome will occur less than 0.00000014% of the time. While the resulting D is not completely uniform, the overall level of bias is enormously reduced.

For elements like cryptographic keys and nonces, we can obtain “cryptographic” levels of security by oversampling by a factor of 2^λ where λ is a security parameter like 128–256. This means to generate an n -bit integer, you’d start with a random string of length $n + \lambda$.

4.4 Algebraic basics

4.4.1 Modular arithmetic

Cryptography makes extensive use of modular arithmetic. Let a, n be integers: the function $a \bmod n$ represents the operation of dividing a by n and outputting only the *remainder* from this division process. This is sometimes referred to as a *modular reduction*.

Consider two integers a, b that may or may not be equal. For some modulus n it can hold that $(a \bmod n) = (b \bmod n)$. Here we say that a, b are *congruent* modulo n , which we express using the notation $a \equiv b \bmod n$.

4.4.2 Finite Fields

4.4.3 Cyclic groups

Many public-key cryptographic algorithms rely on the use of *cyclic groups*, which can be constructed from ingredients such as (integer) finite fields, as well as specific subgroups of points on an elliptic curve.

From a mathematical perspective, a finite cyclic group is an Abelian group that possesses specific properties. The group comprises a finite set of elements $\mathbb{G}$, as well as a *group operation* (which we will denote by $\cdot : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}$) that operates over pairs of elements in $\mathbb{G}$. These must satisfy several axioms:

1. **Commutativity.** For all $a, b \in \mathbb{G}$ it holds that $a \cdot b = b \cdot a$.
2. **Identity element.** There exists an *identity element* $I \in \mathbb{G}$ such that for every $a \in \mathbb{G}$ it holds that $I \cdot a = a$.
3. **Associativity.** For all $a, b, c \in \mathbb{G}$ it holds that $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
4. **Inverses.** For each $a \in \mathbb{G}$, there exists an inverse b such that $a \cdot b = I$. We will denote the inverse of a as a^{-1} .
5. **Generator.** There exists a generator $g \in \mathbb{G}$ such that $\mathbb{G} = \{I, g, (g \cdot g), (g \cdot g \cdot g), \dots\}$, that is, beginning with I , repeatedly applying the group operation on g and the previous element will yield every element of $\mathbb{G}$.

Simple cyclic groups: $\mathbb{Z}_p^*$. A cyclic group can be constructed within a prime-order field, which means that it can be realized using modular arithmetic. Let p be a prime number. Now define $\mathbb{G} = \{1, 2, \dots, p-1\}$, *i.e.*, the elements of the group are the set of integers from 1 to $p-1$ inclusive. Define the group operation $a \cdot b$ to be multiplication modulo p , *i.e.*, the equation $ab \bmod p$. This “multiplicative” group is often referred to using the notation $\mathbb{Z}_p^*$.

The identity element of $\mathbb{Z}_p^*$ is 1, and it is relatively easy to see that $\mathbb{Z}_p^*$ satisfies the requirements of commutativity and associativity. To prove that inverses exist, consider that

Additive vs. multiplicative groups Cyclic group operations can be described using multiplicative notation, where the group operation $\cdot$ is treated as multiplication, and repeated multiplication is written as exponentiation, *e.g.*, $(a \cdot a \cdot a) = a^3$. Sometimes group operations can be written using additive notation, where the group operation is expressed as addition. Here repeated group operations are expressed as multiplication by some integer scalar, *e.g.*, $(a + a + a) = 3a$. In cryptography, the former notation is typically used for protocols implemented in finite fields where the group operation is modular multiplication, while the latter is used for elliptic curves where the group operation is “point addition.” Many academic papers opt to ignore the implementation details and choose one consistent notation for both settings, on the theory that this is much easier to read. Many battles have been fought over the choice between the two notational formats.

Efficient exponentiation and scalar multiplication. Many cryptographic algorithms rely on the ability to compute exponentiation (resp. scalar multiplication) on elements of cyclic groups. For example, the Diffie-Hellman key exchange in a multiplicative group requires the first party to compute $A = g^a$ where a is a very large secret integer.³ While

³Although we will use multiplicative notation in this section, everything here applies to additive groups if you simply replace the group operation with $+$ and exponentiation with scalar multiplication.

this could theoretically be calculated as $A = g \cdot g \cdot \dots \cdot g$, this would require computing a separate group operations, which is impractical when a is large, *e.g.*, $a \approx 2^{256}$.

Fortunately there is a more efficient approach. Consider a simple example where a is a power of 2, such as $a = 2^4$. Then instead of performing 16 separate group operations, we can compute $g^a = (((g^2)^2)^2)^2$, which only requires 4 group operations. If a is not a power of 2, then we can decompose the process into a series of squarings and multiplications. In fact, there is a simple algorithm called the *square and multiply* algorithm that handles this systematically:

From a complexity perspective, square and multiply (sometimes called “double and add” for additive groups) has asymptotic complexity $\mathcal{O}(\log a)$. Concretely, the number of group operations is between 1-2x the number of bits in the description of a .

Subgroups. Cyclic groups can embed other cyclic groups, which are called subgroups. Each subgroup consists of a subset of the elements of the larger group, with one element serving as a generator of just that subgroup. An obvious example present in all cyclic groups is the *trivial subgroup* of order 1 that is generated by the identity element I , which also serves as its own generator. However, cyclic groups may have other subgroups.

The existence of subgroups can be important in some cryptographic protocols. For example, if either contribution g^a or g^b of the Diffie-Hellman protocol generates a small-order subgroup, then the resulting shared secret g^{ab} will be in this subgroup as well. This event can dramatically reduce the entropy of the resulting key, which motivates the use of cyclic groups that do not have large numbers of subgroups, *i.e.*, where the group order has a relatively low number of small factors. Clearly the simplest approach to ensuring this is to use cyclic groups of prime order p , where the only subgroup is the trivial subgroup.

Hard problems in cyclic groups. Cyclic groups are particularly interesting to cryptographers because we conjecture that they contain “hard” problems that cannot be solved efficiently by classical computers.

The discrete logarithm problem. Let $\mathbb{G}$ be a (multiplicative) cyclic group, and let g be a generator. Given an element $h \in \mathbb{G}$, the discrete logarithm is the problem of finding a scalar x such that $h = g^x$. This problem can also be described in additive notation.⁴

There is an obvious but *inefficient* algorithm for solving the discrete logarithm: simply try every possible value x from 0 to $\text{order}(\mathbb{G})$ and see if $h = g^x$. However if h is sampled at random, the best *classical* algorithms we know of in cryptographic groups are much too slow to be effective.⁵ This allows cryptographers to conjecture that calculations of the form $h \leftarrow g^a$ are a *one-way function* (§??): *i.e.*, for random a this can be efficiently computed in

⁴In elliptic curve groups the problem is the same, but it is known technically as the Elliptic Curve Discrete Logarithm Problem (ECDLP).

⁵More formally, known classical algorithms in these groups do not run in *polynomial time*.

one direction, but a reasonable attacker given (g, h) will be unable to *invert* the result to recover a .

Many widely-used protocols depend on the hardness of the discrete logarithm in finite fields and elliptic curves. The most famous of these is the Diffie-Hellman (resp. EC Diffie-Hellman) key exchange, which is entirely broken if discrete logarithms can be efficiently computed. Others include Elgamal encryption and signature schemes such as Schnorr, BLS, ECDSA and EdDSA.

The Diffie-Hellman problem. The Diffie-Hellman protocol relies on a conjectured hard problem known as the (computational) Diffie-Hellman problem (CDH). This problem requires that for random $a, b \in [0, \text{order}(\mathbb{G}))$ an attacker who is given (g, g^a, g^b) must solve for g^{ab} . While the Diffie-Hellman problem is thought to be hard in common groups, it can only be hard if the discrete logarithm is as well. For example, an attacker who can solve the discrete logarithm of g^a base g can simply compute $g^{ab} = (g^b)^a$. It is not known whether there is a way to solve the Diffie-Hellman problem in cryptographic groups without solving the discrete logarithm first, but to date nobody has found one.

The Decisional Diffie-Hellman problem. A related problem is the Decisional Diffie-Hellman problem (DDH). Here the attacker is given (g, g^a, g^b, T) where $T \in \mathbb{G}$ is either a random group element or else is equal to $T = g^{ab}$. The goal of the attacker is to determine which form T has, with probability meaningfully better than simple guessing. Any attacker who can solve CDH will easily solve DDH, but not (as far as we know) the other way round. DDH is used in classical Elgamal encryption and many privacy-preserving protocols.

The “Gap” CDH problem. A weird variant of the computational Diffie-Hellman problem is used to prove security for many public-key encryption systems. This problem is exactly the same as CDH, with a twist: in this problem, the attacker is also given access to an “oracle” that solves the Decisional Diffie-Hellman problem.⁶ This may seem weird and absurd! This assumption is used to prove security for many authenticated Elgamal constructions such as ECIES §7.3.3.

The quantum menace. An unfortunate problem with this discussion is that we *know* there are efficient algorithms for computing the discrete logarithm in the groups we use for cryptography. Fortunately for the Internet, these algorithms require sophisticated quantum computers that we do not yet know how to build. For many years cryptographers have chosen to ignore this problem, and to some extent we still do: most of the Internet’s public-key cryptography is still vulnerable to quantum algorithms. However, in recent years scientists, industry and government have been pushing to deploy new algorithms that are not vulnerable to these attacks. See §?? for a detailed discussion of post-quantum cryptography (PQC).

⁶More precisely, the attacker can send in chosen inputs of the form $(g, g^{\hat{a}}, g^{\hat{b}}, \hat{T})$ and the oracle will return 1 if $\hat{T} = g^{\hat{a}\hat{b}}$ and 0 otherwise.

Classical attacks on Discrete Logarithm. Until quantum computers are available, the major concern for discrete-logarithm cryptography is the existence of various classical attacks. While new attacks can be discovered (or improved) at any time, there are two basic attacks we think about:

1. **Generic attacks.** One class of attacks operates over “generic” groups, which means that they will work in any cyclic group at all: these algorithms use the group operation and basic operations such as comparison, but do not exploit any deeper mathematical structure of the group. The benefit of these algorithms is that we know they will work in both finite-field and elliptic curve groups. The disadvantage is that they do not take advantage of more subtle algebraic structure of the group elements, which might enable faster attacks. In this setting, we know that the best possible algorithm for solving discrete logarithms is Pollard’s rho algorithm [263], which requires approximately $\mathcal{O}(\sqrt{N})$ group operations for groups of order N . This means that if $\text{order}(\mathbb{G}) \approx 2^{256}$ then the best attack requires approximately 2^{128} group operations, which we refer to as “128-bit security.” This attack is in fact the best known attack against properly-designed elliptic curve subgroups, which explains why those groups are able to use relatively short keys and ciphertexts.
2. **The index calculus.** Contemporaneous with the publication of Diffie and Hellman’s protocol, a new class of algorithms was discovered and optimized for finding discrete logarithms in finite fields [262, 6, 156]. This *index calculus* method relies on the fact that the group in this field-based setting has special algebraic structure. The resulting algorithms have sub-exponential (but still super-polynomial) complexity, and are much faster than generic attacks in practice. To compensate for increasingly well-optimized new attacks, practitioners were forced to dramatically increase the size of finite fields used in early cryptographic implementations, from *e.g.*, 256 bits to as much as 3,072 bits in order for attacks to require the same 2^{128} operations, usually called the “128-bit security level.” This development encouraged the widespread use of elliptic curve cryptography, since index calculus methods do not appear to work in common elliptic curve subgroups.⁷

For the most part, the industry and public cryptography community is fairly comfortable using 256-bit elliptic curve groups (such as NIST P-256 or Curve25519) and even 2048-bit finite field groups for most purposes. In 2015, for reasons unknown, the NSA suddenly deprecated P-256 for use in securing classified material, and upgraded this application to require the P-384 curve. Nobody knows why they did this, but see [187] for some excellent speculation.

⁷Some early elliptic curves were vulnerable to pairing-based attacks such as MOV [221], in which group elements were mapped to a related finite field. This sort of thing is why you should never build your own elliptic curves. See §7.3.9.