Operational notes

Document updated on **March 21, 2022**.

The following colors are **not** part of the final product, but serve as highlights in the editing/review process:

- text that needs attention from the Subject Matter Experts: Mirco, Anna,& Jan
- terms that have not yet been defined in the book
- text that needs advice from the communications/marketing team: Aaron & Shane
- text that needs to be completed or otherwise edited (by Sylvia)

Todo list

MoonMath manual

TechnoBob and the Least Scruples crew

March 21, 2022

Contents

Chapter 4

Algebra

Def Subgroup, Fundamental theorem of cyclic groups.

In the previous chapter, we gave an introduction to the basic computational skills needed for a pen-and-paper approach to SNARKs. This chapter provides a more abstract clarification of relevant mathematical terminology on **algebraic types** such as **groups**, **fields**, **rings** and similar.

In a nutshell, algebraic types define sets that are analogous to numbers in various aspects, in the sense that you can add, subtract, multiply or divide on those sets. We know many examples of sets that fall under those categories, such as natural numbers, integers, rational or the real numbers. In some sense, these are the most fundamental examples of such sets.

Papers on cryptography (and mathematical papers in general) frequently contain such terms, and it is necessary to get at least some understanding of these terms to be able to follow these papers. In this chapter, we therefore provide a short introduction to these concepts.

4.1 Groups

Groups are abstractions that capture the essence of mathematical phenomena, like addition and subtraction, multiplication and division, permutations, or symmetries.

To understand groups, let us think back to when we learned about the addition and subtraction of integers (also called whole numbers) in school. We have learned that, whenever we add two integers, the result is guaranteed to be an integer as well. We have also learned that adding zero to any integer means that "nothing happens", that is, result of the addition is the same integer we started with. Furthermore, we have learned that the order in which we add two (or more) integers does not matter, that operations within brackets should be computed before operations outside brackets, and that, for every integer, there is always another integer (the negative) such that we get zero when we add them together.

These conditions are the defining properties of a group, and mathematicians have recognized that the exact same set of rules can be found in very different mathematical structures. It therefore makes sense to give a formal definition of what a group should be, detached from any concrete examples. This lets us handle entities of very different mathematical origins in a flexible way, while retaining essential structural aspects of many objects in abstract algebra and beyond.

Distilling these rules to the smallest independent list of properties and making them abstract, we arrive at the definition of a group:

Definition 4.1.0.1. A **group** $(\mathbb{G}, \cdot)$ is a set $\mathbb{G}$, together with a **map** $\cdot$. The map, also denoted as $\mathbb{G} \times \mathbb{G} \to \mathbb{G}$ and called the **group law**, combines two elements of the set $\mathbb{G}$ into a third one such that the following properties hold:

- **Existence of a neutral element**: There is a $e \in \mathbb{G}$ for all $g \in \mathbb{G}$, such that $e \cdot g = g$ as well as $g \cdot e = g$.
- **Existence of an inverse**: For every $g \in \mathbb{G}$ there is a $g^{-1} \in \mathbb{G}$, such that $g \cdot g^{-1} = e$ as well as $g^{-1} \cdot g = e$.
- **Associativity**: For every $g_1, g_2, g_3 \in \mathbb{G}$ the equation $g_1 \cdot (g_2 \cdot g_3) = (g_1 \cdot g_2) \cdot g_3$ holds.

Rephrasing the abstract definition in layman's terms, a group is something where we can do computations in a way that resembles the behavior of the addition of integers. Specifically, this means we can combine some element with another element into a new element in a way that is reversible and where the order of combining elements doesn't matter.

Notation and Symbols 3. Let $(\mathbb{G}\cdot)$ be a finite group. If there is no risk of ambiguity (whether we are talking about agroup or a set), we frequently drop the symbol $\cdot$ and simply write $\mathbb{G}$ as the notation for the group, keeping the group law implicit.

As we will see in XXX, groups are heavily used in cryptography and in SNARKs. But let us look at some more familiar examples fist: add reference

Example 28 (Integer Addition and Subtraction). The set $(\mathbb{Z}, +)$ of integers with integer addition is the archetypical example of a group, where the group law is traditionally written as $+$ (instead of $\cdot$). To compare integer addition against the abstract axioms of a group, we first see that the neutral element e is the number 0, since $a + 0 = a$ for all integers $a \in \mathbb{Z}$. Furthermore, the inverse of a number is its negative counterpart, since $a + (-a) = 0$, for all $a \in \mathbb{Z}$. In addition, we know that $(a+b)+c = a+(b+c)$, so integers with addition are indeed a group in the abstract sense.

Example 29 (The trivial group). The most basic example of a group is group with just one element $\{\bullet\}$ and the group law $\bullet \cdot \bullet = \bullet$. Add real-life example of 0?

Commutative Groups When we look at the general definition of a group, we see that it is somewhat different from what we know from integers. We know that the order in which we add two integers doesn't matter, as, for example, $4+2$ is the same as $2+4$. However, we also know from example XXX that this is not the case for all groups. add reference

This means that groups where the order in which the group law is executed doesn't matter are a special subcase of groups called **commutative group**s. To be more precise, a group is called commutative if $g_1 \cdot g_2 = g_2 \cdot g_1$ holds for all $g_1, g_2 \in \mathbb{G}$.

Notation and Symbols 4. For commutative groups $(\mathbb{G}, \cdot)$, we frequently use the so-called **additive notation** $(\mathbb{G}, +)$, that is, we write $+$ instead of $\cdot$ for the group law, and $-g := g^{-1}$ for the inverse of an element $g \in \mathbb{G}$.

Example 30. Consider the group of integers with integer addition again. Since $a+b = b+a$ for all integers, this group is the archetypical example of a commutative group. Since there are infinitely many integers, $(\mathbb{Z}, +)$ is not a finite group.

Example 31. Consider our definition of modulo 6 residue classes $(\mathbb{Z}_6, +)$ as defined in the addition table from example 8. As we can see, the residue class 0 is the neutral element in modulo 6 arithmetics, and the inverse of a residue class r is given by $6 - r$, since $r + (6-r) = 6$, which is congruent to 0, since $6 \bmod 6 = 0$. Moreover, $(r_1 + r_2) + r_3 = r_1 + (r_2 + r_3)$ is inherited from integer arithmetic.

We therefore see that $(\mathbb{Z}_6, +)$ is a group, and, since the addition table in example 8 is symmetrical, we see $r_1 + r_2 = r_2 + r_1$, which shows that $(\mathbb{Z}_6, +)$ is commutative.

The previous example of a commutative group is a very important one for this book. Abstracting from this example and considering residue classes $(\mathbb{Z}_n, +)$ for arbitrary moduli n, it can be shown that $(\mathbb{Z}, +)$ is a commutative group with the neutral element 0 and the additive inverse $n - r$ for any element $r \in \mathbb{Z}_n$. We call such a group the **remainder class group** of modulus n.

Of particular importance for pairing-based cryptography in general and SNARKs in particular are so-called **pairing maps** on commutative groups. To be more precise, let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_3$ be three commutative groups. For historical reasons, we write the group law on $\mathbb{G}_1$ and $\mathbb{G}_2$ in additive notation and the group law on $\mathbb{G}_3$ in multiplicative notation. Then a **pairing map** is the following function:

$$e(\cdot,\cdot) : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_3 \tag{4.1}$$

This function takes pairs (g_1, g_2) (products) of elements from $\mathbb{G}_1$ and $\mathbb{G}_2$, and maps them to elements from $\mathbb{G}_3$, such that the **bilinearity** property holds:

Definition 4.1.0.2. **Bilinearity**
For all $g_1, g_1' \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$ we have $e(g_1 + g_1', g_2) = e(g_1, g_2) \cdot e(g_1', g_2)$ and for all $g_1 \in \mathbb{G}_1$ and $g_2, g_2' \in \mathbb{G}_2$ we have $e(g_1, g_2 + g_2') = e(g_1, g_2) \cdot e(g_1, g_2')$.

A pairing map is called **non-degenerat** if, whenever the result of the pairing is the neutral element in $\mathbb{G}_3$, one of the input values is the neutral element of $\mathbb{G}_1$ or $\mathbb{G}_2$. To be more precise, $e(g_1, g_2) = e_{\mathbb{G}_3}$ implies $g_1 = e_{\mathbb{G}_1}$ or $g_2 = e_{\mathbb{G}_2}$.

Informally speaking, bilinearity means that it doesn't matter if we first execute the group law on one side and then apply the bilinear map, or if we first apply the bilinear map and then apply the group law. Moreover, non-degeneracy means that the result of the pairing is zero if and only if at least one of the input values is zero.

Example 32. One of the most basic examples of a non-degenerate pairing involves $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_3$ all to be the groups of integers with addition $(\mathbb{Z}, +)$. Then the following map defines a non-degenerate pairing:

$$e(\cdot,\cdot) : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \; (a, b) \mapsto a \cdot b$$

Note that bilinearity follows from the distributive law of integers, since for $a, b, c \in \mathbb{Z}$, we have $e(a + b, c) = (a + b) \cdot c = a \cdot c + b \cdot c = e(a, c) + e(b, c)$ and the same reasoning is true for the second argument.

To the see that $e(\cdot,\cdot)$ is non-degenrate, assume that $e(a, b) = 0$. Then a$\cdot b = 0$ implies that a or b must be zero.

Exercise 26. Consider example 13 again and let $\mathbb{F}_5^*$ be the set of all remainder classes from $\mathbb{F}_5$ without the class 0. Then $\mathbb{F}_5^* = \{1, 2, 3, 4\}$. Show that $(\mathbb{F}_5^*, \cdot)$ is a commutative group.

check reference

Exercise 27. Generalizing the previous exercise, consider the general modulus n, and let $\mathbb{Z}_n^*$ be the set of all remainder classes from $\mathbb{Z}_n$ without the class 0. Then $\mathbb{Z}_n^* = \{1, 2, \ldots, n-1\}$. Provide a counter-example to show that $(\mathbb{Z}_n^*, \cdot)$ is not a group in general.

Find a condition such that $(\mathbb{Z}_n^*, \cdot)$ is a commutative group, compute the neutral element, give a closed form for the inverse of any element and prove the commutative group axioms.

Exercise 28. Consider the remainder class groups $(\mathbb{Z}_n, +)$ for some modulus n. Show that the map

$$e(\cdot,\cdot) : \mathbb{Z}_n \times \mathbb{Z}_n \to \mathbb{Z}_n \; (a, b) \mapsto a \cdot b$$

is bilinear. Why is it not a pairing in general and what condition must be imposed on n, such that the map will be a pairing?

Finite groups As we have seen in the previous examples, groups can either contain infinitely many elements (such as integers) or finitely many elements (as for example the remainder class groups $(\mathbb{Z}_n,+)$). To capture this distinction, a group is called a **finite group** if the underlying set of elements is finite. In that case, the number of elements of that group is called its **order**.

Notation and Symbols 5. Let $\mathbb{G}$ be a finite group. We write $ord(\mathbb{G})$ or $|\mathbb{G}|$ for the order of $\mathbb{G}$.

Example 33. Consider the remainder class groups $(\mathbb{Z}_6,+)$ from example 8 and $(\mathbb{F}_5,+)$ from example 13, and the group $(\mathbb{F}_5^*,\cdot)$ from exercise 26. We can easily see that the order of $(\mathbb{Z}_6,+)$ is 6, the order of $(\mathbb{F}_5,+)$ is 5 and the order of $(\mathbb{F}_5^*,\cdot)$ is 4.

check references to previous examples

To be more general, considering arbitrary moduli n, we know from Euclidean division that the order of the remainder class group $(\mathbb{Z}_n,+)$ is n.

Exercise 29. The RSA crypto system is based on a modulus n that is typically the product of two prime numbers of size 2048-bits. What is the approximate order of the rainder class group $(\mathbb{Z}_n,+)$ in this case?

RSA crypto system

size 2048-bits

rainder class group

Generators These are sets of elements that can be used to generate the entire group by applying the group law repeatedly to these elements or their inverses only. Generators are of particular interest when working with groups.

Of course, every group $\mathbb{G}$ has a trivial set of generators, when we just consider every element of the group to be in the generator set. The more interesting question is to find the smallest possible set of generators for a given group. Groups that have a single generator are particularly interesting from this perspective. These are groups containing an element $g \in \mathbb{G}$ such that every other element from $\mathbb{G}$ can be computed by the repeated combination of g or its inverse g^{-1}, but no other element. Groups with a single generator are called **cyclic groups**.

Example 34. The most basic example of a cyclic group is the group of integers with integer addition, $(\mathbb{Z},+)$. 1 is a single generator of $\mathbb{Z}$, since every integer can be obtained by repeatedly adding either 1 or its inverse -1 to itself. For example -4 is generated by -1, since $-4 = -1+(-1)+(-1)+(-1)$.

Example 35. Consider a modulus n and the remainder class groups $(\mathbb{Z}_n,+)$ from example 33. These groups are cyclic, with the generator 1, since every other element of that group can be constructed by repeatedly adding the remainder class 1 to itself. Since $\mathbb{Z}_n$ is also finite, we know that $(\mathbb{Z}_n,+)$ is a finite cyclic group of order n.

check reference

Example 36. Let $p \in \mathbb{P}$ be prime number and $(\mathbb{F}_p^*,\cdot)$ the finite group from exercise XXX. Then $(\mathbb{F}_p^*,\cdot)$ is cyclic and every element $g \in \mathbb{F}_q^*$ is a generator.

add reference

The discrete Logarithm problem Observe that, when $\mathbb{G}$ is a cyclic group of order n and $g \in \mathbb{G}$ is a generator of $\mathbb{G}$, then there is a map with respect to the generator g with the following properties:

$$g^{(\cdot)} : \mathbb{Z}_n \to \mathbb{G}\; x \mapsto g^x \tag{4.2}$$

In the map above, g^x means "multiply g x-times by itself" and $g^0 = e_{\mathbb{G}}$. This map, called the **exponential map**, has the remarkable property that it maps the additive group law of the remainder class group $(\mathbb{Z}_n,+)$ in a one-to-one correspondence to the group law of $\mathbb{G}$.

To see this, first observe that, since $g^0 := e_{\mathbb{G}}$ by definition, the neutral element of $\mathbb{Z}_n$ is mapped to the neutral element of $\mathbb{G}$, and, since $g^{x+y} = g^x \cdot g^y$, the map respects the group law.

Because the exponential map respects the group law, it doesn't matter if we do our computation in $\mathbb{Z}_n$ before we write the result into the exponent of g or afterwards: the result will be the same in both cases. This is usually referred to as doing computations “in the exponent". In cryptography in general, and in SNARK development in particular, we often perform computations “in the exponent” of a generator.

Example 37. Consider the multiplicative group $(\mathbb{F}_5^*, \cdot)$ from example 26. We know that $\mathbb{F}_5^*$ is a cyclic group of order 4, and that every element is a generator. If we choose $3 \in \mathbb{F}_5^*$, we then know that the following map respects the group law of addition in $\mathbb{Z}_4$ and the group law of multiplication in $\mathbb{F}_5^*$:

check reference

$$3^{(\cdot)} : \mathbb{Z}_4 \to \mathbb{F}_5^* \; x \mapsto 3^x$$

Let us now perform a computation in the exponent:

$$\begin{aligned} 3^{2+3-2} &= 3^3 \\ &= 2 \end{aligned}$$

This gives the same result as doing the same computation in $\mathbb{F}*_5$:

$$\begin{aligned} 3^{2+3-2} &= 3^2 \cdot 3^3 \cdot 3^{-2} \\ &= 4 \cdot 2 \cdot (-3)^2 \\ &= 3 \cdot 2^2 \\ &= 3 \cdot 4 \\ &= 2 \end{aligned}$$

Since the exponential map is a one-to-one correspondence that respects the group law, it can be shown that this map has an inverse with respect to the base g, called the **discrete logarithm map**:

$$log_g(\cdot) : \mathbb{G} \to \mathbb{Z}_n \; x \mapsto log_g(x) \tag{4.3}$$

Discrete logarithms are highly important in cryptography, because there are groups such that the exponential map and its inverse, the discrete logarithm, which are believed to be one-way functions, that is, while it is possible to compute the exponential map in polynomial time, computing the discrete log takes (sub)-exponential time. We have discussed this briefly following example 3.5 in the previous chapter, and will look at this and similar problems in more detail in the next section.

polynomial time

exponential time

TODO: Fundamental theorem of finite cyclic groups

4.1.1 Cryptographic Groups

In this section, we will look at families of groups that are believed to satisfy certain **computational hardness assumptions**, namely that a particular problem cannot be solved efficiently (where efficiently typically means “in polynomial time of a given security parameter") in the groups under consideration.

Example 38. To highlight the concept of the computational hardness assumption, consider the group of integers $\mathbb{Z}$ from example 3.5. One of the best known and most researched examples of computational hardness is the assumption that the factorization of integers into prime numbers cannot be solved by any algorithm in polynomial time with respect to the bit-length of the integer.

check reference

To be more precise, the computational hardness assumption of integer factorization assumes that, given any integer $z \in \mathbb{Z}$ with bit-length b, there is no integer k and no algorithm with the runtime complexity of $\mathcal{O}(b^k)$ that is able to find the prime numbers $p_1, p_2, \ldots, p_j \in \mathbb{P}$, such that $z = p_1 \cdot p_2 \cdot \ldots \cdot p_j$.

runtime complexity

This hardness assumption was proven to be false, since Shor's (1994) algorithm shows that integer factorization is at least efficiently possible on a quantum computer, since the runtime complexity of this algorithm is $\mathcal{O}(b^3)$. However, no such algorithm is known on a classical computer.

S: what does "efficiently" mean here?

In the realm of classical computers, however, we still have to call the non-existence of such an algorithm an "assumption" because, to date, there is no proof that it is actually impossible to find one. The problem is that it is hard to reason about algorithms that we don't know.

So, despite the fact that there is currently no known algorithm that can factor integers efficiently on a classical computer, we cannot exclude that such an algorithm might exist in principle, and that someone eventually will discover it in the future.

However, what still makes the assumption plausible, despite the absence of any actual proof, is the fact that, after decades of extensive search, still no such algorithm has been found.

In what follows, we will describe a few computational hardness assumptions that arise in the context of groups in cryptography, because we will refer to them throughout the book.

computational hardness assumptions

The discrete logarithm assumption The so-called discrete logarithm problem is one of the most fundamental assumptions in cryptography. To define it, let $\mathbb{G}$ be a finite cyclic group of order r and let g be a generator of $\mathbb{G}$. We know from 4.2 that there is a so-called exponential map $g^{(\cdot)} : \mathbb{Z}_r \to \mathbb{G} : x \mapsto g^x$, which maps the residue classes from module r arithmetic onto the group in a $1:1$ correspondence. The **discrete logarithm problem** is then the task of finding inverses to this map, that is, to find a solution $x \in \mathbb{Z}_r$ to the following equation for some given $h \in \mathbb{G}$:

check reference

$$h = g^x \tag{4.4}$$

In other words, the **discrete logarithm assumption (DL-A)** is the assumption that there exists no algorithm with running time polynomial in the"security parameter $log_2(r)$, that is able to compute some x if only h, g and g^x are given in $\mathbb{G}$. If this is the case for $\mathbb{G}$, we call $\mathbb{G}$ a **DL-A group**.

Rephrasing the previous definition into simple words, DL-A groups are believed to have the property that it is infeasible to compute some number x that solves the equation $h = g^x$ for a given h and g, assuming that the size of the group r is large enough.

Example 39 (Public key cryptography). One the most basic examples of an application for DL-A groups is in public key cryptography, where some pair $(\mathbb{G}, g)$ is publicly agreed on, such that $\mathbb{G}$ is a finite cyclic group of sufficiently large order r, where it is believed that the discrete logarithm assumption holds, and g is a generator of $\mathbb{G}$.

In this setting, a secret key is nothing but some number $sk \in \mathbb{Z}_r$ and the associated public key pk is the group element $pk = g^{sk}$. Since discrete logarithms are assumed to be hard, it is infeasible for an attacker to compute the secret key from the public key, since it is believed to be hard to find solutions x to the equation

$$pk = g^x$$

As the previous example shows, identifying DL-A groups is an important practical problem. Unfortunately, it is easy to see that it does not make sense to assume the hardness of the discrete logarithm problem in all finite cyclic groups: Counterexamples are common and easy to construct.

Example 40 (Modular arithmetics for Fermat's primes). It is widely believed that the discrete logarithm problem is hard in multiplicative groups $\mathbb{Z}_p^*$ of prime number modular arithmetics. However, this is not true in general. To see that, consider any so-called Fermat's prime, which is a prime number $p \in \mathbb{P}$, such that $p = 2^n + 1$ for some number n.

We know from XXX that in this case $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$ is a group with respect to integer multiplication in modular p arithmetics and since $p = 2^n + 1$, the order of $\mathbb{Z}_p^*$ is 2^n, which implies that the associated security parameter is given by $log_2(2^n) = n$.

add reference

We show that, in this case, $\mathbb{Z}_p^*$ is not a DL-A group, by constructing an algorithm, which is able compute some $x \in \mathbb{Z}_{2^n}$ for any given generator g and arbitrary element h of $\mathbb{F}_p^*$, such that

$$h = g^x$$

holds and the runtime complexity of the constructed algorithm is $\mathcal{O}(n^2)$, which is quadratic in the security parameter $n = log_2(2^n)$.

To define such an algorithm, let us assume that the generator g is a public constant and that a group element h is given. Our task is to compute x efficiently.

The first thing to note is that, since x is a number in modular 2^n arithmetic, we can write the binary representation of x as

$$x = c_0 \cdot 2^0 + c_1 \cdot 2^1 + \cdots + c_n \cdot 2^n$$

with binary coefficients $c_j \in \{0, 1\}$. In particular, x is an n-bit number if interpreted as an integer.

explain last sentence more

We then use this representation to construct an algorithm that computes the bits c_j one after another, starting at c_0. To see how this can be achieved, observe that we can determine c_0 by raising the input h to the power of 2^{n-1} in $\mathbb{F}_p^*$. We use the exponential laws and compute

$$\begin{aligned}
h^{2^{n-1}} &= (g^x)^{2^{n-1}} \\
&= \left(g^{c_0 \cdot 2^0 + c_1 \cdot 2^1 + \ldots + c_n \cdot 2^n}\right)^{2^{n-1}} \\
&= g^{c_0 \cdot 2^{n-1}} \cdot g^{c_1 \cdot 2^1 \cdot 2^{n-1}} \cdot g^{c_2 \cdot 2^2 \cdot 2^{n-1}} \cdots g^{c_n \cdot 2^n \cdot 2^{n-1}} \\
&= g^{c_0 2^{n-1}} \cdot g^{c_1 \cdot 2^0 \cdot 2^n} \cdot g^{c_2 \cdot 2^1 \cdot 2^n} \cdots g^{c_n \cdot 2^{n-1} \cdot 2^n}
\end{aligned}$$

Now, since g is a generator and $\mathbb{F}_p^*$ is cyclic of order 2^n, we know $g^{2^n} = 1$ and therefore $g^{k \cdot 2^n} = 1^k = 1$. From this, it follows that all but the first factor in the last expression are equal to 1 and we can simplify the expression into

$$h^{2^{n-1}} = g^{c_0 2^{n-1}}$$

Now, in case $c_0 = 0$, we get $h^{2^{n-1}} = g^0 = 1$ and in case $c_0 = 1$, we get $h^{2^{n-1}} = g^{2^{n-1}} \neq 1$ (To see that $g^{2^{n-1}} \neq 1$, recall that g is a generator of $\mathbb{F}_p^*$ and hence is cyclic of order 2^n, which implies $g^y \neq 1$ for all $y < 2^n$).

So raising h to the power of 2^{n-1} determines c_0, and we can apply the same reasoning to the coefficient c_1 by raising $h \cdot g^{-c_0 \cdot 2^0}$ to the power of 2^{n-2}. This approach can then be repeated until all the coefficients c_j of x are found.

Assuming that exponentiation in $\mathbb{F}_p^*$ can be done in logarithmic runtime complexity $log(p)$, it follows that our algorithm has a runtime complexity of $\mathcal{O}(log^2(p)) = \mathcal{O}(n^2)$, since we have to execute n exponentiations to determine the n binary coefficients of x.

From this, it follows that whenever p is a Fermat's prime, the discrete logarithm assumption does not hold in F_p^*.

The decisional Diffie–Hellman assumption To describe the decisional Diffie–Hellman assumption, let $\mathbb{G}$ be a finite cyclic group of order r and let g be a generator of $\mathbb{G}$. The DDH assumption then assumes that there is no algorithm that has a polynomial runtime complexity in the security parameter $s = log(r)$ that is able to distinguish the so-called DDH- triple (g^a, g^b, g^{ab}) from any triple (g^a, g^b, g^c) for randomly and independently chosen parameters $a, b, c \in \mathbb{Z}_r$. If this is the case for $\mathbb{G}$, we call $\mathbb{G}$ a **DDH-A group**.

It is easy to see that DDH-A is a stronger assumption than DL-A, in the sense that the discrete logarithm assumption is necessary for the decisional Diffie–Hellman assumption to hold, but not the other way around.

To see why, assume that the discrete logarithm assumption does not hold. In that case, given a generator g and a group element h, it is easy to compute some residue class $x \in \mathbb{Z}_p$ with $h = g^x$. Then the decisional Diffie–Hellman assumption cannot hold, since given some triple (g^a, g^b, z), one could efficiently decide whether $z = g^{ab}$ by first computing the discrete logarithm b of g^b, then computing $g^{ab} = (g^a)^b$ and deciding whether or not $z = g^{ab}$.

On the other hand, the following example shows that there are groups where the discrete logarithm assumption holds but the decisional Diffie–Hellman assumption does not hold:

Example 41 (Efficiently computable pairings). Let $\mathbb{G}$ be a finite, cyclic group of order r with generator g, such that the discrete logarithm assumtion holds and such that there is a pairing map $e(\cdot,\cdot) : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ for some target group $\mathbb{G}_T$ that is computable in polynomial time of the parameter $log(r)$.

In a setting like this, it is easy to show that DDH-A cannot hold, since given some triple (g^a, g^b, z), it is possible to decide in polynomial time w.r.t $log(r)$ whether $z = g^{ab}$ or not. To see that check

$$e(g^a, g^b) = e(g, z)$$

Since the bilinearity properties of $e(\cdot,\cdot)$ imply $e(g^a, g^b) = e(g, g)^{ab} = e(g, g^{ab})$ and $e(g, y) = e(g, y')$ implies $y = y'$ due to the non-degenerate property, the equality decides $z = e^{ab}$.

It follows that DDH-A is indeed weaker than DL-A and groups with efficient pairings cannot be DDH-A groups. The following example shows another important class of groups where DDH-A does not hold: multiplicative groups of prime number residue classes.

Example 42. Let p be a prime number and $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$ the multiplicative group of modular p arithmetics as in example XXX. As we have seen in XXX, this group is finite and cyclic of order $p-1$ and every element $g \neq 1$ is a generator. add reference

To see that $\mathbb{F}_p^*$ cannot be a DDH-A group, recall from XXX that the Legendre symbol $\left(\frac{x}{p}\right)$ of any $x \in \mathbb{F}_p^*$ is efficiently computable by Euler's formular. But the Legendre symbol of g^a reveals if a is even or odd. Given g^a, g^b and g^{ab}, one can thus efficiently compute and compare the least significant bit of a, b and ab, respectively, which provides a probabilistic method to distinguish g^{ab} from a random group element g^c. Legendre symbol Euler's formular These are only explained later in the text, ‘‘

The computational Diffie–Hellman assumption To describe the computational Diffie-Hellman assumption, let $\mathbb{G}$ be a finite cyclic group of order r and let g be a generator of $\mathbb{G}$. The computational Diffie–Hellman assumption assumes that, given randomly and independently chosen residue classes $a, b \in \mathbb{Z}_r$, it is not possible to compute g^{ab} if only g, g^a and g^b (but not a and b) are known. If this is the case for $\mathbb{G}$, we call $\mathbb{G}$ a CDH-A group.

In general, it is not known if CDH-A is a stronger assumption then DL-A, or if both assumptions are equivalent. It is known that DL-A is necessary for CDH-A but the other direction is currently not well understood. In particular, there are no groups known where DL-A holds but CDH-A does not hold [Fifield, 2012].

To see why the discrete logarithm assumption is necessary, assume that it does not hold. So, given a generator g and a group element h, it is easy to compute some residue class $x \in \mathbb{Z}_p$ with $h = g^x$. In that case, the computational Diffie–Hellman assumption cannot hold, since, given g, g^a and g^b, one can efficiently compute b and hence is able to compute $g^{ab} = (g^a)^b$ from this data.

The computational Diffie–Hellman assumption is a weaker assumption than the decisional Diffie–Hellman assumption, which means that there are groups where CDH-A holds and DDH-A does not hold, while there cannot be groups such that DDH-A holds but CDH-A does not hold. To see that, assume that it is efficiently possible to compute g^{ab} from g, g^a and g^b. Then, given (g^a, g^b, z) it is easy to decide if $z = g^{ab}$ or not.

Several variations and special cases of the CDH-A exist. For example, the **square computational Diffie–Hellman assumption** assumes that, given g and g^x, it is computationally hard to compute g^{x^2}. The **inverse computational Diffie–Hellman assumption** assumes that, given g and g^x, it is computationally hard to compute $g^{x^{-1}}$.

Cofactor Clearing

TODO: theorem: every factor of order defines a sub-group...

4.1.2 Hashing to Groups

Hash functions Generally speaking, a hash function is any function that can be used to map data of arbitrary size to fixed-size values. Since binary strings of arbitrary length are a general way to represent arbitrary data, we can understand a general **hash function** as a map

$$H : \{0,1\}^* \to \{0,1\}^k \tag{4.5}$$

where $\{0,1\}^*$ represents the set of all binary strings of arbitrary but finite length and $\{0,1\}^k$ represents the set of all binary strings that have a length of exactly k bits. So, in our definition, a hash function maps binary strings of arbitrary size onto binary strings of size exactly k. We call the images of H, that is, the values returned by the hash function **hash values**, **digests**, or simply **hashes**.

A hash function must be deterministic, that is, when we insert the same input x into H, the image $H(x)$ must always be the same. In addition, a hash function should be as uniform as possible, which means that it should map input values as evenly as possible over its output range. In mathematical terms, every string of length k from $\{0,1\}^k$ should be generated with roughly the same probability.

Example 43 (k-truncation hash). One of the most basic hash functions $H_k : \{0,1\}^* \to \{0,1\}^k$ is given by simply truncating every binary string s of size $s.len() > k$ to a string of size k and by filling any string s' of size $s'.len() < 0$ with zeros. To make this hash function deterministic, we define that both truncation and filling should happen"on the left".

For example if $k = 3$, $x_1 = (0000101011101010011101010101)$ and $x_2 = 1$ then $H(x_1) = (101)$ and $H(x_2) = (001)$. It is easy to see that this hash function is deterministic and uniform.

Of particular interest are so-called **cryptographic** hash functions, which are hash functions that are also **one-way functions**, which essentially means that, given a string y from $\{0,1\}^k$ it is practically infeasible to find a string $x \in \{0,1\}^*$ such that $H(x) = y$ holds. This property is usually called **preimage-resistance**.

In addition, it should be infeasible to find two strings $x_1, x_2 \in \{0,1\}^*$, such that $H(x_1) = H(x_2)$, which is called **collision resistance**. It is important to note, though, that collisions always exist, since a function $H : \{0,1\}^* \to \{0,1\}^k$ inevitable maps infinitely many values onto the same hash. In fact, for any hash function with digests of length k, finding a preimage to a given digest can always be done using a brute force search in 2^k evaluation steps. It should just be practically impossible to compute those values and statistically very unlikely to generate two of them by chance.

A third property of a cryptographic hash function is that small changes in the input string, like changing a single bit, should generate hash values that look completely different from each other.

Is there a term for this property?

As cryptographically secure hash functions map tiny changes in input values onto large change in the output, implementation errors that change the outcome are usually easy to spot by comparing them to expected output values. The definitions of cryptographically secure hash functions are therefore usually accompanied by some test vectors of common inputs and expected digests. Since the empty string $''$ is the only string of length 0, a common test vector is the expected digest of the empty string.

Example 44 (k-truncation hash). Considering the k-truncation hash from example 43. Since the empty string has length 0 it follows that the digest of the empty string is string of length k that only contains 0's:

$$H_k('') = (000\ldots000)$$

It is pretty obvious from the definition of H_k that this simple hash function is not a cryptographic hash function. In particular, every digest is its own preimage, since $H_k(y) = y$ for every string of size exactly k. Finding preimages is therefore easy.

In addition, it is easy to construct collusions as all strings of size $> k$ that share the same k-bits"on the right" are mapped to the same hash value.

Also, this hash function is not very chaotic, as changing bits that are not part of the k right-most bits don't change the digest at all.

Computing cryptographically secure hash functions in pen-and-paper style is possible but tedious. Fortunately, Sage can import the **hashlib** library, which is intended to provide a reliable and stable base for writing Python programs that require cryptographic functions. The following examples explain how to use hashlib in Sage.

Example 45. An example of a hash function that is generally believed to be a cryptographically secure hash function is the so-called **SHA256** hash, which in our notation is a function

$$SHA256 : \{0,1\}^* \to \{0,1\}^{256}$$

that maps binary strings of arbitrary length onto binary strings of length 256. To evaluate a proper implementation of the $SHA256$ hash function, the digest of the empty string is supposed to be

$$SHA256('') = e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855$$

For better human readability, it is common practice to represent the digest of a string, not in its binary form but in a hexadecimal representation. We can use Sage to compute $SHA256$ and freely transit between binary, hexadecimal and decimal representations. To do so, we import hashlib's implementation of SHA256:

```
sage: import hashlib
sage: test = 'e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934
   ca495991b7852b855'
sage: hasher = hashlib.sha256(b'')
```

```
sage: str = hasher.hexdigest()
sage: type(str)
<class 'str'>
sage: d = ZZ('0x'+ str) # conversion to integer type
sage: d.str(16) == str
True
sage: d.str(16) == test
True
sage: d.str(16)
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b8
   55
sage: d.str(2)
11100011101100000110001000100000101001100011111100000111000000101
   00100110101111101111110100110010001001100101101111101110010
   01001000010011110101110010000011110010001100100100110111001
   00110100110010100100100101011001100100011011011110000101001
   01011100001010101
sage: d.str(10)
102987336249554097029535212322581322789799900648198034993379939
   7001115665086549
```

Hashing to cyclic groups As we have seen in the previous paragraph, general hash functions map binary strings of arbitrary length onto binary strings of length k for some parameter k. However, it is desirable in various cryptographic primitives to not simply hash to binary strings of fixed length but to hash into algebraic structures like groups, while keeping (some of) the properties like preimage resistance or collision resistance.

Hash functions like this can be defined for various algebraic structures, but, in a sense, the most fundamental ones are hash functions that map into groups, because they are usually easily extended to map into other structures like rings or fields.

To give a more precise definition, let $\mathbb{G}$ be a group and $\{0,1\}^*$ the set of all finite, binary strings, then a **hash-to-group** function is a deterministic map

$$H : \{0,1\}^* \to \mathbb{G} \tag{4.6}$$

Common properties of hash functions, like uniformity are desirable but not always realized in actual real Common properties of hash functions, like uniformity, are desirable but not always realized in actual real world instantiations of hash-to-group functions, so we skip those requirements for now and keep the definition very general. instantiations of hash-to-group functions, so we skip those requirements for now and keep the definition very general.

As the following example shows, hashing to finite cyclic groups can be trivially achieved for the price of some undesirable properties of the hash function:

Example 46 (Naive cyclic group hash). Let $\mathbb{G}$ be a finite cyclic group. If the task is to implement a hash-to-$\mathbb{G}$ function, one immediate approach can be based on the observation that binary strings of size k, can be interpreted as integers $z \in \mathbb{Z}$ in the range $0 \leq z < 2^k$.

To be more precise, choose an ordinary hash function $H : \{0,1\}^* \to \{0,1\}^k$ for some parameter k and a generator g of $\mathbb{G}$. Then the expression

$$z_{H(s)} = H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \ldots + H(s)_k \cdot 2^k$$

is a positive integer, where $H(s)_j$ means the bit at the j-th position of $H(s)$. A hash-to-group function for the group $\mathbb{G}$ can then be defined as a concatenation of the exponential map $g^{(\cdot)}$ of g with the interpretation of $H(s)$ as an integer:

$$H_g : \{0,1\}^* \to \mathbb{G} : \; s \mapsto g^{z_{H(s)}}$$

Constructing a hash-to-group function like this is easy to implement for cyclic groups and might be good enough in certain applications. It is however, almost never adequate in cryptographic applications, as discrete log relations might be constructible between two given hash value $H_g(s)$ and $H_g(t)$.

To see that, assume that $\mathbb{G}$ is of order r and that $z_{H(s)}$ has a multiplicative inverse in modular r arithmetics. In that case, we can compute $x = z_{H(t)} \cdot z_{H(s)}^{-1}$ in $\mathbb{Z}_r$ and have found a discrete log relation between the group hash values, that is, we have found some x with $H_g(t) = (H_g(s))^x$ since

$$\begin{aligned} H_g(t) &= (H_g(s))^x & \Leftrightarrow \\ g^{z_{H(t)}} &= g^{z_{H(s)} \cdot x} & \Leftrightarrow \\ g^{z_{H(t)}} &= g^{z_{H(t)}} & \end{aligned}$$

Applications where discrete log relations between hash values are undesirable therefore need different approaches, and many of those approaches start with a way to hash into the sets $\mathbb{Z}_r$ of modular r arithmetics.

Hashing to modular arithmetics One of the most widely used applications of hash-into-group functions are hash functions that map into the set $\mathbb{Z}_r$ of modular r arithmetics for some modulus r. Different approaches to construct such a function are known, but probably the most widely used ones are based on the insight that the images of arbitrary hash functions can be interpreted as binary representations of integers as explained in example XXX. add reference

It follows from this interpretation that one simple method of hashing into $\mathbb{Z}_r$ is constructed by observing that if r is a modulus with a bit-length of $k = r.nbits()$, then every binary string $(b_0, b_1, \ldots, b_{k-2})$ of length $k-1$ defines an integer z in the rage $0 \le z < 2^{k-1} \le r$, by defining

$$z = b_0 \cdot 2^0 + b_1 \cdot 2^1 + \ldots + b_{k-2} \cdot 2^{k-2} \tag{4.7}$$

Now since $z < r$, we know that z is guaranteed to be in the set $\{0, 1, \ldots, r-1\}$ and hence can be interpreted as an element of $\mathbb{Z}_r$. From this, it follows that if $H : \{0,1\}^* \to \{0,1\}^{k-1}$ is a hash function, then a hash-to-group function can be constructed by

$$H_{r.nbits()-1} : \{0,1\}^* \to \mathbb{Z}_r : \; s \mapsto H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \ldots + H(s)_{k-2} \cdot 2^{k-2} \tag{4.8}$$

where $H(s)_j$ means the j-th bit of the image binary string $H(s)$ of the original binary hash function.

A drawback of this hash function is that the distribution of the hash values in $\mathbb{Z}_r$ is not necessarily uniform. In fact, if $r - 2^{k-1} \neq 0$, then by design $H_{r.nbits()-1}$ will never hash onto values $z \geq 2^{k-1}$. Good moduli r are therefore as close to 2^{k-1} as possible, while less good moduli are closer to 2^k. In the worst case, that is, $r = 2^k - 1$, it misses $2^{k-1} - 1$, that is almost half of all elements, from $\mathbb{Z}_r$. TODO: DOUBLE CHECK THIS REASONING.

An advantage is that properties like preimage resistance or collision resistance of the original hash function $H(\cdot)$ are preserved.

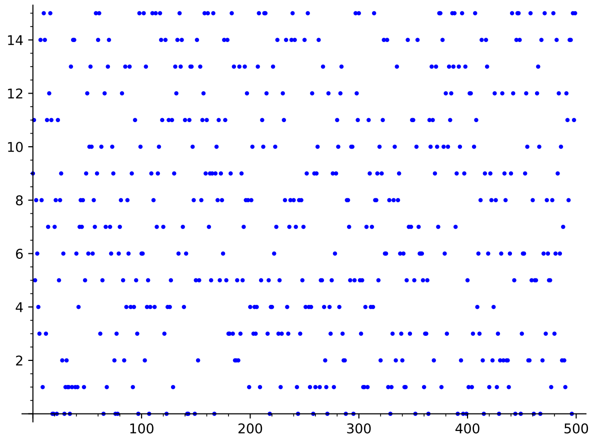
Example 47. To give an implementation of the $H_{r.nbits()-1}$ hash function, we use a 5-bit truncation of the $SHA256$ hash from example 45 and define a hash into $\mathbb{Z}_{16}$ by

$$H_{16.nbits()-1} : \{0,1\}^* \to \mathbb{Z}_{16} : \; s \mapsto SHA256(s)_0 \cdot 2^0 + SHAH256(s)_1 \cdot 2^1 + \ldots + SHA256(s)_4 \cdot 2^4$$

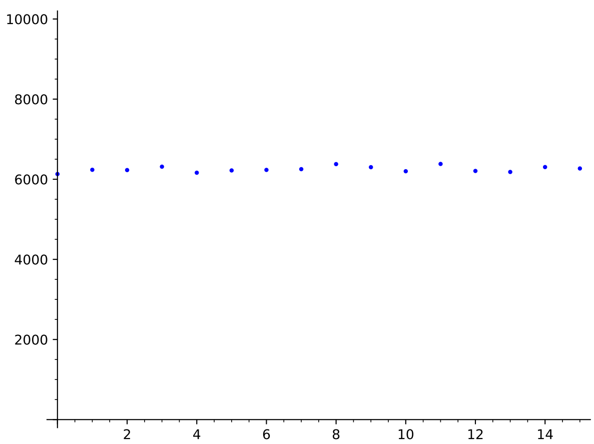
Since $k = 16.nbits() = 5$ and $16 - 2^{k-1} = 0$, this hash maps uniformly onto $\mathbb{Z}_{16}$. We can invoke Sage to implement it e.g. like this:

```
sage: import hashlib
sage: def Hash5(x):
....:     hasher = hashlib.sha256(x)
....:     digest = hasher.hexdigest()
....:     d = ZZ(digest, base=16)
....:     d = d.str(2)[-4:]
....:     return ZZ(d,base=2)
sage: Hash5(b'')
5
```

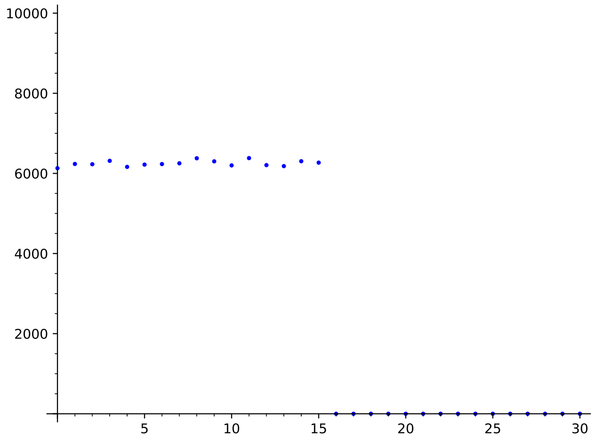
We can then use sage to apply this function to a large set of input values in order to plot a visualization of the distribution over the set $\{0,\ldots,15\}$. Executing over 500 input values gives:



To get an intuition of uniformity, we can count the number of times the hash function $H_{16.nbits()-1}$ maps onto each number in the set $\{0,1,\ldots,15\}$ in a loop of 100000 hashes and compare that to the ideal uniform distribution, which would map exactly 6250 samples to each element. This gives the following result:



The uniformity of the distribution problem becomes apparent if we want to construct a similar hash function for $\mathbb{Z}_r$ for any r in the range $17 \leq r \leq 31$. In this case, the definition of the hash function is exactly the same as for $\mathbb{Z}_{16}$ and hence the images will not exceed the value 16. So, for example, in the case of hashing to $\mathbb{Z}_{31}$, the hash function never maps to any value larger than 16, leaving almost half of all numbers out of the image range.



Mirco: We can do better than this

The second widely used method of hashing into $\mathbb{Z}_r$ is constructed by observing that if r is a modulus with a bit-length of $r.bits() = k_1$ and $H : \{0,1\}^* \to \{0,1\}^{k_2}$ is a hash function that produces digests of size k_2, with $k_2 \geq k_1$, then a hash-to-group function can be constructed by interpreting the image of H as a binary representation of a integer and then taking the modulus by r to map into $\mathbb{Z}_r$. To be more precise

$$H'_{mod_r} : \{0,1\}^* \to \mathbb{Z}_r : \; s \mapsto \left(H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \ldots + H(s)_{k_2} \cdot 2^{k_2}\right) \bmod n \qquad (4.9)$$

where $H(s)_j$ means the j'th bit of the image binary string $H(s)$ of the original binary hash function.

A drawback of this hash function is that computing the modulus requires some computational effort. In addition, the distribution of the hash values in $\mathbb{Z}_r$ might not be even, depending on the difference $2^{k_2+1} - r$. An advantage is that potential properties like preimage resistance or collision resistance of the original hash function $H(\cdot)$ are preserved and the distribution can be made almost uniform, with only negligible bias, depending on what modulus r and images size k_2 are chosen.

Example 48. To give an implementation of the H_{mod_r} hash function, we use k_2-bit truncation of the $SHA256$ hash from example 45 and define a hash into $\mathbb{Z}_{23}$ as follows:

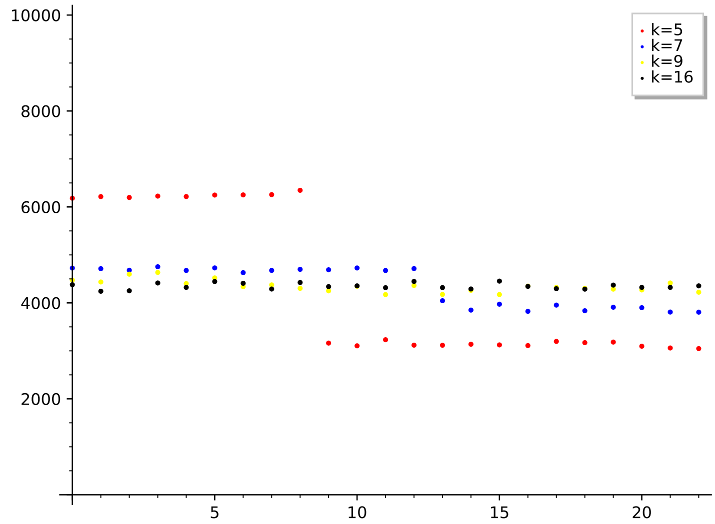
$$H_{mod_{23},k_2} : \{0,1\}^* \to \mathbb{Z}_{23} : \\ s \mapsto \left(SHA256(s)_0 \cdot 2^0 + SHAH256(s)_1 \cdot 2^1 + \ldots + SHA256(s)_{k_2} \cdot 2^{k_2}\right) \bmod 23$$

We want to use various instantiations of k_2 to visualize the impact of truncation length on the distribution of the hashes in $\mathbb{Z}_{23}$. We can invoke sage to implement it e.g. like this:

```
sage: import hashlib
sage: Z23 = Integers(23)
sage: def Hash_mod23(x, k2):
....:     hasher = hashlib.sha256(x.encode('utf-8'))
....:     digest = hasher.hexdigest()
```

```
....:       d = ZZ(digest, base=16)
....:       d = d.str(2)[-k2:]
....:       d = ZZ(d, base=2)
....:       return Z23(d)
```

We can then use Sage to apply this function to a large set of input values in order to plot visualizations of the distribution over the set $\{0,\dots,22\}$ for various values of k_2 by counting the number of times it maps onto each number in a loop of 100000 hashes. We get



A third method that can sometimes be found in implementations is the so-called **"try-and-increment" method**. To understand this method, we define an integer $z \in \mathbb{Z}$ from any hash value $H(s)$ as we did in the previous methods, that is, we define $z = H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \ldots + H(s)_{k-1} \cdot 2^k$.

Hashing into $\mathbb{Z}_r$ is then achievable by first computing z, and then trying to see if $z \in \mathbb{Z}_r$. If it is, then the hash is done; if not, the string s is modified in a deterministic way and the process is repeated until a suitable number z is found. A suitable, deterministic modification could be to concatenate the original string by some bit counter. A "try-and-increment" algorithm would then work like in algorithm 5.

check reference

Algorithm 5 Hash-to-$\mathbb{Z}_n$

Require: $r \in \mathbb{Z}$ with $r.nbits() = k$ and $s \in \{0,1\}^*$

procedure TRY-AND-INCREMENT(r,k,s)

$c \leftarrow 0$

repeat

$s' \leftarrow s||c\_bits()$

$z \leftarrow H(s')_0 \cdot 2^0 + H(s')_1 \cdot 2^1 + \ldots + H(s')_k \cdot 2^k$

$c \leftarrow c+1$

until $z < r$

return x

end procedure

Ensure: $z \in \mathbb{Z}_r$

Depending on the parameters, this method can be very efficient. In fact, if k is sufficiently large and r is close to 2^{k+1}, the probability for $z < r$ is very high and the repeat loop will almost always be executed a single time only. A drawback is, however, that the probability of having to execute the loop multiple times is not zero.

Once some hash function into modular arithmetics is found, it can often be combined with additional techniques to hash into more general finite cyclic groups. The following paragraphs describes a few of those methods widely adopted in SNARK development.

Pedersen Hashes The so-called **Pedersen hash function** [Pedersen, 1992] provides a way to map binary inputs of fixed size k onto elements of finite cyclic groups that avoids discrete log relations between the images as they occur in the naive approach XXX. Combining it with a classical hash function provides a hash function that maps strings of arbitrary length onto group elements.

add reference

To be more precise, let j be an integer, $\mathbb{G}$ a finite cyclic group of order r and $\{g_1,\ldots,g_j\} \subset \mathbb{G}$ a uniform randomly generated set of generators of $\mathbb{G}$. Then **Pedersen's hash function** is defined as

$$H_{Ped} : (\mathbb{Z}_r)^j \to \mathbb{G} : \ (x_1,\ldots,x_j) \mapsto \Pi_{i=1}^{j} g_j^{x_j} \tag{4.10}$$

It can be shown that Pedersen's hash function is collision-resistant under the assumption that $\mathbb{G}$ is a DL-A group. However, it is important to note that Pedersen hashes cannot be assumed to be **pseudorandom** and should therefore not be used where a hash function serves as an approximation of a random **oracle**.

From an implementation perspective, it is important to derive the set of generators $\{g_1,\ldots,g_j\}$ in such a way that they are as uniform and random as possible. In particular, any known discrete log relation between two generators, that is, any known $x \in \mathbb{Z}_r$ with $g_h = (g_i)^x$ must be avoided.

To see how Pedersen hashes can be used to define an actual hash-to-group function according to our definition, we can use any of the hash-to-$\mathbb{Z}_r$ functions as we have derived them in XXX.

add reference

MimC Hashes [Albrecht et al., 2016]

Pseudo Random Functions in DDH-A groups As noted in XXX, Pederson's hash function does not have the properties a random function and should therefore not be instantiated as such. To look at a construction that serves as a random oracle function in groups where the decisional Diffie–Hellman construction is assumed to hold true, let $\mathbb{G}$ be a DDH-A group of order r with generator g and $\{a_0,a_1,\ldots,a_k\} \subset \mathbb{Z}_r^*$ a uniform randomly generated set of numbers invertible in modular r arithmetics. Then a pseudo-random function is given by

$$F_{rand} : \{0,1\}^{k+1} \to \mathbb{G} : \ (b_0,\ldots,b_k) \mapsto g^{b_0 \cdot \Pi_{i=1}^{k} a_i^{b_i}} \tag{4.11}$$

Of course, if $H : \{0,1\}^* \to \{0,1\}^{k+1}$ is a random oracle, then the concatenation of F_{rand} and H, defines a random oracle

$$H_{rand,\mathbb{G}} : \{0,1\}^* \to \mathbb{G} : \ s \mapsto F_{rand}(H(s)) \tag{4.12}$$

4.2 Commutative Rings

Thinking of back to operations on integers, we know that there are two of these: addition and multiplication. As we have seen, addition defines a group structure on the set of integers.

However, multiplication does not define a group structure, given that integers in general don't have multiplicative inverses.

Configurations like this constitute a so-called **commutative ring with unit**. To be more precise, a commutative ring with unit $(R,+,\cdot,1)$ is a set R provided with two maps $+: R\cdot R \to R$ and $\cdot: R\cdot R \to R$, called **addition** and **multiplication**, such that the following conditions hold:

- $(R,+)$ is a commutative group, where the neutral element is denoted with 0. **Commutativity of multiplication**: $r_1 \cdot r_2 = r_2 \cdot r_1$ for all $r_1, r_2 \in R$.
- **Existence of a unit**: There is an element $1 \in R$, such that $1 \cdot g$ holds for all $g \in R$,
- **Associativity**: For every $g_1, g_2, g_3 \in \mathbb{G}$ the equation $g_1 \cdot (g_2 \cdot g_3) = (g_1 \cdot g_2) \cdot g_3$ holds.
- **Distributivity**: For all $g_1, g_2, g_3 \in R$ the distributive laws $g_1 \cdot (g_2 + g_3) = g_1 \cdot g_2 + g_1 \cdot g_3$ holds.

Example 49 (The ring of integers). The set $\mathbb{Z}$ of integers with the usual addition and multiplication is the archetypical example of a commutative ring with unit 1.

Example 50 (Underlying commutative group of a ring). Every commutative ring with unit $(R,+,\cdot,1)$ gives rise to a group, if we disregard multiplication.

The following example is more interesting. The motivated reader is encouraged to think through this example, not so much because we need this in what follows, but more so as it helps to detach the reader from familiar styles of computation.

Example 51. Let $S := \{\bullet, \star, \odot, \otimes\}$ be a set that contains four elements and let addition and multiplication on S be defined as follows:

$\cup$	$\bullet$	$\star$	$\odot$	$\otimes$
$\bullet$	$\bullet$	$\star$	$\odot$	$\otimes$
$\star$	$\star$	$\odot$	$\otimes$	$\bullet$
$\odot$	$\odot$	$\otimes$	$\bullet$	$\star$
$\otimes$	$\otimes$	$\bullet$	$\star$	$\odot$

$\circ$	$\bullet$	$\star$	$\odot$	$\otimes$
$\bullet$	$\bullet$	$\bullet$	$\bullet$	$\bullet$
$\star$	$\bullet$	$\star$	$\odot$	$\otimes$
$\odot$	$\bullet$	$\odot$	$\bullet$	$\odot$
$\otimes$	$\bullet$	$\otimes$	$\odot$	$\star$

Then $(S,\cup,\circ)$ is a ring with unit $\star$ and zero $\bullet$. It therefore makes sense to ask for solutions to equations like this one: Find $x \in S$ such that

$$\otimes \circ (x \cup \odot) = \star$$

To see how such a "moonmath equation" can be solved, we have to keep in mind that rings behaves mostly like normal number when it comes to bracketing and computation rules. The only differences are the symbols and the actual way to add and multiply. With this in mind, we solve the equation for x in the“usual way”:

$$
\begin{aligned}
\otimes \circ (x \cup \odot) &= \star && \text{\# apply the distributive law}\\
\otimes \circ x \cup \otimes \circ \odot &= \star && \text{\#} \otimes \circ \odot = \odot\\
\otimes \circ x \cup \odot &= \star && \text{\# concatenate the } \cup \text{ inverse of } \odot \text{ to both sides}\\
\otimes \circ x \cup \odot \cup -\odot &= \star \cup -\odot && \text{\#} \odot \cup -\odot = \bullet\\
\otimes \circ x \cup \bullet &= \star \cup -\odot && \text{\#} \bullet \text{ is the } \cup \text{ neutral element}\\
\otimes \circ x &= \star \cup -\odot && \text{\# for } \cup \text{ we have } -\odot = \odot\\
\otimes \circ x &= \star \cup \odot && \text{\#} \star \cup \odot = \otimes\\
\otimes \circ x &= \otimes && \text{\# concatenate the } \circ \text{ inverse of } \otimes \text{ to both sides}\\
(\otimes)^{-1} \circ \otimes \circ x &= (\otimes)^{-1} \circ \otimes && \text{\# multiply with the multiplicative inverse}\\
\star \circ x &= \star\\
x &= \star
\end{aligned}
$$

So even though this equation looked really alien at first glance, we could solve it basically exactly the way we solve “normal” equations, like we do for fractional numbers, for example.

Note, however, that in a ring, things can be very different than most of us are used to whenever a multiplicative inverse would be needed to solve an equation in the usual way. For example the equation

$$\odot \circ x = \otimes$$

cannot be solved for x in the usual way, since there is no multiplicative inverse for $\odot$ in our ring. We can confirm this by looking at the multiplication table to see that no such x exits.

As another example, the equation

$$\odot \circ x = \odot$$

does not have a single solution but two: $x \in \{\star, \otimes\}$. Having no solution or two solutions is certainly not something to expect from types like $\mathbb{Q}$.

Example 52. Considering polynomials again, we note from their definition that what we have called the type R of the coefficients must in fact be a commutative ring with a unit, since we need addition, multiplication, commutativity and the existence of a unit for $R[x]$ to have the properties we expect.

Considering R to be a ring with addition and multiplication of polynomials as defined in XXX actually makes $R[x]$ into a commutative ring with a unit, too, where the polynomial 1 is the multiplicative unit. add reference

Example 53. Let n be a modulus and $(\mathbb{Z}_n, +, \cdot)$ the set of all remainder classes of integers modulo n, with the projection of integer addition and multiplication as defined in XXX. It can be shown that $(\mathbb{Z}_n, +, \cdot)$ is a commutative ring with unit 1.

Considering the exponential map from page 43 again, let $\mathbb{G}$ be a finite cyclic group of order n with generator $g \in \mathbb{G}$. Then the ring structure of $(\mathbb{Z}_n, +, \cdot)$ is mapped onto the group structure of $\mathbb{G}$ in the following way: check reference

$$
\begin{aligned}
g^{x+y} &= g^x \cdot g^y && \text{for all } x, y \in \mathbb{Z}_n\\
g^{x \cdot y} &= (g^x)^y && \text{for all } x, y \in \mathbb{Z}_n
\end{aligned}
$$

This of particular interest in cryptography and SNARKs, as it allows for the evaluation of polynomials with coefficients in $\mathbb{Z}_n$ to be evaluated “in the exponent”. To be more precise, let

$p \in \mathbb{Z}_n[x]$ be a polynomial with $p(x) = a_m \cdot x^m + a_{m-1}x^{m-1} + \ldots + a_1 x + a_0$. Then the previously defined exponential laws XXX imply that

add reference

$$\begin{aligned} g^{p(x)} &= g^{a_m \cdot x^m + a_{m-1}x^{m-1} + \ldots + a_1 x + a_0} \\ &= \left(g^{x^m}\right)^{a_m} \cdot \left(g^{x^{m-1}}\right)^{a_{m-1}} \cdot \ldots \cdot (g^x)^{a_1} \cdot g^{a_0} \end{aligned}$$

and hence to evaluate p at some point s in the exponent, we can insert s into the right hand side of the last equation and evaluate the product.

As we will see, this is a key insight to understand many SNARK protocols like e.g. Groth16 [Groth, 2016] or XXX.

add more examples protocols of SNARK

Example 54. To give an example of the evaluation of a polynomial in the exponent of a finite cyclic group, consider the exponential map from example XXX:

add reference

$$3^{(\cdot)} : \mathbb{Z}_4 \to \mathbb{F}_5^* \; x \mapsto 3^x$$

Choosing the polynomial $p(x) = 2x^2 + 3x + 1$ from $\mathbb{Z}_4[x]$, we can evaluate the polynomial at say $x = 2$ in the exponent of 3 in two different ways. On the one hand, we can evaluate p at 2 and then write the result into the exponent, which gives the following:

gives

$$\begin{aligned} 3^{p(2)} &= 3^{2 \cdot 2^2 + 3 \cdot 2 + 1} \\ &= 3^{2 \cdot 0 + 2 + 1} \\ &= 3^3 \\ &= 2 \end{aligned}$$

On the other hand, we can use the right hand side of the equation to evaluate p at 2 in the exponent of 3, which gives the following:

gives

$$\begin{aligned} 3^{p(2)} &= \left(3^{2^2}\right)^2 \cdot \left(3^2\right)^3 \cdot 3^1 \\ &= \left(3^0\right)^2 \cdot 3^3 \cdot 3 \\ &= 1^2 \cdot 2 \cdot 3 \\ &= 2 \cdot 3 \\ &= 2 \end{aligned}$$

Hashing to Commutative Rings As we have seen in XXX, various constructions for hashing-to-groups are known and used in applications. As commutative rings are Abelian groups, when we disregard the multiplicative structure, hash-to-group constructions can be applied for hashing into commutative rings, too. This is possible in general, as the codomain of a general hash function $\{0,1\}^*$ is just the set of binary strings of arbitrary but finite length, which has no algebraic structure that the hash function must respect.

add reference

Abelian groups

codomain

4.3 Fields

We started this chapter with the definition of a group (section 4.1), which we then expanded into the definition of a commutative ring with a unit (section 4.2). Such rings generalize the behavior of integers. In this section, we will look at the special case of commutative rings, where every element, other than the neutral element of addition, has a multiplicative inverse. Those structures behave very much like the rational numbers $\mathbb{Q}$, which are in a sense an extension of the ring of integers, that is, constructed by including newly defined multiplicative inverses (fractions) to the integers.

Now, considering the definition of a ring XXXAdd numbering to definitions again, we define a **field** $(\mathbb{F},+,\cdot)$ to be a set $\mathbb{F}$, together with two maps $+:\mathbb{F}\cdot\mathbb{F}\to\mathbb{F}$ and $\cdot:\mathbb{F}\cdot\mathbb{F}\to\mathbb{F}$, called *addition* and *multiplication*, such that the following conditions hold:

Add numbering to definitions

- $(\mathbb{F},+)$ is a commutative group, where the neutral element is denoted by 0.
- $(\mathbb{F}\setminus\{0\},\cdot)$ is a commutative group, where the neutral element is denoted by 1.
- (Distributivity) For all $g_1,g_2,g_3\in\mathbb{F}$ the distributive law $g_1\cdot(g_2+g_3)=g_1\cdot g_2+g_1\cdot g_3$ holds.

If a field is given and the definition of its addition and multiplication is not ambiguous, we will often simple write $\mathbb{F}$ instead of $(\mathbb{F},+,\cdot)$ to denote the field. We moreover write $\mathbb{F}^*$ to describe the multiplicative group of the field, that is, the set of elements with the multiplication as group law excluding the neutral element of addition.

The **characteristic** of a field $\mathbb{F}$, represented as $char(\mathbb{F})$, is the smallest natural number $n\geq 1$, for which the n -fold sum of 1 equals zero, i.e. for which $\sum_{i=1}^{n}1=0$. If such a $n>0$ exists, the field is also-called to have a **finite characteristic**. If, on the other hand, every finite sum of 1 is such that it is not equal to zero, then the field is defined to have characteristic 0. S: Tried to disambiguate the scope of negation between 1. “It is true of every finite sum of 1 that it is not equal to 0” and 2. “It is not true of every finite sum of 1 that it is equal to 0” From the example below, it looks like 1. is the intended meaning here, correct?

Check change of wording

Example 55 (Field of rational numbers). Probably the best known example of a field is the set of rational numbers $\mathbb{Q}$ together with the usual definition of addition, subtraction, multiplication and division. Since there is no counting number $n\in\mathbb{N}$, such that $\sum_{j=0}^{n}1=0$ in the set of rational numbers, the characteristic $char(\mathbb{Q})$ of the field $\mathbb{Q}$ is zero. In Sage rational numbers are called as follows:

```
sage: QQ
Rational Field
sage: QQ(1/5) # Get an element from the field of rational
   numbers
1/5
sage: QQ(1/5) / QQ(3) # Division
1/15
```

Example 56 (Field with two elements). It can be shown that in any field, the neutral element 0 of addition must be different from the neutral element 1 of multiplication, that is, $0\neq 1$ always holds in a field. From this, it follows that the smallest field must contain at least two elements. As the following addition and multiplication tables show, there is indeed a field with two elements, which is usually called $\mathbb{F}_2$:

Let $\mathbb{F}_2:=\{0,1\}$ be a set that contains two elements and let addition and multiplication on $\mathbb{F}_2$ be defined as follows:

+	0	1
0	0	1
1	1	0

·	0	1
0	0	0
1	0	1

Since $1+1=0$ in the field $\mathbb{F}_2$, we know that the characteristic of $\mathbb{F}_2$ is there, that is, we have $char(\mathbb{F}_2)=0$.

For reasons we will understand better in XXX, Sage defines this field as a so-called Galois field with 2 elements. You can call it in Sage as follows:

add reference

```
sage: F2 = GF(2)
sage: F2(1) # Get an element from GF(2)
1
sage: F2(1) + F2(1) # Addition
0
sage: F2(1) / F2(1) # Division
1
```

Example 57. Both the real numbers $\mathbb{R}$ as well as the complex numbers $\mathbb{C}$ are well known examples of fields.

Exercise 30. Consider our remainder class ring $(\mathbb{F}_5,+,\cdot)$ and show that it is a field. What is the characteristic of $\mathbb{F}_5$?

Prime fields As we have seen in the various examples of the previous sections, modular arithmetics behaves similarly to ordinary arithmetics of integers in many ways. This is due to the fact that remainder class sets $\mathbb{Z}_n$ are commutative rings with units.

However, we have also seen in XXX that, whenever the modulus is a prime number, every remainder class other than the zero class has a modular multiplicative inverse. This is an important observation, since it immediately implies that in case of a prime number, the remainder class set $\mathbb{Z}_n$ is not just a ring but actually a **field**. Moreover, since $\sum_{j=0}^{n} 1 = 0$ in $\mathbb{Z}_n$, we know that those fields have the finite characteristic n.

add reference

To distinguish this important case from arbitrary remainder class rings, we write $(\mathbb{F}_p,+,\cdot)$ for the field of all remainder classes for a prime number modulus $p\in\mathbb{P}$ and call it the **prime field** of characteristic p.

Prime fields are the foundation for many of the contemporary algebra-based cryptographic systems, as they have many desirable properties. One of them is that, since these sets are finite and a prime field of characteristic, p can be represented on a computer in roughly $log_2(p)$ amount of space without precision problems that are unavoidable for computer representations of infinite sets, e.g. rational numbers or integers.

Since prime fields are special cases of remainder class rings, all computations remain the same. Addition and multiplication can be computed by first doing normal integer addition and multiplication, and then taking the remainder modulus p. Subtraction and division can be computed by adding or multiplying with the additive or the multiplicative inverse, respectively. The additive inverse $-x$ of a field element $x\in\mathbb{F}_p$ is given by $p-x$, and the multiplicative inverse of $x\neq 0$ is given by x^{p-2}, or can be computed using the Extended Euclidean Algorithm.

Note that these computations might not be the fastest to implement on a computer. They are, however, useful in this book, as they are easy to compute for small prime numbers.

Example 58. The smallest field is the field $\mathbb{F}_2$ of characteristic 2 as we have seen in example 56. It is the prime field of the prime number 2.

Why are we repeating this example here again?

Example 59. To summarize the basic aspects of computation in prime fields, let us consider the prime field $\mathbb{F}_5$ and simplify the following expression

$$\left(\frac{2}{3}-2\right)\cdot 2$$

A first thing to note is that since $\mathbb{F}_5$ is a field all rules are identical to the rules we learned in school when we where dealing with rational, real or complex numbers. This means we can use e.g. bracketing (distributivity) or addition as usual:

$$
\begin{aligned}
\left(\frac{2}{3}-2\right)\cdot 2 &= \frac{2}{3}\cdot 2 - 2\cdot 2 && \text{\# distributive law}\\
&= \frac{2\cdot 2}{3} - 2\cdot 2 && 4 \bmod 5 = 4\\
&= \frac{4}{3} - 4 && \text{\# multiplicative inverse of 3 is } 3^{5-2} \bmod 5 = 2\\
&= 4\cdot 2 - 4 && \text{\# additive inverse of 4 is } 5-4=1\\
&= 4\cdot 2 + 1 && 8 \bmod 5 = 3\\
&= 3+1 && 4 \bmod 5 = 4\\
&= 4
\end{aligned}
$$

In this computation we computed the multiplicative inverse of 3 using the identity $x^{-1} = x^{p-2}$ in a prime field. This is impractical for large prime numbers. Recall that another way of computing the multiplicative inverse is the Extended Euclidean Algorithm (see 3.10 on page 18). To refresh our memory, the task is to compute $x^{-1}\cdot 3 + t\cdot 5 = 1$, but t is actually irrelevant. We get

k	r_k	x_k^{-1}	$t_k = (r_k - s_k\cdot a)$ div b
0	3	1	·
1	5	0	·
2	3	1	·
3	2	-1	·
4	1	2	·

So the multiplicative inverse of 3 in $\mathbb{Z}_5$ is 2 and indeed if compute $3\cdot 2$ we get 1 in $\mathbb{F}_5$.

Square Roots In this part, we deal with square numbers, also called **quadratic residues** and **square roots** in prime fields. This is of particular importance in our studies on elliptic curves, as only square numbers can actually be points on an elliptic curve.

S: are we introducing elliptic curves in section 1 or 2?

To make the intuition of quadratic residues and roots precise, let $p \in \mathbb{P}$ be a prime number and $\mathbb{F}_p$ its associated prime field. Then a number $x \in \mathbb{F}_p$ is called a **square root** of another number $y \in \mathbb{F}_p$, if x is a solution to the equation

$$x^2 = y \tag{4.13}$$

In this case, y is called a **quadratic residue**. On the other hand, if y is given and the quadratic equation has no solution x , we call y a **quadratic non-residue**. For any $y \in \mathbb{F}_p$, we write

$$\sqrt{y} := \{x \in \mathbb{F}_p \mid x^2 = y\} \tag{4.14}$$

for the set of all square roots of y in the prime field $\mathbb{F}_n$. (If y is a quadratic non-residue, then $\sqrt{y} = \emptyset$ (an empty set), and if $y = 0$, then $\sqrt{y} = \{0\}$)

Informally speaking, quadratic residues are numbers such that we can take the square root of them, while quadratic non-residues are numbers that don't have square roots. The situation therefore parallels the know case of integers, where some integers like 4 or 9 have square roots and others like 2 or 3 don't (as integers).

It can be shown that in any prime field every non zero element has either no square root or two of them. We adopt the convention to call the smaller one (when interpreted as an integer) as the **positive** square root and the larger one as the **negative**. This makes sense, as the larger one can always be computed as the modulus minus the smaller one, which is the definition of the negative in prime fields.

Example 60 (Quadratic (Non)-Residues and roots in $\mathbb{F}_5$). Let us consider our example prime field $\mathbb{F}_5$ again. All square numbers can be found on the main diagonal of the multiplication table XXX. As you can see, in $\mathbb{Z}_5$ only the numbers 0, 1 and 4 have square roots and we get $\sqrt{0} = \{0\}$, $\sqrt{1} = \{1,4\}$, $\sqrt{2} = \emptyset$, $\sqrt{3} = \emptyset$ and $\sqrt{4} = \{2,3\}$. The numbers 0, 1 and 4 are therefore quadratic residues, while the numbers 2 and 3 are quadratic non-residues.

In order to describe whether an element of a prime field is a square number or not, the so-called Legendre Symbol can sometimes be found in the literature, why we will recapitulate it here:

Let $p \in \mathbb{P}$ be a prime number and $y \in \mathbb{F}_p$ an element from the associated prime field. Then the so-called *Legendre symbol* of y is defined as follows:

$$\left(\frac{y}{p}\right) := \begin{cases} 1 & \text{if } y \text{ has square roots} \\ -1 & \text{if } y \text{ has no square roots} \\ 0 & \text{if } y = 0 \end{cases} \tag{4.15}$$

Example 61. Look at the quadratic residues and non residues in $\mathbb{F}_5$ from example XXX again, we can deduce the following Legendre symbols, from example XXX.

$$\left(\tfrac{0}{5}\right) = 0, \quad \left(\tfrac{1}{5}\right) = 1, \quad \left(\tfrac{2}{5}\right) = -1, \quad \left(\tfrac{3}{5}\right) = -1, \quad \left(\tfrac{4}{5}\right) = 1 \,.$$

The Legendre symbol gives a criterion to decide whether or not an element from a prime field has a quadratic root or not. This, however, is not just of theoretic use, as the following so-called **Euler criterion** gives a compact way to actually compute the Legendre symbol. To see that, let $p \in \mathbb{P}_{\geq 3}$ be an odd prime number and $y \in \mathbb{F}_p$. Then the Legendre symbol can be computed as Prime number and $y \in \mathbb{F}_p$. Then the Legendre symbol can be computed as

$$\left(\frac{y}{p}\right) = y^{\frac{p-1}{2}} \,. \tag{4.16}$$

Example 62. Looking at the quadratic residues and non residues in $\mathbb{F}_5$ from example XXX again, we can compute the following Legendre symbols using the Euler criterium:

add reference

$$\left(\frac{0}{5}\right) = 0^{\frac{5-1}{2}} = 0^2 = 0$$
$$\left(\frac{1}{5}\right) = 1^{\frac{5-1}{2}} = 1^2 = 1$$
$$\left(\frac{2}{5}\right) = 2^{\frac{5-1}{2}} = 2^2 = 4 = -1$$
$$\left(\frac{3}{5}\right) = 3^{\frac{5-1}{2}} = 3^2 = 4 = -1$$
$$\left(\frac{4}{5}\right) = 4^{\frac{5-1}{2}} = 4^2 = 1$$

Exercise 31. Consider the prime field $\mathbb{F}_{13}$. Find the set of all pairs $(x,y) \in \mathbb{F}_{13} \times \mathbb{F}_{13}$ that satisfy the equation

$$x^2 + y^2 = 1 + 7 \cdot x^2 \cdot y^2$$

Exponentiation TO APPEAR...

write paragraph on exponentiation

Hashing into prime fields An important problem in SNARK development is the ability to hash to (various subsets) of elliptic curves. As we will see in XXX, those curves are often defined over prime fields, and hashing to a curve then might start with hashing to the prime field. It is therefore important to understand how to hash into prime fields.

add reference

To understand it, note that in XXX we have looked at a few constructions of how to hash into the residue class rings $\mathbb{Z}_n$ for arbitrary $n > 1$. As prime fiedls are just special instances of those rings, all hashing into $\mathbb{Z}_n$ functions can be used for hashing into prime fields, too.

To understand it

add reference

Extension Fields We defined prime fields in the previous section. They are the basic building blocks for cryptography in general and SNARKs in particular.

However, as we will see in, XX so-called **pairing based** SNARK systems are crucially dependent on group pairings XXX defined over the group of rational points of elliptic curves. For those pairings to be non-trivial, the elliptic curve must not only be defined over a prime field, but over a so-called **extension field** of a given prime field.

add reference

group pairings

We therefore have to understand field extensions. To understand them, first observe that the field $\mathbb{F}'$ is called an **extension** of a field $\mathbb{F}$ if $\mathbb{F}$ is a subfield of $\mathbb{F}'$, that is, $\mathbb{F}$ is a subset of $\mathbb{F}'$ and restricting the addition and multiplication laws of $\mathbb{F}'$ to the subset $\mathbb{F}$ recovers the appropriate laws of $\mathbb{F}$.

Now it can be shown that whenever $p \in \mathbb{P}$ is a prime and $m \in \mathbb{N}$ a natural number, then there is a field $\mathbb{F}_{p^m}$ with characteristic p and p^m elements such that $\mathbb{F}_{p^m}$ is an extension field of the prime field $\mathbb{F}_p$.

Similarly to the way prime fields $\mathbb{F}_p$ are generated by starting with the ring of integers and then dividing by a prime number p and keeping the remainder, prime field extensions $\mathbb{F}_{p^m}$ are generated by starting with the ring $\mathbb{F}_p[x]$ of polynomials and then dividing them by an irreducible polynomial of degree m and keeping the remainder.

To be more precise, let $P \in F_p[x]$ be an irreducible polynomial of degree m with coefficients from the given prime field $\mathbb{F}_p$. Then the underlying set $\mathbb{F}_{p^m}$ of the extension field is given by the set of all polynomials with a degree less then m:

$$\mathbb{F}_{p^m} := \{a_{m-1}x^{m-1} + a_{k-2}x^{k-2} + \ldots + a_1 x + a_0 \mid a_i \in \mathbb{F}_p\} \tag{4.17}$$

which can be shown to be the set of all remainders when dividing any polynomial $Q \in \mathbb{F}_p[x]$ by P. So elements of the extension field are polynomials of degree less than m. This is analogous to how $\mathbb{F}_p$ is the set of all remainders when dividing integers by p.

Addition is then inherited from $\mathbb{F}_p[x]$, which means that addition on $\mathbb{F}_{p^m}$ is defined as normal addition of polynomials. To be more precise, we have

$$+ : \mathbb{F}_{p^m} \times \mathbb{F}_{p^m} \to \mathbb{F}_{p^m} \ , \ (\sum_{j=0}^{m} a_j x^j, \sum_{j=0}^{m} b_j x^j) \mapsto \sum_{j=0}^{m} (a_j + b_j) x^j \tag{4.18}$$

and we can see that the neutral element is (the polynomial) 0, and that the additive inverse is given by the polynomial with all negative coefficients.

Multiplication is inherited from $\mathbb{F}_p[x]$, too, but we have to divide the result by our modulus polynomial P, whenever the degree of the resulting polynomial is equal or greater to m. To be more precise, we have

$$\cdot \ \mathbb{F}_{p^m} \times \mathbb{F}_{p^m} \to \mathbb{F}_{p^m} \ , \ (\sum_{j=0}^{m} a_j x^j, \sum_{j=0}^{m} b_j x^j) \mapsto \left(\sum_{n=0}^{2m} \sum_{i=0}^{n} a_i b_{n-i} x^n \right) \bmod P \tag{4.19}$$

and we can see that the neutral element is (the polynomial) 1. It is, however, not obvious from this definition how the multiplicative inverse looks.

We can easily see from the definition of $\mathbb{F}_{p^m}$ that the field is of characteristic p, since the multiplicative neutral element 1 is equivalent to the multiplicative element 1 from the underlying prime field and hence $\sum_{j=0}^{p} 1 = 0$. Moreover, $\mathbb{F}_{p^m}$ is finite and contains p^m many elements, since elements are polynomials of degree $< m$ and every coefficient a_j can have a p number of different values. In addition, we see that the prime field $\mathbb{F}_p$ is a subfield of $\mathbb{F}_{p^m}$ that occurs when we restrict the elements of $\mathbb{F}_p$ to polynomials of degree zero.

One key point is that the construction of $\mathbb{F}_{p^m}$ depends on the choice of an irreducible polynomial, and, in fact, different choices will give different multiplication tables, since the remainders from dividing a product by P will be different.

It can, however, be shown that the fields for different choices of P are **isomorphic**, which means that there is a one-to-one correspondence between all of them, which means that, from an abstract point of view, they are the same thing. From an implementations point of view, however, some choices are preferable, because they allow for faster computations.

To summarize, we have seen that when a prime field $\mathbb{F}_p$ is given, then any field $\mathbb{F}_{p^m}$ constructed in the above manner is a field extension of $\mathbb{F}_p$. To be more general, a field $\mathbb{F}_{p^{m_2}}$ is a field extension of a field $\mathbb{F}_{p^{m_1}}$, if and only if m_1 divides m_2. From this, we can deduce that, for any given fixed prime number, there are nested sequences of fields

$$\mathbb{F}_p \subset \mathbb{F}_{p^{m_1}} \subset \cdots \subset \mathbb{F}_{p^{m_k}} \tag{4.20}$$

whenever the power m_j divides the power m_{j+1}, such that $\mathbb{F}_{p^{m_j}}$ is a subfield of $\mathbb{F}_{p^{m_{j+1}}}$.

To get a more intuitive picture of this, we construct an extension field of the prime field $\mathbb{F}_3$ in the following example, and we can see how $\mathbb{F}_3$ sits inside that extension field.

Example 63 (The Extension field $\mathbb{F}_{3^2}$). In (XXX) we have constructed the prime field $\mathbb{F}_3$. In this example, we apply the definition (XXX) of a field extension to construct $\mathbb{F}_{3^2}$. We start by choosing an irreducible polynomial of degree 2 with coefficients in $\mathbb{F}_3$. We try $P(t) = t^2 + 1$. Maybe the fastest way to show that P is indeed irreducible is to just insert all elements from $\mathbb{F}_3$ to see if the result is never zero. We compute

add reference

$$\begin{aligned} P(0) &= 0^2 + 1 = 1 \\ P(1) &= 1^2 + 1 = 2 \\ P(2) &= 2^2 + 1 = 1 + 1 = 2 \end{aligned}$$

This implies that P is irreducible. The set $\mathbb{F}_{3^2}$ then contains all polynomials of degrees lower than two, with coefficients in $\mathbb{F}_3$, which is precisely

$$\mathbb{F}_{3^2} = \{0, 1, 2, t, t+1, t+2, 2t, 2t+1, 2t+2\}$$

As expected, So our extension field contains 9 elements. Addition is defined as addition of polynomials. For example $(t+2) + (2t+2) = (1+2)t + (2+2) = 1$. Doing this computation for all elements gives the following addition table

+	0	1	2	t	t+1	t+2	2t	2t+1	2t+2
0	0	1	2	t	t+1	t+2	2t	2t+1	2t+2
1	1	2	0	t+1	t+2	t	2t+1	2t+2	2t
2	2	0	1	r+2	t	t+1	2t+2	2t	2t+1
t	t	t+1	t+2	2t	2t+1	2t+2	0	1	2
t+1	t+1	t+2	t	2t+1	2t+2	2t	1	2	0
t+2	t+2	t	t+1	2t+2	2t	2t+1	2	0	1
2t	2t	2t+1	2t+2	0	1	2	t	t+1	t+2
2t+1	2t+1	2t+2	2t	1	2	0	t+1	t+2	t
2t+2	2t+2	2t	2t+1	2	0	1	t+2	t	t+1

As we can see, the group $(\mathbb{F}_3,+)$ is a subgroup of the group $(\mathbb{F}_{3^2},+)$, obtained by only considering the first three rows and columns of this table.

As it was the case in previous examples, we can use the table to deduce the negative of any element from $\mathbb{F}_{3^2}$. For example, in $\mathbb{F}_{3^2}$ we have $-(2t+1) = t+2$, since $(2t+1)+(t+2) = 0$

Multiplication needs a bit more computation, as we first have to multiply the polynomials, and whenever the result has a degree ≥ 2, we have to divide it by P and keep the remainder. To see how this works, compute the product of $t+2$ and $2t+2$ in $\mathbb{F}_{3^2}$

$$\begin{aligned}
(t+2)\cdot(2t+2) &= (2t^2+2t+t+1) \bmod (t^2+1) \\
&= (2t^2+1) \bmod (t^2+1) && \# \, 2t^2+1 : t^2+1 = 2 + \frac{2}{t^2+1} \\
&= 2
\end{aligned}$$

So the product of $t+2$ and $2t+2$ in $\mathbb{F}_{3^2}$ is 2. Doing this computation for all elements gives the following multiplication table:

$\cdot$	0	1	2	t	t+1	t+2	2t	2t+1	2t+2
0	0	0	0	0	0	0	0	0	0
1	0	1	2	t	t+1	t+2	2t	2t+1	2t+2
2	0	2	1	2t	2t+2	2t+1	t	t+2	t+1
t	0	t	2t	2	t+2	2t+2	1	t+1	2t+1
t+1	0	t+1	2t+2	t+2	2t	1	2t+1	2	t
t+2	0	t+2	2t+1	2t+2	1	t	t+1	2t	2
2t	0	2t	t	1	2t+1	t+1	2	2t+2	t+2
2t+1	0	2t+1	t+2	t+1	2	2t	2t+2	t	1
2t+2	0	2t+2	t+1	2t+1	t	2	t+2	1	2t

As it was the case in previous examples, we can use the table to deduce the multiplicative inverse of any non-zero element from $\mathbb{F}_{3^2}$. For example in $\mathbb{F}_{3^2}$ we have $(2t+1)^{-1} = 2t+2$, since $(2t+1)\cdot(2t+2) = 1$.

From the multiplication table, we can also see that the only quadratic residues in $\mathbb{F}_{3^2}$ are the set $\{0,1,2,t,2t\}$, with $\sqrt{0} = \{0\}$, $\sqrt{1} = \{1,2\}$, $\sqrt{2} = \{t,2t\}$, $\sqrt{t} = \{t+2,2t+1\}$ and $\sqrt{2t} = \{t+1,2t+2\}$.

Since $\mathbb{F}_{3^2}$ is a field, we can solve equations as we would for other fields, like the rational numbers. To see that lets find all $x \in \mathbb{F}_{3^2}$ that solve the quadratic equation $(t+1)(x^2+(2t+2)) =$

2. So we compute:

$$
\begin{aligned}
(t+1)(x^2+(2t+2)) &= 2 && \#\ 2 \text{ distributive law}\\
(t+1)x^2+(t+1)(2t+2) &= 2 &&\\
(t+1)x^2+(t) &= 2 && \#\ 2 \text{ add the additive inverse of } t\\
(t+1)x^2+(t)+(2t) &= (2)+(2t) &&\\
(t+1)x^2 &= 2t+2 && \#\text{ multiply with the multiplicative invers of } t+1\\
(t+2)(t+1)x^2 &= (t+2)(2t+2) && \#\text{ multiply with the multiplicative invers of } t+1\\
x^2 &= 2 && \#\ 2 \text{ is quadratic residue. Take the roots.}\\
x &\in \{t,2t\}
\end{aligned}
$$

Computations in extension fields are arguably on the edge of what can reasonably be done with pen and paper. Fortunately, Sage provides us with a simple way to do the computations.

```
sage: Z3 = GF(3) # prime field
sage: Z3t.<t> = Z3[] # polynomials over Z3
sage: P = Z3t(t^2+1)
sage: P.is_irreducible()
True
sage: F3_2.<t> = GF(3^2, name='t', modulus=P)
sage: F3_2
Finite Field in t of size 3^2
sage: F3_2(t+2)*F3_2(2*t+2) == F3_2(2)
True
sage: F3_2(2*t+2)^(-1) # multiplicative inverse
2*t + 1
sage: # verify our solution to (t+1)(x^2 + (2t+2)) = 2
sage: F3_2(t+1)*(F3_2(t)**2 + F3_2(2*t+2)) == F3_2(2)
True
sage: F3_2(t+1)*(F3_2(2*t)**2 + F3_2(2*t+2)) == F3_2(2)
True
```

Exercise 32. Consider the extension field $\mathbb{F}_{3^2}$ from the previous example and find all pairs of elements $(x,y) \in \mathbb{F}_{3^2}$, such that

$$y^2 = x^3 + 4$$

Exercise 33. Show that the polynomial $P = x^3 + x + 1$ from $\mathbb{F}_5[x]$ is irreducible. Then consider the extension field $\mathbb{F}_{5^3}$ defined relative to P. Compute the multiplicative inverse of $(2t^2+4) \in \mathbb{F}_{5^3}$ using the extended Euklidean algorithm. Then find all $x \in \mathbb{F}_{5^3}$ that solve the equation

$$(2t^2+4)(x-(t^2+4t+2)) = (2t+3)$$

Hashing into extension fields In XXX we have seen how to hash into prime fields. As elements of extension fields can be seen as polynomials over prime fields, hashing into extension fields is therefore possible, if every coefficient of the polynomial is hashed independently.

add reference

4.4 Projective Planes

Projective planes are a certain type of geometry defined over a given field. In a sense, projective plances extend the concept of the ordinary Euclidean plane by including “points at infinity.”

a certain type of geometry

Such an inclusion of infinity points makes them particularly useful in the description of elliptic curves, as the description of such a curve in an ordinary plane needs an additional symbol “the point at infinity” to give the set of points on the curve the structure of a group. Translating the curve into projective geometry, then includes this “point at infinity” more naturally into the set of all points on a projective plane.

To understand the idea for the construction of projective planes, note that in an ordinary Euclidean plane, two lines either intersect in a single point, or are parallel. In the latter case both lines are either the same, that is, they intersect in all points, or do not intersect at all. A projective plane can then be thought of as an ordinary plane, but equipped with additional “points at infinity” such that two different lines always intersect in a single point. Parallel lines intersect “at infinity".

To be more precise, let $\mathbb{F}$ be a field, $\mathbb{F}^3 := \mathbb{F}\times\mathbb{F}\times F$ the set of all three tuples over $\mathbb{F}$ and $x\in\mathbb{F}^3$ with $x=(X,Y,Z)$. Then there is exactly one *line* in $\mathbb{F}^3$ that intersects both $(0,0,0)$ and x. This line is given by

$$[X:Y:Z] := \{(k\cdot X, k\cdot Y, k\cdot Z) \mid k\in\mathbb{F}\} \tag{4.21}$$

A **point** in the **projective plane** over $\mathbb{F}$ is then defined as such a *line* and the projective plane is the set of all such points, that is

$$\mathbb{FP}^2 := \{[X:Y:Z] \mid (X,Y,Z)\in\mathbb{F}^3 \text{ with } (X,Y,Z)\neq(0,0,0)\} \tag{4.22}$$

It can be shown that a projective plane over a finite field $\mathbb{F}_{p^m}$ contains $p^{2m}+p^m+1$ many elements.

To understand why $[X:Y:Z]$ is called a line, consider the situation where the underlying field $\mathbb{F}$ are the real numbers $\mathbb{R}$. Then $\mathbb{R}^3$ can be seen as the three dimensional space and $[X:Y:Z]$ is then an ordinary line in this 3-dimensional space that intersects zero and the point with coordinates X, Y and Z.

The key observation here is that points in the projective plane are lines in the 3-dimensional space $\mathbb{F}^3$, also for finite fields, the terms space and line share very little visual similarity with their counterparts over the real numbers.

It follows from this that points $[X:Y:Z]\in\mathbb{FP}^2$ are not simply described by fixed coordinates (X,Y,Z), but by **sets of coordinates** rather, where two different coordinates (X_1,Y_1,Z_1) and (X_2,Y_2,Z_2), with describe the same point, if and only if there is some field element k, such that $(X_1,Y_1,Z_1)=(k\cdot X_2, k\cdot Y_2, k\cdot Z_2)$. Point $[X:Y:Z]$ are called **projective coordinates**.

Notation and Symbols 6 (Projective coordinates). Projective coordinates of the form $[X:Y:1]$ are descriptions of so-called **affine points** and projective coordinates of the form $[X:Y:0]$ are descriptions of so-called **points at infinity**. In particular the projective coordinate $[1:0:0]$ describes the so-called **line at infinity**.

Example 64. Consider the field $\mathbb{F}_3$ from example XXX. As this field only contains three elements, it does not take too much effort to construct its associated projective plane $\mathbb{F}_3\mathbb{P}^2$, as we know that it only contain 13 elements.

add reference

To find $\mathbb{F}_3\mathbb{P}^2$, we have to compute the set of all lines in $\mathbb{F}_3\times\mathbb{F}_3\times\mathbb{F}_3$ that intersect $(0,0,0)$.

Since those lines are parameterized by tuples (x_1,x_2,x_3). We compute:

$$\begin{aligned}
[0:0:1] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(0,0,1),(0,0,2)\}\\
[0:0:2] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(0,0,2),(0,0,1)\} = [0:0:1]\\
[0:1:0] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(0,1,0),(0,2,0)\}\\
[0:1:1] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(0,1,1),(0,2,2)\}\\
[0:1:2] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(0,1,2),(0,2,1)\}\\
[0:2:0] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(0,2,0),(0,1,0)\} = [0:1:0]\\
[0:2:1] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(0,2,1),(0,1,2)\} = [0:1:2]\\
[0:2:2] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(0,2,2),(0,1,1)\} = [0:1:1]\\
[1:0:0] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(1,0,0),(2,0,0)\}\\
[1:0:1] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(1,0,1),(2,0,2)\}\\
[1:0:2] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(1,0,2),(2,0,1)\}\\
[1:1:0] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(1,1,0),(2,2,0)\}\\
[1:1:1] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(1,1,1),(2,2,2)\}\\
[1:1:2] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(1,1,2),(2,2,1)\}\\
[1:2:0] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(1,2,0),(2,1,0)\}\\
[1:2:1] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(1,2,1),(2,1,2)\}\\
[1:2:2] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(1,2,2),(2,1,1)\}\\
[2:0:0] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(2,0,0),(1,0,0)\} = [1:0:0]\\
[2:0:1] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(2,0,1),(1,0,2)\} = [1:0:2]\\
[2:0:2] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(2,0,2),(1,0,1)\} = [1:0:1]\\
[2:1:0] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(2,1,0),(1,2,0)\} = [1:2:0]\\
[2:1:1] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(2,1,1),(1,2,2)\} = [1:2:2]\\
[2:1:2] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(2,1,2),(1,2,1)\} = [1:2:1]\\
[2:2:0] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(2,2,0),(1,1,0)\} = [1:1:0]\\
[2:2:1] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(2,2,1),(1,1,2)\} = [1:1:2]\\
[2:2:2] &= \{(k\cdot x_1,k\cdot x_2,k\cdot x_3)\mid k\in\mathbb{F}_3\} = \{(2,2,2),(1,1,1)\} = [1:1:1]
\end{aligned}$$

Those lines define the 13 points in the projective plane $\mathbb{F}_3\mathbb{P}$ as follows

$$\begin{aligned}
\mathbb{F}_3\mathbb{P} = \{&[0:0:1],[0:1:0],[0:1:1],[0:1:2],[1:0:0],[1:0:1],\\
&[1:0:2],[1:1:0],[1:1:1],[1:1:2],[1:2:0],[1:2:1],[1:2:2]\}
\end{aligned}$$

This projective plane contains 9 affine points, three points at infinity and one line at infinity.

To understand the ambiguity in projective coordinates a bit better, let us consider the point $[1:2:2]$. As this point in the projective plane is a line in $\mathbb{F}_3^3$, it has the projective coordinates $(1,2,2)$ as well as $(2,1,1)$, since the former coordinate give the latter, when multiplied in $\mathbb{F}_3$ by the factor 2. In addition, note that for the same reasons the points $[1:2:2]$ and $[2:1:1]$ are the same, since their underlying sets are equal.

Exercise 34. Construct the so-called *Fano plane*, that is, the projective plane over the finite field $\mathbb{F}_2$.

Bibliography

Jens Groth. On the size of pairing-based non-interactive arguments. *IACR Cryptol. ePrint Arch.*, 2016:260, 2016. URL http://eprint.iacr.org/2016/260.

P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994. doi: 10.1109/SFCS.1994.365700.

David Fifield. The equivalence of the computational diffie–hellman and discrete logarithm problems in certain groups, 2012. URL https://web.stanford.edu/class/cs259c/finalpapers/dlp-cdh.pdf.

Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 129–140, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. ISBN 978-3-540-46766-3. URL https://fmouhart.epheme.re/Crypto-1617/TD08.pdf.

Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. Cryptology ePrint Archive, Report 2016/492, 2016. https://ia.cr/2016/492.