One-way functions from factoring, discrete logarithms, and SIS

Noah Stephens-Davidowitz

May 29, 2023

1 One-way functions recap

Recall from the previous lecture notes the definition of a one-way function. In fact, we had two definitions, one strong and one weak, though weak one-way functions are only interesting because we can use them to construct strong one-way functions. (To be clear, in the future when we say "one-way functions" without qualification, we will always mean strong one-way functions.)

Definition 1.1. *A function $f : \{0,1\}^* \to \{0,1\}^*$ is called* (strongly) one-way *if it satisfies the following.*

1. **Easy to compute.** *There is a probabilistic polynomial-time algorithm computing f.*
2. **Hard to invert.** *For all probabilistic polynomial-time adversaries $\mathcal{A}$, there exists a negligible $\varepsilon(n)$ such that*

$$\Pr_{\boldsymbol{x}\sim\{0,1\}^n}[\mathcal{A}(1^n, f(\boldsymbol{x})) = \boldsymbol{x}' \ : \ f(\boldsymbol{x}') = f(\boldsymbol{x})] \le \varepsilon(n)$$

for all $n \ge 1$.

f is called weakly one-way *if it satisfies the following instead of Item 2.*

2'. **Weakly hard to invert.** *There exists a polynomial $p(n)$ such that for all probabilistic polynomial-time adversaries $\mathcal{A}$, there exists $n_0 \ge 1$ such that*

$$\Pr_{\boldsymbol{x}\sim\{0,1\}^n}[\mathcal{A}(1^n, f(\boldsymbol{x})) = \boldsymbol{x}' \ : \ f(\boldsymbol{x}') = f(\boldsymbol{x})] \le 1 - 1/p(n)$$

for all $n \ge n_0$.

In the previous lecture notes, we saw a candidate for a weak one-way function: multiplication. We also saw that weak one-way functions imply strong one-way functions. So, we already know how to build strong one-way functions (assuming that factoring is suitably hard). Here, we introduce more constructions of one-way functions, as well as some of the mathematical background that will be useful for the rest of the course.

2 "Efficiently sampleable one-way functions" and a better one-way function from the hardness of factoring

Remember that, in our attempt to build a one-way function from the hardness of factoring, we ran into an annoying problem: it is *not* hard to factor a uniformly random n-bit integer (or, more accurately, the product of two uniformly random $n/2$-bit integers). One way to view the problem is as a sort of type mismatch between our definition of a one-way function, which asks for hardness to invert on *uniformly random input*, and the hardness of factoring, which applies to a different distribution—e.g., it is likely hard to find a factor of the product of two random n-bit primes, but not to find a factor of a random integer.

More generally, imagine that we have some function f and some distribution D such that it is hard to invert $f(x)$ when x is sampled from D. (Really, we should consider a family of distributions D_n, one for each security parameter. But, we often ignore such formalities.) Shouldn't this be just as useful as a one-way function?

The answer in general is no! In particular, if there is no algorithm that efficiently samples from D (or, more accurately, a PPT algorithm that samples from D_n on input 1^n), then how could a PPT algorithm make use of this function? However, if D is efficiently sampleable, then we're in business.

For example, we think that it is hard to find a non-trivial factor of pq, where p and q are uniformly random n-bit *primes*. So, perhaps instead of running our one-way function $f_{\mathsf{mult}}(p, q) = pq$ on random n-bit integers, why not simply run it on uniformly random n-bit primes? This doesn't quite fit our definition of a one-way function, so let's change the definition!

In particular, we define what we will call "an efficiently sampleable one-way function." We will then show that one-way functions exist if and only if efficiently sampleable one-way functions exist. This will allow us to move between different definitions. E.g., sometimes, we will show that one-way functions exist under a certain assumption by constructing something that satisfies the definition below.

Definition 2.1. *An* efficiently sampleable one-way function *consists of a domain D, range R, function $f : D \to R$, and a PPT algorithm* Samp *with the following properties.*

1. **Correct.** Samp *takes as input 1^n and always outputs an element in the domain D.*
2. **Easy to compute.** *There is a PPT algorithm $\mathcal{B}$ such that for any $x \in D$, $\mathcal{B}(x) = f(x)$.*
3. **Hard to invert.** *For any PPT adversary $\mathcal{A}$ there exists negligible ε such that*

   $$\Pr_{x \leftarrow \mathsf{Samp}(1^n)}[x' \leftarrow \mathcal{A}(1^n, f(x)) \ : \ f(x') = f(x)] \le \varepsilon(n)$$

   for all $n \ge 1$.

In other words, an efficiently sampleable one-way function is just like a one-way function except that (1) its domain and range are not necessarily $\{0,1\}^*$; and (2) it is hard to invert when its input distribution is $\mathsf{Samp}(1^n)$, which is not necessarily uniformly random over $\{0,1\}^n$.

We will now show that one-way functions exist if and only if efficiently sampleable one-way functions exist. Notice that one direction is easy: if a one-way function f exists, then we can build an efficiently sampleable one-way function by just taking $D = R = \{0,1\}^*$ and having the sampling algorithm $\mathsf{Samp}(1^n)$ output a uniformly random n-bit string. The more difficult thing to prove is that any efficiently sampleable one-way function implies the existence of a standard one-way function.

To prove this, we use a simple and common trick. Recall that a PPT algorithm like Samp flips $m \le n^C$ coins during its computation (when the input consists of n bits), where $C > 0$ is some constant. For $r \in \{0,1\}^m$, we write $\mathsf{Samp}(1^n; r)$ for the output of the Samp algorithm when its input is 1^n and its coins are fixed to be r. Then, $g(r) := \mathsf{Samp}(1^{\lfloor |r|^{1/C} \rfloor}; r)$ is a function (i.e., each input corresponds to exactly one output). Furthermore, if $r \sim \{0,1\}^m$, then $g(r)$ is distributed exactly as $\mathsf{Samp}(1^{\lfloor |r|^{1/C} \rfloor})$. We therefore use this to move between uniformly random bit strings, and arbitrary efficiently sampleable distributions.

Here is the formal theorem.

Theorem 2.2. *One-way functions exist if and only if efficiently sampleable one-way functions exist.*

Proof. As we observed above, one direction is easy. In particular, suppose that f is a one-way function. Then, consider the candidate efficiently sampleable one-way function with the same function f, $D = R = \{0,1\}^*$, and $\mathsf{Samp}(1^n) \sim \{0,1\}^n$. In other words, we take the Samp algorithm to simply be the algorithm that flips n coins and outputs the result. A quick check of the definitions shows that this is in fact an efficiently sampleable one-way function.

Now, suppose that D, R, Samp, and f form an efficiently sampleable one-way function. Let $C > 0$ be such that $\mathsf{Samp}(1^n)$ uses at most n^C random coins. Consider the function f' defined by $f'(r) := f(\mathsf{Samp}(1^{\lfloor |r|^{1/C} \rfloor}; r))$. In other words, $f'(r)$ is what you get from applying f to the output of Samp when Samp is run with random coins fixed to r.

It is immediate that f' is efficiently computable. (Specifically, computing f' consists of computing $\lfloor |r|^{1/C} \rfloor$, running Samp, and then computing f. All of these operations can be done in polynomial time by the assumption that f is efficiently computable and Samp is PPT. We may also assume without loss of generality that the output of f' lies in $\{0,1\}^*$, since anyway it has to be something that a computer program can write down.)

To prove that f' is hard to invert, we suppose for contradiction that there exists some PPT adversary $\mathcal{A}$ such that

$$\varepsilon(m) := \Pr_{r \sim \{0,1\}^m}[r' \leftarrow \mathcal{A}(1^m, f'(r)) \ : \ f'(r') = f'(r)]$$

is non-negligible. Then, we construct an adversary $\mathcal{A}'$ in the efficiently sampleable one-way function game against f as follows. Let $n := \lfloor m^{1/C} \rfloor$. $\mathcal{A}'$ takes as input 1^n and $y^* := f(x^*)$, where $x^* := \mathsf{Samp}(1^n; r^*)$ with $r^* \sim \{0,1\}^m$. $\mathcal{A}'$ then runs $\mathcal{A}$ on input 1^m and y^*,[1] receiving as output r'. $\mathcal{A}'$ then outputs $x' := \mathsf{Samp}(1^n; r')$.

[1]I am cheating here a tiny bit in that I am implicitly assuming that $\mathcal{A}'$ "knows" m. But, notice that $\mathcal{A}'$ only takes 1^n (and y^*) as input, and that m is not uniquely determined by n (because of the floor function). Of course, $\mathcal{A}'$ could fix $m := n^C$, but it could be the case that $\mathcal{A}$ has non-negligible advantage for general m but, e.g., never succeeds when m is a Cth power, in which case $\mathcal{A}'$ would never succeed even though $\mathcal{A}$ has non-negligible advantage in general. (Adversaries can be very annoying. In fact, they are adversarial!) One solution to this is to have $\mathcal{A}'$ guess m uniformly at random from e.g., the integers m between n^C and $(n+1)^C$ (which are all integers m such that $\lfloor m^{1/C} \rfloor = n$). Then, with probability at least, say, $1/(n+1)^C$, we will guess correctly, and since this probability is non-negligible, our resulting advantage is also non-negligible. We can also simply run $\mathcal{A}$ many times, once for each m in this interval, check if any of the outputs are valid inverses, and if so, output the valid inverse. But, anyway, we ignore these issues here because they are super annoying and our patience is limited.

Notice that $\mathcal{A}'$ is in fact PPT.

More interestingly, notice that if $f'(r') = f'(r^*)$, then $f(x^*) = f(x')$. It follows that

$$\varepsilon'(n) := \Pr[f(x') = y^*] \geq \Pr[f'(r') = f'(r^*)] = \varepsilon(m) = \varepsilon(n^C) ,$$

which is non-negligible. This contradicts the assumption that f is an efficiently sampleable one-way function. So, f' must be one way. □

(Notice a clever trick that we use in this proof. We observe that, in order to find a preimage x' of f, it suffices to find coins r' such that $x' = \mathsf{Samp}(1^n; r')$.)

This theorem is quite useful, and we will often invoke it implicitly. For example, we will often consider one-way functions whose range is restricted to n-bit strings where, e.g., n is even or n is a perfect square or something like this. This is justified by noting that these fit the definition of efficiently sampleable one-way functions (with the sampling algorithm sampling a uniformly random bit string of even or square length or whatever), and therefore their existence is equivalent to the existence of one-way functions whose domains are arbitrary bit strings.

2.1 Finishing our factoring example

For completeness, we now show how to construct an efficiently sampleable one-way function from the hardness of factoring. Specifically, we use the following assumption.

Assumption 2.3 (One version of hardness of factoring)**.** *For any PPT $\mathcal{A}$ there exists negligible ε such that*

$$\Pr_{p,q \sim \mathbb{P}_n}[(p', q') \leftarrow \mathcal{A}(1^n, pq) \ : \ p'q' = pq, \ p', q' > 1] \leq \varepsilon(n) ,$$

where $\mathbb{P}_n = \{2 \leq p \leq 2^n - 1 \ : \ p \text{ is prime}\}$ is the set of n-bit primes.

We now recall two crucial facts. The first crucial fact is that a uniformly random n-bit integer is prime with probability $\Theta(1/n)$. (This is known as the Prime Number Theorem.) The second crucial fact is that there are polynomial-time algorithms that determine whether a given integer is prime. Taken together, these two facts imply that there is a PPT algorithm Samp such that $\mathsf{Samp}(1^n)$ outputs a uniformly random n-bit prime. In particular, this algorithm repeatedly samples a uniformly random n-bit integer, tests if its prime, and if it isn't repeats. This runs in polynomial time because its probability of success in each attempt is $\Theta(1/n)$, which means that its expected number of steps is $\Theta(n)$.[2]

So, we can build an efficiently sampleable one-way function whose domain is the set of primes $D := \mathbb{P}$, range is $R := \mathbb{N}$, sampling function is as described above, and the function itself is simply f_{mult} (restricted to this domain). Assumption 2.3 is exactly equivalent to the assumption that this is hard to invert.

[2]I am being slightly misleading here. In the proof of Theorem 2.2, we assumed that there was some fixed C such that the number of coins flipped by $\mathsf{Samp}(1^n)$ was globally bounded by n^C. But, the Samp algorithm that I have described here has some tiny but non-zero probability of sampling any number of coins, since it might get unlikely and keep failing to find a prime. To make the two definitions fit, we must, e.g., declare that $\mathsf{Samp}(1^n)$ always outputs the prime 2 if it fails to find a prime after, say, n^2 tries. Since this event happens with negligible probability, it only alters the output distribution of Samp by a negligible amount and therefore does not affect the security of the resulting one-way function. However, we typically ignore this minor distinction between expected polynomial-time randomized algorithms and guaranteed polynomial-time randomized algorithms.

3 The discrete logarithm

3.1 Background: commutative groups

We will need to introduce the notion of a group. Actually, we will only need a special kind of group, which we will call commutative groups. They are often also called "Abelian groups" in honor of Abel (pronounced uh-beel'-yun and ah'-bull respectively).

Definition 3.1. *A* commutative group $(G, \cdot)$ *is a set G with a binary operation $\cdot$ over G, written $g \cdot h$, with the following properties.*

1. **Closure.** *For all $g, h \in G$, $g \cdot h \in G$.*
2. **Identity.** *There exists an element $e \in G$ (called the identity element) such that $e \cdot g = g \cdot e = g$ for all $g \in G$.*
3. **Inverses.** *For every $g \in G$, there exists a $g^{-1} \in G$ such that $g \cdot g^{-1} = e$.*
4. **Associativity.** *For every $g_1, g_2, g_3 \in G$, $(g_1 \cdot g_2) \cdot g_3 = g_1 \cdot (g_2 \cdot g_3)$.*
5. **Commutativity.** *For every $g, h \in G$, $g \cdot h = h \cdot g$.*

We are only interested in groups with finitely many elements, and we refer to the number of elements $|G|$ as the order *of the group.*

Our simplest example is $(\mathbb{Z}_q, +)$, the additive group modulo an integer $q \geq 2$. I.e., the set is simply $\mathbb{Z}_q := \{0, 1, \ldots, q-1\}$, and the group operation is addition modulo q, $g + h \bmod q$. Clearly, this is a commutative group. In particular, 0 is the identity element, and every $g \in \mathbb{Z}_q$ has as its inverse $q - g \in \mathbb{Z}_q$. Its order is q.

The other example that interests us is the multiplicative group modulo q, $(\mathbb{Z}_q^*, \cdot)$. The set is $\mathbb{Z}_q^* := \{g \in \mathbb{Z}_q \ : \ \gcd(g, q) = 1\}$, and the group operation is multiplication modulo q, i.e., $g \cdot h \bmod q$. It is a bit less clear that $\mathbb{Z}_q^*$ is a group. The identity is of course 1, and it is easy to check that it is closed, associative, and commutative. But, it's less obvious that elements in this group have inverses. Notice that an element h is an inverse of g modulo q if and only if there exists some integer k such that $gh + kq = 1$. We recall that the extended Euclidean algorithm finds integers h, k satisfying this identity (sometimes called Bézout's identity). So, $h = g^{-1} \bmod q$ certainly exists, and we can even find it efficiently, given g and q. Notice that this holds if and only if $\gcd(g, q) = 1$, i.e., if and only if $g \in \mathbb{Z}_q^*$.

The order of $\mathbb{Z}_q^*$ is often written as $\phi(q) := |\mathbb{Z}_q^*|$, and this function ϕ is called Euler's ϕ function or Euler's totient function. I.e., $\phi(q)$ is the number of natural numbers less than q that are coprime to q. For prime q, $\phi(q) = q - 1$. More generally, if $q = p_1^{a_1} \cdots p_\ell^{a_\ell}$ for distinct primes $p_1, \ldots, p_\ell$ with $a_i \geq 1$, then $\phi(q) = p_1^{a_1 - 1}(p_1 - 1) \cdot p_2^{a_2 - 1}(p_2 - 1) \cdots p_\ell^{a_\ell - 1}(p_\ell - 1)$.

3.2 Background: exponentiation in a group

We write $g^k := g \cdot g \cdots g$ for the product of $g \in G$ with itself $k \geq 1$ times—or, if the operation is addition, we write $kg := g + g + \cdots + g$. We also define $g^{-k} := (g^{-1})^k$ and $g^0 := e$. With these conventions, this operation satisfies the basic properties of exponentiation: $(g^k)^\ell = g^{k\ell}$, $g^{-k} = (g^k)^{-1}$, $g^k \cdot g^\ell = g^{k+\ell}$, and $(hg)^k = h^k g^k$.

Now, G is finite. Therefore, the sequence $g^1, g^2, g^3, \ldots,$ must eventually repeat itself. I.e., $g^k = g^\ell$ for some $k > \ell$. Multiplying by $g^{-\ell}$ on both sides, we see that $g^{k-\ell} = e$. So, for every element g in the group, there exists some $k \geq 1$ such that $g^k = e$. The minimal such k is called the *order* of g. (This overuse of the word order is a bit annoying.) A key fact (known as Lagrange's theorem and proven in Appendix A) is that the order k of the element must divide the order $|G|$ of the group.

A group is *cyclic* if all its elements can be written as a power of one fixed element $g \in G$, i.e., $G = \{e, g, g^2, \ldots, g^{|G|-1}\}$. Equivalently, a group is cyclic if it has an element with order $|G|$. We call such a g a *generator*.

For example, the additive group $\mathbb{Z}_q$ is cyclic, with 1 as a generator. More generally, any element coprime to q is a generator. So, there are actually $\phi(q)$ generators of $\mathbb{Z}_q$.

The multiplicative group $\mathbb{Z}_q^*$ is not always cyclic, but it *is* cyclic when q is prime. (See Appendix B for a proof.) I.e., there is some element g whose order equals the order $q-1$ of the entire group. In fact, since one such element exists, there must be many. To see this, we recall that if g is a generator of $\mathbb{Z}_q^*$, then every element in the group can be written as g^k for some k. Notice that $g^{k\ell} = 1$ if and only if $k\ell$ is a multiple of $q-1$. It follows that the order of g^k is exactly $(q-1)/\gcd(k, q-1)$. In particular, g^k is a generator whenever k and $q-1$ are coprime. So, the number of generators is $\phi(q-1)$. (More generally, the number of generators of a cyclic group G is always $\phi(|G|)$.)

3.3 Groups as computational objects, and the repeated squaring algorithm

Since we are cryptographers interested in asymptotic complexity, we need a security parameter n, and our group operation should be efficiently computable in the security parameter. This notion does not really make sense for a fixed group G, so, we will actually need a sequence $G_1, G_2, G_3, \ldots,$ of groups, say with $|G_n| \approx 2^n$. Let's assume that group elements are represented by n-bit strings and that group operations are computable in $\mathrm{poly}(n)$ time, given a description of the group G_n as advice. We will quickly get tired of dragging the parameter n around, and we will drop it. (This is very common even in formal papers about cryptography. We are often formally interested in sequences of objects $A_1, A_2, A_3, \ldots,$, one for each possible security parameter. But, we often lazily pretend that the security parameter is fixed and that we are interested in only a single object A.) But, we will keep it for now to make clear that it is formally necessary.

For example, G_n could be $\mathbb{Z}_{q_n}$ or $\mathbb{Z}_{q_n}^*$ for some n-bit number q_n. Addition modulo q_n takes $O(\log q_n) = O(n)$ time, and multiplication modulo q_n can be done in $O(\log^2 q_n) = O(n^2)$ time (or even in $O(n \log n)$ time [HvdH21]). So, both of these groups have efficiently computable group operations. For these two groups, the inverse $g^{-1} \in G_n$ is efficiently computable as well.

But, what about computing $g^k \in G_n$? The naive algorithm just computes $g, g^2, g^3, \ldots, g^k$, which requires us to compute k group operations. This is not a polynomial in the bit length $\approx \log k$ of k. But there is a better algorithm for this—called the repeated squaring algorithm—that allows us to use only $O(\log k)$ group operations! We can compute $g^1, g^2, g^4, \ldots, g^{2^{\ell-1}}, g^{2^\ell} \in G_n$, where $\ell := \lfloor \log_2 k \rfloor$, using a total of ℓ group operations by noticing that $g^{2^{i+1}} = g^{2^i} \cdot g^{2^i}$. Since we can write k as a sum of $O(\log k)$ numbers of the form $1, 2, 4, \ldots, 2^\ell$ (by writing k in binary), we can write g^k as $O(\log k)$ products of the $g^{2^i} \in G_n$. This allows us to compute g^k in just $O(\log k)$ total group operations, as claimed.

3.4 The discrete logarithm

We are now ready to present our one-way function! Assume we have a sequence $G_1, G_2, G_3, \ldots,$ of groups together with generators $g_1, g_2, g_3, \ldots,$. Our one-way function f is simply $f(k) := g_n^k \in G_n$, where $k \in \{0,1\}^n$ is an n-bit string interpreted as an integer. Notice that this is efficiently computable if the group operation in G_n is efficiently computable, since as we argued above, g_n^k is computable using $O(\log k) = O(n)$ group operations.

The problem of inverting this function f is called the discrete logarithm problem. I.e., given a generator $g_n \in G_n$ and another element $h \in G_n$, the discrete logarithm problem is to find k such that $g_n^k = h$. We write $\log_{g_n}(h)$ for the unique $0 \le k \le |G_n| - 1$ such that $g_n^k = h$. (Notice the similarity with the "continuous logarithm." E.g., $\log_2 128$ is the number x such that $2^x = 128$.)

So, when is this hard? For the additive group $\mathbb{Z}_{q_n}$, the discrete logarithm problem is easy, i.e., solvable in time $\mathrm{poly}(\log q_n) = \mathrm{poly}(n)$. Indeed, the discrete logarithm over $\mathbb{Z}_{q_n}$ is the following. We are given $g, h \in \mathbb{Z}_{q_n}$ with g coprime to q_n, and we are asked to find k such that $kg = h \bmod q_n$. To do so, it suffices to compute the inverse of $g \bmod q_n$, i.e., the element $r \in G$ such that $gr = 1 \bmod q_n$. We observed earlier that the Euclidean algorithm lets us compute this efficiently. Then, we can find k by computing $k = hg^{-1} \bmod q_n$.

For the multiplicative group $\mathbb{Z}_{q_n}^*$, things are more interesting. As far as we know, the discrete logarithm is in fact hard over $\mathbb{Z}_{q_n}^*$. Specifically, the best known algorithm for the discrete logarithm over $\mathbb{Z}_{q_n}^*$ runs in time $2^{O(\log^{1/3} q_n (\log\log q_n)^{2/3})} = 2^{O(n^{1/3} \log^{2/3} n)}$, which is superpolynomial in n. So, we can fix some sequence q_n of n-bit primes and generators g_n of $\mathbb{Z}_{q_n}^*$, and we believe that the function $f(k) := g_n^k \bmod q_n^*$ is in fact one way. (There are groups of size roughly 2^n over which the fastest known algorithm for the discrete logarithm runs in time $2^{n/2}$. Because of this, these groups, which are based on elliptic curves, are used in practice—and sometimes in theory as well.)

3.5 Where do G_n and g_n come from?

If we are comfortable with non-uniform algorithms, then we do not formally need to worry so much about where G_n and g_n come from. Technically, we can just provide them as advice to the algorithm that computes our one-way function f. In other words, we can hard code G_n and g_n into our algorithm.

But, it's a bit unsatisfying to say that G_n and g_n just fall from the sky, and if we want to work with uniform algorithms, it's unacceptable. For this one-way function to be useful, there obviously has to be some way to find G_n and g_n efficiently. So, we now show how to efficiently find an n-bit prime q together with a generator g of $\mathbb{Z}_q^*$.

So, first of all, how do we find n-bit primes q? Even this is not entirely trivial. There is no deterministic $\mathrm{poly}(n)$-time algorithm known (though very simple algorithms work under certain very strong number-theoretic conjectures). But, with randomness, it is relatively straightforward. We pick a random n-bit number, use our favorite efficient primality testing algorithm to test if it is prime, and repeat this until we find one. The Prime Number Theorem tells us that our guess will be prime with probability roughly $1/n$, so that we are likely to find a prime after n or so tries. (We mentioned this above when we built and efficiently sampleable one-way function from factoring.)

Finding generators is harder. We mentioned earlier that $\mathbb{Z}_q^*$ has a lot of generators, $\phi(q-1)$ of them. ($\phi(m)$ is always at least $\Omega(m/\log\log m)$. I.e., for n-bit primes, at least a $\Omega(1/\log n)$ fraction of the elements are generators.) So, if we had some way to test whether an element $g \in \mathbb{Z}_q^*$ is a generator, then we could find one using the same guess-and-check trick that allows us to find primes. And, since the order of g must divide the order of the group, the possible orders for the element g correspond to the factors of $|\mathbb{Z}_q^*| = q-1$. If we knew the non-trivial factors of $q-1$, say, $d_1, \ldots, d_\ell$, then we could test whether g is a generator by checking whether $g^{d_i} = 1 \bmod q$ for all i. g is a generator if and only if $g^{d_i} \neq 1 \bmod q$ for all i. There are at most $\log_2 q$ factors, so we could do this efficiently, given the factors.

Unfortunately, factoring $q-1$ seems to be hard. (Maybe that's actually fortunate.) But, we can use a trick to get around this: instead of sampling a prime q and then trying to factor $q-1$, we can sample a factorization of $q-1$ first and then test whether q is prime. There are beautiful algorithms to do this that achieve uniformly random q [Kal02], but in practice, we do the following. We sample a uniformly random $(n-1)$-bit prime p using the guess-and-check technique described above. If $q = 2p+1$ is prime, then take this to be q. Otherwise, resample p until this is true.

Primes p, q satisfying $q = 2p+1$ are called Sophie Germain primes (after Marie-Sophie Germain). More specifically, p is called a Sophie Germain prime, and q is called a safe prime. We believe that a random n-bit number will be a Sophie Germain prime with probability roughly $1/n^2$ (i.e., the probability that two random n-bit numbers p and q are prime). But, like many things in number theory (and many things in cryptography), we do not know how to prove this. (Sophie Germain primes are used quite a lot in practice, and in practice, this works.)

Notice that we conveniently know the factorization of $q-1$ when q is a safe prime. Specifically, $q-1 = 2p$ is a complete factorization. So, to find a generator of $\mathbb{Z}_q^*$, we sample a random element g and check if $g^2 = 1 \bmod q$ or $g^p = 1 \bmod q$. If not, then g is a generator. (Safe primes are quite useful in number-theoretic cryptography in general because of this nice property.)

4 One-way function families

To make the approach described above work, we need to modify the definition of one-way function slightly. Instead of one function f, we have a family of functions f_k indexed by a key k and an efficient algorithm Gen that takes as input 1^n and outputs k. E.g., Gen can be the algorithm described above to find an n-bit safe prime q and a generator g of $\mathbb{Z}_q^*$. It outputs q and g. The formal definition is below.

Definition 4.1. *A* one-way function family *is an efficient algorithm* Gen *that takes as input* 1^n *and outputs* $k \in \mathcal{K}$ *and a function* $f : \mathcal{K} \times \{0,1\}^* \to \{0,1\}^*$ *satisfying the following properties.*

1. **Easy to compute.** *There is a probabilistic polynomial-time algorithm computing* f*.*

2. **Hard to invert.** *For all probabilistic polynomial-time adversaries* $\mathcal{A}$*, there exists a negligible* $\varepsilon(n)$ *such that*

$$\Pr_{\boldsymbol{x} \leftarrow \{0,1\}^n, k \leftarrow G(1^n)}[\mathcal{A}(1^n, k, f(k, \boldsymbol{x})) = \boldsymbol{x}' \ : \ f(k, \boldsymbol{x}') = f(k, \boldsymbol{x})] \leq \varepsilon(n)$$

for all $n \geq 1$*.*

In fact, the existence of a one-way function family implies the existence of a one-way function. We will not show this here, but the proof is relatively simple. Given f and Gen, we construct f' whose input consists of the randomness r used by $\mathsf{Gen}(1^n)$ together with the input $\boldsymbol{x}$ to f. f' outputs the key $k = \mathsf{Gen}(1^n; r)$ together with $f(k, \boldsymbol{x})$.

5 Short Integer Solutions

Our next candidate family of functions is simpler. The key is a suitable modulus q (e.g., taking $q = n$ is fine) and a uniformly random matrix $A \sim \mathbb{Z}_q^{n \times m}$ for some m (more on how to choose m later). The function f itself just applies the linear transformation A to its input, $f_{A,q}(\boldsymbol{x}) := A\boldsymbol{x} \bmod q$, where we interpret our input as a vector $\boldsymbol{x} \in \mathbb{Z}_q^m$ and the resulting output lies in $\mathbb{Z}_q^n$.

Of course, as described, this function is certainly not secure. In particular, inverting f is equivalent to solving a system of linear equations over $\mathbb{Z}_q$, which is easy. It can be done, e.g., using Gaussian elimination. (You might have only seen Gaussian elimination described over the real numbers, the complex numbers, or the rationals. But, it works modulu q as well.) To make this function secure, we modify it in two clever ways, due to Ajtai [Ajt96].

First, we take $m \geq 10n \log q$ to be much larger than n. As a result, every output vector $\boldsymbol{y} \in \mathbb{Z}_q^n$ has many different preimages $\boldsymbol{x} \in \mathbb{Z}_q^m$ such that $A\boldsymbol{x} = \boldsymbol{y} \bmod q$ (assuming that A has full rank, which will be the case with high probability). Second, instead of allowing arbitrary input vectors $\boldsymbol{x} \in \mathbb{Z}_q^m$, we only allow $\{0,1\}$-vectors as input, $\boldsymbol{x} \in \{0,1\}^m$. (We do this for security, but it is also rather convenient to work with bit vectors.)

Notice that, if we did one of these things but not the other, then f would not be secure. For example, if $n \geq m$ then $A\boldsymbol{x}$ will typically have a single preimage, which can be found efficiently by solving the system of linear equations. Restricting our input to bit vectors will not change that. On the other hand, if we take m to be large but allow arbitrary input, then we can still use Gaussian elimination to find a preimage $\boldsymbol{x}' \in \mathbb{Z}_q^m$ with $A\boldsymbol{x} = A\boldsymbol{x}' \bmod q$. In fact, we can find a whole affine subspace of preimages $\boldsymbol{x}'$.

But, if there are *many* preimages, it seems hard to pick out one whose coordinates happen to be bits. In fact, Ajtai showed that this problem is hard if certain well-studied and seemingly hard geometric problems called lattice problems are hard. (Lattice problems happen to be the focus of much of my own research.) This gives quite a simple one-way function family.

The problem of finding $\boldsymbol{x} \in \{0,1\}^n$ such that $A\boldsymbol{x} = \boldsymbol{y} \bmod q$ is called *Short Integer Solutions* or just SIS.

A Proof that the order of an element divides the order of the group

For each $h \in G$, let $C_h := \{h, h \cdot g, h \cdot g^2, h \cdot g^3, \ldots, h \cdot g^{k-1}\}$, where k is the order of g. Notice that $|C_h| = k$, i.e., all of the elements in this set are distinct (since if $h \cdot g^i = h \cdot g^j$ for some $j < i \leq k-1$, then clearly $g^{i-j} = e$, contradicting the fact that the order of g is k). Such sets are called *cosets.*

We claim that for any two $h, h' \in G$, either $C_h = C_{h'}$ or $C_h \cap C_{h'} = \emptyset$, i.e., any two cosets are either identical or they share no elements in common. Indeed, suppose that $r \in C_h \cap C_{h'}$. I.e., $r = h \cdot g^\ell = h' \cdot g^{\ell'}$ for some ℓ, ℓ'. Then, $h = h' \cdot g^{\ell' - \ell} \in C_{h'}$, and it follows that $h \cdot g^i = h' \cdot g^{i + \ell' - \ell} \in C_{h'}$ for any i. (Which parts of the definition of a group did we use here?)

Furthermore, notice that every element $h \in G$ must lie in at least one coset—specifically, C_h itself contains h.

So, G can be partitioned into disjoint sets C_h with $|C_h| = k$. It follows that k divides $|G|$. (The quotient $|G|/k$ is the number of distinct sets of the form C_h.)

B Proof that $\mathbb{Z}_q^*$ is cyclic for prime q

The elements of $\mathbb{Z}_q^*$ must have order dividing $|\mathbb{Z}_q^*| = \phi(q) = q-1$. For a divisor d of $q-1$, let N_d be the number of elements $g \in \mathbb{Z}_q^*$ with order d. We claim that $N_d = \phi(d)$. In particular, $N_{q-1} = \phi(q-1)$, so that there are actually many generators of $\mathbb{Z}_q^*$, and it is therefore cyclic.

To see that $N_d = \phi(d)$, first suppose that there exists one element $g \in \mathbb{Z}_q^*$ with order d. Then, g^k has order d for any k coprime to d. There are of course $\phi(d)$ such elements g^k. Furthermore, for all $1 \le k \le d$, g^k is a distinct root of the polynomial $x^d - 1 \bmod q$. Since $\mathbb{Z}_q$ is a field, this polynomial has at most d roots. So, there cannot be any other elements with order d.

The above shows that either $N_d = 0$ or $N_d = \phi(d)$. To finish the proof, we use Euler's identity:

$$\sum_{d|m} \phi(d) = m \ .$$

Applying this to $m = q-1$ and using the fact that $N_d \le \phi(d)$, we see that $\sum N_d \le q-1$ with equality if and only if $N_d = \phi(d)$ for all d. Since $\sum N_d = |\mathbb{Z}_q^*| = q-1$, we must have $N_d = \phi(d)$ for all d.

References

[Ajt96] Miklós Ajtai. Generating hard instances of lattice problems. In *STOC*, 1996. 9

[HvdH21] David Harvey and Joris van der Hoeven. Integer multiplication in time $o(n \log n)$ time. *Annals of Mathematics*, 193(2):563–617, 2021. 6

[Kal02] Adam Kalai. Generating random factored numbers, easily. In *SODA*, 2002. 8