ONE DAY AT A TIME

DESMOND COLES

ABSTRACT. I have recently decided I would like to better understand various topics in and around cryptography, including the used of zero-knowledge proofs, formal verification, and the applications in modern tech. This coincides with a foray into programming and machine learning, so from time-to-time I may write about those topics as well. And of course, my heart will always be with a certain corner of pure mathematics that consists of combinatorial algebraic geometry and geometric representation theory, if I am lucky enough to stop even the most vague connection with one of those topics I will likely mention it. In any case, these notes are mostly to hold myself accountable.

AUGUST 8TH 2025

My friend has introduced me to two different topics recently, zero-knowledege proofs and formal verification. A zero-knowledge proof, loosely speaking, is where to you verify to another party that a statement is true, and the only thing the other party learns is that the statement is true, without telling them anything else. For example, you may want to show someone that you know the passcode to your phone without telling them the passcode. Formal verification is where you have some kind of mathematically sound way to prove that some systems satisfies a specification, i.e. an algorithm is correct. These two interact as one may want to verify that certain ZK protocols are in fact correct and work as intended.

AUGUST 9TH 2025

Today I read about *signatures and zero-knowledege proofs*. A signature is a kind of *cryptographic primtitive*, a primitive is some precise reusable mathematically sound tool for doing cryptography that would be used to make a cryptographic protocol, like a hash-function or a given cipher. A signature should be some way for you to "sign" something so that only you can give the signature and the reader can verify that you actually signed it. Here's one way to do sign a message:

- Make a public and private key pair.
- Hash the message.
- Encrypt the hash with the private key, this is the signature

To verify the signature for the message you decrypt the signature using the public key and then hash the message, if they match then the signature is considered verified. There's three things going on here, key creation, signing, and verifying. One could use something like this for authenticated key exchange, for example.

How does this relate to zero-knowledge proofs (ZKPs)? One might say they want to show someone else they know something, but not reveal the thing. For example, you and someone else know that $h = g^x$ where g and h are in some group. You know x and want to prove this without telling the other person x. The information x is called the *witness*. ZKPs are proofs defined as being:

- Complete: if the statement is true you can convince an observer it is true
- Sound: if the statement is not true there is a vanishingly small chance you can convince someone it is true
- Zero-knowledge: after seeing the proof, you cannot produce a better proof using.

Remark 1. There is some notion of *simulator* here that I don't fully understand.

Signatures are an example of ZKPs!
Notes taken from [Won21, Chapter 7]. There are some other notes here

1. August 12th 2025

A little more on formal verification. AWS uses FV to prove that their code runs correctly, the idea being that it is very important that their large scale systems run correctly and that it isn't a viable task to check this manually. See `https://aws.amazon.com/blogs/security/an-unexpected-discovery-automated-reasoning-often-mak` There is tool called SAW that can be used for formal verification. In this video for example they verify pop-count runs correctly `https://www.youtube.com/watch?v=TE4UtU8bfq0`.

Idea 2. *I could just write some code for some of those other coding challenges like fizzbuzz and try to verify them.*

Another specific example of where formal verification is used is in showing that some straightforward, clearly correct code, is equivalent to some optimized yet harder to interpet code. For example in the video you want to show two computations of multiplication mod p are equivalent, or AWS wanted to change part of their software, so they proved it was equivalent to something else before swapping the two pieces.

2. August 15th 2025

A particular kind of zero-knowledge proof is a non-interactive zero-knowledge proof. One might have "interaction" in a zero-knowledge proof, this means that the verifier is somehow communicating with the prover. For example, they may ask questions of the prover such that the prover couldn't possibly answer them all correctly without possessing the relevant knowledge (such as the solution to a discreet log problem). By asking lots of questions the verifier can be sure it is unlikely the prover is lying. The issue is that real-time communication might not be feasible for whatever reason. Consider the example given in [Won21, Section 7.2.1]. I am speaking with someone and we both know that for some group elements h and g, we have that $h = g^n$. I know n but they do not, I want to prove to them I know n without revealing n. In this case I am the prover they are the verifier. Here's a way to prove that I know n.

(1) I pick a random m and let $i = g^m$, I do not tell the other person what m is.
(2) I know that $hi = g^{m+n}$, so I say look I know $m + n$ so I must know the value of n
(3) The issue then is that I could do something like take $i = h^{-1}g^n$
(4) To avoid this we can make it interactive letting the other person say "okay great, here's some value ℓ" find the discreet log when for hi^ℓ. I could not reasonably solve this without know n.

The problem here is that back and forth communication might not be very feasible for the prover and verifier. So we may want a **non-interactive zero-knowledge proof**.

An actual protocol called zk-SNARKs was built for non-interactive zero-knowledge proofs. A distinctive feature is that it can be verified in milliseconds. This is used in Zcash to verify transactions in the blockchain as legitimate without revealing information about the transactions. This repository contains a rust library that implements some of the algebra behind zkSNARKs: `https://github.com/arkworks-rs/algebra`.

3. August 19th 2025

The topic of today is computation complexity, hardness, factoring primes, and quantum computation. I am trying to understand Schor's algorithm [Sho97]. We will start this discussion by roughly getting at what it means to compute something. A function is *computable* if there is an algorithm that will take any value in the range and compute the value of the function. The Church-Turing thesis says that any computable function can be computed by a Turing machine, though thesis cannot be made mathematically rigorous (the problem seems to be hiding in the definition of algorithm). A turing machine is a model of a computer that runs with a finite set of instructions and possible, and which has unlimited memory, the machine works by writing onto tape. As put by Schor, this distinction is much too coarse for real-world applications. Not only do we want to compute things we want to do it fast [Sho97]. An algorithm is said to run in polynomial time if it's runtime is bounded above by a polynomial in the size of the input. The class of problems that can be solved with a polynomial time algorthim is called P. There is a larger class of problems denoted NP which is given by problems that can be solved in polynomial time by a non-deterministic Turing machine, or in other words decisions problems whose solutions can be check in polynomial time. For example, "given n and k, is there a factor of n which is smaller than k?", solutions can be checked in polynomial time but you can't

necessarily find solutions easily. There are other kind of complexity that arise in computing, in particular space complexity, how much memory an algorithm uses.

This all matters for cryptography because in general we want to use 'hard problems' in cryptography. In particular we would like things like the discrete log problem (or more specifically prime factorization) which are easy to check solutions to but are hard to solve. In particular you don't want to use problems that are P, because if the problem can be solved in polynomial time the a large enough computer could in principal solve the problem quickly and thus break your cryptographic protocol. But if we take a problem like prime factorization then a computer cannot do it in a reasonable amount of time if we take large enough inputs. This is what RSA, a widely used public-key protocol, is based on [RSA78].

Quantum computers actually have another type of computational constraint which is accuracy, the probabilistic nature of quantum mechanics means that you can't compute things precisely. If one allows for some inaccuracy to grow, then one can compute things with less time or memory. Here in lies the problem for cryptography: quantum computing gives a polynomial time solution to the discrete log problem. My rough understanding of what Shor's algorithm does is the following. First, we reduce the question of factoring primes into one about computing the order r of an element x of $\mathbb{Z}/n\mathbb{Z}$. Then to solve this problem we define a unitary operator who eigenphases correspond to the order of x. This unitary contains all the different values of x^a simultaneously. You then use quantum phase estimation on this unitary and some other processing techniques to extract the order of x with some probability. A computation using probability estimates from quantum mechanics and some estimates on the number of prime factors of an integer from number theory that you can repeat this process $O(\log\log r)$ times to find r. It's a neat paper, it makes use of some good classical number theory like Euler's totient function and repeated fractions, and also uses the Quantum mechanical theory to do a slick computation. The introduction is good.

4. August 20th 2025

I am reading part I of [CLRS22] to brush up a bit more on the idea of algorithms and computational complexity. The basic idea is that algorithms, such as instructions you might in software. These instructions might be things like arithmetic, logical operations, bit manipulations, and numerical comparisons. Something like a GPU might be specialized to run particular computations. Roughly speaking, a program loads as instructions onto the RAM and then the CPU executes the instructions. Complexity of algorithms is then measured by how many steps are performed. Space complexity is how much RAM is consumed as the program runs, for example merge sort needs to break the original array into smaller arrays, so it requires memory while it runs. Selection sort does not require memory.

Big O notation gives an upper bound on a function. I.e. we say that $f(n)$ is $O(g(n))$ if there is some constant c such that we always have that $f(n) < cg(n)$. The book [CLRS22] uses Ω notation to denote a lower bound, and we say that f is $\Theta(g(n))$ is it is both $\Omega(g(n))$ and $O(g(n))$. Insertion-sort says roughly, take a list, start at the second position, if the number is smaller than the first number swapm their places, then go to the third spot, if that number is larger than the second number move it back, and if it's larger than the first number move it back again, and so on. This is a nested loop, you could say it's a for loop and a while loop, looping k over 2 to n and for each k looping over 1 to $k-1$. Because of this runtime has an upper bound of n^2 for a list of n elements. If you take the worst possible list fo this, a decreasing list, then the runtime is the triangular number $\frac{n^2-2}{2}$. So in fact we have that the runtime is $\Theta(n^2)$.

Tying this back to RSA, the reason this protocol works is that any known algorithm to factor is very slow when we increase the number of digits in the prime number. The most naive algorithm to factor n is to check division by each number less than $\sqrt{n}$. The runtime here is $O(2^{L/2})$ where L is the number of digits in n. If n has say 256 digits this would take unfathomably long time even on a very fast computer. One of the best known algorithms for large numbers the general purpose number field sieve still runs in time $O(e^{(64/9L)^{1/3}(\log L)^{2/3}})$, which is still slow. Shor's algorithm is $O(L^3)$ (even faster in fact)! A huge improvement.

5. August 21st 2025

I am reading [RSA78], it's a bit wild to think that only 50 years ago we didn't have the means to send email. Here's the basic idea of public key cryptography:

(1) Every user has some encryption protocol and some secret decryption method.
(2) The encryption method is made public. Anyone can encrypt a message.
(3) Only the original user can decrypt a message that was encrypted using their encryption process.
(4) It should totally impractical to break the encryption through brute force.

This paper gives an explicit recipe for this using prime factorization. The authors include an important criterion which is that if take a message, apply the decryption function to it, and then encrypt the decrypted message, I get the original message. This is essential for secure signing. If I want to sign a message I first *decrypt* it, then I encrypt it with the *other* persons public key, and then they can decrypt it with their private ket and re-encrypt with *my* public key, which would result in a readable message. This counts as signing because it could only work out that you get a readable message if someone with the private key sent it.

Remark 3. A naive way to understand this is that 'symmetric encryption', like the disk encryption on my computer requires one key to lock and unlock it. You would use something like AES or SHA for this. Public key has *two* keys, one for locking and one for unlocking. One fun fact is that there is a symmetric cryptography technique called a "one-time" pad that is actually impossible to break. It is unreasonable to use in practice though.

Here is how you do RSA encryption:
(1) Secretly pick two very large primes p and q, and a large number d that is coprime with $p-1$ and $q-1$.
(2) Let $n = pq$ and e be the multiplicative inverse of d modulo n.
(3) The public recipe for encryption is a pair (e, n).
(4) Represent your message M as an integer between 0 and $n-1$.
(5) The encrypted message C is $M^e \mod n$.
(6) To decrypt a received message simply compute C^d.

We can break large messages into blocks. This is secure because of the hardness assumption about factoring a number into primes: due to the fact that our best factoring algorithms are superpolynomial we can assume that n hides p and q.

The encryption time for this is polynomial (L^3) in the length of digits in the message M, overall this is a very fast way to encrypt then.

The paper closes with some arguments as to why this hardness assumption is sufficient, which have held up until now! Schor's algorithm is still not able to be used at scale due to hardware limitations with quantum computing, though small numbers have been factored.

This paper comes after [**?**] where they propose this kind of abstract machinery. In [**?**] they also use a discrete-log problem to deal with key exchange. Let's say I want to develop a shared password with someone, K.

(1) Let p be a big prime, and let's publicly agree on $g \in \mathbb{Z}/p\mathbb{Z}$.
(2) I send you $g^x \mod p$ and you send me $g^y \mod p$. Then we can both compute $g^{xy} \mod p$. An observer cannot do this because they can't find x or y.
(3) We agree that our key is $K = g^{xy}$. We could then use that for some symmetrically encrypted thing that we transfer to each other.

This was big but doesn't protect against a man in middle attack, i.e. I could intercept both g^x and g^y and send both people g^z!

Unrelated: here's a paper using tropical algebra for cryptography! [GS13].

6. August 26th 2025

Okay today I got github desktop, VS Code, a tex editor, rust, and python set up on my computer. I pushed these notes to github (yikes, anyone can read them now).

I made the classic "Hello, world!" project. This is pretty much the most simple project you could do. Let's unpack this cargo "ba dum tss". There's the cargo.toml file, this contains dependencies i.e. outside packages, and metadata like name and version number. This project has no dependencies. You've got the cargo.lock file, this file keeps track of the exact builds that were used to run the program, this way we get

reproducible builds. Then you've got the source and target, The source is where I write my code and then target is where the output goes. So in my case I have a main file in source and that gives the command to print 'hello, world'. In this case the output appears to just be some files that are used for debugging.

References

[Bar16]   Boaz Barak, *Cs 127 / csci e-127: Cryptography (notes)*, https://www.boazbarak.org/cs127spring16/, 2016, Accessed: 2025.

[CLRS22] Thomas H. Cormen, Charles Eric Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms / thomas h. cormen, charles e. leiserson, ronald l. rivest, clifford stein.*, fourth edition. ed., The MIT Press, Cambridge, Massachusetts, 2022 (eng).

[GS13]   Dima Grigoriev and Vladimir Shpilrain, *Tropical cryptography*, CoRR **abs/1301.1195** (2013).

[RSA78]   R. L. Rivest, A. Shamir, and L. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Commun. ACM **21** (1978), no. 2, 120–126.

[Sho97]   Peter W. Shor, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer*, SIAM Journal on Computing **26** (1997), no. 5, 1484–1509.

[Won21]   David Wong, *Real-world cryptography*, 1st ed., Manny, Shelter Island, NY, 2021.