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-knowledege 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 popcount 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 [DH76] where they propose this kind of abstract machinery. In [DH76] 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.

7. September 2nd 2025

Today I'm gonna solve fizzbuzz in Rust. Remembering some *really* basic things. I cd into the folder I would like to keep my project in. I go to my project and I open the main.rs file in my editor, I'm at home so I'm doing this quick and dirty and just writing straight into a text file, at some point I will get my act together and properly setup rust-analyzer with emacs.

8. September 3rd 2025

I read through [DH22]. For context this paper is written when good and fast cryptography is becoming a commercial need as a result of the digital age. In this paper Diffie and Hellman argue that the proposed standard from the National Bureau of Standards is insufficient. In particular they argue that the recommended length for encryption key is too short and is vulnerable to a known plain-text brute force attack. As in, an attacker that has some plaintext and the encrypted version of that text could reasonably find the key via brute force. The argument really is one about hardware limitations (or lack thereof).

9. September 4th 2025

Wrote a little bit of Rust code that checks if a string is a palindrome. A couple important things I learned. First, if you want to import a module, which I did, you write "use". The "use" command defines your 'scope': there are modules, or collections of functions or commadnds and to use them you would need to specifcy the module, using "use" for a module or functionality means you don't need to specify that module every time. In my case I wanted to use "std::io" which gives tools for bringing in inputs and giving outputs. Second, you want to use "let" to define a variable. Rust demands that you define all your variables and their type explicitly. This allows rust to compile the code before running it, where a language like python compiles and runs code line by line.

10. September 7th 2025

Today I started reading this expository articel on zk-SNARKS [Pet19]. This paper is about **Zero-knowledge succinct non-interactive arguments of knowledge**. This is the notion of performing a proof that you performed a computation, in a manner that is zero-knowledge (you don't reveal information about the computation), non-interactive (you don't require back and forth communication between the prover and verifier), and the proof is succinct (the proof is quick even if the computation is large). More generally one might be interested in something called a zkVM, which is a "zero-knowledge virtual machine". a zkVM is a system that can execute zero-knowledge proofs that arbitrary programs are run. This kind of technology is really useful for things like the blockchain. The non-interactive proof component allows you to verify that a computation (that represents a transaction or contract) took place. Succinctness means that you can perform proofs of computations in a manner that is computationally cheaper and thus scalable. The zero-knowledge component is of course important for privacy. The idea of zk-SNARKS was originally introduced in [BCCT12]. I hope to revisit this with a more-detailed explanation, but here is a vague sketch of how the protocol works. First you have something that you want to prove you computed, represent this as a polynomial. You then make a proving key and verification key, which generally are tuples of points in a finite group or elliptic curve. You then encrypt the steps of the computation as elements of your group, and compute group elements that will satisfy some relations with the public verification key, these elements are the proof. The verifier then computes some relations which can only be satisfied if you computed ran the computation correctly. The zero-knowledge component is easy to program in, the succinctness is impressive though: even if the computation is very complex you can use a small number of relations to prove that you did it.

As a note the idea that encryption of the form $x \mapsto g^x$ is *homomorphic* is very important, we can add encrypted values by multiplying in the group. This observation is essential for this all to work

11. September 9th 2025

I am going to try to play around with formal verification techniques in software. Kevin Buzzard is running a fairly large project to use Lean to formalize number theory results. I am going to focus on some applications to industry applications and things that may overlap with cryptography. For this reason I might try my hand at working with the Galois' tool SAW which can formally verify Rust programs. They have a tutorial here. I may also this coursera course. I may also go through Kevin Buzzard's Lean website as the Ethereum Foundation is currently running a project to use formal verification for zkVMs. It looks like if I want to contribute the most pressing thing would be to just learn Lean.

So here is my rough understanding of how you check things about software in Lean. You need to define what you are interested in as mathematical objects and you need to specify their properties in Lean. You then phrase the relevant property of the program as a theorem, and then you write a proof of this theorem. You then write a proof and Lean checks this. One goofy exercise I could do is to formally verify my leetcode solutions, they are kind of well-adjusted for this. I could also go back to my roots and contribute to this. This is different from SAW which takes actual Rust code and checks that said Rust codes satisfies though specifications.

I do like this little game for Lean.

12. September 16th 2025

I am going to use the Open Quantum Safe library to make a little toy application to try out their ope-source library for post-quantum secure cryptography.

13. October 3rd 2025

More or less got the script for the post-quantum secure encryption running! I I can symmetrically encrypt a file and then encrypt that file the Kyber512, encryption protocol, which relies on the problem of learning with errors over a module lattice. The biggest difficulty was getting the liboqs library to work properly on my computer, the next challenge will be to make this script run with a nicer user interface, I would like something similar to a PGP application, if possible.

I have also started learning Lean, in particular I have proved some basic facts about matrices. A fun side-effect of this is that I thought harder than usual about some linear algebra problems. For example over a general commutative ring, having non-zero determinant implies full rank, but it does not imply invertibility. Conversely a matrix can be full rank but have zero determinant and not be invertible. Determinants still nicely correspond to invertibility over commutative rings, a matrix is a unit iff the determinant is.

14. October 9th 2025

So lean is based on dependent type theory, which is a system for doing symbolic logic, where everything can be represented as a type. For example, the statement $P \to Q$ has the type of a function whose domain is proofs of P and whose range is proofs of Q. The idea then is that proofs for a proposition P are things of type P, and there are rules for how types can be formally manipulated, which is checked by the lean kernel, when we prove something we actually are constructing a proof object and lean is checking that we followed the rules. The theoretical underpinning of all this seems to be something called the Curry-Howard correspondence. There is a subtle idea about programs being the same thing as proofs.

For example, consider this definition in lean:

```
def verified_sort (l : List bbN) :
  Sigma r : List bbN, sorted r and r ~ l :=
⟨merge_sort (· leq ·) l, ⟨merge_sort_sorted (· leq ·) l, merge_sort_perm (· leq ·) l⟩⟩
```

What this gives is something called "verified sort" that is of type "a sorted list, with a proof that it is sorted, and a permutation that gives the sorting". This object is defined as a merge sorted list, the proof that merge sort works, and the resulting rearrangement from merge-sort. You could then evaluate this on a list to get the sorted list. We know the list will always be sorted because we know it is of that type, and if lean compiled are proof then it is correct. Wild stuff, simultaneously feels obvious that you can write formal logic for a computer to evaluate, but also feels shocking somehow. I learned this stuff from this lean manual

15. October 13th 2025

Went to a talk today from Elizaveta Pertseva on using SMT solvers to prove theorems in lean. SMT solvers are programs that can check if a given formula (typically without quantifiers) can be satisfied. In this case the idea is that an SMT solver could be used to prove something in lean, in particular it could be used to construct a proof that some kind of software satisfies a certain property. One major goal for this project was to streamline the process of translating types in lean for the SMT solver, and creating an interface that does this but allows user input. The end output of this project was verification of the front of a zkVM called jolt. This work was done at Galois.

16. October 15th 2025

I have been working on contributing to the zkEVM Formal Verification Project. Generally this project aims to build more tools for formal verification and to integrate FV into maintenance of software projects. In general this goal wants to verify certain things about zkEVMs. I have been contributing to Arklib which is part of the cryptography track. This track aims to verify the security and implementation of cryptographic primitives used in zkEVMs. This project is primarily interested in RISC-V zkVMs. One difficulty with formal verification in lean is that you need to translate things from another language into lean, which could be time consuming. A tool called hax is being developed to do this just this, to translate rust code into lean so it can be verified.

Project idea: it would be cool to try to verify some code with hax.

To reiterate, a virtual machine is software that effectively simulates a computer inside of your computer. A zkVM is something that runs a programs like a VM but it produces a zk-proof that it ran the specified program correctly and it also produces the output, this allows outside observers to verify things without revealing sensitive information, and it's computationally efficient when implemented correctly.

To do: I need to understand RISC-V's contribution, what zk-circuits are, and why Rust is so central to this ecosystem.

17. October 16th 2025

I am reading a bit about lean to get a better understanding of how it works.

When you state a theorem or use the "have" command in a proof you introduce a goal. The goal will be constructing a proof term of the relevant type. A proof term is just an expression in lean. Tactics tell lean how to construct a proof term from other proof terms. When you prove a theorem P you demonstrate that a proof of type P exists. You then assemble proof terms together into other proof terms into a new proof term that has a different type. For example "intro" helps you assemble a proof term of type P implies Q. A "sorry" says "this proof type may be empty, but pretend like it is not and continue to assemble other proof terms that rely on proof terms of this type". The 'apply' tactic takes some other expression and unifies it with the current goal for example:

```
lemma conjcomm (P Q : Prop) : P and Q rightarrow Q and P := lambda h, <h.2, h.1>

example (P Q : Prop) (h : P and Q) : Q and P :=
begin
  apply conjcomm,
  exact h,
end
```

The 'rewrite' tactic changes specific terms in the goal by using theorems. I.e. things in an iff or on the other end of an equality will be swapped out. More generally, simp, will look throughout the goal and rewrite it using things in mathlib tagged as simp. Though, sometimes you have to tell simp where to look.

18. October 17th 2025

I have been reading a bit about zkML. The basic idea is that you want to do ML, and you want to prove to someone that you ran model A and got output Y but not reveal A or X. This allows someone to protect the model they built, and allows someone using the model to verify that a company isn't claiming to use

model when running another (which they may do to reduce computational expenses). The workflow would basically be, commit to some weights, possibly via a hash, and then when you return an output of the model A you can run a zk-proof that you did in fact use A.

On the other hand, if you are really concerned about private data, you could perform fully homomorphic encryption on this data and then train a model on that encrypted data. This is difficult though because computing on encrypted data isn't necessarily easy, only algebraic operations can be used for activation functions, for example.

19. October 18th 2025

I was reading about zkML a bit more, and the main use case that I can think of is you have a situation where you have someone with a model, and you want to ensure people using the model that the model is not changing. So this involves building the model, committing to the model, running the model, and then providing proof that you ran the model you committed to and got output Y. There is a package for example, ezkl for doing this, there is also something called zkpytorch, or I could try to do ML in rust and then use RISCV but that may be hard as I may have to do a lot of manual input.

If one is interested in doing ML where data is protected, it's probably better to do some kind of ML on encrypted data. Encrypting some data homomorphically and then doing some EDA on it could be interesting, how, for example should PCA work? Could one use an ML model built from polynomial functions very easily?

Note to self: I would like to read more about, why rust is so important for crypto, why lean is becoming popular in formal verification, what formal verification for hardware looks like, and what a more routine approach to debugging software is like. I would also need to read up on the math of FHE more and research on FHE + ML more if I was to spitball about projects.

20. October 19th 2025

Mini-project idea: try to use hax, or SAW to to convert my leetcode problems into lean or some other language for FV and then check my solutions that way. The leetcode problems are logically simple, have clear constraints, are usually short to state and solve, and they don't require outside packages, so this might not be too hard.

21. October 21st 2025

I have been spending quite a lot of time trying to prove inequalities in lean; calculations over the reals are difficult for me, need to pin down some ways to do this better.

22. October 22nd 2025

I have been reading a bit about the basics of bug testing in software. One on the hand there are relatively obvious paths to checking for bugs: including investigating user reports of bugs and simply reading error logs. One can also do "unit testing" i.e. testing code in very small chunks. One can also release beta versions or create a 'staging environment' to test code. There are also ways to catch bugs before they crop up. Continuous integration for example is where developers push to code to a min database often, multiple times a day, and automated tests are run or other formatting features like linters are used to make sure the code doesn't cause issues. Good documentation during this process makes it easier to isolate bugs and good testing prevents many bugs. There is also 'test-driven development' where you write tests for your code before you write the code itself, and then check that your code passes the test. One can also do static analysis, so analyzing the code without running it, or property-based testing, where a wide-range of inputs are tested. Formal verification fits into this picture by guaranteeing certain bugs don't happen, the drawback is that FV can be time consuming and really hard, so it's only worth doing if you have something really critical.

23. October 23rd 2025

I am having trouble using hax I may try Aeneas instead.

24. October 24th 2025

I got hax to translate the kids with candies problem successfully. This requires using arrays instead of vectors, avoiding any methods, and using for loops and not while loops. The hax documentation was helpful, the issue seems to be that some features are just not implemented in rust, features that more have more support are pretty clear once you look around.

25. October 25th 2025

Played around more with hax today. I translated a post-condition. It's pretty time-consuming to do some of this stuff even in trivial cases. Proving that algorithms work also doesn't feel like it's verifying much because my programs are so simple. It might be more interesting to check if two algorithms really always run the same. This would be an easy post-condition t state.

26. October 26th 2025

Why Rust for cryptography? According to this CISA post seventy percent of vulnerabilities are related to lack of memory safety. This includes things like using uninitialized memory or not respecting type-boundaries. The borrow checker in Rust keeps track of this and preemptively handles many of these issues. Though Python, Go, and Java have similar features to prevent things like this. C and C++ do not have these kinds of features and whoever writes the code has to manually manage the memory. Rust is different in this regard though because it does not use a garbage collector, which is part of the runtime system, which is code that the language uses to run. The issue is that this can slow down programs. Rust instead deals with memory at compile time: there are ownership rules that the compiler enforces at compile time, this is where the concepts of ownership, borrowing, and lifetime come in. Because of this Rust can run faster. Rust also allows for control at a level closer to C or C++. The result then is that for performance sensitive and security sensitive code Rust ends up being a good choice. The other thing is that many of the protocols people want to code are fairly new, so this opens the possibility to write in a new language, as developers are not relying on a large established codebase for this stuff.

27. October 29th 2025

In this AWS article about how Cedar was constructed (Cedar is a langauge for designing permissions for applications, developed by AWS) the authors discuss how they do 'verification-guided development' with lean. The idea is that they construct a model of important components of the code in Lean, and then they use 'differential random testing' to show that the models match the actual code. Differential random testing is where you generate many inputs randomly and run them through both models to see that they behave the same. In a sense, leetcode basically does this to check solutions. In principle a highly reliable translation tool (like hax) should be able to remove the need for this, but on some level your formal verification is only as good as your formal verification. Here is some lean code where they model the 'authorizer' from Cedar:

```
def isAuthorized
  (req : Request)
  (entities : Entities)
  (policies : Policies) : Response
:=
   let forbids := satisfiedPolicies .forbid policies req entities
   let permits := satisfiedPolicies .permit policies req entities
   if forbids.isEmpty && !permits.isEmpty
   then { decision := .allow, policies := permits }
   else { decision := .deny,  policies := forbids }

theorem forbid_trumps_permit
   (request : Request) (entities : Entities) (policies : Policies) :
   (thereexists (policy : Policy),
     policy in policies = and
     policy.effect = forbid and
```

```
    satisfied policy request entities) \rightarrow
  (isAuthorized request entities policies).decision = deny
:= by
  intro h
  unfold isAuthorized
  simp [if_satisfied_then_satisfiedPolicies_non_empty
       forbid policies request entities h]
```

There longest proof is around 4,500 lines and took 18 days to write!

Part of what AWS likes about lean is the versatility and the speed at which it verifies things. Some other reasons people like lean are that it relies on a fairly small and well-understood kernel, so it is highly trusted. It also has a large ecosystem around it, and there is a clear directive by the lean community to make it more popular. The fact that lean can both be used for formal verification and as a programming language is also cited as a benefit. There are still people that prefer F*, Coq, or Isabelle. For example https://blog.mozilla.org/security/2020/07/06/performance-improvements-via-formally-verified-cryptography-in-firefox/. In general though FV is generally seen as difficult to do in practice and the enthusiastic community around lean may be the reason for its success.

28. October 30th 2025

Now that I have managed to deal with the way rust returns an option, I think that formally verifying more code should be easy, as all proofs should boil down to proving that two pieces of code panic at the same time and that they return the same result when panicking.

References

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

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer, *From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again*, Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (New York, NY, USA), ITCS '12, Association for Computing Machinery, 2012, p. 326–349.

[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).

[DH76] W. Diffie and M. Hellman, *New directions in cryptography*, IEEE Transactions on Information Theory **22** (1976), no. 6, 644–654.

[DH22] Whitfield Diffie and Martin E. Hellman, *Exhaustive cryptanalysis of the nbs data encryption standard*, 1 ed., p. 391–414, Association for Computing Machinery, New York, NY, USA, 2022.

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

[Pet19] Maksym Petkus, *Why and how zk-snark works*, 2019.

[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.