Cryptographic Engineering

working draft

Matthew D. Green
Johns Hopkins University

Excerpt: Chapter 16, Privacy enhancing technologies

§16.2 Zero-knowledge: proof systems; the GMW protocol (book pages 279–290)

Posted for 601.445/601.645 Practical Cryptographic Systems, Fall 2026
Lecture 22, Wed Nov 18: zero-knowledge proofs

`https://practicalcrypto.org`

This is an excerpt from an unfinished book draft: expect gaps, rough edges, and errors, and note that section numbers and page numbers may change in later drafts. Cross-references to other chapters point to material that is not included here. Copyright © 2026 Matthew D. Green. All rights reserved. Posted for the students of the course above; **please do not redistribute** — link to the course site instead. Built 2026-11-15.

Chapter 16

Privacy Enhancing Technologies

16.1 Private Set Intersection

16.2 Zero-knowledge

DRAFT

Zero-knowledge proofs are one of the few areas where truly advanced research cryptography has escaped containment and found purchase in the real world. These mathematical systems allow one machine (called a Prover) to convince a second party (called a Verifier) that a particular mathematical statement is true. While this explanation, by itself, seems kind of dry, it's a bit easier to understand if we replace "mathematical statement" with "computer program", and instead phrase the goal as follows: *zero-knowledge proofs allow me to convince you that I know the input to a specific computer program that causes it to produce a particular output, without revealing that input.* These proofs have enormous applications in areas as diverse as cryptocurrency, anonymous authentication, and private computation.

16.2.1 What's a proof system?

A short history lesson. In the early 1980s, MIT researchers Shafi Goldwasser and Silvio Micali were studying the problem of building *interactive proof systems* and interactive *argument systems*. Here a Prover and a Verifier, send messages back and forth until the Prover has convinced the Verifier of the truth of some statement. (In an *argument* system, the Verifier doesn't have to be perfectly convinced, just convinced that the Prover is telling the truth with some high probability.) These proofs generally involved dull mathematical or logical statements about numbers or graphs: for example, "this number is composite", or "these two graphs are isomorphic[1]." For this proof to be meaningful, the Prover typically had some special knowledge, called a *witness*, that allowed it to actually prove the truth of a statement. (For example, it might be powerful enough to compute the factors of a large

[1] Two graphs are isomorphic if they're structurally identical.

composite number, or a mapping between two graphs.) Hence the question was whether the Verifier could detect a lying Prover, often in cases where the Prover was much more powerful.

Most of the early research in this area on the *soundness* of the proof system: namely, the probability that the Verifier could reject a proof involving a false statement. Goldwasser and Micali decided to ask a different question: was it possible for the Prover to convince a Verifier of the truth of a statement *without revealing any information about its witness?* While this started as a theoretical question, it quickly gained some real practical applications. For example, imagine that I want to convince you that I've generated an RSA modulus N correctly: this is very easy to do if I hand you the "witness" consisting of N's secret factorization p, q. But (as we discussed in §7.2.1) doing this means that you now have my secret key! It would be much better if I could prove to you that I've generated a valid RSA key *without* handing over my secrets. In principle, Goldwasser and Micali's proofs allow me to do to this.

What can we prove in zero-knowledge? In 1984, Oded Goldreich, Silvio Micali and Avi Wigderson (henceforth GMW) published a very famous paper: it showed how to build zero-knowledge proofs for every problem in the complexity class NP [**?**]. Since NP contains every single efficient (polynomial-time) program with a true/false output, this result instantly meant that a Prover and Verifier could agree on a program, and the Prover could prove they know a valid input that makes it output TRUE. Critically, the Prover would not have to reveal its input. While the specific protocols that GMW invented are not terribly practical (and we have many more efficient options today), this paper proved that zero-knowledge proofs are possible for most programs we care about. We'll talk about GMW a bit further below, because it's a good way to understand how these protocols work.

DRAFT

Where are zero-knowledge proofs used? Many places! Zero-knowledge is one of the true breakout successes of the theoretical cryptography research community. In research land, these proofs are incredibly useful "on paper" for making weak protocols into strongly secure ones. But out in the real world they're also used for everything from constructing signatures (like the Schnorr signature) to building anonymous credentials, like the PrivacyPass protocol (§16.3.1) or Google's new age verification systems [**?**]. Zero-knowledge has become extremely popular in the cryptocurrency area, partly because they allow for private transactions on blockchains, and partly because a variant of these techniques (called *succinct* proofs) can be used to help make blockchains much more scalable. We'll talk about several of these examples further below.

Proofs of knowledge and non-interactivity. While we're speaking at a high level, it's worth pointing out that zero-knowledge proofs come in various flavors. The proofs we've been discussing all refer to some mathematical statement, such as "this number is composite" or "there *exists* an input to this program that makes it output TRUE." A variant of this idea is something called a "proof of knowledge". Here the prover shows that they know a

specific witness that satisfies the statement. While this might seem like a subtle difference (after all, doesn't the Prover need a witness either way?) it has some major implications. For example, consider the statement "there exists a message M such that this encrypted message C decrypts to M." For many encryption schemes, this statement is always trivially true — that is, for many schemes we can prove that there always *exists* such some message, but not necessarily that we know it. A *proof of knowledge* tells me something more specific: it says "I actually know the specific message M that this ciphertext decrypts to." For many purposes, that's often a more useful statement to prove.

A second area where proofs differ is on their degree of *interactivity*. The early proofs that GMW considered were interactive proofs: a Prover and Verifier needed to send multiple messages back and forth to complete the proof. *Non-interactive* proofs remove this requirement. Here the Prover can formulate their proof into a single message (either large or small) and send it to the Verifier to be checked. Even better, since non-interactive proofs are don't require interacting with a specific Verifier, this means that *anyone* can verify that a given proof is valid. This makes non-interactive proofs very useful for applications like blockchains, where thousands or millions of different computers will need to verify a proof made by one person.

Succinctness and SNARKs. The final area where proof systems get interesting is related to the size of a zero-knowledge proof (or *argument*.) The early proof systems like GMW required the Prover to send a very large amount of data. Put in a more general way, the size of these earlier proofs was at least as large as (the larger of) the witness and/or the size of the circuit/program being proven (in fact, they were usually many times larger than either number!) This meant that as a general rule, only relatively short witnesses and simple programs were feasible to prove things about. This changed in the late 1990s and 2000s, as a series of researchers developed a new class of proof systems called *Succinct Non-interactive ARguments of Knowledge*, or SNARKs.[2]

The main benefit of a SNARK is that it produces a short proof, often much smaller than the witness or the size of the program being proven. As a second benefit, this proof can also be verified quickly, sometimes more quickly than the time required to re-run the original computation. This makes it possible to prove the correct execution of very complex functions, even complicated computer programs. To give a real-world example, imagine that I want to outsource some large computation to a cloud provider, who sends me back the result; ordinarily I would have to trust that the provider did the calculation correctly. But a provider who returns a SNARK with their computation can *prove* to me that the result is correct, without forcing me to re-do all that work. Even better, the "witness" to the computation can contain a huge amount of data that the provider might keep on their servers. Instead of sending me all that data, they could send me a *hash* of the data, along

[2] As the name implies, SNARKs are technically *argument* systems, which means they don't provide 100% soundness. However, they can be made sound with error low enough that they work effectively as well as proofs. We're going to call them proofs because life's too short for pendantry.

DRAFT

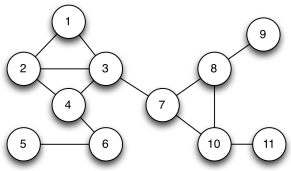
with a proof that the data they used to run the program matches the hash. While Amazon doesn't do this, the idea has become very useful in the world of blockchains, where many different computers share a "ledger" of millions of transactions, and all computers must be convinced that all of the transactions are valid. Here the entire ledger can be replaced with a short hash, and a proof that the hash represents a valid list of transactions.

16.2.2 An introduction to zero-knowledge: the GMW protocol

Many discussions of zero-knowledge protocols start by trying to give "physical world" analogies and metaphors.[3] While I love that people have tried to come up with simpler explanations of how these protocols work, I don't think the simplifications really help. If you want to actually understand a real zero-knowledge protocol, I think you actually need to see one in action. For this reason I usually teach my students how the GMW protocol itself works. To do this, I'm going to use some ideas that I've borrowed from Silvio Micali himself.[4]

The setup. The GMW protocol is designed to prove statements about graphs (of the computer science form.) It's specifically a protocol for proving the following statement: *I know a way to color a specific graph, such that I only need to use three colors, and no adjacent nodes use the same color.*

If that doesn't immediately make sense to you, let's use a more colorful analogy. For the purposes of this example, I'd like you to imagine that I'm a telecom magnate in the process of deploying a new cellular communications network. My network's layout is represented by the graph below. Each vertex in this graph represents a cellular radio tower, and the connecting lines (edges) indicate locations where two cells' broadcast areas overlap, meaning that their transmissions are likely to interfere with each other. Here's a very tiny example:



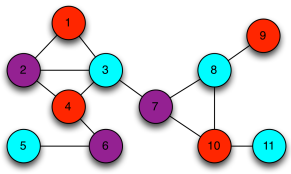
This overlap is problematic, since it means that signals from adjacent towers are likely to scramble reception. Fortunately my network design allows me to configure each tower to one of three different frequency bands to avoid such interference. That means that the real challenge in deploying my network is to assign those three frequency choices to the towers such that no two overlapping (meaning, connected by lines) cells share the same frequencies.

[3] Two of the most popular are the "where's Waldo" analogy, and the Aladdin's cave explanation [271]. These mostly leave me cold.

[4] In addition to plagiarizing Micali's ideas, I'm also self-plagiarizing some parts of this from a blog post I wrote a few years ago [145].

DRAFT

If we use colors to represent the frequency bands, we can quickly work out one solution to the problem:



Of course, what I'm showing you here is an example of a famous graph-theory problem called the *three-coloring problem.* If you've taken a CS theory course, then you might also know what makes this problem so interesting to computer scientists: for some graphs, it can be quite hard to find a solution, or even to *determine whether any solution is possible.* In fact, graph three-coloring — specifically, the decision problem of whether a given graph can be colored with only three colors — is known to be in the complexity class NP-complete. The beautiful thing about this problem is that *any other* problem in NP can be converted into a graph, and the existence of a three-coloring for that graph implies that there is a solution to the original problem![5]

Proving the existence of a solution. Now, it goes without saying that the toy example above is easy to solve by hand. But what if it wasn't? For example, imagine that my cellular network was very large and complex, so much so that the computing power at my disposal was not sufficient to find a solution. In this instance, it would be desirable to outsource the problem to someone else who has plenty of computing power. For example, I might hire my friends at Google to solve it for me on spec.

But this leads to a problem. Imagine that Google devotes a large percentage of their computing infrastructure to searching for a valid coloring for my graph. I'm certainly not going to pay them until I know that they really have such a coloring. At the same time, Google isn't going to give me a copy of their solution until I've paid up. We'd wind up at an impasse.[6]

A first zero-knowledge protocol (with paper and hats!) The engineers at Google consult with Silvio Micali at MIT. Micali consults with his colleagues Oded Goldreich and Avi Wigderson, and they come up with the following clever protocol, one so elegant that it doesn't even require any computers. All it requires is a large warehouse, lots of crayons, and plenty of paper. Oh yes, and a whole bunch of hats.

[5] If this surprises you, see this blog post [**?**] for a really good, intuitive explanation.

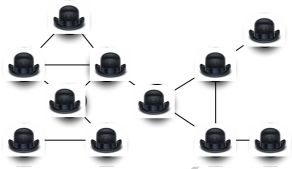
[6] In real life there's probably a common-sense answer to this dilemma, one that involves lawyers and escrow accounts. But this is not a blog about real life, it's a blog about cryptography. And if you've ever read a crypto paper, you'll understand that the right way to solve this problem is to dream up an absolutely crazy technical solution.

DRAFT

I'm the Verifier, and Google is the Prover. Here's how our protocol will work:

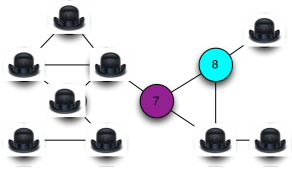
1. First I will enter the warehouse, cover the floor with paper, and draw a blank representation of my cell network graph. Then I'll exit the warehouse.
2. The Prover (here Google) can now enter enter, shuffle a collection of three crayons to pick a random assignment of the three agreed-upon crayon colors (red/blue/purple, as in the example above), and color in the graph in with their solution. Note that it doesn't matter which specific crayons they use, only that the coloring is valid.
3. Before leaving the warehouse, the Prover covers up each of the vertices with a hat.

When I come back in, this is what I'll see:



Obviously this approach protects the Prover's secret coloring perfectly. But it doesn't help me at all. For all I know, the Prover might have filled in the graph with a random, invalid solution. They might not even have colored the graph at all!

Challenging the Prover. To address my valid concerns, Google now gives me (as the Verifier) an opportunity to 'challenge' their solution to the graph coloring. I'm allowed to pick a single *edge* of this graph at random (meaing, one line between two adjacent hats). To convince me that Google isn't lying, the Prover will then remove the two corresponding hats, revealing a small portion of their solution:



Notice that there are two ways that this challenge can turn out:

1. If the two revealed vertices are the same color (or aren't colored in at all!) then I definitely know that the Prover is lying to me.
2. If the two revealed vertices are different colors, then the Prover *might not be* lying to me.

DRAFT

I hope that the first proposition is pretty clear. The second one requires a bit more thought. The problem is that even if the Prover successfully shows me two different colors, that doesn't mean that their whole solution is valid. They could still be lying to me! After all, I only looked under two of the hats. If there are E edges in the graph, then the Prover could fill in an invalid solution and still get away with it most of the time, specifically, with probability up to $(1 - 1/E)$. For a 1,000 edge graph, that works out to 99.9% of the time.

Let's run the protocol again! Fortunately the Prover has an answer to this. We'll just run the protocol again! We put down fresh paper with a new, blank copy of the same graph. The Prover now shuffles its crayons to pick a new ordering of the colors. Next they fill in the graph with the same valid solution, but using the new random ordering of the three colors. The hats go back on.

When I come back into the warehouse, I can pick a new random edge and repeat the challenge process again. Once again the same logic applies. If I catch the Prover in a lie, then I know they are definitely trying to cheat me. If the hats are colored with two different colors, I should be slightly more confident that the Prover is being honest. That's because in order to cheat me, the Prover would have had to get lucky twice in a row. This can happen, but the probability is lower now. The chance that the Prover fools me twice is $((E-1)/E)^2$ (or about 99.8% if our graph has 1,000 edges.)

Of course we don't have to stop at two challenges. In fact, we can keep trying this over and over again until I'm confident that the Prover is *probably* telling me the truth. Note that I'll never be perfectly certain that the Prover is being honest; there's always going to be a tiny probability that they're cheating me and just got *very* lucky. But after a sufficient number t of repeated rounds I can eventually reduce the probability of being cheated to an arbitrarily low (but non-zero) value.[7] This can help me reduce the probability to a point where it's concretely low enough that for all practical purposes I can stop worrying. And then I'll be able to safely hand Google my money.

DRAFT

How does this protect the Prover's witness? For this protocol to be zero-knowledge, I also need to believe that the Prover's solution is protected. This is particularly important, since we run the same coloring/hat-removal protocol many times and I (the Verifier) might try to piece together the full graph coloring by taking off different hats at every round in the protocol.

At an intuitive level, the argument here is simple. What you should notice is that the Prover *shuffles* its crayons each time it colors in the graph. Even though the Prover is using the same overall solution to coloring the graph, the specific ordering of the three crayons changes each time through. Let's just consider the case where the Prover is honest (we don't provide any guarantees to Provers who behave dishonestly.) When the Verifier removes two

[7] More concretely, let $|E|$ be the number of edges in the graph. To drive the worst-case probability below some arbitrary value ϵ, set the number of rounds t to be $t \geq |E|ln(1/\epsilon)$. Setting $t \approx |E|log^2|E|$ will ensure that this probability is *negligible* in $|E|$.

hats on a given edge, they will always see two different colors. However *which* colors they see depends on the Prover's random choice of crayons: there are six possible choices of color combinations, and the one the Verifier sees is equally probable to be any of the six. This means the "information" the Verifier learns is essentially random itself. More importantly, it should be very hard for the Verifier to link together any consistent picture of the full solution by trying to take notes about the coloring over many rounds, since none of the colorings will be the same between rounds. (If this explanation isn't completely satisfying, don't worry, I'll try to do better in a minute.)

Properties of zero-knowledge proofs. To talk about why the GMW "hats" protocol is secure, it's important to understand what properties a zero-knowledge protocol needs to have. Goldwasser, Micali and Rackoff proposed three following properties that every zero-knowledge protocol must satisfy [**?**]. Stated informally, they are:

1. **Completeness.** If the Prover is being honest, then they will eventually convince the Verifier (at least with high probability).
2. **Soundness.** The Prover can only convince the Verifier (with high probability) if they're actually telling the truth.
3. **Zero-knowledgeness.** (Yes it's really called this.) The Verifier doesn't learn anything else about the Prover's solution.

We've already discussed the argument for completeness. The protocol will eventually convince a Verifier (at least with high probability), provided that we run it enough times. Soundness is also pretty easy to argue: if the Prover ever tries to cheat, a Verifier will detect their treachery with overwhelming probability. The hardest part to be really convinced of is the *zero knowledgeness* property. To do this, we need to conduct a very strange thought experiment.

DRAFT

Simulating knowledge (with time machines.) To explain this idea, I need to start with a crazy science-fiction scenario. Imagine that Google's engineers aren't quite as capable as people make them out to be. They work on this problem for weeks and weeks, but they never manage to come up with a solution that three-colors the graph. With twelve hours to go until showtime, the Googlers get desperate. They decide to trick the Verifier into thinking they have a coloring for the graph, even though they don't.

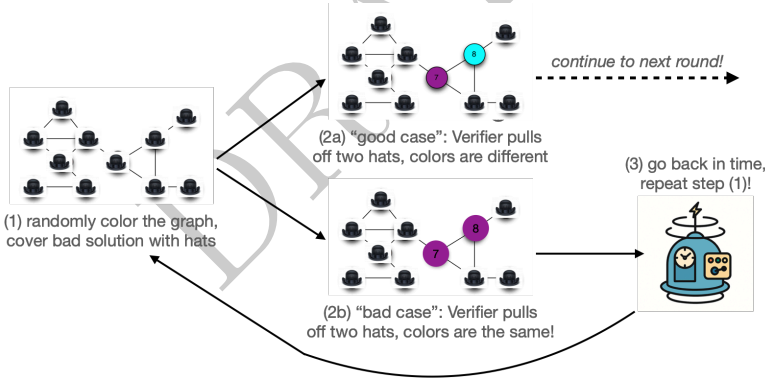
Intuitively this should be hard: we've already argued that the protocol is *sound*, meaning that the Verifier should eventually catch on to any cheating with high probability. However, this assumes that the Universe works the way we think it does. The Googlers have an ace up their sleeve: their idea is to sneak into the GoogleX workshop and borrow Google's prototype time machine. A time machine seems like it would solve most problems, but unfortunately this one is a bit more limited. Most critically: it's only capable of going backwards in time *four and a half minutes.* That means they can't use the machine to buy a bunch more time

to actually solve the problem. Still, it turns out that even this very limited capability can still be used to cheat at the hats-based proving protocol we described above.

The plan they come up with works like this:

1. Since the Prover doesn't actually know a valid three-coloring for the graph, they'll simply color the vertices of the graph with a bunch of random colors, then put the hats on.
2. If by sheer luck, the Verifier happens to "challenge" on an edge where the vertices are different colors, the Prover will heave a sigh of relief and move on to the next round. This will happen about 2/3 of the time, since (if the vertices are colored at random) that's the probability any two vertices have different colors.
3. If, eventually, the Verifier picks an edge where the vertices *are the same color*, this is where the time machine comes in.

If the Prover ever gets caught with two vertices that are the same color, they can "solve" their problem very easily: simply flip the switch on the time machine, and travel back in time by about four minutes. This gives them enough time to quickly fill in the entire graph with an *entirely new set of random colors*.[8] Now they let time roll forward and they try again.



In effect, the time machine allows the Prover to "have a second chance" any time they get caught faking their bogus protocol execution. Notice that the Prover spends slightly more subjective time running this protocol, due to the extra time they spend time traveling and repeating rounds, but the good news is that they only need to do this about one third of the time, since that's the probability that two vertices will have the same color. What's

[8] Alternatively, the Prover could just "repair" the part of the graph that the Verifier challenged them on, coloring it in with two different colors. Which solution you choose depends on whether you think the Verifier will stick with the original edge it chose, or if "the butterfly effect" is real, and the effect of the time machine will cause the Verifier to change it's challenge point. Since I don't know how real time travel works, I'm going to be cautious and assume the worst.

most amazing about this time-machine cheating approach is that from the perspective of the Verifier, the resulting protocol execution looks exactly like a real, successful run of the protocol. And I mean this mathematically: the distribution of the results the Verifier seems should be *statistically identical* to the distribution it sees during the normal protocol.

What does this mean about "knowledge"? Notice that in the real protocol, there is knowledge to be gained from the Prover. After all, it starts the protocol with a *witness*, which is a real solution to coloring the graph. In that version of the protocol, we have a legitimate concern that some of the Prover's knowledge might leak through the protocol to the Verifier. What's important to note about this "time machine" version of the protocol is that the amount of "knowledge" the Prover started with *is literally zero.* We can be certain that the Verifier doesn't get any knowledge about the solution from the Prover, simply because there's no knowledge to obtain![9] Yet the Verifier cannot learn more from the transcript of the real version of the protocol than from the "time machine" one, because we already stipulated that the distribution of the protocol transcripts is statistically identical. (If the Verifier could learn more about the witness from the non-cheating protocol, they would clearly be able to distinguish the distribution of that version of the experiment from the other one, and they would not be statistically identical.) This means that, on average, the amount of knowledge about the graph coloring that a Verifier can extract from the Prover must be *identical* between the two different scenarios, and in one of the scenarios the amount of knowledge is zero. That must also be the amount of information it can obtain from the Prover in the real protocol.

What if we don't have time machines? The obvious problem with the "time machine" protocol is that it requires, well, time machines. Computer scientists don't have those. But in the setting we care about, we have something almost as good. To explain what I mean, consider what the Verifier actually is. So far we've been talking about the Verifier being a person. In practice, the kind of Verifier we care about is actually an *algorithm*, one that might be running on a computer, usually modeled as a Turing machine. When we talk about the possibility of a clever Verifier that "learns" some knowledge about the solution (*i.e.,* defeats the zero-knowledgeness property), we're suggesting there might exist an algorithm that contains a strategy to learn something about the witness from seeing the results of this protocol. For a protocol to be zero-knowledge, what we must actually show is that there *exists no algorithm* that can do this.[10]

[9] This is called the "you can't get blood from a stone" principle. Note that it does not mean that the Verifier can't learn something about the solution; they could try to solve the graph on their own. What it means is that any knowledge the Verifier gained comes from itself, and not from the Prover.

[10] Notice that this formulation rules out a lot of things. For example, we're going to explicitly leave out *side channels* that might leak out of the Prover (*e.g.,* via EM radiation, timing, or power consumption, see §13.3.) By specifying an algorithm running on a Turing machine *and only sees the intended outputs of the protocol*, we also get to rule out crazy things like supernatural powers and ESP, since those would be outside of our model.

DRAFT

A digital "hats" protocol. Before we can talk about algorithms and computers, we'll first need to bring our protocol into the digital world. We don't need paper, because a graph is just a set of vertices and a set of edges. Every vertex can be labeled with one of three possible "colors" (just the integers $0, 1$ or 2.) Building our protocol digitally also means we need the digital equivalent of a "hat": something that both hides a digital value, while simultaneously binding the maker to its value, so she can't change her mind after the fact. Fortunately we've already discussed the tool for his application: this is simply a *commitment scheme*, which we discussed in §6.2.

Given a commitment scheme, we now have all the ingredients we need to run the "hats" protocol electronically.[11] It looks like this:

1. The Prover first randomly shuffles the three "colors" and makes a list of integers representing the coloring for each node.
2. The Prover commits to each integer using a commitment scheme, holds on to the *opening* value.
3. The Verifier receives the list of commitments from the Prover.
4. The Verifier picks a specific edge in the range $\{0, 1, \ldots, |E| - 1\}$ and sends it to the Prover.
5. The Prover then sends back (a) the integer color values for those two vertices, and (b) the commitment opening for each.
6. The Verifier checks that the openings match the commitments it receives, and that the two vertices have different colors.
7. This whole process repeats as many times as necessary.

This new protocol runs entirely digitally and we've managed to eliminate the hats. But how do we prove that this protocol is zero knowledge? To do this, we need to think about the statement we want to prove. More formally, what we're really interested in arguing is the following:

> *For all possible Verifier algorithms, no algorithm learns any information about the Prover's witness (that it didn't figure out by itself.)*

Rewinding. Since we're now proving statements about algorithms, we no longer need a time machine to prove things about this protocol. Let's imagine that our statement above is incorrect: that is, *there does* exist some Verifier algorithm that learns information about the Prover's witness. Then we want to investigate what that means, and decide if the claim is even possible.

The main trick here is to recognize that since the Verifier algorithm we're interested in is just a classical algorithm,[12] then we can always "rewind" that algorithm to an earlier stage

[11] We'll assume that both sides have a description of the full (uncolored) graph.

[12] Generally we specify that it's running on a computer design that we understand, like a classical probabilistic Turing machine.

DRAFT

in its execution. The way we do that is to emulate our "time machine" trick above, randomly coloring in the graph. Whenever we get challenged on a pair of vertices where we don't have a good answer, we simply *rewind* the algorithm to the point before it issues a challenge; then we recolor the graph with a new random coloring and try again. To do this rewinding on any computer, in the worst case we can simply start running the Verifier algorithm over from beginning — feeding it all the same inputs (and random bits, if it uses any) — until it winds forward to the point where we wanted to rewind to.

What we can prove by this is the following statement: if there exists an algorithm that somehow gains "knowledge" about the witness, from seeing the inputs and outputs of the real protocol, then that algorithm must be able to get *as much information* from running it against our a version of the protocol where the Prover doesn't even know a valid witness. We show this by demonstrating that the values the Verifier sees are distributed in a manner that's statistically identical in both protocols. From this, we can argue that since the amount of "knowledge" we have in the fake protocol is *zero*, then the amount of knowledge this Verifier algorithm gains after participating in the real protocol is about the same.

From interactive to non-interactive: Fiat-Shamir. One obvious limitation of the proof systems we've discussed is that they're *interactive*. The Prover and Verifier must exchange messages, and they may need to do this many times. Even worse, if I conduct a proof with some Verifier, then that proof can't be used to convince a second Verifier![13] A much nicer idea would be to get rid of all that interaction and somehow build a *non-interactive* proof: this proof would be a single message that can be checked by any Verifier without interaction.

Fortunately there is a technique that allows us to take an interactive proof in the style of GMW, then turn it into a non-interactive one. The technique was invented by Amos Fiat and Adi Shamir, and it relies on the idea that the Prover can run *its own* copy of the Verifier algorithm, and then hand out the transcript of this interaction. Of course, that doesn't sound secure at all, since the Prover could simply cause its "Verifier" to issue challenges that it can answer! To prevent the Prover from doing this, Fiat and Shamir came up with the following clever trick:

1. The Prover sends the first message of the interactive protocol to the "Verifier" (this is the list of commitments to the graph colors, if we're talking about GMW.)
2. The "Verifier" then *hashes* the first message from the Prover, and converts the resulting hash into a *challenge* (in GMW, this is an integer representing the edge to open.)[14]

[13] This is a real issue with interactive proofs: even if one Verifier records a transcript of the protocol it conducted with the Prover, it cannot convince a second Verifier that the proof was valid! For example, the GMW protocol relies on the idea that the honest Verifier chooses the "challenge" edges at random. If the Verifier wanted to *collude* with the Prover, it could pick the challenge edges in such a way that the Prover would always be able to answer the responses! So a proof transcript is only guaranteed to be useful to the Verifier who actually chooses the challenges.

[14] For GMW we can do this, for example, by hashing the commitment message to an integer between 0

DRAFT