Proofs of knowledge, identification protocols, and Fiat-Shamir

Noah Stephens-Davidowitz

October 9, 2019

1 Proofs of knowledge

1.1 NP statements are not enough

The zero-knowledge proofs that we saw in the last two classes are extremely useful for proving facts that can be represented as NP statements. “Are these two graphs isomorphic?” “Is this graph 3-colorable?” “Is there a proof of the Riemann hypothesis?” If the answer to any of these questions is yes, then there is a zero-knowledge proof for this fact. (There are zero-knowledge proofs when the answer is negative as well, but probably not with efficient provers in the case of 3-colorability.)

Formally, these statements are all of the form “there exists $y \in \{0, 1\}^{\text{poly}(|x|)}$ such that $R(x, y) = 1$,” where $R$ is some efficiently testable relation.

But, what if we want to prove something else? For example, suppose we want to login to some website. Presumably, the website has some information about us. E.g., the website might know $x := f(y)$, where $f$ is some (public) one-way function and $y \in \{0, 1\}^n$ is some secret random string (i.e., a password). How do we prove that we possess $y$?

A zero-knowledge proof seems like it should be useful here. E.g., we could use a zero-knowledge proof to show that $x$ is in the image of $f$. But, what if $f$ is surjective? Then, that fact is trivial. Even if $f$ is not surjective, it might be possible to efficiently test whether $x$ is in its image. And, even if that’s not possible, the fact that we can prove that $x = f(y)$ for some $y$ does not mean that we know $y$.

As another example, suppose we’re engaged in some protocol, and we wish to prove that we did something honestly. E.g., maybe our first message in the protocol was supposed to be $m := G(s)$ for some (public) PRG $G$ and (secret) string $s$. We would like to prove that we did in fact generate $m$ in this way. How do we do this?

What we really need here are not proofs of NP statement, but proofs of knowledge. A proof of knowledge is (as the name suggests), a proof that we know something. In the password example, what we really want is a proof of knowledge of $y$. This is exactly what websites check when we log in. They don’t force us to prove that we are in fact who we claim to be; they just ask us to prove that we do in fact know what we claim to know: our password, $y$, a preimage of $x$. And, though it’s less clear, in the second example we similarly want a proof that we know $s$ such that $m = G(s)$.

\footnote{To see why this is the case, we have to step back and ask what it would mean to “prove that we generated $m$ by computing $m := G(s)$ for some $s$.” Maybe there are many different efficient algorithms for $G$. Maybe there is some clever way to simultaneously generate $m$ and $s$ such that $m = G(s)$. The best that we can do is prove that we know $s$ such that $m = G(s)$. Then, while we might have found $m$ in some other way, we could have found $m$ by computing $m = G(s)$. This is the best that we can hope for. (And, as we will see later in the course, this is a very
1.2 Proofs of knowledge

Of course, we are now again saddled with the task of worrying about what it means to “know” something. When we defined zero-knowledge, we settled on the definition that what we “know” corresponds to what we can compute efficiently. So, a proof conveys no “knowledge” if interacting with the prover does not allow us to compute anything new (or, well, anything that is computationally distinguishable from what we were already able to compute).

So, to prove that we know something, we should prove that we can compute it. To prove that we can compute something, we could just send it to the verifier, but that would certainly not be a zero-knowledge proof. (Of course, this is often done in practice.) A more clever solution is to engage in a protocol that demonstrates our ability to compute something, without just sending it directly. Specifically, we engage in a protocol from which a witness could be extracted.

The specific definition that we use is not ideal, but it will suffice for our purposes. (The correct definition takes the constant $C$ to be 1.)

Definition 1.1. A protocol between a prover $P$ and a verifier $V$ is a proof of knowledge with knowledge error $\varepsilon(n)$ of a witness for an NP relation $R$ if there exists a PPT extractor $E$ and a constant $C > 0$ such that for every string $x \in \{0, 1\}^*$ and every (malicious, not necessarily efficient) prover $P^*$

$$\Pr[R(x, y) : y \leftarrow E^{P^*}(x)] \geq \left( \Pr[\langle P^*, V \rangle(x) = 1] - \varepsilon(|x|) \right)^C.$$  

Notice that a proof of knowledge with knowledge error $\varepsilon$ must also be sound with soundness error $\varepsilon$. So, we can think of this as a strengthening of soundness.

Let’s try to unpack this definition. The first term $\Pr[\langle P^*, V \rangle(x) = 1]$ is the probability that $P^*$ convinces $V$ that it knows a witness. The second term $\Pr[E^{P^*}(x) \in \text{Wit}(x)]$ is the probability that an extractor with oracle access to $P^*$ successfully extracts a witness. So, intuitively, the above definition says that, if $P^*$ has decent probability of convincing $V$, then there is some efficient algorithm $E$ with oracle access to $P^*$ that actually finds such a witness. In other words, $P^*$ must “know” a witness, since an efficient algorithm could “use $P^*$” to find a witness.

1.3 Graph isomorphism

Let’s recall our zero-knowledge proof for graph isomorphism. The protocol went as follows. (We write $S_n$ for the permutations on $n$ elements.)
This isn’t a great proof of knowledge of the witness \( \pi \), but it is a proof of knowledge with knowledge error \(1/2\). To see this, we construct an extractor \( E \) that behaves as follows.

When the extractor \( E \) receives the first message \( H \) from \( P^* \), it first responds with the bit 0, receiving in response the permutation \( \pi_0 \). It then “rewinds” \( P^* \) back to its state immediately after it sent the first message \( H \) and sends it an alternative challenge 1, receiving in response the permutation \( \pi_1 \).

\( E \) then outputs \( \pi_1 \circ \pi_0 \).

To see that this extractor works, first notice that we may assume that \( \delta := \Pr[\langle P^*, V \rangle(G_0, G_1) = 1] - 1/2 > 0 \). (Otherwise, the definition is vacuous.) We want to prove that \( E \) succeeds with probability at least \( \delta^2 \).

Let \( \text{GOOD} \) be the set of all states \( \sigma \) of \( V^* \) after the first message such that, conditioned on that state, this probability is at least, say, \( 1/2 + \delta/2 \). By Markov’s inequality, the probability that \( P^* \) is in such a state after the first message is at least, say, \( \delta \). Fix such a state \( \sigma \), and let \( p_b \) be the probability that \( \pi_b(G_b) = H \) for \( b \in \{0, 1\} \). Then, \( p_0 + p_1 \geq 1 + \delta \), and it follows that \( p_0 p_1 \geq \delta \) (since the function \( x(1 + \delta - x) \) is minimized at \( x = 1 \) in the interval \( x \in [1/2 + \delta/2, 1] \)). Notice that \( p_0 \cdot p_1 \) is exactly the probability that \( E \) succeeds conditioned on the state \( \sigma \).

So, the probability that \( E \) succeeds is at least the probability that the state after the first message lies in \( \text{GOOD} \) times the probability that \( E \) succeeds conditioned on this event. Each of these probabilities is at least \( \delta \), so their product is at least \( \delta^2 \), as needed.

(There is a more clever extractor for this function that achieves success probability exactly \( \delta \).)

### 1.4 Schnorr’s protocol for discrete logarithm

We now present Schnorr’s (beautiful, well-known) honest-verifier zero-knowledge proof of knowledge for the discrete logarithm [Sch90]. We assume that \( G \) is some group with prime order \( q \) generated by \( g \). (Formally, we should define a sequence \( G_n, g_n \) of groups and generators, but at this point in the course, we are comfortable dropping these formalities.)

---

2There are many different ways to view this “rewinding” procedure depending on how we imagine \( E \) interacts with its oracle. For example, perhaps \( P^* \) is implemented as a first-message function and a second-message function. The first-message function outputs a state \( \sigma \) together with the first message \( H \). The second-message function takes as input \((H, \sigma, b)\) and returns a second message \( P^* \). Then “rewinding” amounts to calling the second-message function twice with the same \( H \) and \( \sigma \), but different \( b \).
Correctness of the protocol is easy to see. In fact, it's quite easy to see that it's honest-verifier zero-knowledge as well. Indeed, for a simulator $S$ to reproduce the view of the honest verifier $V$, it suffices to simply sample $c, z \sim \mathbb{Z}_q$ uniformly at random and set $u = g^zh^{-c}$. Clearly this new view is identically distributed to the view of an honest verifier. (This is certainly not a zero-knowledge protocol against arbitrary verifiers, but for our most important use case, we will only need honest verifiers.)

To prove that the protocol is actually a proof of knowledge with negligible knowledge error, we have to construct an extractor $E$. The intuition is as follows. For fixed $u \in G$ and $c \neq c'$, $z, z' \in \mathbb{Z}_q$ such that $g^z = uh^c$ and $g^{z'} = uh^{c'}$. Let $t := (c - c')^{-1}$ be the inverse of $c - c'$ modulo $q$. Then, $g^{t(z-z')} = h$. I.e., $t(z - z') = x \mod q$.

To make this formal, we do another “rewinding” proof. I.e., our extractor $E$ will first run $P^*$, receiving some message $u$. It then samples distinct $c, c' \sim \mathbb{Z}_q$ and send $P^*$ both messages, receiving as its answers $z, z'$. (I.e., it sends it $c$, receives the answer, rewinds, and then sends it $c'$ and receives the answer to that.) If $g^z = uh^c$ and $g^{z'} = uh^{c'}$, then the extractor outputs $t(z - z') = x \mod q$.

The proof that this works is essentially identical to the proof in the previous section. We therefore leave it as an exercise.

## 2 Identification protocols

An identification protocol is a protocol that allows an efficient prover $P$ to prove that $P$ really is, well, $P$.

Of course, what we actually prove is that we know something (not that we are someone). Specifically, we prove that we know some secret key that is related to some public key. The primitive that allows us to do this, an identification protocol, is our first example of a public-key primitive. (Often, when we say “public-key primitive,” what we really mean is a primitive that is so strong that its existence would imply public-key encryption or something similar. Here, we just mean that there is a public key.)

Again, we use a relatively weak notion of security for now.

**Definition 2.1.** An identification protocol consists of an efficient key-generation algorithm $G$, which outputs a secret key $sk$ and a (public) verification key $vk$, and a protocol between an efficient prover $P$ and an efficient verifier $V$ satisfying the following.
• Correctness:  
$$\Pr_{(sk,vk) \sim G(1^n)} [(P(sk,vk), V(vk)) = 1] = 1.$$ 

• Security:  
For every PPT adversary $A$ there exists a negligible $\varepsilon(n)$ such that  
$$\Pr_{(sk,vk) \sim G(1^n)} [(A(P(sk,vk), V(vk)) (vk), V(vk)) = 1] \leq \varepsilon(n),$$  
where we write $A(P(sk,vk), V(vk))$ to represent an adversary $A$ that may request as many transcripts of the protocol between $P$ and $V$ as it likes.

(A much stronger definition would give $A$ arbitrary oracle access to $P(sk,vk)$, rather than just transcripts between $P(sk,vk)$ and an honest verifier $V(vk)$.)

In fact, Schnorr's protocol essentially already is an identification scheme (if discrete log is hard). The key generation algorithm simply picks a group $G$ with prime order $p$ and generator $g$, samples uniformly random $x \in \mathbb{Z}_p$, and sets $vk = (G,g,g^x)$ and $sk = x$. The protocol itself is exactly Schnorr’s protocol. (There is actually a more general theorem showing that any zero-knowledge proof of knowledge for an appropriate language implies an identification scheme, but we will not bother to introduce the definitions.)

**Theorem 2.2.** Schnorr’s protocol is a secure identification protocol (if, given $G,g,g^x$, it is hard to find $x$).

**Proof.** We suppose not. I.e., we suppose that we have some efficient adversary $A$ with non-negligible advantage in the above game. The proof goes in two steps. First, we use the honest-verifier zero-knowledge property of Schnorr’s protocol to argue that the output of $A(P(sk,vk), V(vk))$ is indistinguishable from the output of $A_{S(vk)}$, where $S$ is some PPT simulator. Then, we notice that $A_{S(vk)}$ can be implemented entirely by a PPT algorithm $A'$, since $S$ is itself PPT and $A$ knows $vk$.

So, we have converted our adversary $A$ into an $A'$ that $Pr[(A'(G,g,g^x), V) = 1]$ is non-negligible. Notice that this is exactly the security game in the definition of a proof of knowledge for Schnorr’s discrete log protocol. In particular, the fact that this protocol is a proof of knowledge means that there exists some PPT extractor $E$ such that $E_{A'}$ outputs $x$ with non-negligible probability.

Then, like we did earlier with $S$, we observe that $A'$ is PPT, so that the entire oracle algorithm $E_{A'}$ can be run by a single PPT algorithm $B$. I.e., $B$ is a PPT algorithm that takes as input $(G, g, g^x)$ and outputs $x$ with non-negligible probability. This is a contradiction, since we assumed that the discrete logarithm is hard over $G$.  

3 An informal overview of Fiat-Shamir, identification to signatures, and ZK to NIZK in the random oracle model

Finally, we show how to use the Fiat-Shamir heuristic to convert certain interactive protocols into non-interactive protocols.

A $\Sigma$ protocol (pronounced “sigma protocol”) is a “commit-challenge-response” protocol. I.e., a $\Sigma$ protocol consists of three messages, the first sent by the prover, which we think of as a “commitment.” (Sometimes it is literally a commitment, but it need not be.) The second message—the “challenge”—is of course sent by the verifier, and it should be uniformly random over some domain.
The final message—the “response”—should be publicly verifiable. I.e., it should be possible to efficiently determine whether the verifier accepts based only on the transcript (and not on any secret state kept by the verifier). (The Greek letter Σ is meant to look sort of like the flow of messages in a three-message protocol.)

In fact, every single zero-knowledge protocol that we have seen is a Σ protocol. This gives a very useful framework for designing protocols. It also turns out that Σ protocols (with negligible soundness error and $|m_2| > \omega(\log n)$) can be made non-interactive. I.e., we can collapse the three messages $m_1, m_2, m_3$ into one message $(m_1, m_2, m_3)$. The catch is that we need a random oracle.

To accomplish this we replace the second message $m_2$—the “challenge”—with the output of the random oracle $H$. The output of the random oracle on what input? Well, the output when the oracle is called on the first message, $m_2 := H(m_1)$. Then, given access to $H$, a verifier can verify the proof $(m_1, m_2, m_3)$ by checking that $m_2 = H(m_1)$ and then using the public verification algorithm to check that $(m_1, m_2, m_3)$ form a valid transcript.

This is known as the Fiat-Shamir heuristic [FS87], and it is extremely useful. Clearly, the resulting non-interactive proof is still complete (if the original protocol was complete). Intuitively, it should still be sound too. To see why, imagine some malicious prover $P^*$. No matter what first “commit” message $m_1$ is sent by $P^*$, the random oracle will output a uniformly random challenge $m_2 := H(m_1)$. So, if $P^*$ could find a valid response to $m_2 := H(m_1)$ with non-negligible probability then, intuitively, one would expect there to be a similar prover in the original protocol.

To make this formal, we “program” the oracle. I.e., we first observe that, since the random oracle is random, any cheating prover $P^*$ for this non-interactive game can only output a proof $(m_1, m_2, m_3)$ with $H(m_1) = m_2$ with non-negligible probability if it queries $H$ on $m_1$. (Here, we need the assumption that $|m_2| > \omega(\log n)$.) So, we build a cheating prover $B$ in the soundness game of the original protocol as follows. Let $q = \text{poly}(n)$ be a bound on the number of queries made by $P^*$.

$B$ chooses an index $i \in [q]$ uniformly at random. It then simulates $P^*$. Each time that $P^*$ makes a query to the random oracle $H$, $B$ answers with a uniformly random string (handling repeated queries appropriately). But, in the $i$th query, $B$ does something different. It takes the query $m_1$ made by $P^*$ and sends it to its verifier $V$. The verifier responds with a uniformly random message $m_2$, and $B$ sets $H(m_1) = m_2$. It then continues to simulate $P^*$ as before until it outputs $(m_1', m_2', m_3)$. If $m_1' = m_1$ and $m_2' = m_2$, then $B$ sends $m_3$ to its verifier. (Otherwise, it gives up.)

Let $\varepsilon$ be the advantage of $P^*$ in the non-interactive game. We claim that the advantage of $B$ is at least $\varepsilon/q - 2^{-|m_2|}$. Notice that the view of $P^*$ is identical to its view in an honest game. I.e., the output of the simulated random oracle $H$ is uniformly random. (Here, we are using the fact that the second message is uniformly random in the original protocol.) Therefore, the probability that $P^*$ outputs a proof $(m_1', m_2', m_3)$ is exactly $\varepsilon$. Furthermore, $i$ is a uniformly random element from $[q]$, independent of the view of $P^*$. Therefore, if this valid proof has $m_1'$ equal to one of the queries made by $P^*$ to its random oracle, then we have $m_1' = m_1$ with probability at least $1/q$. On the other hand, conditioned on $m_1'$ not being equal to any query made to the oracle, the success probability of $P^*$ is at most $2^{-|m_2|}$. The result follows by union bound.

### 3.1 Fiat-Shamir signatures

Of course, the fact that these proofs are complete and sound isn’t very interesting on its own. NP statements already have non-interactive proofs that are complete and sound; the “prover” just
sends a witness.

We would like to say that these proofs have some sort of zero-knowledge property (when the original protocol does). This can be done, but rather than dive into the thorny world of non-interactive zero knowledge, let’s build another primitive: signatures. We won’t bother to give a formal definition now; we will see it over the next two lectures. Instead, we just give an intuitive idea of what’s going on.

A signature allows someone with a secret key \(sk\) to efficiently “sign” any plaintext \(p \in \{0,1\}^*\), i.e., to compute a signature \(\sigma\) of \(p\) such that anyone with a verification key \(vk\) can verify that \(\sigma\) is in fact a valid signature of \(p\). Security of a signature scheme says that an adversary that has seen many signed plaintexts \((p_1,\sigma_1),\ldots,(p_\ell,\sigma_\ell)\) should not be able to produce a valid signature for some other plaintext \(p \neq p_i\). (This definition is not very formal, and there are multiple ways to make it formal that result in distinct primitives.)

Notice that a signature scheme has a lot in common with an identification protocol, except that (1) a signature scheme is non-interactive; and (2) signatures are associated with plaintexts \(p\). To convert an identification scheme into a signature scheme in the random oracle model, we use the Fiat-Shamir trick, but we include \(p\) as input to the random oracle. I.e., we compute the first message \(m_1\) of an identification scheme as normal. We then set \(m_2 := H(m_1,p)\), and compute \(m_3\) as normal. The signature is then \((m_1,m_2,m_3)\).

This simple idea, instantiated with Schnorr’s identification protocol is used quite a bit because of its simplicity and efficiency, with the random oracle replaced by some hash function. The proof of security in the random oracle model is also simple—it is essentially the same as the proof of soundness that we gave above. However, proving security when the random oracle is replaced by any explicit hash function \(H\) is quite challenging.

Indeed, Yael and Shafi showed that there exist secure identification protocols that do not yield secure signature schemes under the Fiat-Shamir heuristic when they are instantiated with any efficiently computable function \(H\) [GK03]. (Of course, if we use a random oracle for \(H\), then the scheme is secure.)

References

