# Privacy Enhancing Technologies FS2025
# Lecture 5-6-7-8 – Zero-Knowledge Proofs

Florian Tramèr

AGENDA

1. Interactive Proof Systems
2. Zero-Knowledge Proofs
3. Proofs of Knowledge
4. Sigma Protocols
5. Non-interactive Zero-Knowledge

## 1 Interactive Proof Systems

The goal of a proof is to convince someone that a statement is true. This is useful in many cryptographic settings, e.g., proving that "$N$ is a product of two primes" or "the input $x$ to the program `prog` causes a crash", etc.

**Languages.** To formalize proof systems, we will treat statements with respect to a *language* $\mathcal{L}$. A language is a set of strings $\mathcal{L} \subseteq \{0,1\}^*$ and a statement is of the form $x \in \mathcal{L}$.

Examples:

- "$N$ is a product of two 1024-bit primes": $\mathcal{L} = \{pq \mid p, q \text{ are 1024-bit primes}\}$
- "the input $x$ to the program `prog` causes a crash": $\mathcal{L} = \{x \mid \texttt{prog}(x) \text{ crashes}\}$

**One-shot proofs and NP.** Let's first consider standard one-shot proofs, where the prover "writes down" a proof $\pi$ and the verifier checks its correctness. We can view this as a single round protocol between two algorithms, a (possibly unbounded) prover $P$ and an efficient verifier $V$, on some statement $x$:

$$\text{Prover } P(x) \xrightarrow{\text{proof } \pi} \text{Verifier } V(x) \downarrow \text{accept / reject}$$

A useful proof system should satisfy the following properties:

- *Completeness*: If $x \in \mathcal{L}$, then an honest prover $P$ that follows the protocol will convince the verifier $V$.
- *Soundness*: If $x \notin \mathcal{L}$, then any malicious prover $P^*$ (that may arbitrarily deviate from the protocol) cannot convince the verifier $V$.

Which languages can be proven in this way? If the verifier $V$ is deterministic, then this is exactly the class NP. Recall that NP is the class of languages $\mathcal{L}$ such that there exists an efficient, deterministic algorithm $M$ such that:

$$x \in \mathcal{L} \iff \exists w \in \{0,1\}^{\mathsf{poly}(|x|)} \text{ s.t. } M(x,w) = 1$$

In our protocol above, the prover $P(x)$ would first compute the witness $w$ and then send it to the verifier $V(x)$, i.e., $\pi = w$. The verifier would then check that $M(x,w) = 1$.

**The power of interaction.** So if we can already prove everything in NP with one-shot proofs, what's the point of interactivity? It turns out that interactive proofs are (under plausible assumptions) much more powerful than one-shot proofs. For example, they allow us to:

- Prove more things than NP: In a celebrated result, Shamir [Sha92] showed that the class of languages that have an interactive proof is exactly PSPACE, i.e., the class of languages that can be decided in polynomial *space*. This includes many problems that we believe are not in NP, such as *counting* the number of satisfying assignments to a 3-SAT formula.
- Make proofs succinct: in a "one-shot" NP proof, the proof size (the size of the witness $w$) is $\mathsf{poly}(|x|)$. Using interaction, we can sometimes bring down the communication cost to *sublinear* in the witness size. We will talk about this more when we introduce SNARGs.
- Prove in zero-knowledge: a "one-shot" NP proof reveals the entire witness to the verifier. Using interaction, we can prove things without revealing anything aboutthe witness!

**Defining interactive proofs.**

**Definition 1** (Interactive Proof)**.** Consider a (possibly multi-round) protocol between an unbounded prover $P$ and an efficient (possibly randomized) verifier $V$. Given a statement $x$, let $\langle P, V\rangle(x) \in \{0,1\}$ denote the output of the protocol, i.e., the verifier's final answer. We say that the protocol $(P,V)$ is an interactive proof system for $\mathcal{L}$ if it satisfies the following properties:

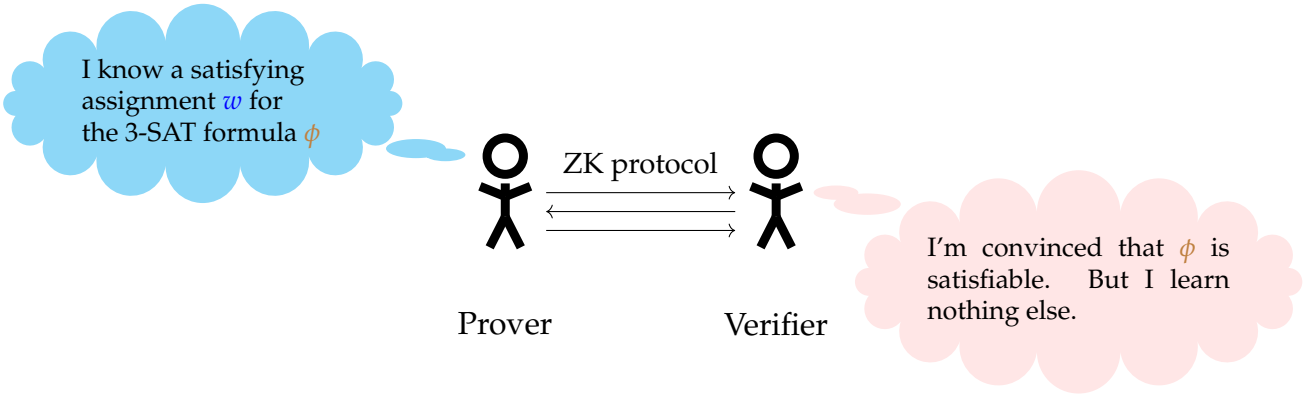
- *Completeness*: $\forall x \in \mathcal{L}$,
$$\Pr[\langle P, V\rangle(x) = 1] \geq \frac{2}{3}.$$
- *Soundness*: $\forall x \notin \mathcal{L}$, for all (possibly malicious) provers $P^*$,
$$\Pr[\langle P^*, V\rangle(x) = 1] \leq \frac{1}{3}.$$

The probabilities $\frac{2}{3}$ and $\frac{1}{3}$ (which are called the *completeness probability* and *soundness error*) are mostly arbitrary, as we can always *amplify* them by repeating the protocol $O(\lambda)$ times, leading to completeness probability $1 - \mathsf{negl}(\lambda)$ and soundness error $\mathsf{negl}(\lambda)$.

## 2 Zero-Knowledge

Zero-knowledge (ZK) is one of the most beautiful concepts in computer science. It's one of those things that seems impossible at first, but can actually be done (under reasonable assumptions) in a way that's practical enough for deployment. It's also one of those concepts where the most challenging (and beautiful) part was actually to come up with the right definitions.

A ZK proof system [GMR85] allows the prover to convince the verifier of some statement, *without revealing any information apart from the fact that the statement is true.*




Note that such a proof system provides different forms of security for both parties. On one hand, a cheating prover shouldn't be able to convince the verifier of a false statement (e.g., that some formula $\phi$ is satisfiable when it is not). On the other hand, a malicious verifier shouldn't learn anything about the prover's witness beyond the fact that the statement is true.

**Example applications.** There are many practical applications of zero-knowledge proofs. Here we list some of them:

- *Verifying passwords:* Servers typically store your password as a hash $H(p)$ (often with some additional randomness to prevent dictionary attacks). A login can thus be seen as a "proof" that the user knows the password $p$. But in practice, we do this proof in a very naive and straightforward way, by simply sending the password $p$ over the wire. This is bad in case the server is compromised (or if you fell for a phishing attack), as the server learns your password in the clear. A better approach, in principle, would be to use a ZK proof to prove that you know the password $p$ with hash $H(p)$ without revealing it. This leads to protocols known as Password-Authenticated Key Agreement (PAKEs), which are unfortunately only rarely used in practice.
- *Verifiable computation:* A client may wish to outsource the computation of some expensive function $f(x)$ to a powerful remote server, but be able to verify the correctness of the result. The goal here is for the client to be able to verify the result without having to re-run the computation themselves. As we will see, the property we need here is not zero-knowledge (as the server has no secrets), but rather a form of *succinctness* (the server's proof should be short and easy to verify).

  One variant of verifiable computation that would require zero-knowledge is a "proof-of-exploit". Say you know an exploit command $x$ that gives you root access to Linux. You could use a ZK proof system to convince someone that you have such an exploit (and maybe get paid a bounty), without revealing what the exploit is.

- *Private payments:* In cryptocurrencies such as Bitcoin, transactions are inherently *public*. That is, when one party sends money to another, everyone learns about the transaction taking place. This is not desirable in many cases.

  In Bitcoin, a coin is (roughly) represented as a tuple $(v, \texttt{pk})$ where $v$ is the value of the coin and `pk` is the public key of the owner. To send the money to another owner with key `pk'`, we create a new coin $(v, \texttt{pk'})$ and sign it using the owner's private key `sk`.

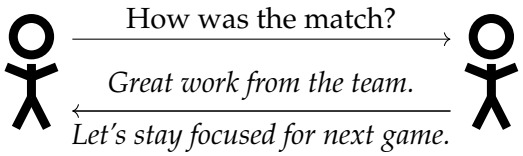
  Privacy-preserving cryptocurrencies such as ZCash use ZK proofs to make payments private, i.e., they hide the transacted value and the identities of the parties. In ZCash, a coin is of the form $c = \texttt{Enc}_{pk}(v)$, and a transaction creates a new coin $c = \texttt{Enc}_{pk'}(v')$, along with a ZK proof that $v' = v$, and that $\texttt{Dec}_{sk}(c) = v$. All participants can verify the proof to ensure that the transaction is valid, but don't learn anything else.

## 2.1 Defining ZK proofs

Recall the proofs for `NP` languages we saw above. The prover just sends the entire witness $w$ to the verifier. So the verifier doesn't just learn that the statement is true. It also learns the entire witness (which in turn would allow the verifier to convince someone else). That's where ZK comes in to play.

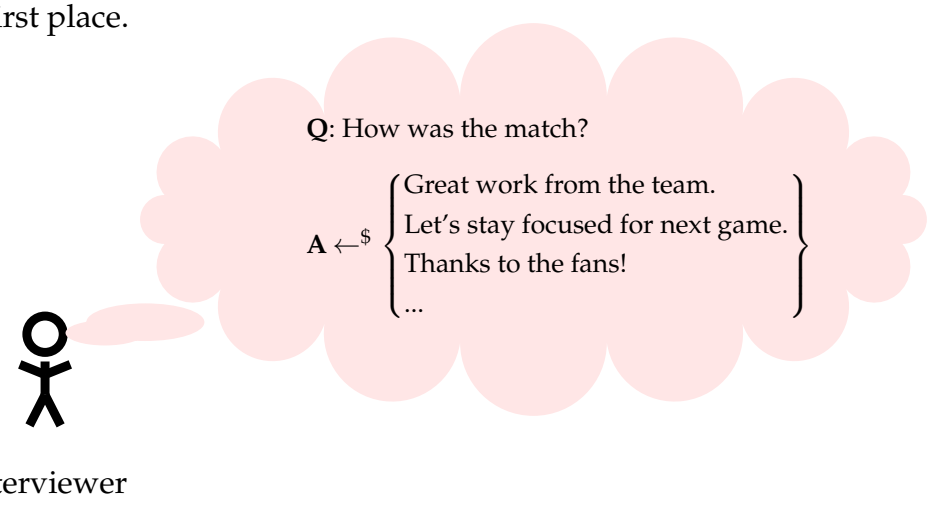
So what does it mean to learn "nothing" from an interaction?

**An analogy: the footballer's interview.** Interviews at the end of a football match typically go like this:




Interviewer Footballer

Neither the interviewer (or the audience) really learns anything from such interviews. Intuitively, we could just "simulate" the entire conversation without even needing the footballer to be there in the first place.




Interviewer

Clearly, in this case the interviewer learns nothing from the footballer since the footballer is not even there! To define zero-knowledge, we'll essentially say that whatever the interviewer discusses with the footballer, is *indistinguishable* from a discussion that the interviewer could have had by themselves.

**A formal definition.** To formalize this property, we first introduce the notion of a *View* of the verifier, which is all the information that the verifier gets access to during the execution of the protocol (i.e., all the messages that the verifier receives and sends). To argue zero-knowledge, we then construct a *simulator*. This is a probabilistic algorithm that takes as input everything that the verifier gets (i.e., just the statement $x$) and outputs a simulated view that is indistinguishable from the protocol view.[1] What this means is that anything the verifier can learn from the protocol could just as well have been simulated given just the statement $x$.

**Definition 2** (ZK Proof)**.** A zero-knowledge proof system for a language $\mathcal{L}$ is an interactive proof system $\langle P, V \rangle$ that satisfies completeness and soundness, and additionally:

- *Zero-Knowledge*: $\exists$ a simulator $\texttt{Sim}$ such that $\forall$ (possibly malicious) $V^*$ and $\forall x \in \mathcal{L}$:

$$\texttt{View}[\langle P, V^* \rangle(x)] \approx^c \texttt{Sim}^{V^*}(x).$$

The notation $\texttt{Sim}^{V^*}$ means that the simulator can run the verifier $V^*$ as a subroutine.

Here, we consider that soundness and zero-knowledge are computational properties (i.e., they hold against any PPT adversary). A proof with computational soundness is also called an *argument*. We can also define *statistical* notions of soundness and zero-knowledge, although we won't be using these in this class.

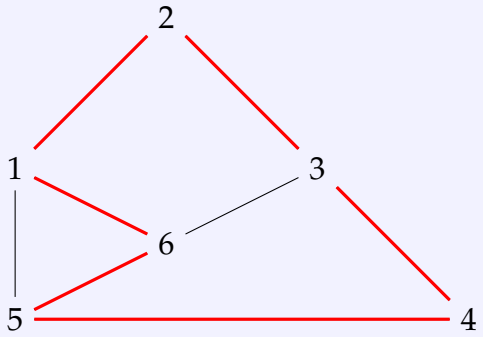
## 2.2 A ZK proof for Graph Hamiltonicity

To familiarize ourselves with these new definitions, let's look at a classical example of a ZK-proof for Graph Hamiltonicity due to Blum [Blu86].

This is an interesting example for a few reasons:

- The decision version of the problem is NP-complete. So if we have a ZK proof for Graph Hamiltonicity, we can use it to construct a ZK proof for any language in NP [GMW91].
- It is a fairly simple protocol, which is actually an instance of a more general class of protocols known as *Sigma protocols* which we will see later.

**Definition 3** (Graph Hamiltonicity)**.** Given an (undirected) graph $G = (V, E)$, the graph has a hamiltonian cycle if there exists a cycle that visits all vertices exactly once.




[1] A simulator is just an (efficient) algorithm that we define for our security proof. This is not an actual algorithm that we use in practice.

So our language is $\mathcal{L} = \{G \mid G \text{ has a hamiltonian cycle}\}$.

**The protocol.**

| $P(G, \textbf{Ham-Cycle})$ | | $V(G)$ |
|---|---|---|
| pick a random permutation $\Pi$ of the $n$ vertices; For $1 \le i \le j \le n$, let $B_{ij} = 1$ if $(\Pi(i), \Pi(j)) \in E$ and $B_{ij} = 0$ otherwise; | | |
| | $\xrightarrow[\texttt{Commit}(\Pi)]{\texttt{Commit}(B_{11}), \ldots, \texttt{Commit}(B_{nn})}$ | |
| | | $c \leftarrow^{\$} \{0, 1\}$ |
| | $\xleftarrow{c}$ | |
| If $c = 0$, open all commitments Else, open all commitments $B_{ij}$ where $(\Pi(i), \Pi(j))$ is in the Hamiltonian cycle. | | |
| | $\xrightarrow{\text{Openings}}$ | |
| | | Verify commitments if $c = 0$, check that the committed graph is isomorphic to $G$ Else, check that a cycle was opened. |

**Theorem 1.** Assume the existence of a commitment scheme with perfect binding and computational hiding. Then the protocol above is a ZK proof for Graph Hamiltonicity with perfect completeness, soundness error at most $\frac{1}{2}$, and computational zero-knowledge.

By running the protocol $O(\lambda)$ times, we can reduce the soundness error to $\mathsf{negl}(\lambda)$.

*Proof.*

- *Completeness:* If the prover knows a Hamiltonian cycle, then it can always open the correct commitments and the verifier will accept.
- *Soundness:* If the graph does not have a Hamiltonian cycle, then the prover must commit in step 1 to a graph that is either not isomorphic to $G$ or does not have a Hamiltonian cycle. Since the commitment scheme is perfectly binding,[2] the prover cannot open the commitments in any other way, and thus gets "caught" with probability at least $\frac{1}{2}$.

[2] This is important if we want soundness to hold even against unbounded provers.

- *Zero-knowledge:* We construct a simulator $\texttt{Sim}$ that can simulate the verifier's view given just the statement $x$. The simulator works as follows:

$\texttt{Sim}^{V^*}(G)$

---

Guess $\hat{c} \leftarrow^{\$} \{0,1\}$
  If $\hat{c} = 0$, commit to a random permutation $\Pi$ of $G$
  If $\hat{c} = 1$, commit to a complete graph
Send the commitments to the verifier who returns $c$
If $\hat{c} \neq c$, abort and restart
Else, open the commitments as requested by the verifier
Output the view $(G, \texttt{Commit}, c, \texttt{Open})$

Note that the simulator's first message $\texttt{Commit}$ is computationally indistinguishable from the prover's first message, as the commitments are computationally hiding.

The simulator's second message is statistically indistinguishable from the prover's second message: if $c = 0$, the simulator does exactly what the prover does, and if $c = 1$, the simulator opens all commitments of some arbitrary cycle, which is the image of the Hamiltonian cycle under *some* permutation $\Pi$.

Finally, we need to argue that the simulator is efficient. Because the commitments in the first message are computationally hiding, the verifier cannot guess $\hat{c}$ with non-negligible advantage, and thus $\Pr[\hat{c} \neq c] \leq 1/2 - \mathsf{negl}(\lambda)$. □

**A note on amplifying soundness.** Remember that we previously said that we could always amplify soundness and correctness by running the protocol multiple times. But once we add zero-knowledge, this requires more care!

There are two standard ways to compose interactive proofs: in parallel and in sequence. In parallel means that we run $\lambda$ instances of the protocol where we have $\lambda$ provers send their first message to the verifiers, then the verifiers send their first message to the provers, and so on. In sequence means that we run the protocol $\lambda$ times one-after-the-other.

Both strategies are equivalent if we only care about completeness and soundness. Composing in parallel is nicer because it doesn't increase the number of rounds of interaction.

Yet, once we add zero-knowledge, we need to be careful that we still can build an efficient simulator! It turns out that composing ZK-proofs in parallel is remarkably tricky because of this. We'll discuss this more in the exercises.

# 3 Proofs of Knowledge

Let's take a closer look at the soundness property of ZK proofs. It essentially says that if the verifier accepts some statement $x$, then with high probability $x \in \mathcal{L}$. But this doesn't necessarily imply that the prover "knows" the witness $w$ for $x$, which is a property we would like to have in many of the applications we listed above.

Consider an identification protocol as an example. Suppose a server holds a value $X = g^{\alpha}$, where $g$ is a generator of a cyclic group $\mathbb{G}$, and the client wants to authenticate by proving to the server that they know $\alpha$ without revealing it (this is Schnorr's identification protocol, which we will see in more detail in the next exercise session). A zero-knowledge proof for

the language $\mathcal{L} = \{X \in \mathbb{G} : \exists \alpha \text{ s.t. } X = g^\alpha\}$ is trivial: all elements of $\mathbb{G}$ are in $\mathcal{L}$ since $g$ is a generator. What we want is something stronger: we want the prover to prove that they "know" the witness $\alpha$.

What should it mean for some algorithm $P$ to "know" a witness $w$? Should that mean that $w$ appears somewhere in the prover's memory? Or what if $w$ is encoded in some special way? We'll get around these issues by defining knowledge in a more abstract way. Essentially, if we can recover the knowledge through ... **torture!**

We say that a prover $P$ knows a witness $w$ for $x$ if there exists an efficient algorithm $\texttt{Ext}$, called the *extractor*, that can recover $w$ by interacting with a successful prover $P$ in a special way. In particular, the extractor $\texttt{Ext}$ cannot simply interact with $P$ in the same way as the verifier, as zero-knowledge implies that such an interaction leaks nothing about $w$. The extra "power" we'll give to the extractor is that it can *rewind* the prover: that is, it can run the prover for a while, save the prover's state, and then restart the prover from that state at a later stage.

We'll now define Proofs of Knowledge (PoKs) in a more formal way, for $\texttt{NP}$ languages. Recall that such a language $\mathcal{L}$ has an efficient deterministic verifier $M$ such that $M(x, w) = 1$.

**Definition 4** (Proof of Knowledge)**.** A proof system $\langle P, V \rangle$ for an $\texttt{NP}$ language $\mathcal{L}$ is a proof of knowledge with knowledge error $\varepsilon$ if there exists an efficient extractor $\texttt{Ext}$ such that for all $x$ and all provers $P^*$

$$\Pr\left[M(x, w) = 1 : w \leftarrow \texttt{Ext}^{P^*}(x)\right] \geq \Pr\left[\langle P^*, V\rangle(x) = 1\right] - \varepsilon\,.$$

Note that the proof of knowledge property subsumes the soundness property: if a witness is extractable, then clearly $x \in \mathcal{L}$. So if the knowledge error is negligible, then there is only a negligible probability that a prover convinces a verifier on a false statement.

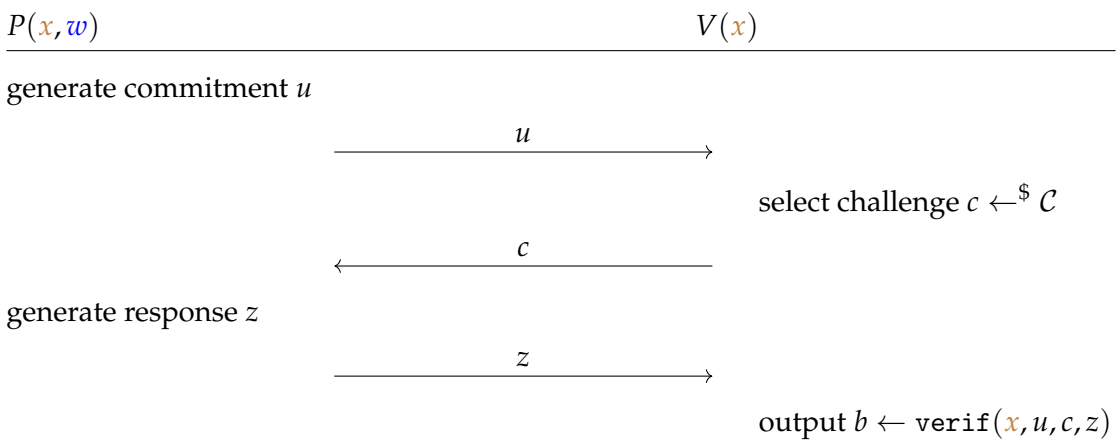
**Our protocol for Graph Hamiltonicity is a PoK.** We only give an informal argument here. Consider a prover $P^*$ that succeeds in convincing the verifier that $G$ has a Hamiltonian cycle with probability 1. Then we can extract a Hamiltonian cycle from $P^*$ with probability 1 as well.

Our extractor $\texttt{Ext}$ simply runs $P^*$ until it gets the first message of commitments. It then sends challenge $c = 0$ to $P^*$. Since $P^*$ succeeds with probability 1, it will reveal a correct permutation $\Pi$ of the graph $G$. Now the extractor rewinds $P^*$ and sends a different challenge $c = 1$ to $P^*$. Since $P^*$ succeeds with probability 1, it will reveal a Hamiltonian cycle in the permuted graph $\Pi(G)$. Given both of these, the extractor can reconstruct the original Hamiltonian cycle in $G$.

# 4 Sigma Protocols

Our protocol for Graph Hamiltonicity (and Schnorr's identification protocol which you'll see in the exercises) are examples of general class of protocols known as *Sigma protocols*.

A Sigma protocol is a three-round protocol:




where the verifier's final decision is made by a *deterministic* function `verif`.

**Definition 5** (Sigma protocol)**.** A Sigma protocol is a three-round protocol $\langle P, V \rangle$ of the form above, such that the following properties hold:

- *Perfect completeness:* For all $x \in \mathcal{L}$ and all $w$, $\Pr[\langle P, V \rangle(x) = 1] = 1$.
- *"Special" soundness:* There exists an efficient algorithm `Ext` such that given two accepting transcripts $(u, c, z)$ and $(u, c', z')$ for $x$, where $c \neq c'$, it outputs a witness $w$ such that $M(x, w) = 1$.
- *Honest verifier zero-knowledge:* There exists an efficient simulator `Sim` that on input $(x, c)$ where $x \in \mathcal{L}$ is a statement and $c \in \mathcal{C}$ is a challenge, outputs $(u, c, z)$ that is computationally indistinguishable from the verifier's view in an *honest* execution of the protocol.

**Theorem 2.** A Sigma protocol for an `NP` language $\mathcal{L}$ gives an interactive proof for that language with soundness error at most $\frac{1}{|\mathcal{C}|}$.

*Proof.*

- Completeness is immediate.
- For soundness, we claim that if $x \notin \mathcal{L}$, then for any commitment $u$, there exists at most one "good" challenge $c$ that will cause the verifier to accept. Indeed, if there were two such challenges $c$ and $c'$, then running the extractor on $(u, c, z)$ and $(u, c', z')$ would output a valid witness $w$ which is impossible since $x \notin \mathcal{L}$. Therefore, the probability that the verifier accepts is at most $\frac{1}{|\mathcal{C}|}$.

□

**Composition of Sigma protocols.** For Sigma protocols, it turns out that both composition in parallel and in sequence preserve all the security properties (see your exercises for a proof)! The main reason for this is that the zero-knowledge property we consider for Sigma protocols is only *honest-verifier* zero-knowledge.

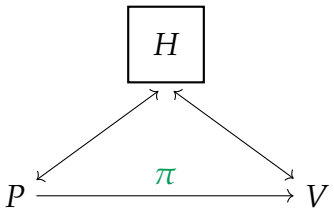
# 5 Non-interactive Zero-Knowledge

Sigma protocols are nice because they give us ZK-proofs with just 3 rounds. Could we do better, and get a ZK-proof in just a single message? This is what we call a *non-interactive zero-knowledge proof* [BFM88].

This seems impossible (except for simple languages) at a first glance. If the proof consists of a single message $\pi$, then by the ZK property we know that for all $x \in \mathcal{L}$ there exists an efficient simulator $\texttt{Sim}$ that can simulate a proof given just the statement $x$. By soundness, we know that for an $x \notin \mathcal{L}$, the output of $\texttt{Sim}(x)$ should be rejected by the verifier (with high probability). And so $V(\texttt{Sim}(x))$ is an efficient (possibly randomized) algorithm for deciding the language $\mathcal{L}$ (which means that $\mathcal{L} \in$ BPP).

The problem, intuitively, is that the simulator and the honest prover have the exact same "power": anything that the prover does, the simulator should be able to do as well because of zero-knowledge.

Surprisingly, we can get around this by changing our computation model! We'll show how to build NIZKs from Sigma protocols in the Random Oracle Model.

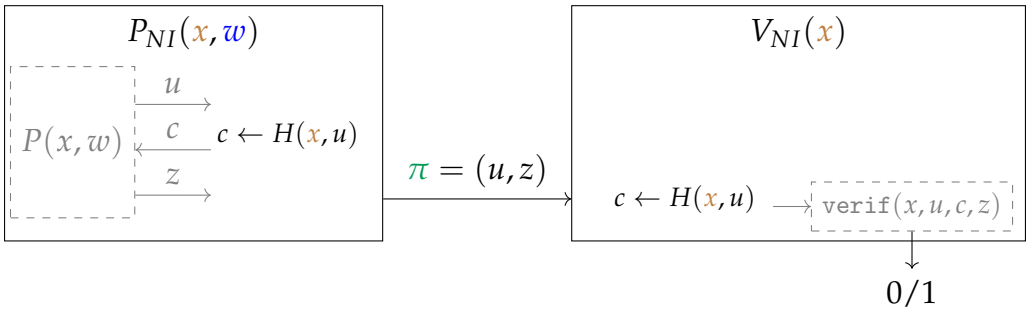



The extra power we'll give to the simulator is that it can *program* the random oracle: that is, it can choose how to respond to the verifier's RO queries when simulating the verifier's view, as long as the RO responses are indistinguishable from random.

**The Fiat-Shamir transform.** The Fiat-Shamir transform [FS86] is a simple heuristic that allows us to turn a Sigma protocol into a NIZK in the random oracle model.

The key insight is that in a Sigma protocol, all the (honest) verifier does is sample some uniformly random challenge and sends it to the prover, and the verifier's final decision is deterministic. Such a protocol is also called a *public coin* protocol.

The idea of Fiat and Shamir is to compute the challenge as a hash of prior messages from the prover $c = H(x, u)$. Then, the prover can simulate the entire interaction locally, and send a single proof $\pi = \{u, z\}$ to the verifier. The verifier recomputes the challenge $c$ using the hash function, and checks that the message transcript is accepting.



**Security of the Fiat-Shamir Transform (informal).**

- Completeness: this is easy to see.
- Zero-knowledge: The simulator picks a random challenge $c \in \mathcal{C}$ and then runs the simulator from the Sigma protocol on $(x, c)$ to get $(u, c, z)$. It then programs the RO to return $c$ when queried with $(x, u)$, and outputs $\pi = (u, z)$.

  Note that here we don't need to make any assumption about the verifier being honest or not, since the protocol is non-interactive. It was also enough for the Sigma protocol to be honest-verifier zero-knowledge, since the honest prover now makes sure to pick a random challenge using the RO.
- Proof of knowledge: we can build an extractor `Ext` that runs the prover until it queries the RO with $(x, u)$. It then programs the RO to return some random $c \in \mathcal{C}$ and obtains a transcript $(u, c, z)$. It then rewinds the prover to the point where it queried the RO, and now returns a different challenge $c' \in \mathcal{C}$ and obtains a transcript $(u, c', z')$. It then runs the Sigma protocol extractor on $(u, c, z)$ and $(u, c', z')$ to get the witness.

**A note on soundness.** Recall that when defining interactive zero-knowledge above, we allowed soundness (and completeness) to fail with some constant probability. This was fine as we can always repeat the interaction a constant number of times to make the probability of failure negligible. But when we deal with non-interactive proofs, we don't get to repeat the interaction as there isn't any! And so importantly, when defining non-interactive proofs, we require the soundness and completeness error to be negligible.

**NIZKs in practice: the Fiat-Shamir *heuristic*.** In practice, we of course don't have a random oracle. And so we replace it by a concrete cryptographic hash function $h$. As with other schemes in the random oracle model, the security of the NIZK then becomes heuristic—based on the assumption that an adversary cannot somehow exploit any special structure in the hash function that distinguishes it from a truly random function.

# References


[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 103–112. Association for Computing Machinery (ACM), 1988.

[Blu86] Manuel Blum. How to prove a theorem so no one else can claim it. In *Proceedings of the International Congress of Mathematicians*, volume 1, page 2. Citeseer, 1986.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194. Springer, 1986.

[GMR85] Shafi Goldwasser, Silvio Micali, and Chales Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, STOC '85, pages 291–304. Association for Computing Machinery (ACM), 1985.

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(3):690–728, 1991.

[Sha92] Adi Shamir. IP=PSPACE. *Journal of the ACM (JACM)*, 39(4):869–877, 1992.