Privacy Enhancing Technologies
Lecture 3 – Zero-Knowledge Proofs

Florian Tramèr

AGENDA

1. Recap on MPC
2. Defining ZK
3. SNARGs
4. Building a SNARG

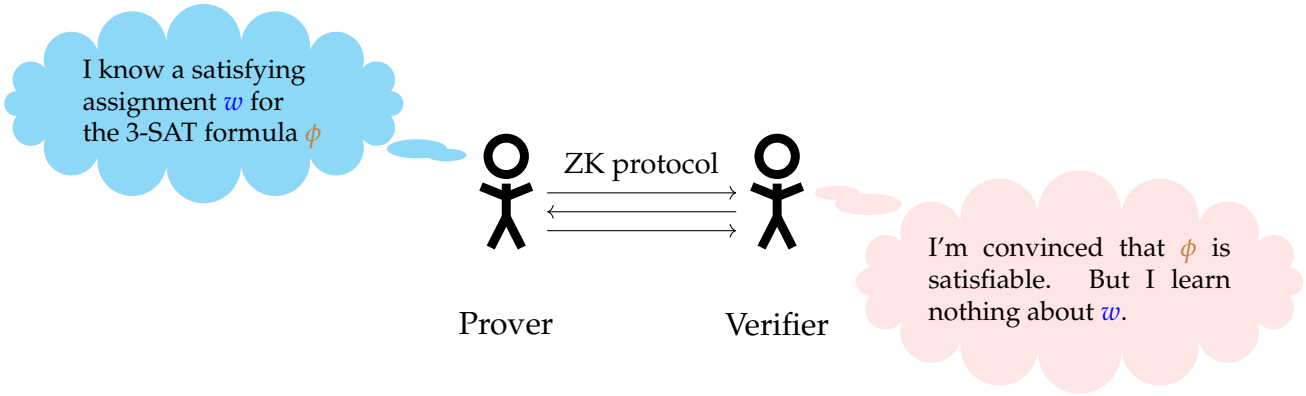
1 Recap on MPC

In the last lecture, we showed how to build protocols that allow n parties to compute *any* NP function between them while learning only the output. So are we done with this class? What else could we possibly want? Well, it turns out that MPC is not the end of the story. First, as we'll see in the second half of the class, there are stronger forms of "privacy" that MPC does not provide (in particular, any kind of privacy leakage that stems from the function's output itself). Second, general-purpose MPC is very expensive (in computation and communication), especially if we want malicious security. We will see that for specific functions, or in the special case of $n = 2$ parties, we can build much more efficient protocols.

2 Zero-Knowledge Proof Systems

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] is a protocol that allows one party (the *prover*) to convince some other party (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 ϕ 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.

There are a number of other classes that cover the theory of interactive proofs and zero-knowledge. These are the classes you should take if you want to learn about fundamental theory results such as IP=PSPACE [Sha92] or how to generically build ZK proofs for all of NP [GMW91].

Here, we will instead focus on giving you a flavor of what modern, practical ZK proofs looks like. These modern systems have been developed and deployed primarily in the context of blockchain applications, but the proof systems themselves are universal and can be used in many different contexts.

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.

- *Maliciously-secure MPC:* One of the first applications of ZK was to transform MPC protocols with semi-honest security into protocols with malicious security. The idea is quite simple: since the protocol is secure when parties follow the protocol steps, whenever a party generates a new message, it proves to all other parties that m is the output of calling the protocol on the party's current (secret) view. As you will see in your homework, however, this is not how modern maliciously-secure MPC protocols work as this approach is too expensive in practice.

In general, we can view the proofs of such statements as expressions about *arithmetic circuits* (the reason we use arithmetic circuits, as in the last lecture on MPC, is because they are a "crypto-friendly" way of expressing arbitrary functions).

Compared to last lecture, we will make it explicit that the circuit has two inputs: a public input x (known to the prover and the verifier) and a private witness w (known to the prover). We say an arithmetic circuit $\mathcal{C} : \mathbb{F}^m \times \mathbb{F}^n \to \mathbb{F}$ is *satisfiable* on an input $x \in \mathbb{F}^m$ if there exists an assignment $w \in \mathbb{F}^n$ such that $\mathcal{C}(x, w) = 0$.

Definition of ZK proofs. If the prover has a satisfying witness w, they can easily convince the verifier that $\mathcal{C}$ is satisfiable on x by just sending them w. The verifier can then check that $\mathcal{C}(x, w) = 0$ and accept the proof. By definition, all NP languages have such a proof. But note that this proof completely reveals the satisfying assignment to the other party. That's where ZK comes in to play.

So what would it means for a proof system to reveal "nothing more" than the fact that $\mathcal{C}$ is satisfiable on x? We actually saw a very similar notion when we discussed the definition of secure MPC: we want to learn the output of some function (here $\mathcal{C}(x, w)$) without the parties learning anything about each other's inputs (here x is public and w is private). Recall that we defined security by introducing a *simulator*, that can simulate the adversary's view without knowing the private inputs. So here, this will mean that we can simulate the verifier's view without knowing the witness w.

Definition 1 (ZK Proof)**.** An zero-knowledge proof system for a family of circuits $\mathbb{C}$ is a protocol between a prover P and verifier V on some circuit $\mathcal{C} \in \mathbb{C}$ and input $x \in \mathbb{F}^m$, denoted as $\langle P, V \rangle(\mathcal{C}, x)$, that satisfies the following properties:

- *Completeness*: $\forall \mathcal{C} \in \mathbb{C}, x \in \mathbb{F}^m$ such that $\mathcal{C}$ is satisfiable on x,

$$\Pr[\langle P, V \rangle(\mathcal{C}, x) = \text{``accept''}] \geq \frac{2}{3}.$$

- *Soundness*: $\forall \mathcal{C} \in \mathbb{C}, x \in \mathbb{F}^m$ such that $\mathcal{C}$ s non-satisfiable on x, for all (malicious) provers P^*,

$$\Pr[\langle P^*, V \rangle(\mathcal{C}, x) = \text{``accept''}] \leq \frac{1}{3}.$$

- *Zero-Knowledge*: Informally, V learns nothing more than the fact that $\mathcal{C}$ is satisfiable. Formally, $\forall$ malicious V^*, $\exists$ a simulator $\mathtt{Sim}_{V^*}$ such that $\forall \mathcal{C} \in \mathbb{C}, x \in \mathbb{F}^m$ such that $\mathcal{C}$ is satisfiable on x:

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

The probabilities $\frac{2}{3}$ and $\frac{1}{3}$ are mostly arbitrary, as we can always *amplify* the completeness and

soundness success probabilities to $1 - \mathsf{negl}(\lambda)$ by repeating the protocol $O(\lambda)$ times.

Here, 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.

Again, note here that there are two different adversaries at play. For soundness, the adversary is the prover P^* who tries to convince the verifier that $\mathcal{C}$ is satisfiable on x when it is not. For zero-knowledge, the adversary is the verifier V^* who tries to learn something about the witness w from the proof.

Proofs of knowledge. You may have noted that in some of the example applications we listed above, the prover convinces the verifier of a "fact" (e.g., "the computation $f(x)$ produces the output y"), while in other cases, the prover convinces the verifier that they have "knowledge" of some secret (e.g., a password p).

The soundness property we defined above covers the former case (i.e., the verifier is convinced of the fact that $\mathcal{C}$ is satisfiable on x). The second property is actually a strictly stronger notion of soundness, known as a *Proof of Knowledge* (PoK). To see the difference, consider our example of password verification. Suppose the prover proves to you the fact that there *exists* a password p such that $H(p)$ matches the stored hash. By definition, such a password must exist and so this statement is vacuously true. What we really want is for the prover to prove that they possess this password.

We won't cover proofs of knowledge explicitly in this class. But it's good to remember this distinction. Most proof systems used in practice today are actually proofs of knowledge. As with zero-knowledge, the definition of a proof-of-knowledge is somewhat subtle, and involves defining a special algorithm (called an *extractor*) that can recover a valid witness w by interacting with a successful prover P in a special way.

3 SNARGs

In many of the applications listed above, a standard ZK proof system is impractical and we actually want further properties from our proof system.

For example, suppose you want a server to compute the function $\mathrm{SHA}^{1'000'000}(x)$ for you (i.e., applying SHA-256 a million times to an input x). One way for the server to "prove" that the answer is y is just to send you all $1'000'000$ hashes and let the verifier check each one (or just send only y and let the verifier re-run the entire computation...) So we need a proof system where the proof is *succinct*, i.e., much faster to verify than to generate.

In the cryptocurrency setting, we also do not want the prover (the party making a transaction) to have to *interact* with every other party in the system to convince them the transaction is valid. Here, we want a *non-interactive proof* [BFM88], i.e., a proof string that can just be sent to anyone.

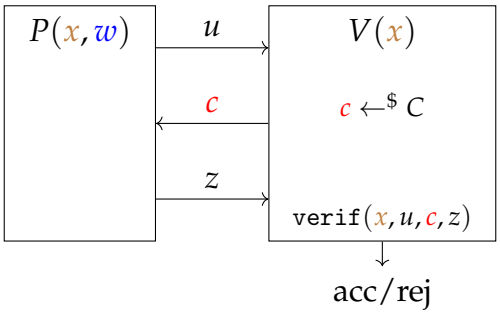
Proofs that satisfy these two properties are called *SNARGs* (Succinct Non-Interactive Arguments) [BCCT12]. Let's talk about how we could achieve these two properties in isolation. (notice that for now, we've left zero-knowledge out of the picture. It turns out that for modern SNARGs, zero-knowledge is typically the easy part. Once we have a proof system that is non-interactive and succinct, getting zero-knowledge as well can often be done by appropriately randomizing the prover).

3.1 Non-Interactive Zero-knowledge Proofs (NIZKs)

We tend to think of mathematical proofs as non-interactive (e.g., someone writes a proof in a paper, and you read it and verify its correctness). Naturally, all NP statements have non-interactive proofs (*why?*).

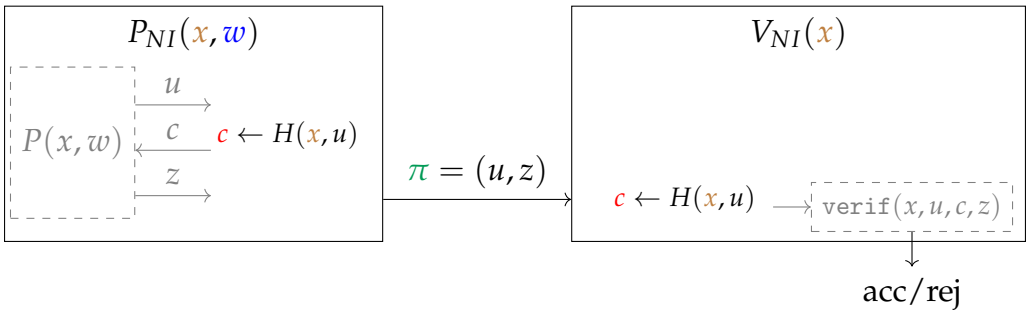
The notion of interaction between a prover and a verifier was studied in order to achieve additional properties not covered by NP, such as zero-knowledge, or proofs for statements outside of NP.

It turns out that in some (common) scenarios, an interactive zero-knowledge proof can be turned non-interactive through a very simple heuristic, the *Fiat-Shamir transform* [FS86]. This heuristic is typically applied to interactive proof systems of the following form, called *Sigma protocols*:



That is, all the verifier does during the protocol is sample some uniformly random *challenge* c from a finite set C and sends it to the prover. The final accept/reject decision is a *deterministic* function of the verifier's view. Such a protocol is also called a *public coin* protocol.

To make the protocol non-interactive, we could have the prover pick the random challenge c themselves. But we have to be careful to not give the prover too much power, or we lose soundness. 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.



To prove security of this transform, we have to make some strong assumptions about the hash function H. The simplest way is to assume that H is a *truly random function*, i.e., to prove security in the random-oracle model. Of course, in practice we cannot use a truly random function as this would be computationally intractable. And so we replace the random oracle by a concrete hash function (e.g., SHA-256). We cannot prove that the Fiat-Shamir transform is secure when instantiated with SHA-256 (this is why it's called a heuristic), but we know that an attack would have to exploit some surprising structure in the hash function.

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.

3.2 Succinct Proofs

There are different ways to define succinctness. The one we'll consider here is as follows:

- *Short proofs:* The size of the proof π is *poly-logarithmic* in the size of the circuit $\mathcal{C}$ (the number of gates in $\mathcal{C}$). In other words, the size of the proof is at most $O(\lambda, \text{polylog}(|\mathcal{C}|))$, where λ is the security parameter.
- *Efficient verification:* The time required to check the proof is $O(\lambda, |x|, \text{polylog}(|\mathcal{C}|))$, i.e., linear in the length of the public input x and poly-logarithmic in the size of the circuit $\mathcal{C}$ (note that the linear dependence on $|x|$ is unavoidable as the verifier has to be able to read the statement to be proven).

In contrast to non-interactivity, succinctness is a highly non-trivial property even without considering zero-knowledge. Indeed, a typical proof for an NP statement is *not* succinct: e.g., to prove that a circuit $\mathcal{C}$ is satisfiable, the proof would be the whole assignment and the verification time is proportional to the circuit size. But it turns out that this is possible by using cryptography! (under some strong assumptions)

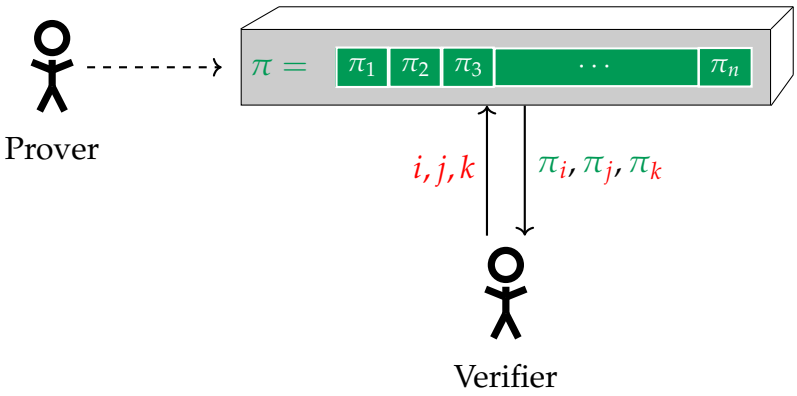
4 Blueprint for a SNARG

So how do we construct SNARGs? Modern constructions that aim to maximize efficiency are quite complicated. We will give an overview of a popular approach next week, but we won't be able to cover all the details. You'll build an older and conceptually simpler (but incredibly inefficient) scheme in your homework!

All these schemes actually have a common *blueprint*. We start by building a proof system where the security is *information theoretic* (i.e., we don't need any cryptographic assumptions). Because standard NP proofs are not succinct, this requires working in some (often weird) model that constrains how the verifier and prover interact (you can think of this a bit like the random oracle model, where we assume access to some imaginary object that helps us prove security). So we first build a proof system that is unconditionally secure in this weird computational model.

Then, cryptography comes into mix. We use cryptographic tools to force the prover to adhere to the constraints, without having the ability to cheat. Then we further apply Fiat-Shamir to make the whole thing non-interactive. And we're done! Sounds easy, right...

The box game. Let's take a little detour first, and consider the following problem: The verifier wants to test whether some witness w satisfies a circuit $\mathcal{C}$ on input x. The prover puts their proof π in a special "box", that lets the verifier see the value of π at only three randomly chosen bits. The verifier can then decide whether to accept or reject the proof.



Is this possible?

At first glance, this seems impossible. You're only allowed to see three bits of the proof! That's a bit like saying that I can grade your homework by only reading three random lines of your submission. And yet, it turns out that this is possible, if we encode the proof in a special way!

This follows from one of the most celebrated results in theoretical computer science, the PCP theorem [ALM+98] (PCP stands for Probabilistically Checkable Proofs). The PCP theorem says, informally, that you can create a proof π for any NP statement (where the length of π is polynomial in the size of the instance), so that you can verify the proof by reading only 3 random bits of it [Hås01]! The proof has perfect completeness, and soundness error of at most $1/2$ (i.e., incorrect proofs are accepted with probability at most $1/2$).

There's a flavor of succinctness that is apparent in the PCP theorem. For a proof of size n, the verifier only needs to sample $O(\log n)$ random bits, and receives $O(1)$ bits in return, from which they can decide whether to accept the proof.

So note that we now have an information-theoretic proof system that is succinct, but where the prover and verifier interact in some special way. In particular, the prover has to encode the proof once and for all and cannot reply "adaptively" to the verifier's queries. This concludes the first step of our blueprint. Now the second step is to use cryptography to actually instantiate this "box".

From PCPs to SNARGs. Note that the PCP theorem does not give us a SNARG just yet. In practice, this "box" doesn't exist, so we still have to figure out how the prover and verifier should exchange the proof π.

One option would be for the prover to send the verifier the entire proof, and the verifier to sample three random bits from it. But this would defeat the point of succinctness, as the PCP proof is as long (actually significantly longer) than the original proof.

Another way would be for the verifier to send the prover the three random indices, but then we somehow need to ensure that the prover doesn't just return arbitrary values. In particular, note that there are only $\binom{n}{3}$ possible choices for the three indices, so the prover could just adaptively choose the values of the proof to satisfy the verifier's queries. This is where cryptography comes in.

So how do we instantiate such a box in cryptography? This should remind you of commitment schemes! But to get succinctness, we'll need a special form of commitment scheme called a *vector commitment scheme*. This allows the prover to send a *short* commitment to an entire vector of values $x = (x_1, x_2, \ldots, x_n)$, and later let the verifier *open* the commitment at a

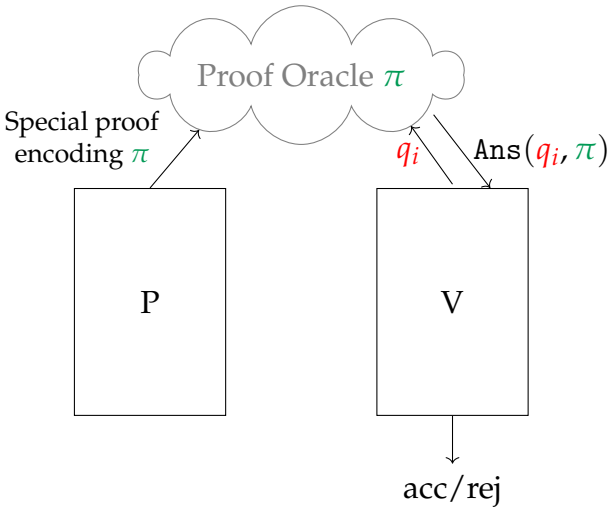
specific index i to obtain x_i. You'll show how to construct such a scheme in your homework, and how to finalize the SNARG construction using it.

4.1 Abstracting the Box Game: Interactive Oracle Proofs

This SNARG construction used a special type of "box", where the verifier interacts with the proof through queries at individual indices. We can generalize this idea to other types of "boxes", that constrain the form that the verifier's interaction with the proof can take.

Formally, we refer to the box as an *oracle*, which you can think of as trusted third party that takes as input a proof string from the prover, and allows the verifier to ask only specific queries to the proof. Such proof systems are called *Interactive Oracle Proofs* (IOPs) [BSCS16].

Again, note that IOPs are information-theoretic proofs. That is, we first show that in a special model where this oracle exists, the proof system is sound and complete. Then, we use cryptography (typically a special form of commitment scheme) to instantiate the oracle.



Beyond PCPs, other types of commonly used IOPs include:

- *Linear IOPs*, where the proof is encoded as a *vector* π in some field $\mathbb{F}$, and the verifier asks for a constant number of *linear combinations* of the proof values (i.e., $< q, \pi >= \sum_{i \in [n]} q_i \cdot \pi_i$ for some vector $q \in \mathbb{F}^n$).

  We will get back to linear IOPs in a few lectures, when we discuss protocols for aggregating private statistics.

- *Polynomial IOPs*, where the proof is encoded as a *polynomial* $\pi(x)$ over some field $\mathbb{F}$, and the verifier asks for *polnomial evaluations* at a constant number of points $z \in \mathbb{F}$.

  We will see Polynomial IOPs in more detail next week, when we describe a modern practical SNARG.

References

[ALM+98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM (JACM)*, 45(3):501–555, 1998.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowl-

edge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 326–349. Association for Computing Machinery (ACM), 2012.

[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, page 103–112. Association for Computing Machinery (ACM), 1988.

[BSCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *Theory of Cryptography: 14th International Conference, TCC 2016-B, Beijing, China, October 31-November 3, 2016, Proceedings, Part II 14*, pages 31–60. Springer, 2016.

[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.

[Hås01] Johan Håstad. Some optimal inapproximability results. *Journal of the ACM (JACM)*, 48(4):798–859, 2001.

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