Extending Zero-Knowledge PCPs Beyond NP

A Thesis
Presented to
The Established Interdisciplinary Committee for
Mathematics and Computer Science
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Patrick Norton

April 30, 2025

Contents

Chapter 1: Introduction **11**
1.1 What is a computational problem? 11
1.2 Turing machines 12
1.3 Relativization and algebrization 13
1.4 Computational proof systems 14
1.5 Zero-knowledge 15
1.6 The PCP theorem 16

Chapter 2: Preliminaries **17**
2.1 Turing machines 17
2.2 Complexity classes 19
2.2.1 Time complexity 21
2.2.2 Space complexity 25
2.2.3 Completeness 26
2.2.4 Randomized complexity 28
2.2.5 Impagliazzo's five worlds 29
2.3 Polynomials 30
2.4 Statistics 37
2.5 Error-correcting codes 39

Chapter 3: Relativization and algebrization **43**
3.1 Defining relativization 45
3.2 Query complexity 46
3.3 Relativization of P vs. NP 47
3.4 Consequences on proof techniques 50
3.5 Algebrization 51
3.6 Algebraic query complexity 52
3.7 Algebrization of P vs. NP 53

Chapter 4: Interactive proof systems **59**
4.1 Interactive Turing machines 60
4.2 Single-prover systems 62
4.3 Multi-prover systems 65
4.4 Zero-knowledge proofs 68
4.4.1 Commitment schemes 72

4.5 Probabilistically-checkable proofs . . . 73
4.5.1 PCPs of proximity . . . 77
4.5.2 Robustness . . . 78
4.6 Zero-knowledge probabilistically-checkable proofs . . . 81
4.7 Interactive probabilistically-checkable proofs . . . 81
4.8 Zero-knowledge IPCPs . . . 83

Chapter 5: The PCP theorem . . . 85
5.1 Algebraic circuits . . . 85
5.2 The composition theorem . . . 89
5.3 Alphabet reduction . . . 90
5.4 Robust total-degree test . . . 91
5.5 PCPPs for polynomial summation . . . 92
5.6 Properties of O3SAT . . . 96
5.7 A PCP for NP . . . 97

Chapter 6: Low-degree zero-knowledge IPCPs . . . 103
6.1 AQC of polynomial summation . . . 103
6.2 The sumcheck problem . . . 106
6.2.1 A non-zero-knowledge sumcheck protocol . . . 107
6.2.2 Making the sumcheck protocol zero-knowledge . . . 108
6.3 Extending the sumcheck algorithm to NEXP . . . 114
6.4 Zero-knowledge MIP$*$ for NEXP . . . 114

Chapter 7: A zero-knowledge PCP theorem . . . 117
7.1 Locally-computable proofs . . . 117
7.2 Zero-knowledge proof composition . . . 119
7.3 Zero-knowledge alphabet reduction . . . 121
7.4 Zero-knowledge PCPPs for Sum . . . 122
7.5 A zero-knowledge PCP for NP and NEXP . . . 124
7.5.1 Reducing query complexity to constant . . . 128

Appendix A: More on extension polynomials . . . 131
A.1 A proof of Equation (2.6) . . . 131
A.2 Algebra behind Equation (2.8) . . . 132

Bibliography . . . 133

Index . . . 137

List of Algorithms

2.1 A NEXP-time algorithm for determining O3SAT 25
2.2 An algorithm to reduce A to B 27

3.1 An algorithm for constructing B 49
3.2 Determiner for $\tilde{L}$ 54
3.3 An algorithm for constructing $\tilde{A}$ 56

4.1 An interactive proof for the language GNI 64
4.2 A 2-prover MIP simulating a k-prover MIP 67
4.3 A perfect zero-knowledge IP for GI 70
4.4 A simulator for Algorithm 4.3 70
4.5 A bit-commitment scheme based on a one-way function f 74
4.6 A PCP for GNI 76
4.7 The IPCP protocol 82

5.1 A composed PCP [8, Theorem 2.7] 90
5.2 A boolean reduction of a PCP [17, Construction 3.6] 91
5.3 A robust low-degree test [27, Prop. 5.7] 92
5.4 A robust PCPP for Sum [17, Construction 4.3] 94
5.5 A PZK-PCP for O3SAT [17, Construction 6.4] 99

6.1 The standard sumcheck protocol [26, Thm. 1] 107
6.2 Strong zero-knowledge sumcheck [10, Construction 3] 109
6.3 An inefficient simulator for Algorithm 6.2 [10, p. 15:33] 113
6.4 A low-degree IPCP for O3SAT [10, p. 15:36] 115
6.5 A simulator for Algorithm 6.4 [10, p. 15.37] 115

7.1 A hybrid simulator for a locally-computable PCP [17, Construction 3.3] 118
7.2 A PZK simulator for a locally-computable PCP [17, Construction 3.4] 118
7.3 An algorithm for π_r from π_0 119
7.4 A zero-knowledge robust PCPP for Sum [17, Construction 5.2] 123
7.5 A PZK-PCP for O3SAT [17, Construction 6.4] 126
7.6 A simulator for Algorithm 7.5 [17, Construction 6.7] 127

Abstract

This expository paper examines the interaction between zero-knowledge proofs and probabilistically-checkable proofs (PCPs). To start, we build up much of the language and tools of modern complexity theory, in particular the variations on interactive proofs and how they can be made zero-knowledge. We also build up much of the algebraic language necessary for working with zero-knowledge proofs and PCPs. Next, we show a proof of the PCP theorem, which relates the power of probabilistically-checkable proofs and the class NP. After that, we unpack several pieces of recent literature on zero-knowledge probabilistically-checkable proofs, including a zero-knowledge version of the aforementioned PCP theorem.

Chapter 1

Introduction

Complexity theory, to wit, is the study of what problems are easy and what problems are hard. More specifically, it is the study of computational problems and how they can be classified according to the amount of effort needed to solve them. While there are many facets of complexity theory, the one we will be focusing on is that of computational proof systems: different mechanisms by which one computer can prove a statement to another computer in a way that removes reasonable doubt. Along with that, we look at proof systems that reveal no potentially-sensitive information, besides the exact statement we care about proving. We show that in several useful cases, this can be done without making the problem substantially harder.

The type of computational proof system we will be working with the most is called a probabilistically-checkable proof. The idea behind these is that instead of reading the entire proof end-to-end, in the way we would for a "standard" mathematical proof, we only check some random portion of it. The question therefore becomes how little of the proof we can check while still reliably getting the right answer, and with the ability to catch out "liar" proofs that purport to prove some incorrect statement. It has been shown that for many problems, one needs to read only a very small portion of the proof in order to still be correct; we also show that this can be done in a way that reveals no potentially-sensitive information. We further show that by combining a probabilistically-checkable proof with a different proof system, we can gain even more power, still while leaking nothing sensitive.

1.1 What is a computational problem?

In this thesis, we focus on *decision problems*: problems where we are given some input text, and then we have to decide either yes or no. We care about these because mathematically, they are the simplest type of problem. Importantly for us, the yes/no answer for a given input never changes, and every input value has some correct answer.

The next layer in setting up a computational problem is to define an *alphabet*. Every input text needs to be written down in some particular form, and it is important to know how it is written ahead of time. The only true restriction on our alphabet is that it must be finite (an infinite amount of symbols would make it really hard to do any useful analysis), but in practice, because we are dealing with computers our alphabet almost always consists of just the symbols 0 and 1.

The second important property of computational problems is that of input length. Of course, when we ask for the answer to a question, our query could be as long as we like. However, in general we expect longer questions to take longer to process, so our computation time is almost always written in terms of the input length. Knowing that, determining the input length is simple—it is just the number of alphabet letters in the query.

1.2 Turing machines

A *Turing machine* is the principal model we use to underpin our computations. A Turing machine is a theoretical box that has access to some arbitrarily-long *tape*[1], a list of blank spaces to which it can read and write data, one space at a time. The computation of the machine proceeds in individual steps—in each step the machine can know what is currently on the tape, and what state the machine is currently in; after it receives this information it may choose to "halt" (i.e., stop running and make a decision), or otherwise it may write a single bit to the tape and then move its view one square left or right.

It may at first seem like this is a rather silly definition of a computer—after all our computers have things like screens and keyboards, and there is definitely not an infinite tape sticking out the side. However, it turns out this is still useful to us! As we discussed in the last section, computational problems do not always line up with what we consider computational tasks in the real world, and just answering yes/no requires neither a screen nor keyboard.

Further objection might be taken to the fact that to access different parts of the tape, we need to slowly move the machine over, step by step. This concern is less easy to intuitively dissuade, but it turns out that at the level of granularity we will care about, this is not actually a concern. Essentially, we only care about classifying problems at a very coarse level, and at some point the differences in our computational model pale in comparison to the differences in the complexity of the problem itself.[2] Since it is indistinguishable for our purposes from a normal computer, we use it because it is mathematically nicer than many other models of computation.

This classical notion of a Turing machine has of course spawned many variants to reflect different ways of interacting in the real world. Perhaps the most famous is the *quantum* Turing machine, which attempts to emulate the computational abilities of a computer that can leverage specific quantum effects. A variant we will be focusing on somewhat in this work is the *interactive* Turing machine, a machine with the ability to pass messages back and forth with another interactive machine. Another variant we will be focusing on is the *oracle* Turing machine, a machine with access to some "oracle" (a function that can answer some hard problem in a single step).

[1]The naming of a tape comes by analogy to a ticker tape or audio tape, not masking tape.

[2]This is no accident—these classes were picked to some extent *because* of their independence of model, not in spite of it.

1.3 Relativization and algebrization

Oracle machines have a particular use in how they interact with the most famous open hypothesis in complexity theory: P vs. NP. In brief, this asks whether any problem that can be verified quickly (which we call NP) can also be solved quickly (which we call P). A good example of this kind of problem is Sudoku: taking a solved board and verifying if it is actually valid does not take that long (all you need to do is go through each row, column, and 3x3 square and check every number appears exactly once), but taking an unsolved one and coming up with a solution is much harder.

Unfortunately, what these oracle machines tell us is that we are not very close to solving P vs. NP. What these have shown us is that entire classes of techniques—in particular *every single* proof technique that had been tried for problems similar to P vs. NP—would not work here.

Even worse for us, this has now happened twice! After the first paper [6] showing this came out (calling its technique *relativization*), mathematicians got to work looking for relativization-proof techniques. Over time they found many, until another paper [1] came out, building on the previous with a new technique called *algebrization*. Similarly to before, this showed that every relativization-proof technique was not algebrization-proof, sending researchers back on a new series of quests for better proof techniques.

The general idea behind relativization is based on observing what problems become easy when we add different oracles to a Turing machine. It is worth noting that no problem ever becomes harder when we add an oracle: a machine can always choose to never talk to an oracle and then it could solve any problem just the same as a non-oracle machine. However, if we give a machine a powerful enough oracle, it turns out that any problem that it could verify quickly would also be solvable quickly. But we can also pick a particularly tricky oracle to show the converse—a machine with that oracle can verify some problems quickly that we are able to prove that it cannot solve quickly. What this all means is that any proof of the P vs. NP problem needs to have its logic break somewhere if we introduce an oracle: if we can replace every instance of "P" with "P with some oracle" (and similarly for NP) and the logic holds up, we have created a contradiction no matter what we do.

Algebrization works similarly to relativization, but with a slightly different model of an oracle. In this case, instead of a normal oracle (which we can think of as a function that returns either a 0 or a 1), we think of our oracle as a large collection of polynomials. Because we can now get back arbitrary numbers from our oracle, instead of just 0 or 1, our oracle gains more power. Again, this leads us to be able to construct oracles similar to before, where under some oracle every problem that is easy to verify is easy to solve, and under some other oracle this is false. This gives us the same issue as with relativization. It turns out that every proof technique devised *after* the discovery of relativization is still vulnerable to algebrization, so once again this reset the proof technique search.

1.4 Computational proof systems

Alongside using computational models to help us determine how to prove statements, we can also look at computers as a model of proof themselves. In this model, we have to shift our thinking a little bit. So far, we have been thinking of our inputs as an arbitrary piece of text, where every piece of text has either “yes” or “no” associated with it. Now, we would like to think of the input as being some *statement*, where the idea is that the response is “yes” if the statement is true, and “no” if the statement is false. More specifically, we will be working to classify all the statements of a given type: for example, all statements of the form “x is even”, where x is replaced with some integer. In this example, we would want our machine to output “yes” when we are given an even number, and “no” when we are not.

Interactive proofs operate with a pair of the interactive Turing machines mentioned earlier, passing messages between them. We call the two machines the “prover” and the “verifier”. The two machines are not interachangable—we say the prover can take as much time as it needs, and only the verifier is required to be fast. A reasonable response to this would be to wonder: if the prover can be arbitrarily powerful, why could it not just compute the correct answer itself and then just send “yes” or “no” to the verifier? Well, the compromise we get for giving the prover unlimited power is we lose the prover’s *trustworthiness*. While our proofs still define a correct prover, and our verifier needs to reliably work correctly when talking with the correct prover, we also require that our verifier can reliably *reject* any other prover that may be trying to trick it into giving a “yes” answer when it should be outputting a “no”.

Something to notice in the last paragraph is that we said “reliably”, not “always” when talking about the verifier’s outputs. This is the other compromise we make with interactive proofs: randomness. Randomness in itself is not a compromise—we could always ignore the randomness capabilities if it gets in the way. However, the compromise that comes with randomness is non-deterministic results. After all, in the real world we only ever get a “yes” or “no” response, and not any sort of information about how improbable that result might be. Further, we said before that any input has exactly one correct response, and so if our results are non-deterministic then at least *sometimes* our computer must necessarily be wrong in its conclusion. One might be tempted to work around this by allowing our laptop to use randomness internally, but say that its *result* must be deterministic. While this would be nice, it turns out that doing this means our computer is no more powerful than if we did not use randomness at all. In brief, if the result is the same no matter what random bits we roll, then if we just replaced every random roll with a non-random result of, e.g., 4,[3] then we would have a machine that uses no randomness and yet outputs the same answer as our random machines.

Interactive proofs are one model of a proof system, but it is not the only one. Probabilistically-checkable proofs (PCPs) are another model, and one that we will be looking at quite a lot. Here, instead of a verifier that responds to our requests, we have a static *proof* (often modeled as a particular oracle) that we can request some

[3]As chosen by fair dice roll: `https://xkcd.com/221/`.

limited number of bits from. Similarly to with interactive proofs, we not only care about what happens when our verifier is given the proper, "honest" proof, but we also want it to reliably be able to sniff out false proofs of false statements and reject them as well.

1.5 Zero-knowledge

The traditional method of proving a statement to someone else is to reveal enough information about each step along the way that the provee can recreate each logical step. This necessarily means revealing lots of additional information that is not the statement being proven. One thing that mathematicians have wondered is whether it is possible to prove a statement *without* revealing any of the information besides the specific statement to be proven.

Intuitively, it would be reasonable to expect this is impossible. After all, one definitely cannot just state something and expect people to believe it is a mathematical truth. However, when we say zero-knowledge, we do not necessarily mean that *no* knowledge is leaked, just that no complicated (i.e., something we can not compute efficiently on our own) information is leaked. In this model, what we allow for is the verifier to challenge the prover: they can make requests, with the idea that if the prover is reliably answering them correctly, then it must be the case that the prover has proven the statement.

Perhaps the most important property of these interactions is that they are *not* generally acceptable as strict mathematical proofs—for any set of queries and responses, it is always possible that a sneaky prover could answer them all correctly by accident. However, for a zero-knowledge proof it must always be statistically unlikely for a prover to answer correctly, and thus with enough tests, it will eventually become a near impossibility for any correctly-answering prover to be lying.

Alongside their mathematical properties, zero-knowledge proofs have many real-world applications. One common example is that of authentication: in general, I would like to be able to prove that I have the password for my account without sending the password publicly. After all, if I sent my password publicly anyone listening in to my conversations could just write it down, and then it is no longer much of a password. However, if we have a zero-knowledge proof that I have my password, it would leak no information about what my password is, and thus I can log in with peace of mind.

A zero-knowledge requirement can be applied to just about any model of computation that involves multiple machines communicating in some fashion. To this end, we will be looking at zero-knowledge versions of nearly every computational model we discuss in this thesis. While the specifics of how we define zero-knowledge differ slightly from model to model, the broad strokes are the same: no information may ever be leaked through the messages between the machines.

1.6 The PCP theorem

The main way we measure the difficulty of probabilistically-checkable proofs is through *query complexity*: in brief, how much (or how little) of the proof we need to look at in order to get a reliable response. In general, we would like to minimize this—we want to find the most efficient way to answer the question posed. It turns out, luckily for us, that if a problem is easy to verify (i.e., it is in the class NP we mentioned earlier), then the query complexity required is astonishingly low [3]. More specifically, there exists a proof for which we only need to read *3 bits* that will allow us to answer the question reliably [20]. This statement is what we call the *PCP theorem*.

This result is astonishing, even to some seasoned computer scientists. There are various proofs of this, including one we will work through in this thesis. While the proof itself is somewhat complicated, we still replicate it because it will closely parallel our technique for proving another result later on.

Of course, this is a thesis about *zero-knowledge* PCPs, and so we would be remiss were we not to look for a zero-knowledge version of the PCP theorem. As we see later on, there is such a statement for zero-knowledge PCPs [17]. While it does not give us the exact constant of 3 (at least, not yet), it does tell us that there is some constant number of queries that will always work. While it might be a little disappointing that we do not get an exact number, the fact that it is constant at all is still a huge result. Remember, we in computer science like to measure things by asymptotic complexity, so a constant complexity is as low as we can go.[4]

[4]Technically, there are sub-constant complexities out there, but they would require the number of queries to go *down* as the input gets longer, which doesn't really make sense here.

Chapter 2

Preliminaries

2.1 Turing machines

Central to our definitions of complexity is that of a Turing machine. This is the most common mathematical model of a computer, and is the jumping-off point for many variants. There are many ways to think of a Turing machine, but the most common is that of a small machine that can read and write to an arbitrarily-long "tape" according to some finite set of rules. We give a more formal definition below, and then we will attempt to take this definition into a more manageable form.

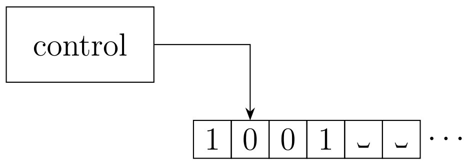

Figure 2.1: A Turing machine

Definition 2.1.1 ([31, Def. 3.1])**.** A *Turing machine* (abbreviated TM) is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ where Q, Σ, and Γ are all finite sets, and

1. Q is the set of *states*,
2. Σ is the *input alphabet*,
3. Γ is the *tape alphabet*,
4. $\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the *transition function*,
5. $q_0 \in Q$ is the *start state*,
6. $q_a \in Q$ is the *accept state*,
7. $q_r \in Q$ is the *reject state*, with $q_a \neq q_r$.

While we have this formalism here as a useful reference, even here we will most frequently refer to Turing machines in a more intuitionisitc form. There are several ways we will think about Turing machines.

The first way to think about a Turing machine is as a little computing box with a tape. We let the box read and write to the tape, and each step it can move the tape one space in either direction. At some point, the machine can decide it is done, in which case we say it "halts"; however it does not necessarily need to halt. For this paper, we will only think about machines that *do* halt, and in particular we will care about how many it takes us to get there. Further, we will use this informalism as a base from which we can define our Turing machine variants intuitively, without needing to deal with the (potentially extremely convoluted) formalism.

Another way we think about a Turing machine is as an algorithm. Perhaps the foundational paper of modern computer science theory, the *Church-Turing thesis* [33], states that any actually-computable algorithm has an equivalent Turing machine, and vice versa. We will use this fact liberally; in many cases we will simply describe an algorithm and not deal with putting it into the context of a Turing machine. If we have explained the algorithm well enough that a reader can execute it (as we endeavor to do), then we know a Turing machine must exist.

Definition 2.1.2 ([31, p. 178]). A *nondeterministic Turing machine* is a Turing machine, but where the transition function has signature

$$\delta\colon Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\}),$$

where $\mathcal{P}(X)$ is the power set of X; that is, the set of all subsets of X.

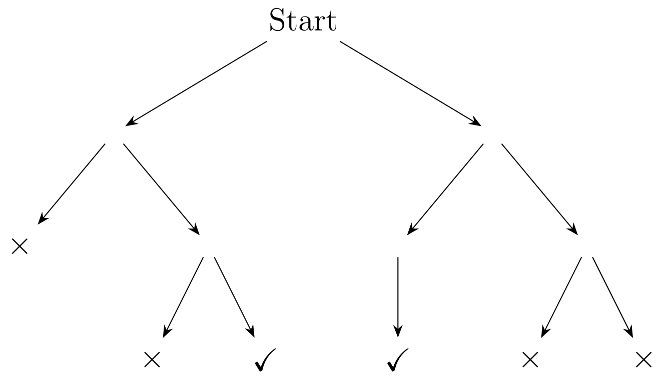

Figure 2.2: Computation graph of a nondeterministic Turing machine

With a nondeterministic machine, instead of returning a single action, we return a set of them. The idea is that these represent a collection of possible actions we could take given the present state. As the computation progresses, these compound on each other, until every possible branch has reached a terminal state. At this point, we check if *any* of the branches are in the accept state, in which case we accept, and reject otherwise. We can see an example of this in Figure 2.2. This computation graph would accept since it has two branches ending in ✓; contrastingly, if every graph ended in $\times$, then it would reject.

Definition 2.1.3. A *multitape Turing machine* is a Turing machine, but where the transition function has signature

$$\delta \colon Q \times \Gamma^n \to Q \times \Gamma^n \times \{L, R\}^n,$$

for some $n \in \mathbb{N}$.

Definition 2.1.4. A *probabilistic Turing machine* is a Turing machine, but with two transition functions δ_1 and δ_2, instead of one.

In each step, instead of applying its sole transition function, a probabilistic machine flips a coin and then applies either δ_1 or δ_2, based on its result. This is the mechanism behind which we can introduce randomness into our machines. The end result of this is that the result of this machine is not a fixed answer, but a *random variable*, something we will discuss in more detail in Section 2.4.

2.2 Complexity classes

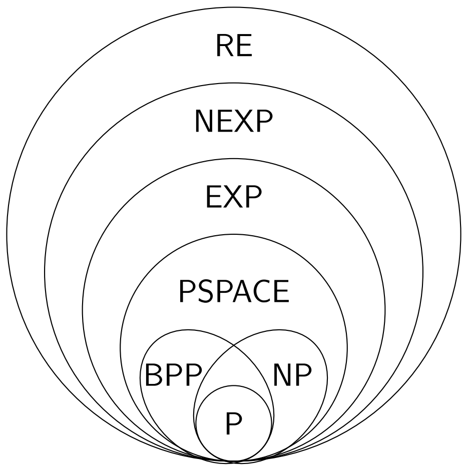

Figure 2.3: The relationship between the complexity classes for this paper

Complexity classes are the main mechanism we use to think about the hardness of problems in computer science. We will build complexity classes out of languages, which have a natural correspondence to decision problems that we will soon see. To define a language, first we need some new notation.

Definition 2.2.1. Let Σ be a finite set. The set Σ^* is the set of all tuples with elements in Σ. That is,

$$\Sigma^* = \bigcup_{i=1}^{\infty} \Sigma^i.$$

Next, we can define what a language actually is.

Definition 2.2.2. Let Σ be a finite set. A *language* (over the *alphabet* Σ) is a subset of Σ^*.

Most frequently, we will be using the alphabet $\{0, 1\}$, as we like to think of computers as binary machines. However, there will be a few places where we need to be explicit about our alphabet, and we will be sure to flag those as they arise.

Important to us is the relationship between languages and decision problems. The important thing about this correspondence is that once we have it, we no longer need to care about the difference between the two. That is, anything that we can say about languages, we can say about decision problems, and vice versa. As such, for the rest of this thesis we will no longer be differentiating between the two.

Definition 2.2.3. Let L be a language over an alphabet Σ. The *decision problem corresponding to* L is the decision problem where the answer of an input $x \in \Sigma^*$ is true if $x \in L$ and false if $x \notin L$.

Finally, we get to define a complexity class.

Definition 2.2.4. A *complexity class* is a set of languages.

While this formal definition is very broad, in practice every complexity class we will see is a set of languages grouped by some complexity-theoretic property. We start with a relatively straightforward example of a complexity class: the class of languages that a Turing machine can recognize. First, we need to define what recognition is in order to make a complexity class out of it.

Definition 2.2.5 ([31, Def. 3.2])**.** A language L is *recognized* by a Turing machine M if for all strings $s \in L$, M halts in the accept state when given s as input.

Now, since our complexity classes are about *languages*, we naturally wish to extend our notion of recognition to a statistic on languages.

Definition 2.2.6. A language L is *Turing-recognizable* (frequently just *recognizable*) if it is recognized by some Turing machine.

Now that we have a property of languages, it is straightforward for us to turn it into a complexity class.

Definition 2.2.7. The class RE is the class of all Turing-recognizable languages.

For most other classes, we want our Turing machines to halt on *all* inputs, not just those in the class. From a practical perspective, this is useful because it tells us that we can be certain about whether any given string is in the given language. From here on, we will generally care about how much of some resource our machines take when making their decision, as opposed to whether or not they can.

2.2.1 Time complexity

The most intuitive (and most important) notion of complexity is that of time complexity. Time complexity is the answer of the question of how long it takes to solve a problem. We begin with an abstract base for our time classes, and will then introduce some specific ones that we care about. Before that, though, we need to talk about *how* we compare times.

For many problems, the length of time taken is not particularly straightforward. While *in general*, problems will tend to get more difficult to solve as their inputs get longer, this is not necessarily true for any given input length. One could imagine a problem where there is a special-case fast algorithm for inputs of lengths that are a power of 2 (for example), but any other length takes much longer to compute. Given that, one might wonder how we would compare this to a problem that takes slightly less time than the non-special-case of the previous problem, but does so consistently. This is the framework to which we will apply big-O notation.

Definition 2.2.8 ([13, p. 47])**.** Let $f, g\colon \mathbb{N} \to \mathbb{N}$ be functions. Then $g(n) \in O(f(n))$ if and only if there exists $c, n_0 \in \mathbb{N}$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

Big-O notation is widely used across computer science to compare attributes of problems. In general, the way we will think of functions like f and g is representing the maximum number of steps (or maximum amount of some other resource) used by our computer when given an input of length n. When we think about functions in this context, $g(x) \in O(f(x))$ translates roughly to "$g(x)$ is no harder than $f(x)$". At first, this might seem slightly confusing, since $2f(x) \in O(f(x))$, but $2f(x)$ definitely represents a harder problem than $f(x)$. While this is true, it is not harder in a meaningful way to us: scalar multiples are often not preserved by a change in our computational model, and thus we want to ignore them whenever possible.

In addition to big-O, we will want one more class of functions, this being the one corresponding to all polynomials. We want this because there is no function f such that $O(f(n))$ corresponds to exactly the set of polynomials: $f(n) \in O(f(n))$ always, so f must be a polynomial, but for any k, $n^{k+1} \notin O(n^k)$. Because of this, we need a separate definition, and cannot simply reuse our big-O.

Definition 2.2.9. The set $\mathsf{poly}(n)$ is

$$\mathsf{poly}(n) = \bigcup_{i=1}^{\infty} O(n^k).$$

Now, we can finally begin to start looking at actual complexity classes. We will start off by showing a general correspondence between complexity classes and big-O classes of functions. The idea is that the complexity class corresponding to a function is the set of all languages that are computable in that time. Because we have multiple definitions of Turing machines, there are multiple such correspondences. The two we will care about in particular are those for deterministic and nondeterministic Turing machines.

Definition 2.2.10 ([2, Def. 1.19])**.** Let $f\colon \mathbb{N} \to \mathbb{N}$ be a function. The class $\mathsf{DTIME}(f(n))$ is the class of all languages computable by a deterministic Turing machine in $O(f(n))$ steps.

Definition 2.2.11 ([2, Def. 2.5])**.** Let $f\colon \mathbb{N} \to \mathbb{N}$ be a function. The class $\mathsf{NTIME}(f(n))$ is the class of all languages computable by a nondeterministic Turing machine in $O(f(n))$ steps.

While DTIME and NTIME are useful bases to start from, it is rare that we deal with these classes directly. Instead, we will deal with some classes with slightly nicer mathematical properties, which we will build using DTIME and NTIME as a base.

Definition 2.2.12 ([2, Def. 1.20])**.** The complexity class P is the class

$$\mathsf{P} = \bigcup_{c>0} \mathsf{DTIME}(n^c).$$

If a language is in P, we say it is *polynomial-time.*

The class P is perhaps the most important complexity class. Mathematically, we care about P because it is closed under composition: a polynomial-time algorithm iterated a polynomial number of times is still in P. Further, P turns out to generally be invariant under change of (deterministic) computation model, which allows us to reason about P problems easily without needing to resort to the formal definition of a Turing machine. More philosophically, P generally represents the set of “efficient” algorithms in the real world.[1] For that matter, in this paper if we ever refer to an algorithm as “efficient” we mean that is polynomial-time.

Theorem 2.2.13. *The language*

$$\{(p, x, y) \mid p \text{ a polynomial and } p(x) = y\}$$

is in P*.*

Proof. We can calculate whether a string is in this language by calculating $p(x)$ (which we can do in polynomial time), and then comparing it to y. □

As we have defined P in terms of DTIME, the question arises of whether there is an equivalent in terms of NTIME. Naturally, there is, and we call it NP.

Definition 2.2.14 ([31, Cor. 7.22])**.** The complexity class NP is the class

$$\mathsf{NP} = \bigcup_{c>0} \mathsf{NTIME}(n^c).$$

[1] It is worth mentioning that this is a *mathematical* efficiency—there are plenty of algorithms in P that a real-world computer scientist would never dare to call efficient.

While this definition demonstrates how NP is similar to P, there are other equivalent ones that we can use. In particular, we very often like to think of NP in terms of deterministic *verifiers*. Since nondeterministic machines do not exist in real life, this definition gives a practical meaning to NP.

By way of an example, we briefly sketch out a very important NP language. While this language is important in general and makes for a good example, it is only tangentially important to the rest of this thesis. For this reason, we will only be providing an informal sketch of the relevant proof; there are many more formal treatments, for example in [31].

Definition 2.2.15 ([31, p. 299])**.** A *Boolean formula* is any expression where the only operations used are the three Boolean operations logical and ($\wedge$), logical or ($\vee$), and logical not ($\neg$).

Definition 2.2.16 ([31, p. 299])**.** The language SAT is the language of Boolean formulas with at least one assignment of Boolean values to its variables such that the Boolean formula evaluates to 1.

Theorem 2.2.17. *The language* SAT *is in* NP.

Proof. Nondeterministically pick an assignment of values to variables, then evaluate the formula. Since evaluating the formula can be done efficiently, then our nondeterministic algorithm will complete in polynomial time. Further, at least one branch will accept if and only if at least one assignment of values results in the formula outputting 1. $\square$

The definition of NP we gave in Definition 2.2.14 is not the only useful one. We have already referred to NP a few times as "the set of languages that can be verified easily", and the definition we are about to give is the definition that corresponds to that. Before we do that, however, we we need to define what it means for a Turing machine to verify a language—after all, so far all we have seen formally defined is recognition.

Definition 2.2.18 ([31, Def. 7.18])**.** A *verifier* for a language L is an algorithm A where for any $x \in \Sigma^*$, $x \in L$ if and only if there exists a $c \in \Sigma^*$ (which we call the *certificate*) such that A accepts when given (x, c) as input.

With our definition of a verifier in hand, we are now free to give our alternate definition of NP.

Theorem 2.2.19 ([31, Def. 7.19])**.** NP *is exactly the class of all languages verifiable by a polynomial-time Turing machine.*

As an example, the language SAT we defined in Definition 2.2.16 can be verified efficiently, where the certificate is an accepting set of variables. Since we can evaluate a Boolean formula efficiently, if we already have an accepting set of variables we can therefore verify it in polynomial time.

The precise relationship between P and NP is not very well-known. We know $\mathsf{P} \subseteq \mathsf{NP}$, as any deterministic algorithm corresponds exactly to a nondeterministic algorithm of the same speed. However, whether $\mathsf{P} = \mathsf{NP}$ is still unknown, and one of the largest open problems in complexity theory.

The next step up from polynomial complexities is that of exponential complexities. For these, instead of having the classes bounded above by a polynomial, we have the classes bounded above by 2 to the power of a polynomial. While we use 2 as the base, the value of the base turns out not to matter since for any $a, b > 1$,

$$a^{n^c} = b^{n^c \log_b(a)} \in O\left(b^{n^{c+1}}\right). \tag{2.1}$$

Definition 2.2.20 ([2, §2.6.2])**.** The complexity class EXP is the class

$$\mathsf{EXP} = \bigcup_{c>0} \mathsf{DTIME}\left(2^{n^k}\right).$$

Similarly, the complexity class NEXP is the class

$$\mathsf{NEXP} = \bigcup_{c>0} \mathsf{NTIME}\left(2^{n^k}\right).$$

It is immediate that $\mathsf{P} \subseteq \mathsf{EXP}$ and $\mathsf{NP} \subseteq \mathsf{NEXP}$ (since the exponential classes allow the use of more of the same resource). Of slightly less-trivial interest is the relationship between NP and NEXP.

Theorem 2.2.21. $\mathsf{NP} \subseteq \mathsf{EXP}$.

Proof. If a nondeterministic machine solves a problem in $p(n)$ steps, it follows that the total number of branches is less than $a^{p(n)}$, where a is the maximum number of nondeterministic choices we make at any given step. Hence, we can simulate the machine deterministically by simply enumerating every branch, giving us a total computation time of $p(n)a^{p(n)}$, which is in $O(2^{q(n)})$ for some other polynomial $q(n)$. Hence any NP problem is in EXP. □

It is perhaps illustrative to see an example of a problem in NEXP. Here, we present an language whose relationship to NEXP is very similar to how SAT is related to NP. Unlike SAT, we will be seeing a lot of this language throughout the rest of the thesis, and finding algorithms for it will be a major part of several upcoming chapters.

Definition 2.2.22 ([10, Def. 14.1])**.** The *oracle 3-satisfiability problem*, denoted $\mathsf{O3SAT}$, is the language of all triplets (r, s, B), where $r, s \in \mathbb{N}^+$ and $B\colon \{0,1\}^{r+3s+3} \to \{0,1\}$ a boolean function, such that there exists a boolean function $A\colon \{0,1\}^s \to \{0,1\}$ having the property that for all $z \in \{0,1\}^r$ and $b_1, b_2, b_3 \in \{0,1\}^s$,

$$B(z, b_1, b_2, b_3, A(b_1), A(b_2), A(b_3)) = 1.$$

Theorem 2.2.23. $\mathsf{O3SAT} \in \mathsf{NEXP}$.

Proof. We present the following non-deterministic algorithm to determine if $(r, s, B) \in$ O3SAT:

```
    Input: A triplet $(r, s, B)$
    Output: Whether or not $(r, s, B) \in$ O3SAT
 1  Nondeterministically choose a function $A\colon \{0,1\}^s \to \{0,1\}$;
 2  for $z \in \{0,1\}^r$ do
 3      for $b_1, b_2, b_3 \in \{0,1\}^s$ do
 4          Compute $B(z, b_1, b_2, b_3, A(b_1), A(b_2), A(b_3))$;
 5          if the above is not 1 then
 6              reject;
 7          end
 8      end
 9  end
10  accept;
```

Algorithm 2.1: A NEXP-time algorithm for determining O3SAT

First, we need to show that Algorithm 2.1 is in NEXP. Nondeterministically choosing a function from $\{0,1\}^s$ can be done in time 2^s; and the two loops will run a total of $2^r 2^s$ times, respectively. Computation of a function can be done in polynomial time relative to its length; hence the runtime of this function is in exponential time relative to $r + s$.

One might be tempted to think that since we are given a function B as input, that our function A can be no longer than $\mathsf{poly}(|B|)$, but this is not necessarily true. We are given B in 3SAT form, and thus there are expressions of B that are polynomial with respect to $r + s$. Despite this, there are polynomial-length B instances whose A is *not* polynomial in length (since that is an arbitrary function); since runtime complexity is about the worst case there thus exist inputs that cannot be computed in polynomial time relative to their length.

Next, we want to show that Algorithm 2.1 actually recognizes O3SAT. If $(r, s, B) \in$ O3SAT, then there exists a function A such that for any values z and b_i, B evaluates to 1. Since we nondeterministically choose A, one of our nondeterministic branches will pick it, and in that branch every one of the computations in line 4 will be 1. Thus, that branch will accept and therefore so will the entire machine.

If $(r, s, B) \notin$ O3SAT, then no matter what nondeterministic branch we are on, the value of A means that the computation in line 4 will be 0. Hence, every branch will reject and thus Algorithm 2.1 as a whole will.

Since Algorithm 2.1 runs in exponential time and will accept if and only if its input is in O3SAT, it follows that O3SAT $\in$ NEXP. □

2.2.2 Space complexity

Similar to time complexity, space complexity is the question of how much space on its memory tape a machine needs in order to compute a problem. In many ways, our definitions of space complexity are analogous to those for time complexity that we have already defined. In particular, DSPACE will correspond nicely to DTIME, and NSPACE to NTIME.

Definition 2.2.24 ([2, Def. 4.1])**.** Let $f\colon \mathbb{N} \to \mathbb{N}$ be a function. A language L is in $\mathsf{DSPACE}(f(n))$ if there exists a deterministic Turing machine M such that the number of locations on the tape that are non-blank at some point during the execution of M is in $O(f(n))$.

Similarly, a language L is in $\mathsf{NSPACE}(f(n))$ if there exists a nondeterministic Turing machine M such that the number of locations on the tape that are non-blank at some point during the execution of M is in $O(f(n))$.

Analogously to P and NP, our two main classes of space complexity are PSPACE and $\mathsf{NPSPACE}$.

Definition 2.2.25 ([2, Def. 4.5])**.** The complexity class PSPACE is the class

$$\mathsf{PSPACE} = \bigcup_{c>0} \mathsf{DSPACE}(n^c),$$

and the complexity class $\mathsf{NPSPACE}$ is the class

$$\mathsf{NPSPACE} = \bigcup_{c>0} \mathsf{NSPACE}(n^c).$$

Unlike with P and NP, the relationship between PSPACE and $\mathsf{NPSPACE}$ is well known. Due to the complexity of the proof of the theorem, we will not prove it here, as it is mostly not relevant to what we will be doing.

Theorem 2.2.26 (Savitch's theorem; [29])**.** $\mathsf{PSPACE} = \mathsf{NPSPACE}$.

Upon seeing this, one might ask why it is that we believe $\mathsf{P} \neq \mathsf{NP}$ if we know that $\mathsf{PSPACE} = \mathsf{NPSPACE}$, given they are defined analogously. The answer to this question boils down to the fact that we are able to reuse space, while we are not able to reuse time. Space on the tape that is no longer needed can be overwritten, while time that is no longer needed is gone forever.

Since PSPACE and $\mathsf{NPSPACE}$ are equal classes, it is relatively rare to see $\mathsf{NPSPACE}$ referred to. Here, we will only refer to it when it makes a class relationship clearer; most frequently when comparing $\mathsf{NPSPACE}$ to some other nondeterministic class.

2.2.3 Completeness

Even within a complexity class, not all problems are created equal. The notion of *completeness* gives us a mathematically-rigorous way to talk about which problems in a class are the hardest. Since putting upper bounds on hard problems naturally puts those same bounds on any easier problems, complete problems can be useful in reasoning about the relationship between complexity classes.

Definition 2.2.27 ([31, Def. 7.29])**.** A language A is *polynomial-time reducible* to a language B if a polynomial-time computable function $f\colon \Sigma^* \to \Sigma^*$ exists such that for all $w \in \Sigma^*$, $w \in A$ if and only if $f(w) \in B$.

Polynomial-time reductions are important because they give us a way to say that A is *no harder* than B. In particular, if we have an algorithm M that determines B, we can construct the following algorithm that determines A with only a polynomial amount of additional work:

```
    Input: A string $w \in \Sigma^*$
    Output: Whether $w \in A$
1   Compute $f(w)$;
2   Use $M$ to check whether $f(w) \in B$;
3   return the result of $M$;
```

Algorithm 2.2: An algorithm to reduce A to B

Definition 2.2.28 ([31, Def. 7.34])**.** A language L is NP-complete if $L \in \mathsf{NP}$ and every $A \in \mathsf{NP}$ is polynomial-time reducible to L.

This is a practical use of our polynomial-time reductions: since an NP-complete language has a reduction from every other language in NP, it follows that it is *at least as hard* as any other language in NP. Of particular interest to complexity theorists is the fact that $\mathsf{P} = \mathsf{NP}$ if and only if *any* NP-complete language is in P.

Example 2.2.28.1. A famous result of Cook [12], also proved around the same time by Levin and thus called the *Cook-Levin theorem*, is that the SAT problem we defined earlier in Definition 2.2.16 is NP-complete.

The notion of completeness is very important to complexity theorists. Since these are the "hardest" problems in NP, this means that if we can do anything interesting to an NP-complete problem, we can leverage these reductions to do that interesting thing to *any* other problem in NP with only a little (i.e. polynomial) more effort. This will come in especially handy when we want to prove that complexity classes are equal or that NP is a subset of some other complexity class—since most complexity classes allow for things to change polynomially, we only need to prove that a single NP-complete element is in the other class for the subset relation to follow.

Along with completeness for NP, we have a notion of completeness for NEXP. While you might expect that the reducibility constraints might loosen (i.e. allow more complex reductions) since NEXP is more complex for NP, but this turns out not to be the case. In particular, while it might initially seem logical to allow for EXP-reductions, it turns out that NEXP is not closed under EXP-reductions, which makes a notion of completeness challenging. Despite this, we can still learn interesting things about NEXP by studying completeness under polynomial reductions.

Definition 2.2.29. A language L is NEXP-complete if $L \in \mathsf{NEXP}$ and every $A \in \mathsf{NEXP}$ is polynomial-time reducible to NEXP.

NEXP-completeness has many of the same nice properties of NP-completeness. Of particular interest to us will again be the ease with which NEXP-completeness allows us to determine subset relations, simply by proving the inclusion of a single complete language.

For our language O3SAT we defined earlier (in Definition 2.2.22), we actually have an even *stronger* notion of completeness, with a polynomial-time reduction for arbitrary time functions $T(n)$. We will get the standard NEXP-completeness of $\mathsf{O3SAT}$ as a corollary, but we will actually find the stronger characterization here useful in later chapters.

Theorem 2.2.30 (Cook-Levin)**.** *Let M be a $\mathsf{NTIME}(T(n))$ Turing machine, with $T \in \Omega(n)$. Then, there exists a polynomial-time reduction R_M such that for any input $x \in \{0,1\}^n$, $R_m(x) \in \mathsf{O3SAT}$ if and only if there exists a w with $M(x,w) = 1$. Furthermore, $R_m(x)$ outputs a formula in $O(\log(T(n)))$ variables, with size $\mathsf{poly}(n, \log(T(n)))$.*

Corollary 2.2.31 ([4, Proposition 4.2])**.** *The language $\mathsf{O3SAT}$ is NEXP-complete.*

Proof. Let $T(n) = 2^{n^k}$ for some k. Then, the output formula of $R_m(x)$ is a formula in $O(\log(2^{n^k})) = O(n^k)$ variables, with size

$$\mathsf{poly}(n, \log(2^{n^k})) = \mathsf{poly}(n^k) = \mathsf{poly}(n). \tag{2.2}$$

This is the normal definition of $\mathsf{O3SAT}$; hence it is NEXP-complete. □

Just as we have NP-completeness and NEXP-completeness for time complexity, we also have notions of completeness for space complexity. Since $\mathsf{PSPACE} = \mathsf{NPSPACE}$, instead of calling the class $\mathsf{NPSPACE}$-complete, we call it PSPACE-complete.

Definition 2.2.32 ([31, Def. 8.8])**.** A language L is PSPACE-complete if $L \in \mathsf{PSPACE}$ and every $A \in \mathsf{PSPACE}$ is polynomial-time reducible to L.

While this definition is mostly analagous to that of NP-completeness, one might wonder why we use a time complexity for our reduction when PSPACE is a space-complexity class. This is because if we were to use space complexity, we would want to use PSPACE-reductions, but that would make every language in PSPACE trivially PSPACE-complete. Since that is not a useful definition, we instead restrict ourselves to polynomial-time reductions.

2.2.4 Randomized complexity

All the complexity classes we have seen so far are *non-randomized*: they do not allow a Turing machine to consult any source of randomness. In these models, a Turing machine's output is always the same when it is given the same input. However, there exist Turing machines that can consult a random-bit generator. Because of their different capabilities, we can define separate complexity classes to the languages corresponding to those machines.

The main probabilistic class we care about is BPP (short for "bounded-error probabilistic polynomial"). This class is similar to P in that it requires its machines to run in polynomial time, but it has one large difference. Because these are probabilistic machines, we want to be able to leverage that randomness. As such, we do not insist that we *always* accept when given a string in the language, only most of the time. So long as we accept with a probability noticeably greater than 1/2 (and reject similarly), we can always boost the probability arbitrarily high by repeating the simulation multiple times and taking the majority vote.

Definition 2.2.33. A language L is in BPP if there exists a probabilistic Turing machine M such that

1. M runs in polynomial time,
2. for all $x \in L$, M accepts x with probability at least $2/3$,
3. for all $x \notin L$, M rejects x with probability at least $2/3$.

Next, it is important to see how BPP relates to the other complexity classes. We have also included BPP on the Venn diagram in Figure 4.1 earlier. It is important to note that neither of the following inclusions are known to be strict.

Theorem 2.2.34. $\mathsf{P} \subseteq \mathsf{BPP}$.

Proof. A deterministic Turing machine is equivalent to a randomized machine that never consults its oracle. In this way, a polynomial-time Turing machine fulfills all the requirements of Definition 2.2.33: it runs in polynomial time, it accepts each $x \in L$ with probability $1 > 2/3$, and it rejects each $x \notin L$ with probability $1 > 2/3$. $\square$

Theorem 2.2.35. $\mathsf{BPP} \subseteq \mathsf{PSPACE}$.

Proof. Consider a BPP machine M that reads r random bits on input x. Then, consider a machine M' that loops through every $\rho \in \{0,1\}^r$ and simulates M on input x and random coins r, finally accepting if at least $2/3$ of the instances accepted and rejecting if $2/3$ of the instances rejected. This machine is deterministic and would accept the same language as M, but it would run in polynomial space, since simulating M is polynomial space and after each iteration we can throw out the scratchwork, only keeping the total accept/reject count. $\square$

2.2.5 Impagliazzo's five worlds

The question of P vs NP is a large and important one, and one that is not particularly close to being resolved. However, the possible resolutions to this central dilemma are more nuanced than just "yes" or "no". In particular, there are many, many results that depend on the answer to the P vs NP question, either a positive result or a negative one.

A 1995 paper by Impagliazzo [21] talks about this problem in more depth. In this paper, he identifies five "worlds", ranging the full spectrum from "$\mathsf{P} = \mathsf{NP}$ efficiently" all the way to "$\mathsf{P} \neq \mathsf{NP}$ and cryptography is possible in full". This is relevant to us because we will be working with zero-knowledge a lot for this paper, and as we will see in Section 4.4, several important building blocks of zero-knowledge proofs are impossible in many of Impagliazzo's worlds. For this reason, for the rest of this paper we will be assuming we are in "cryptomania", his most cryptographic world. The most important assumption we will be pulling from this, besides that $\mathsf{P} \neq \mathsf{NP}$, is that one-way functions (which we will be defining in more depth in Section 4.4.1) exist.

2.3 Polynomials

Many of the theorems we will be working with will make heavy use of various properties of polynomials. We will prove several of those here.

Definition 2.3.1 ([25]). Let P be a mathematical statement. The function $[P]$ is the the function

$$[P] = \begin{cases} 1 & P \text{ is true} \\ 0 & \text{otherwise.} \end{cases} \tag{2.3}$$

This is called the *Iverson bracket*, after its inventor Kenneth Iverson, who originally included it in the programming language APL[2] [22, p. 11].

As an example, we can use the Iverson bracket to define the Kronecker delta function as

$$\delta_{ij} = [i = j].$$

Traditionally, many people think of polynomials as being functions over the real numbers: a polynomial takes a real number as input and gives a real number as output. We would like to generalize this notion to be as broad as possible: what kind of other mathematical things can we make a polynomial out of? To some extent, all we really need to make a polynomial out of something is to allow ourselves to do addition and multiplication on it.

Definition 2.3.2 ([32, Def. 2.6.1]). A *field* is a set $\mathbb{F}$ with functions $+, \cdot : \mathbb{F} \to \mathbb{F}$ such that there exist elements $0, 1 \in F$ and for all $a, b, c \in F$,

1. $a + 0 = a = 0 + a$.
2. $a + (b + c) = (a + b) + c$.
3. $a + b = b + a$.
4. $a \cdot 1 = a = 1 \cdot a$.
5. $a + (b \cdot c) = a \cdot b + a \cdot c$.
6. $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
7. $a \cdot b = b \cdot a$.
8. There exists $d \in F$ such that $a + d = d + a = 0$.
9. If $a \neq 0$, there exists $d \in F$ such that $a \cdot d = d \cdot a = 1$.
10. $0 \neq 1$.

[2]The original notation used parentheses, but square brackets are much less ambiguous, so that has become the standard and what we will use here.

These ten conditions presented above are what we need for addition and multiplication to "work nicely", so to speak. The real numbers are our favorite example of a field, but there exist many others. Because we generally like finite sets in computer science, we will often be restricting ourselves to work with specifically finite fields. Our favorite examples of these are the modular-arithmetic fields $\mathbb{Z}/p\mathbb{Z}$, where p is prime. These work like the integers, but with addition and multiplication done mod p instead of freely.

Much of our work will deal with multivariate polynomials. For a given field $\mathbb{F}$, we will denote the set of m-variable polynomials over $\mathbb{F}$ with $\mathbb{F}[x_{1,\dots,m}]$. The terminology of multivariate polynomials generally coincides with that of single-variable polynomials, but not always. Hence, we will introduce some important multivariable-specific terminology and definitions here. First up, perhaps the most important property of a polynomial is its degree. Degree is well-defined for multivariate polynomials just as it is for monovariate ones, but there is a second, closely-related definition we call *multidegree.*

Definition 2.3.3 ([1, p. 8])**.** The *multidegree* of a multivariate polynomial p, written $\mathrm{mdeg}(d)$, is the maximum degree of any variable x_i of p.

We denote by $\mathbb{F}[x_{1,\dots,m}^{\leq d}]$ the subset of $\mathbb{F}[x_{1,\dots,m}]$ of polynomials with multidegree at most d. Similarly, if we want to refer to polynomials with degree at most d, we will write $\mathbb{F}^{\leq d}[x_{1,\dots,m}]$.

It is worth noting that for monovariate polynomials, multidegree and degree coincide. The difference between multidegree and degree is subtle, but important. To help make the difference more clear, we will often refer to degree as *total degree* instead of just degree. We shall illustrate the difference with a simple example. We also need two special cases of these polynomials, which we will want to quickly be able to reference throughout the paper.

Definition 2.3.4 ([1, p. 8])**.** A polynomial is *multilinear* if it has multidegree at most 1. Similarly, a polynomial is *multiquadratic* if it has multidegree at most 2.

Consider the polynomial $x_1^2x_2 + x_2^2$. The multidegree of this polynomial is 2, while its degree is 3. The polynomial $x_1x_2 + 4x_2x_3 + x_1x_2x_3$ is multilinear. The polynomial $x_1^2x_2x_3 - 2x_1x_3 + 3x_2^2$ is multiquadratic.

From here, we need to define the notion of an *extension polynomial.* This gives the ability to take an arbitrary multivariate function defined on a subset of a field and extend it to be a multivariate polynomial over the *whole* field.

Definition 2.3.5 ([1, p. 8])**.** Let $\mathbb{F}$ be a finite field, $H \subseteq \mathbb{F}$, $m \in \mathbb{N}$ a number, and $f\colon H^m \to \mathbb{F}$ be a function. An *extension polynomial* of f is any polynomial $f' \in \mathbb{F}[x_{1,\dots,m}]$ such that $f(h) = f'(h)$ for all $h \in H$.

Example 2.3.5.1. Define $H = \{0,1\}^3 \subseteq \mathbb{R}^3$. Further define

$$\begin{aligned} f\colon H^3 &\to \mathbb{R} \\ (a,b,c) &\mapsto a \oplus b \oplus c, \end{aligned}$$

where $\oplus$ is the xor function; equivalently addition mod 2. Then one extension polynomial of f is the function

$$f'(x, y, z) = xyz - (x - y)(y - z)(z - x).$$

A second extension polynomial of f is the function

$$f''(x, y, z) = x + y + z - 2xy - 2yz - 2xz + 4xyz.$$

There are (at least) two important things to be gleaned from this example. First, extension polynomials are not unique: f' and f'' are not equal to each other (they are not even of the same multidegree). Second, f'' is in fact multilinear, which might be a somewhat lower multidegree than expected given we need to interpolate 8 different points. It turns out that this is not particularly unusual: while our choice of H is particularly nice, functions from this particular H still happen to be quite nice in general. Even for less-nice values of H, extension polynomials need only to be of a surprisingly low multidegree. Since polynomials of lower degree are generally easier to compute, we would like to see exactly what these low-degree extension polynomials look like and how they work.

Definition 2.3.6 ([10, §5.1])**.** Let $\mathbb{F}$ be a finite field, $H \subseteq \mathbb{F}$, $m \in \mathbb{N}$ a number, and $f\colon H^m \to \mathbb{F}$ be a function. A *low-degree extension* $\tilde{f}$ of f is an extension of f with multidegree at most $|H| - 1$.

It turns out that this is the minimum possible degree of any extension polynomial. Further, it turns out that for any f, there is a *unique* low-degree extension. Neither of these statements are particularly important for our further work, so we will not endeavor to prove them here. Something of practical use to us is an explicit formula for the low-degree extension, which we shall now calculate.

Theorem 2.3.7 ([10, §5.1])**.** *Let $\mathbb{F}$ be a finite field, $H \subseteq \mathbb{F}$, $m \in \mathbb{N}$ a number, and $f\colon H^m \to \mathbb{F}$. Then a low-degree extension $\tilde{f}$ of f is the function*

$$\tilde{f}(x) = \sum_{\beta \in H^m} \delta_\beta(x) f(\beta), \tag{2.4}$$

where δ is the polynomial

$$\delta_x(y) = \prod_{i=1}^{m} \left(\sum_{\omega \in H} \left(\prod_{\gamma \in H \setminus \{\omega\}} \frac{(x_i - \gamma)(y_i - \gamma)}{(\omega - \gamma)^2} \right) \right). \tag{2.5}$$

Proof. First, we must show $\tilde{f}$ has multidegree $|H| - 1$. First, note that $\tilde{f}$ is a linear combination of some $\delta_x$es; hence asking about the multidegree of $\tilde{f}$ is really just asking about the multidegree of δ_x. Looking at δ_x, the innermost product has $|H| - 1$ terms, each with the same y_i; thus those terms have multidegree $|H| - 1$. Summing terms preserves their multidegree, and the outer product iterates over the variables, thus it preserves multidegree as well. Thus, δ_x has multidegree $|H| - 1$.

To understand why $\tilde{f}(x)$ agrees with $f(x)$ on H, we first should look at $\delta_\beta(x)$. In particular, for all $x, y \in H^m$,

$$\delta_y(x) = [x = y] = \delta_{xy}. \tag{2.6}$$

This can be shown through some algebra which we have worked through in full detail in Appendix A. This is the reason why we have named the polynomial in Equation (2.5) as we have; it functions as the Kronecker delta function over the set H^m.

Taking the above statement, we get that for all $x \in H^m$, the only nonzero term of $\tilde{f}(x)$ is the term where $\beta = x$; thus $\tilde{f}(x) = f(x)$. Hence, $\tilde{f}$ is a low-degree extension of f. □

Of particular interest to us will be the case of low-degree extensions where $H = \{0, 1\}$. Since every field contains both 0 and 1, this will allow us to construct a set consisting of an extension for *every* field. Further, since $|H| = 2$ here, it means our low-degree extensions will be multilinear. Not only do we thus constrain our polynomial to have a very low multidegree, the δ function also dramatically simplifies in this case, which makes it much easier to reason about.

Corollary 2.3.8 ([1, §4.1])**.** *Let $\mathbb{F}$ be a finite field, $m \in \mathbb{N}$ a number, and $f\colon \{0,1\}^m \to \mathbb{F}$. Then*

$$\tilde{f}(x) = \sum_{\beta \in \{0,1\}^m} \delta_\beta(x) f(\beta) \tag{2.7}$$

is a low-degree extension of f, where δ is the polynomial

$$\delta_y(x) = \left(\prod_{i : y_i = 1} x_i \right) \left(\prod_{i : y_i = 0} (1 - x_i) \right). \tag{2.8}$$

As we can see, the form of δ in Equation (2.8) is much more manageable than the form in Equation (2.5), and it is perhaps more immediately apparent here why δ has the property it does. Further, since this equation has no division, it turns out that it is valid for arbitrary (non-trivial) rings, while the more complex equation is only valid for fields. We show the algebra that brings us from the first to the second in Appendix A.

The form of δ_y defined in Equation (2.8) has further use to us than just being simpler. In particular, these δ_y form a basis of multilinear polynomials (and hence a generating set for the ring of all polynomials). This is a particularly useful basis because it allows us to reason about multilinear polynomials based solely on their outcomes on the Boolean cube.[3]

Theorem 2.3.9 ([1, §4.1])**.** *For any field $\mathbb{F}$, the set $\{\delta_x \mid x \in \{0,1\}^n\}$ forms a basis for the vector space of multilinear polynomials $\mathbb{F}^n \to \mathbb{F}$.*

[3]As an aside, this fact provides a relatively slick proof of the special case of our unproven statement earlier that low-degree extensions are both of minimal degree and unique.

Proof. Since $\delta_y(x) = 0$ for all $y \neq x \in \{0,1\}^n$, it follows that the only way to get

$$\sum_{y \in \{0,1\}^n} a_y \delta_y = 0 \tag{2.9}$$

is to have each $a_y = 0$. Hence the set of δ_x is linearly independent. Further, the vector space of multilinear polynomials has 2^n dimensions; since there are 2^n distinct δ_x polynomials, it follows that they form a basis. □

Now, we can use this fact to prove some cases where our low-degree extensions turn out to have a particularly low degree. Unfortunately, these do have a lot of qualifiers to them, but they will be useful in later theorems (in particular Lemma 2.3.12).

Lemma 2.3.10 ([1, Lemma 4.2]). *Let $\mathbb{F}$ be a field and $Y \subseteq \mathbb{F}^n$ be a set of t points. Then, there exists a multilinear polynomial $m : \mathbb{F}^m \to \mathbb{F}$ such that*

1. *$m(y) = 0$ for all $y \in Y$, and*
2. *$m(z) = 1$ for at least $2^n - t$ Boolean points z.*

This lemma should look similar to Corollary 2.3.8, but it does have a few key differences. The main difference is that in that lemma, *every* point we care about is in $\{0,1\}^n$, but here the points in Y (that we can pick) are not necessarily Boolean. The compromise is that we do not get to pick the z in the condition; we know how many of them there will be, but not necessarily what they are.[4]

Proof. As per Theorem 2.3.9, we know that we can write

$$m(x) = \sum_{z \in \{0,1\}^n} m_z \delta_z(x) \tag{2.10}$$

for some constants $m_z \in \mathbb{F}$. Using this, for each $y_i \in Y$ we can write $m(y_i) = 0$ as a linear equation relating the m_z. Since there are 2^n m_z and t equations, it follows that our solution space has dimension $2^n - t$. Hence, we can find a solution to these equations with at least $2^n - t$ coefficients equal to 1. For any z where $m_z = 1$, it follows that $m(z) = 1$; thus our conditions hold. □

We can use this proof to give us another polynomial construction. For this, after we pick our y, instead of getting $2^n - t$ points in $\{0,1\}^n$ to be 1, we get exactly one of them equal to 1 and then *every* other Boolean point evaluates to 0. We will be using this as an "adversary" polynomial: the idea is *almost* everywhere relevant to us it evaluates to 0, but there is a single tricky Boolean point where it evaluates to 1.

Lemma 2.3.11 ([1, Lemma 4.3]). *Let $\mathbb{F}$ be a field and $Y \subseteq \mathbb{F}^n$ be a set of t points $y_1, \dots, y_t$. Then for at least $2^n - t$ Boolean points $w \in \{0,1\}^n$, there exists a multi-quadratic extension polynomial $p \colon \mathbb{F}^n \to \mathbb{F}$ such that*

[4] Until we actually construct the polynomial, that is.

1. *$p(y_i) = 0$ for all $y_i \in Y$,*
2. *$p(w) = 1$,*
3. *$p(z) = 0$ for all Boolean $z \neq w$.*

Proof. Define m to be the multilinear polynomial from Lemma 2.3.10. Let $w \in \{0,1\}^n$ be any of the $2^n - t$ such points where $m(w) = 1$. Then, define

$$p(x) = m(x)\delta_w(x), \tag{2.11}$$

where δ is the multilinear polynomial from Equation (2.8). Since $p(x)$ is the product of two multilinear polynomials, it is quadratic; from the definition of δ it is equal to 0 for all $z \in \{0,1\}^n \setminus \{w\}$, and from the definition of m it is equal to 0 for all $y_i \in Y$. □

Now that we have it for one polynomial, we would like to extend this to a collection of polynomials. It may be a little strange to be looking at a collection of polynomials over different fields, but this construction will come in useful later on. In particular, keep this in mind when we define *extension oracles* in Definition 3.5.1; there, we again will be dealing with functions over a collection of fields.

Lemma 2.3.12 ([1, Lemma 4.5])**.** *Let $\mathcal{F}$ be a collection of fields. Let $f\colon \{0,1\}^n \to \{0,1\}$ be a Boolean function, and for every $\mathbb{F} \in \mathcal{F}$, let $p_{\mathbb{F}}\colon \mathbb{F}^n \to \mathbb{F}$ be a multiquadratic polynomial over $\mathbb{F}$ extending f. Also let $\mathcal{Y}_{\mathbb{F}} \in \mathbb{F}^n$ for each $\mathbb{F} \in \mathcal{F}$, and define $t = \sum_{\mathbb{F}\in\mathcal{F}} |\mathcal{Y}_{\mathbb{F}}|$.*

Then, there exists a subset $B \subseteq \{0,1\}^n$, with $|B| \leq t$, such that for all Boolean functions $f'\colon \{0,1\}^n \to \{0,1\}$ that agree with f on B, there exist multiquadratic polynomials $p'_{\mathbb{F}}\colon \mathbb{F}_n \to \mathbb{F}$ (one for each $\mathbb{F} \in \mathcal{F}$) such that

1. *$p'_{\mathbb{F}}$ extends f', and*
2. *$p'_{\mathbb{F}}(y) = p_{\mathbb{F}}(y)$ for all $y \in \mathcal{Y}_{\mathbb{F}}$.*

Proof. Call $z \in \{0,1\}^n$ *good* if for every $\mathbb{F} \in \mathcal{F}$ there exists a multiquadratic polynomial $u_{\mathbb{F},z}\colon \mathbb{F}^n \to \mathbb{F}$ such that

(a) $u_{\mathbb{F},z}(y) = 0$ for all $y \in \mathcal{Y}_{\mathbb{F}}$,

(b) $u_{\mathbb{F},z}(z) = 1$, and

(c) $u_{\mathbb{F},z} = 0$ for all $w \in \{0,1\}^n \setminus \{z\}$.

From Lemma 2.3.11, each $\mathbb{F} \in \mathcal{F}$ can prevent at most $|\mathcal{Y}_{\mathbb{F}}|$ points from being good. Since $t = |\mathcal{Y}_{\mathbb{F}}|$, there are at least $2^n - t$ good points.

Let G be the set of good points, and thus let $B = \{0,1\}^n \setminus G$ be the set of not-good points. Define

$$p'_{\mathbb{F}}(x) = p_{\mathbb{F}}(x) + \sum_{z\in G} (f'(z) - f(z)) u_{\mathbb{F},z}(x). \tag{2.12}$$

Now, all we need is to show that $p'_{\mathbb{F}}(x)$ satisfies the two conditions from the theorem statement.

First, we show that $p'_{\mathbb{F}}$ extends f'; that is, $p'_{\mathbb{F}}(x) = f'(x)$ for all $x \in \{0,1\}^n$. There are two cases here: $x \in G$ and $x \in B$. If $x \in B$, then the sum term of Equation (2.12) is 0; hence $p'_{\mathbb{F}}(x) = p_{\mathbb{F}}(x)$. Since $p_{\mathbb{F}}(x)$ extends $f(x)$, and since $f(x) = f'(x)$ on B, this means $p'_{\mathbb{F}}(x) = f'(x)$. If $x \in G$, then the only term of the sum where $u_{\mathbb{F},z}(x)$ is nonzero is where $x = G$, as per Items (b) and (c) above. Hence, we have

$$p'_{\mathbb{F}}(x) = p_{\mathbb{F}}(x) + f'(x) - f(x),$$

and since $p_{\mathbb{F}}(x) = f(x)$, it follows that $p'_{\mathbb{F}}(x) = f'(x)$.

Next, we show that $p'_{\mathbb{F}}(y)$ and $p_{\mathbb{F}}(y)$ agree for all $y \in \mathcal{Y}_{\mathbb{F}}$. Since by Item (a) above, we have that $u_{\mathbb{F},z}(y) = 0$, it follows that the entire sum term is zero. Therefore, $p'_{\mathbb{F}}(y) = p_{\mathbb{F}}(y)$ for all $y \in \mathcal{Y}_{\mathbb{F}}$.

As such, we have constructed a polynomial $p'_{\mathbb{F}}$ and a set B that satisfy our conditions of the theorem. $\square$

What this lemma tells us is that if we get a collection of extension polynomials to some function, for any collection of points we pick we can construct a "false" function f' and give it a false collection of extensions, such that anywhere on $\mathcal{Y}$, our "false" f' agrees with f. The terminology "false" may make more sense in the framework where we think about $\mathcal{Y}$ as being exactly the set of points we can test from $f_{\mathbb{F}}$ (i.e., we can see $f_{\mathbb{F}}(\mathcal{Y})$ but no other values of $f_{\mathbb{F}}$): in this context, we would be completely unable to tell the difference between $f_{\mathbb{F}}$ and $f'_{\mathbb{F}}$, but they would be different polynomials, and even stronger, f and f' would be different functions!

Next, we introduce an interesting lemma about the sum of multilinear monomials over a set. So far, we have seen a lot about how surprisingly flexible multilinear polynomials are: we can construct multilinear polynomials that can extend any Boolean function, we can construct polynomials that send any (smallish) set of points to zero but some particular other points to one, and more. But here, we can provide a restriction on the power of multilinear polynomials: no matter what we do, their sum over $\{\pm 1\}^n$ will always be zero.

Lemma 2.3.13 ([23, Lemma 7]). *Let $m(x_1, \ldots, x_n)$ be a multilinear monomial. Over a field of characteristic other than 2, we have*

$$\sum_{b \in \{-1,1\}^n} m(b) = 0. \tag{2.13}$$

Proof. For some x_i, we can write $m = x_i \cdot m'$, where the degree of x_i in m' is 0. Then

$$\begin{aligned}\sum_{b \in \{1,-1\}^n} m(b) &= \sum_{a \in \{-1,1\}} \sum_{b' \in \{1,-1\}^{n-1}} a \cdot m'(b') \\ &= \sum_{a \in \{-1,1\}} a \cdot \left(\sum_{b' \in \{1,-1\}^{n-1}} m'(b') \right) \\ &= \left(\sum_{b' \in \{1,-1\}^{n-1}} m'(b') \right) - \left(\sum_{b' \in \{1,-1\}^{n-1}} m'(b') \right) \\ &= 0.\end{aligned}$$

□

We note briefly that this proof only actually requires m to be linear in one variable; we have given the lemma as-is since it is still strong enough for our purposes and slightly easier to explain.

2.4 Statistics

In this paper, we will be dealing quite a bit with computers that have access to randomness. Because these computers now have access to randomness, their outputs are no longer deterministic: they can return different results depending on the exact rolls of their random dice. To talk about computers in this context, we will need to use a few ideas from statistics.

Definition 2.4.1. A *random variable* is a function from some *probability space* Ω to a set of outcomes O.

In this thesis, we will assume Ω is finite; this makes much of the statistics we plan to do easier. While random variables very frequently look like variables (and we will rather frequently treat them as such), they are actually functions. One particular example of this distinction is in applying functions to random variables: we will denote it as $\varphi(X)$, for some function φ and random variable X, but the result of this is itself a random variable whose actual value is $\varphi \circ X$.

Definition 2.4.2. An *event* is a subset of a sample space Ω.

While we define an event as a simple subset, we will often think of an event as being a condition on some random variable. If X is a random variable with probability space Ω and outcome set O, the event $X = o$ for some $o \in O$ is the set $\{X(\omega) = e \mid \omega \in \Omega\}$.

Definition 2.4.3. Let E be an event. The *probability* of E is

$$\mathbb{P}[E] = \frac{|E|}{|\Omega|}.$$

Next, we introduce the notion of statistical independence. In the abstract, two events are statistically independent if one event occurring does not affect the likelihood of the other occurring.

Definition 2.4.4. Two events are *statistically independent* if

$$\mathbb{P}[A \cap B] = \mathbb{P}[A]\mathbb{P}[B].$$

When we have a random variable, one of the things we would like to measure is what the average outcome is. We capture this value through the notion of an expected value.

Definition 2.4.5. The *expected value* of a random variable X is the value

$$\mathbb{E}[X] = \frac{1}{|\Omega|}\sum_{\omega\in\Omega} X(\omega).$$

Note that expected value requires the outcome set to be a field; by default we will assume it to be $\mathbb{R}$ in this paper, unless specified otherwise. Next, we give some nice properties of probability and expected value. First, Boole's inequality tells us that the probability of any one of a set of events occurring is no more than the sum of the probabilities of each of the individual events.

Theorem 2.4.6 (Boole's inequality)**.** *For any countable set of events $\{A_i\}$, we have*

$$\mathbb{P}\left(\bigcup_{i=1}^{\infty} A_i\right) \le \sum_{i=1}^{\infty} \mathbb{P}(A_i). \tag{2.14}$$

Next, Jensen's inequality tells us how concave-down functions relate to random variables.

Theorem 2.4.7 (Jensen's inequality)**.** *Let φ be a function that is concave down on its domain. Then, for any random variable X,*

$$\mathbb{E}[\varphi(X)] \le \varphi(\mathbb{E}[X]). \tag{2.15}$$

Now, we introduce a specific lemma about how independence relates to vector spaces. In essence, what we are planning to show is that linear independence is the same as statistical independence. To do this, first we must briefly introduce one piece of terminology: that of the restriction of a vector.

Definition 2.4.8. Let $V \subseteq \mathbb{F}^D$ be a vector space (with corresponding basis $\{e_i \mid i \in D\}$) and subset $S \subseteq D$. We define the *restriction* of V to S to be the set

$$V|_S = \operatorname{span}_V(\{e_i \mid i \in S\}). \tag{2.16}$$

Similarly, for a vector $v \in V$ we define the restriction $v|_S$ to be the vector $(v_i)_{i\in S} \in V|_S$.

Now, we may state our theorem.

Theorem 2.4.9 ([10, Claim 2])**.** *Let $\mathbb{F}$ be a finite field and D a finite set. Let $V \subseteq \mathbb{F}^D$ be a vector space, and let v be a uniform random variable over V. For any subdomains $S, S' \subseteq D$, the restrictions $v|_S$ and $v|_{S'}$ are statistically dependent if and only if there exist constants $c \in V|_S$ and $d \in V|_{S'}$ such that*

1. *There exists $w \in V$ such that $c \cdot w|_S \neq 0$, and*

2. *For all $w \in V$, $c \cdot w|_S = d \cdot w|_{S'}$.*

Proof. Let $x \in \mathbb{F}^S$ and $x' \in \mathbb{F}^{S'}$. Define

$$p_{x,x'} = \mathop{\mathbb{P}}_{v \in V}[v|_S \mid v|_S = x \text{ or } v|_{S'} = x']. \tag{2.17}$$

Further let $B \subseteq \mathbb{F}^D$ be a basis for V. Define

$$\begin{aligned} B_S &= \{b|_S \mid b \in B\} \\ B_{S''} &= \{b|_{S''} \mid b \in B\} \end{aligned}$$

Finally, define $B_{S,S'} \in M_{|S|+|S'|,d}(\mathbb{F})$ be the matrix where each row vector is $b|_S$ concatenated with $b|_{S'}$, for each $b \in B$. Hence,

$$p_{x,x'} = \mathop{\mathbb{P}}_{z \in \mathbb{F}^d}[B_{S,S'} \cdot z = (x, x')]. \tag{2.18}$$

For any matrix $A \in M_{m,n}(\mathbb{F})$,

$$\mathop{\mathbb{P}}_{z \in \mathbb{F}^n}[Az = b] = \begin{cases} |\mathbb{F}|^{-\operatorname{rank}(A)} & b \in \operatorname{img}(A) \\ 0 & \text{otherwise.} \end{cases} \tag{2.19}$$

If b is not in the image of A, then by the definition of the image there is no z such that $Az = b$. Otherwise, we know that there are $|\mathbb{F}|^{\operatorname{rank}(a)}$ elements in the image of A, and a random z has an equal chance of falling on any of them; hence the probability of Az being equal to b is the inverse of that.

Next, note that $\operatorname{img}(B_{S,S'}) \subseteq \operatorname{img}(B_S) \times \operatorname{img}(B_{S'})$, and by the definition of rank, these are equal if and only if $\operatorname{rank}(B_{S,S'}) = \operatorname{rank}(B_S) + \operatorname{rank}(B_{S'})$. Hence,

$$p_{x,x'} = \mathop{\mathbb{P}}_{v \in V}[v|_S = x] \mathop{\mathbb{P}}_{v \in V}[v|_{S'} = x'] \tag{2.20}$$

if and only if $\operatorname{rank}(B_{S,S'}) = \operatorname{rank}(B_S) + \operatorname{rank}(B_{S'})$. By the rank-nullity theorem, this is true if and only if $\operatorname{nullity}(B^T_{S,S'}) \subseteq \operatorname{nullity}(B^T_S) \times \operatorname{nullity}(B^T_{S'})$. Lastly, note that the two conditions in the theorem statement are true exactly when there exists $c \in \mathbb{F}^S$ and $d \in \mathbb{F}^{S'}$ such that $c \notin \operatorname{nullity}(B^T_S)$ but $(c, -d) \in \operatorname{nullity}(B^T_{S'})$. □

2.5 Error-correcting codes

Error-correcting codes are a concept with broad applications in both theoretical and practical computer science. An error-correcting code is some function applied to a string, such that any two elements of the image of that function are sufficiently far from each other (for some definition of "far" that we will see soon). These are called "error-correcting" because if a valid output is edited a relatively small amount, the large distance to other valid outputs means that the edited string is not valid, and what the original output was can still be guessed reasonably effectively, since it is highly likely to still be the closest valid string.

To work with error-correcting codes, we first need to define a notion of distance. Computer scientists use several notions of distance, all of which can be useful in different contexts. We will be using one of the simpler ones, because it is more mathematically elegant (as opposed to being more practically useful) and works well with the definitions we will be using in the rest of the text.

Definition 2.5.1 ([19]). Let $x, y \in \Sigma^n$ be strings of the same length. We say the *Hamming distance* between x and y is the value

$$\Delta(x, y) = \frac{|\{i \in [n] \mid x_i \neq y_i\}|}{n}.$$

Note that $\Delta(x, y) \in [0, 1]$ for any strings x and y. This normalization is not strictly necessary in general, since Δ is only defined between strings of equal length (and there do exist cases where it is much nicer to keep the distance as a natural number). In our case, however, we would like to keep the distances all in the same bounded range; hence we normalize.

1	0	0	0	1	0	0	1
1	1	0	0	0	1	0	1

Figure 2.4: Two strings with Hamming distance 3/8

The definition of Hamming distance between two strings also generalizes to the notion of distance from a set. Informally, we say the distance from a string to a set is simply the distance to the closest element of the set.

Definition 2.5.2. Let $\varepsilon > 0$. A vector $x \in \Sigma^n$ is *ε-far* from a set $S \subseteq \Sigma^n$ if

$$\min_{y \in S}(\Delta(x, y)) \geq \varepsilon.$$

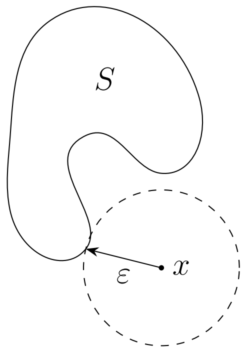

Figure 2.5: A point x that is ε-far from a set S

Now that we have some notions of distance, we can define a useful error-correcting code. Like with distance metrics, there are lots of useful error-correcting codes; we will pick one that has particularly nice mathematical properties.

Definition 2.5.3. Let $\mathbb{F}$ be a finite field, and let $k, r, m \in \mathbb{N}^+$. Then the set $\mathrm{RM}^k[\mathbb{F}, r, m]$ is

$$\mathrm{RM}^k[\mathbb{F}, r, m] = \left\{ \begin{aligned} F\colon \mathbb{F}^m &\to \mathbb{F}^k \\ x &\mapsto (p_1(x), \dots, p_k(x)) \end{aligned} \;\middle|\; p_i \in \mathbb{F}^{\leq d}[X_{1,\dots,m}] \right\}. \tag{2.21}$$

In a strict set-theoretic sense, this means $\mathrm{RM}^k[\mathbb{F}, r, m] = \left(\mathbb{F}^{\leq d}[X_{1,\dots,m}]\right)^k$; however, we will be referring to this set in a specific context: that of encoding values. More specifically, when we talk about $\mathrm{RM}^k[\mathbb{F}, r, m]$, we are often referring to a specific encoding of these polynomials as strings. Something to note is that while we have almost always been using the language $\{0, 1\}$ so far, here we will be using the language $\mathbb{F}^k$.

Definition 2.5.4. Let φ be an injection from a set S to $\mathrm{RM}^k[\mathbb{F}, r, m]$.Then the *Reed-Muller code* of a value $s \in S$ is the vector

$$(\varphi(s)(x))_{x \in \mathbb{F}^m} \in (\mathbb{F}^k)^{|\mathbb{F}|^m}. \tag{2.22}$$

Less formally, what this means is that for our input $s \in S$, we associate it with some polynomial F; then our encoding is the *entire evaluation table* of F over its domain. If we consider our alphabet to be $\mathbb{F}^k$ (i.e., the codomain of F), then the Hamming distance from F to any other arbitrary function $F'\colon \mathbb{F}^m \to \mathbb{F}^k$ is exactly the proportion of inputs x on which $F'(x) \neq F(x)$. The following lemma formalizes this notion.

Lemma 2.5.5. *Let $F \in \mathrm{RM}[\mathbb{F}, m, d]$, and let $F'\colon \mathbb{F}^m \to \mathbb{F}^k$, where F' is also laid out as an evaluation table, in the same order as F. Then, the Hamming distance $\Delta(F, F')$, when written over the alphabet $\mathbb{F}^k$, is*

$$\frac{|\{x \in \mathbb{F}^m \mid F(x) \neq F'(x)\}|}{|\mathbb{F}|^m}. \tag{2.23}$$

Proof. First, note that since our alphabet is exactly the codomain of F (and F'), each entry in the evaluation table has length 1. Hence, there is a bijection between inputs $x \in \mathbb{F}^m$ and indexes in the evaluation table. Further, two items in the table are equal if and only if $F(x) = F'(x)$. Hence, the total number of distinct values in their representations is exactly $|\{x \in \mathbb{F}^m \mid F(x) \neq F(x)\}|$. The total length of both evaluation tables is exactly the number of inputs, i.e. $|\mathbb{F}^m| = |\mathbb{F}|^m$. Hence, by the definition of the Hamming distance, Equation (2.23) is true. □

Corollary 2.5.6. *Let $F \in \mathrm{RM}[\mathbb{F}, m, d]$, and let $F'\colon \mathbb{F}^m \to \mathbb{F}^k$. Then,*

$$\Delta(F, F') = \mathbb{P}[F(x) \neq F'(x)]. \tag{2.24}$$

Proof. The probability over x that $F(x) \neq F'(x)$ is exactly the number of inputs in which $F(x) \neq F'(x)$, divided by the number of values x can be. We know x can be any of $|\mathbb{F}|^m$ choices, and by definition the number of values of x where $F(x) \neq F'(x)$ is $|\{x \in \mathbb{F}^m \mid F(x) \neq F(x)\}|$. Hence, we get that Equation (2.23) is exactly $\mathbb{P}[F(x) \neq F'(x)]$, and thus by Lemma 2.5.5, Equation (2.24) is true. □

We will be using Reed-Muller codes as a way to "space out" our values—that is, we will be using Corollary 2.5.6 to our advantage when we deal with proofs of proximity. We will talk about these in much more depth in Section 4.5.1, but the important point is that these have restrictions on how close different valid inputs can be to each other, and we will want to use these to help keep them relatively far apart.

Chapter 3

Relativization and algebrization

An important prerequisite to understanding algebrization is the similar, but simpler, concept of *relativization*, also called *oracle separation*. To do this, we first must define an *oracle*.

Definition 3.0.1 ([1, Def. 2.1])**.** An *oracle* A is a collection of Boolean functions $A_m\colon \{0,1\}^m \to \{0,1\}$, one for each natural number m.

There are several ways to think of an oracle; this will extend the most naturally when it comes time to define an extension oracle in Definition 3.5.1.

Another way to think of an oracle is as a subset $A \subseteq \{0,1\}^*$. This allows us to think of A as a language. Since we can do this, it gives us the ability to think of the complexity of the oracle. If we want to think about the subset in terms of our functions, we can write A as

$$A = \bigcup_{m\in\mathbb{N}} \{x \in \{0,1\}^m \mid A_m(x) = 1\}. \tag{3.1}$$

We will use the Iverson bracket defined in Definition 2.3.1 for this purpose: allowing us to think of A as the set and $[A]$ as the function.

The third way to think of an oracle is as a list of bits—this is how a Turing machine thinks of an oracle. In this context, we consider the oracle to be a string of 2^n bits b_i, where b_i is the result of A when given the binary representation of i as input. We will mostly not think of A this way explicitly, but for practical purposes this is how an oracle is encoded whenever we pass it to a Turing machine as an input.

Example 3.0.1.1. Let $m = 3$. The function

$$\begin{aligned} f\colon \{0,1\}^3 &\to \{0,1\} \\ abc &\mapsto b \end{aligned} \tag{3.2}$$

is an oracle function. We can think of f as corresponding to the set $\{010, 011, 110, 111\}$.

Example 3.0.1.2. For each $n \in \mathbb{N}$, define

$$\begin{aligned} f_n\colon \{0,1\}^n &\to \{0,1\} \\ a_1a_2\cdots a_n &\mapsto a_n. \end{aligned} \tag{3.3}$$

Then the set $\{f_n\}$ forms an oracle, whose corresponding language is the set of all binary representations of odd numbers.

An oracle is not particularly interesting mathematical object on its own (after all, it is simply a set of arbitrary Boolean functions); its utility comes from when it interacts with a Turing machine. A normal Turing machine does not have the facilities to interact with an oracle, so we need to define a small extension to a standard Turing machine to allow for this.

Definition 3.0.2 ([2, Def. 3.6])**.** A *Turing machine with an oracle* is a Turing machine with an additional tape, called the *oracle tape*, as well as three special states: q_{query}, q_{yes}, and q_{no}. Further, each machine is associated with an oracle A. During the execution of the machine, if it ever moves into the state q_{query}, the machine then (in one step) takes the output of A on the contents of the oracle tape, moving into q_{yes} if the answer is 1 and q_{no} if the answer is 0.

Of course, the question now becomes how we can effectively use an oracle in an algorithm. The previously-mentioned conception of an oracle as a set of strings is useful here. If we consider the set of strings as being a *language* in its own right, then querying the oracle is the same as determining whether a string is in the language, just in one step. If the language is computationally hard, this means our machine can get a significant power boost from the right oracle.

$\mathcal{O}(0)$	$\mathcal{O}(1)$	$\mathcal{O}(00)$	$\mathcal{O}(01)$	$\mathcal{O}(10)$	$\mathcal{O}(11)$	$\mathcal{O}(000)$	$\cdots$

Figure 3.1: Bit representation of an oracle

Definition 3.0.3 ([1, Def. 2.1])**.** For any complexity class $\mathcal{C}$, the complexity class $\mathcal{C}^A$ is the class of all languages determinable by a Turing machine with access to A using the amount of resources defined for $\mathcal{C}$.

We will be using this definition in many places, so we should take a moment to look at it in more depth. First, it is important to realize that $\mathcal{C}^A$ is a set of *languages*, not *machines*: despite the notation, augmenting $\mathcal{C}$ with an oracle does not modify any languages, it just adds new ones that are computable. Second, since a machine can always ignore its oracle, it follows that adding an oracle can only increase the number of languages in the class, never decrease it.

Lemma 3.0.4. *For any complexity class $\mathcal{C}$ and oracle A, $\mathcal{C} \subseteq \mathcal{C}^A$.*

Proof. Let $L \in \mathcal{C}$ and M be a machine that determines L. Then the oracle machine M' that simulates M on its input and makes no queries to the oracle will also accept exactly L. Since M' is a $\mathcal{C}^A$ machine for any oracle A, it follows that $L \in \mathcal{C}^A$ and hence $\mathcal{C} \in \mathcal{C}^A$. $\square$

While the above lemma tells us that $\mathcal{C} \subseteq \mathcal{C}^A$ always, another interesting question is when $\mathcal{C} = \mathcal{C}^A$. We do have a notion for this, called *lowness*. Lowness can be defined for both individual languages and complexity classes; we will define both here.

Definition 3.0.5. A language L is *low* for a class $\mathcal{C}$ if $\mathcal{C}^L = \mathcal{C}$. A complexity class $\mathcal{D}$ is *low* for a class $\mathcal{C}$ if each language in $\mathcal{D}$ is low for $\mathcal{C}$.

Of particular interest to us will be classes that are low for *themselves*. We care about these classes because they can use other problems from the same class as a subroutine without issue; in particular recursion and iteration both work here. Thankfully, both P and PSPACE are low for themselves (it turns out NP is probably not); this allows us to easily write algorithms that recurse for classes in both of our most common classes.

Theorem 3.0.6. P *and* PSPACE *are low for themselves.*

Proof. Let $L \in \mathsf{P}$ and let $K \in \mathsf{P}^L$. Let $M(L)$ be a determiner for L and $M(K)$ be a determiner for K. Further, let $\hat{M}(K)$ be a determiner of K, but with access to L as an oracle. Our goal is to show that $K \in \mathsf{P}$.

Let $p_L(n)$ be a polynomial upper bound of the runtime of $M(L)$ on an input of length n, and let $p_{\hat{K}}(n)$ be similar. Since $M(K)$ can call $M(L)$ no more than $p_{\hat{K}}(n)$ times, it follows that $p_K(n) \leq p_{\hat{K}}(p_L(n))$. Hence, the runtime of $M(K)$ is bounded above by a polynomial, and thus $K \in \mathsf{P}$.

The proof for PSPACE is very similar to that of P, but with space instead of time. Since memory usage is bounded above by some polynomial, and polynomials are closed under composition, it follows that PSPACE is low for itself. □

3.1 Defining relativization

We are now ready to define what relativization is. First, note that relativization is a statement about a *result*: we talk about inclusions relativizing, not sets themselves.

Definition 3.1.1. Let $\mathcal{C}$ and $\mathcal{D}$ be complexity classes such that $\mathcal{C} \subseteq \mathcal{D}$. We say the result $\mathcal{C} \subseteq \mathcal{D}$ *relativizes* if $\mathcal{C}^A \subseteq \mathcal{D}^A$ for all oracles A. Conversely, if there exists A such that $\mathcal{C} \nsubseteq \mathcal{D}$, we say that the result $\mathcal{C} \subseteq \mathcal{D}$ *does not relativize.*

Definition 3.1.2. Let $\mathcal{C}$ and $\mathcal{D}$ be complexity classes such that $\mathcal{C} \nsubseteq \mathcal{D}$. We say the result $\mathcal{C} \nsubseteq \mathcal{D}$ *relativizes* if $\mathcal{C}^A \nsubseteq \mathcal{D}^A$ for all oracles A. Conversely, if there exists A such that $\mathcal{C} \subseteq \mathcal{D}$, we say that the result $\mathcal{C} \nsubseteq \mathcal{D}$ *does not relativize.*

We start with a very straightforward example of a relativizing result.

Lemma 3.1.3. *For any oracle A,* $\mathsf{P}^A \subseteq \mathsf{NP}^A$. *Equivalently, the result* $\mathsf{P} \subseteq \mathsf{NP}$ *relativizes.*

Proof. Since any deterministic Turing machine is also a nondeterministic machine, it follows that a machine that solves a P^A problem is also an NP^A machine. Hence, $\mathsf{P}^A \subseteq \mathsf{NP}^A$. □

This result tells us that not *everything* is weird in the world of relativization (although we will soon do our best to find all the weird bits): if we have a machine that can do more operations without an oracle, it can still do more operations with an oracle. Further, for the question of P vs. NP that we will discuss in Section 3.3, this means that the question we care about is whether $\mathsf{NP} \subseteq^? \mathsf{P}$ relativizes. As such, the question we are asking simplifies to determining where $\mathsf{P}^A = \mathsf{NP}^A$ and where $\mathsf{P}^A \subsetneq \mathsf{NP}^A$.

Now that we have talked about set inclusions relativizing, we need to define the other side of the coin: *proofs* can relativize as well as results. Unfortunately, this needs to be a somewhat informal definition as formally delineating different types of proof is far beyond the scope of this paper. However, the definition we offer here will be sufficient for our purposes.

Definition 3.1.4. We say a *proof relativizes* if it is not made invalid if the relevant classes are replaced with oracle classes, i.e., a proof that $\mathcal{C} \subseteq \mathcal{D}$ *relativizes* if the same proof can be used to show $\mathcal{C}^A \subseteq \mathcal{D}^A$ for all oracles A with minimal modifications.

This gives us a reason to care about relativization as a concept: if our proofs are relativizing then we know not to try to use them to prove nonrelativizing results. In particular, we will show in Section 3.3 that the famous P vs. NP problem will not relativize regardless of the outcome, and then in Section 3.4 we will show that the common proof technique of diagonalization *does* in fact relativize.

3.2 Query complexity

Now that we have given ourselves a reason to care about oracles and how they interact with Turing machines, we now turn to the question of how a machine can gain information about the oracle it queries. We will do this with the notion of *query complexity*.

The goal of query complexity is to ask questions about some Boolean function $A\colon \{0,1\}^n \to \{0,1\}$ by querying A itself. For this, we will interchangeably think of A as a *function* as well as a bit string of length $N = 2^n$, where each string element is A applied to the ith string of length n, arranged in some lexicographical order.

We can further think of the property itself as being a Boolean function; a function that takes as input the bit-string representation of A and outputs whether or not A has the given property. We will call the function representing the property f. When viewed like this, f is a function from $\{0,1\}^N$ to $\{0,1\}$. We define three types of query complexity for three of the most common types of computing paradigms: deterministic, randomized, and quantum. Nondeterministic query complexity is interesting, but it is outside the scope of this paper.

Definition 3.2.1 ([1, p. 17])**.** Let $n \in \mathbb{N}$ and let $A\colon \{0,1\}^n \to \{0,1\}$ be an oracle. We can write A using 2^n bits, where bit i is the output of A when given the binary representation of i as input. Define $N = 2^n$, and let $f\colon \{0,1\}^N \to \{0,1\}$ be a Boolean function. Then the *deterministic query complexity* of f, which we write $D(f)$, is the minimum number of queries made by any deterministic algorithm with access to an oracle A that determines the value of $f(A)$.

To make this more clear, let us give an example problem.

Definition 3.2.2. The OR problem is the following oracle problem:

> Let $A\colon \{0,1\}^n \to \{0,1\}$ be an oracle. The function $\mathsf{OR}(A)$ returns 1 if there exists a string on which A returns 1, and 0 otherwise.

The question is then what the deterministic query complexity of the OR function is.

Theorem 3.2.3. *The* OR *problem has a deterministic query complexity of* 2^n.

Proof. First, note that any algorithm that determines the OR problem can stop as soon as it queries A and gets an output of 1. Hence, for any algorithm M, let $\{s_i\}$ be the sequence of queries M makes to A on the assumption that it always receives a response of 0. If $|\{s_i\}| \leq 2^n$, there exists some $s \in \{0,1\}^n$ not queried. In that case, M will not be able to distinguish the zero oracle from the oracle that outputs 1 only when given s. Hence, M must query every string of length n and thus the query complexity is 2^n. □

From this, we get that the OR problem cannot be solved any better than by enumerative checking. This makes intuitive sense because none of the results we get by querying A imply anything about what A will do on other values, since A can be an arbitrary function. In Section 3.6, we will look at what happens when we give ourselves access to a *polynomial*, where querying one point could tell us information about others.

For the next two definitions, since their Turing machines include some element of randomness, we only require that they succeed with a 2/3 probability. This is in line with most definitions of complexity classes involving random computers.

Definition 3.2.4 ([1, p. 17])**.** Let $f\colon \{0,1\}^N \to \{0,1\}$ be a Boolean function. Then the *randomized query complexity* of f, which we write $D(f)$, is the minimum number of queries made by any randomized algorithm with access to an oracle A that evaluates $f(A)$ with probability at least 2/3.

3.3 Relativization of P vs. NP

An important example of relativization is that of P and NP. While the question of if $\mathsf{P} = \mathsf{NP}$ is still open, we aim to show that *regardless of the answer*, the result does not relativize. To do this, we show that there are some oracles A where $\mathsf{P}^A = \mathsf{NP}^A$, and some where $\mathsf{P}^A \neq \mathsf{NP}^A$.

Additionally, it should be noted that the similarity of relativization to algebrization means that the structure of these proofs will return in Section 3.7 when we show the algebrization of P and NP.

The more straightforward of the two proofs is the oracle where $\mathsf{P}^A = \mathsf{NP}^A$, so we shall begin with that.

Theorem 3.3.1 ([6, Theorem 2])**.** *There exists an oracle A such that $\mathsf{P}^A = \mathsf{NP}^A$.*

Proof. For this, we can let A be any PSPACE-complete language. By letting our machine in P be the reducer from A to any other language in PSPACE, we therefore get that $\mathsf{PSPACE} \subseteq \mathsf{P}^A$. Similarly, if we have a problem in NP^A, we can verify it in polynomial space without talking to A at all (by having our machine include a determiner for A). Hence, we have that $\mathsf{NP}^A \subseteq \mathsf{NPSPACE}$. Further, a celebrated result of Savitch [29] (which we briefly discussed as Theorem 2.2.26) is that $\mathsf{PSPACE} = \mathsf{NPSPACE}$. Combining all these results, we get the chain

$$\mathsf{NP}^A \subseteq \mathsf{NPSPACE} = \mathsf{PSPACE} \subseteq \mathsf{P}^A \subseteq \mathsf{NP}^A. \tag{3.4}$$

This is a circular chain of subset relations, which means everything in the chain must be equal. Hence, $\mathsf{P}^A = \mathsf{NP}^A = \mathsf{PSPACE}$. □

For a slightly more intuitive view of what this proof is doing, what we have done is found an oracle that is so powerful that it dwarfs any amount of computation our actual Turing machine can do. Hence, the power of our machine is really just the same as the power of our oracle, and since we have given both the P and NP machine the same oracle, they have the same power.

Having shown that an oracle exists where $\mathsf{P}^A = \mathsf{NP}^A$, we now endeavor to find one where $\mathsf{P}^A \neq \mathsf{NP}^A$. This piece of the proof is less simple than the previous section, and it uses a diagonalization argument to construct the oracle. Before we dive in to the main proof, however, we need to define a few preliminaries.

Definition 3.3.2 ([6, p. 436])**.** Let X be an oracle. The language $L(X)$ is the set

$$L(X) = \{x \mid \text{there is } y \in X \text{ such that } |y| = |x|\}.$$

For example, consider the language $X = \{0, 11, 0100\}$. The language $L(X)$ is the language consisting of all strings of length 1, 2, and 4.

Our eventual goal will be to construct a language X such that $L(X) \in \mathsf{NP}^X \setminus \mathsf{P}^X$. Of particular note is that we can rather nicely put a upper bound on the complexity of $L(X)$ when given X as an oracle, regardless of the value of X. This fact is what gives us the freedom to construct X in such a way that $L(X)$ will not be in P^X.

Lemma 3.3.3 ([6, p. 436])**.** *For any oracle X, $L(X) \in \mathsf{NP}^X$.*

Proof. Let S be a string of length n. If $S \in L(X)$, then a witness for S is any string S' such that $|S| = |S'|$ and $S' \in X$. Since a machine with query access to X can query whether S' is in X in one step, it follows that we can verify that $S \in L(X)$ in polynomial time. □

With this lemma as a base, we can now move on to our main theorem.

Theorem 3.3.4 ([6, Theorem 3])**.** *There exists an oracle A such that $\mathsf{P}^A \neq \mathsf{NP}^A$.*

Proof. Our goal is to construct a set B such that $L(B) \notin \mathsf{P}^B$. We shall construct B in an interactive manner. We do this by taking a sequence $\{P_i\}$ of all machines that recognize some language in P^A, and then constructing B such that for each machine in the sequence, there is some part of $L(B)$ it cannot recognize. This technique is called *diagonalization*, and it is used in many places in computer science theory.[1] Additionally, we define $p_i(n)$ to be the maximum running time of P_i on an input of length n. We aim to show that Algorithm 3.1 constructs B.

```
    Input: A sequence of P oracle machines $\{P_i\}_{i=1}^{\infty}$
    Output: A set $B$ such that $L(B) \notin \mathsf{P}^B$
 1  $B(0) \leftarrow \varnothing$;
 2  $n_0 \leftarrow 0$;
 3  for $i$ starting at 1 do
 4      Let $n > n_i$ be large enough that $p_i(n) < 2^n$;
 5      Run $P_i^{B(i-1)}$ on input $0^n$;
 6      if $P_i^{B(i-1)}$ rejects $0^n$ then
 7          Let $x$ be a string of length $n$ not queried during the above computation;
 8          $B(i) \leftarrow B(i-1) \sqcup \{x\}$;
 9      end
10      $n_{i+1} \leftarrow 2^n$;
11  end
12  $B \leftarrow \bigcup_i B(i)$;
```

Algorithm 3.1: An algorithm for constructing B

To begin, let us demonstrate the algorithm's soundness. First, note that since P_i runs in polynomial time, $p_i(n)$ is bounded above by a polynomial, and hence there will always exist an n as defined in line 4. Next, since there are 2^n strings of length n and since $p_i(n) < 2^n$, we know that there must be some x to make line 7 well-defined. While our algorithm allows x to be any string, if it is necessary to be explicit in which we choose, then picking x to be the smallest string in lexicographic order is a standard choice.

We should also briefly mention that this algorithm does not terminate. This is okay because we are only using it to construct the set B, which does not need to be bounded. If this were to be made practical, since the sequence of n_is is monotonically increasing, the set could be constructed "lazily" on each query by only running the algorithm until n_i is greater than the length of the query.

Next, we demonstrate that $L(B) \notin \mathsf{P}^B$. The end goal of our instruction is a set B such that if P_i^B accepts 0^n then there are no strings of length n in B, and if P_i^B rejects, then there is a string of length n in B. This means that no P_i accepts $L(B)$, and hence $L(B) \notin \mathsf{NP}^B$.

The central idea behind the proper functioning of our algorithm is that adding

[1]This argument style is named after *Cantor's diagonal argument*, which was originally used to prove that the real numbers are uncountable [28, Thm. 2.14].

strings to our oracle *cannot change the output if they are not queried*. This is what we do in line 4: we need our input length to be long enough to guarantee that a non-queried string exists. Since the number of queried strings is no greater than $p_i(n)$, and there are 2^n strings of length n, there must be some string not queried.

Next, we run $P_i^{B(i-1)}$ on all the strings we have already added. If it accepts, then we want to make sure that no string of length n is in B; that is, 0^n is not in $L(B)$. Hence, in this particular loop we add nothing to $B(i)$. If $P_i^{B(i-1)}$ rejects, we then need to make sure that $0^n \in L(B)$ but in a way that does not affect the output of $P_i^{B(i-1)}$. Hence, we find a string that $P_i^{B(i-1)}$ did not query (and thus will not affect the result) and add it to $B(i)$.

Having done this, we then set n_{i+1} to be 2^n. Since $p_i(n) < 2^n$, it follows that no previous machine could have queried any strings of length n_{i+1}.[2] This way, we ensure our previous machines do not accidentally have their output change due to us adding a string they queried.

Having run this over all polynomial-time Turing machines, we have a set $L(B)$ such that no machine in P^B accepts it, which tells us $L(B) \notin \mathsf{P}^B$. But, Lemma 3.3.3 already told us $L(B) \in \mathsf{NP}^B$. Hence, $\mathsf{P}^B \neq \mathsf{NP}^B$. □

3.4 Consequences on proof techniques

Of course, determining that P vs NP does not relativize is only important if the proof techniques used in practice *do* in fact relativize. Rather unfortunately, it turns out that simple diagonalization is a relativizing result.

While diagonalization itself does not have a formal definition, we can still think about it informally. Looking at our construction of B, which we did using diagonalization, notice that our definition never really cared about how the P_i worked, just about the results it produced. Hence, if it were to be possible to modify Algorithm 3.1 to construct $B \in \mathsf{NP} \setminus \mathsf{P}$, the proof would remain the same if we were to replace our sequence $\{P_i\}$ with a sequence of machines in P^A for some PSPACE-complete A. However, this would lead to a contradiction, as we showed in Theorem 3.3.1 that in that case, $\mathsf{P}^A = \mathsf{NP}^A$! This tells us that a simple diagonalization argument would not suffice to determine separation between P and NP.

While we know that diagonalization relativizes, in the years since the Baker, Gill, and Solovay paper researchers have discovered proof techniques that do not in fact relativize. One of these techniques is *arithmetization*, introduced by [5].

The idea behind arithmetization is that we want to be able to reduce computational problems to algebraic ones. More specifically, we would like to reduce our problems to ones involving low-degree polynomials over a finite field (such as those seen in Section 2.3). In this paper, we will care about arithmetization for two reasons: because it is a non-relativizing technique (as we are about to see) and because we will be using it later on in this paper as an important part of several proofs.

[2] A word of caution: we only care about what P_i does on input n_i, *not any other input*. This is because we only need each machine to be incorrect for some i, not all i.

3.5 Algebrization

Algebrization, originally described by Aaronson and Wigderson [1], is an extension of relativization. While relativization deals with oracles that are Boolean functions, algebrization extends oracles to be a collection of polynomials over finite fields. Since any field contains the set $\{0,1\}$, we can think about our new oracles as *extending* some specific oracle A, so that both oracles agree on the set $\{0,1\}^n \subseteq \mathbb{F}^n$. We formalize this notion below.

Definition 3.5.1 ([1, Def. 2.2]). Let $A_m \colon \{0,1\}^m \to \{0,1\}$ be a Boolean function and let $\mathbb{F}$ be a finite field. Also, given an oracle $A = (A_m)$, an *extension* $\tilde{A}$ of A is a collection of polynomials $\tilde{A}_{m,\mathbb{F}} \colon \mathbb{F}^m \to \mathbb{F}$, one for each positive integer m and finite field $\mathbb{F}$, such that

1. $\tilde{A}_{m,\mathbb{F}}$ is an extension polynomial of A_m for all $m, \mathbb{F}$, and
2. there exists a constant c such that $\tilde{A}_{m,\mathbb{F}} \in \mathbb{F}[X^{\leq c}_{1,\ldots,n}]$ for all $m, \mathbb{F}$.

Take note that an oracle can have many different extension oracles, since one can construct an infinite number of polynomials that go through a set of points. For this reason, when dealing with oracles in practice, we will also often be interested in oracles of a particular multidegree, which limits our options for oracles in potentially-interesting ways.

Example 3.5.1.1. Consider the function we defined in Example 3.0.1.1:

$$\begin{aligned} f \colon \{0,1\}^3 &\to \{0,1\} \\ abc &\mapsto b. \end{aligned} \tag{3.5}$$

An extension of that function is the polynomial

$$\begin{aligned} \tilde{f} \colon \mathbb{F}^3 &\to \mathbb{F}^3 \\ (a,b,c) &\mapsto b. \end{aligned} \tag{3.6}$$

While this is a relatively trivial polynomial, there are more non-trivial ones, for example

$$\begin{aligned} \tilde{f} \colon \mathbb{F}^3 &\to \mathbb{F}^3 \\ (a,b,c) &\mapsto a^3c^3 + b^2 - ac. \end{aligned} \tag{3.7}$$

Notice that on $\{0,1\}$, $x^2 = x$, which allows us to see that $\tilde{f}$ is a valid extension of f.

Definition 3.5.2 ([1, Def. 2.2]). For any complexity class $\mathcal{C}$ and extension oracle $\tilde{A}$, the complexity class $\mathcal{C}^{\tilde{A}}$ is the class of all languages determinable by a Turing machine with access to $\tilde{A}$ with the requirements for $\mathcal{C}$.

Next, we need to formally define what algebrization is.

Definition 3.5.3 ([1, Def. 2.3]). Let $\mathcal{C}$ and $\mathcal{D}$ be complexity classes such that $\mathcal{C} \subseteq \mathcal{D}$. We say the result $\mathcal{C} \subseteq \mathcal{D}$ *algebrizes* if $\mathcal{C}^A \subseteq \mathcal{D}^{\tilde{A}}$ for all oracles A and finite field extensions $\tilde{A}$ of A. Conversely, if there exists A and $\tilde{A}$ such that $\mathcal{C} \nsubseteq \mathcal{D}$, we say that the result $\mathcal{C} \subseteq \mathcal{D}$ *does not algebrize.*

Definition 3.5.4 ([1, Def. 2.3])**.** Let $\mathcal{C}$ and $\mathcal{D}$ be complexity classes such that $\mathcal{C} \not\subseteq \mathcal{D}$. We say the result $\mathcal{C} \not\subseteq \mathcal{D}$ *algebrizes* if $\mathcal{C}^A \not\subseteq \mathcal{D}^{\tilde{A}}$ for all oracles A and finite field extensions $\tilde{A}$ of A. Conversely, if there exists A and $\tilde{A}$ such that $\mathcal{C} \subseteq \mathcal{D}$, we say that the result $\mathcal{C} \not\subseteq \mathcal{D}$ *does not algebrize.*

3.6 Algebraic query complexity

Similarly to how we defined query complexity in Section 3.2, our notion of algebrization requires a definition of *algebraic* query complexity.

Definition 3.6.1 ([1, Def. 4.1])**.** Let $f\colon \{0,1\}^N \to \{0,1\}$ be a Boolean function, $\mathbb{F}$ be a field, and c be a positive integer. Also, let $\mathbb{M}$ be the set of deterministic algorithms M such that $M^{\tilde{A}}$ outputs $f(A)$ for every oracle $A\colon \{0,1\}^n \to \{0,1\}$ and every finite field extension $\tilde{A}\colon \mathbb{F}^n \to \mathbb{F}$ of A with $\mathrm{mdeg}(\tilde{A}) \leq c$. Then, the *deterministic algebraic query complexity* of f over $\mathbb{F}$ is defined as

$$\tilde{D}_{\mathbb{F},c}(f) = \min_{M\in\mathcal{M}} \left(\max_{A,\tilde{A}:\mathrm{mdeg}(\tilde{A})\leq c} T_M(\tilde{A}) \right), \tag{3.8}$$

where $T_M(\tilde{A})$ is the number of queries to $\tilde{A}$ made by $M^{\tilde{A}}$.

Our goal here is to find the *worst*-case scenario for the *best* algorithm that calculates the property f. The difference between this and Definition 3.2.1 is twofold: first, our algorithm M has access to an extension oracle of A, and second, that we can limit our $\tilde{A}$ in its maximum multidegree. For the most part, we will focus on equations with multidegree 2, which is enough to get the results we want.

As an example, let us look at the same OR problem we defined in Definition 3.2.2.

Theorem 3.6.2 ([1, Thm. 4.4])**.** *$\tilde{D}_{\mathbb{F},2}(\mathsf{OR}) = 2^n$ for every field $\mathbb{F}$.*

Proof. First note that 2^n is an upper bound for the number of queries necessary since we can query every point in $\{0,1\}^n$, of which there are 2^n.

Let M be a deterministic algorithm and let $\mathcal{Y}$ be the set of points queried by M in the case where M always receives 0 as a response. So long as $|\mathcal{Y}| < 2^n$, there exists by Lemma 2.3.11 a multiquadratic extension polynomial $\tilde{A}\colon \mathbb{F}^n \to \mathbb{F}$ such that $\tilde{A}(y) = 0$ for all $y \in \mathcal{Y}$ but $\tilde{A}(w) = 1$ for some $w \in \{0,1\}^n$. As such, if M queries less than 2^n points then it would not be able to tell the difference between $\tilde{A}$ and the zero function. However, $\mathsf{OR}(A) = 1$ and $\mathsf{OR}(0) = 0$, so it would get the incorrect answer for one of them. Hence if M queries fewer than 2^n points it cannot solve the OR problem. □

Note that this works even if M is adaptive: if M ever receives a nonzero response it (correctly) knows $\mathsf{OR}(A) = 1$, so it can accept immediately. As such, we know that any contradiction must come when M has only ever seen zeros as responses.

This gives us a potentially counterintuitive property of algebraic query complexity: while it would seem that giving our machine a polynomial (and a polynomial of multidegree only 2, at that) would give us the ability to solve the hardest problems more quickly, that turns out not to be the case.

Now, while this is true for polynomials of multidegree 2, it turns out that if we restrict our oracles to being simply *multilinear* polynomials, we do get a speedup.

Theorem 3.6.3 ([23, Thm. 3])**.** $\tilde{D}_{\mathbb{F},1}(\mathsf{OR}) = 1$ *for every field* $\mathbb{F}$ *with characteristic not equal to* 2*.*

Proof. Let $A\colon \{0,1\}^n \to \{0,1\}$ and $\tilde{A}$ be our extension polynomial. Consider the value of $p(1/2,\ldots,1/2)$. We aim to show that this value is equal to 0 if and only if A is the zero oracle.

Consider the function

$$p'(x_1,\ldots,x_n) = p(1-2x_1,\ldots,1-2x_n). \tag{3.9}$$

Since $1-2x$ is a linear polynomial, it follows that p' is itself a multilinear polynomial. Further, since the sum over $\{1,-1\}^n$ of a non-constant multilinear monomial is 0 as per Lemma 2.3.13, it follows that

$$\sum_{b\in\{-1,1\}^n} p'(b) = p'(0,\ldots,0), \tag{3.10}$$

i.e., the constant term of p'. Further, from our definition of p', we have that $p'(0,\ldots,0) = p(1/2,\ldots,1/2)$. Hence, we have

$$\sum_{b\in\{0,1\}^n} p(b) = p(1/2,\ldots,1/2). \tag{3.11}$$

Since $p(b) \geq 0$ for all $b \in \{0,1\}^n$, it follows that $p(1/2,\ldots,1/2)$ is 0 if and only if $p(b) = 0$ for all $b \in \{0,1\}^n$, i.e. exactly when A is the zero function. □

3.7 Algebrization of P vs. NP

As with relativization, an important application of algebrization is in regards to the P vs. NP problem.

In order to work with algebrization, first we need a stronger definition than completeness.

Definition 3.7.1 ([4, Def. 6.1])**.** A language L is PSPACE-*robust* if $\mathsf{P}^L = \mathsf{PSPACE}^L$.

The idea behind robustness is that a robust language is one that is complex enough that PSPACE algorithms that use it as an oracle can not glean any more information than a P algorithm with the same oracle. As a side note, the existence of at least one PSPACE-robust language L implies that the statement P = PSPACE (which is probably true but has not been proven) is non-relativizing.

Lemma 3.7.2. *Any* PSPACE-*complete language is also* PSPACE-*robust.*

Proof. First, we know from Lemma 3.0.4 that $\mathsf{P}^L \subseteq \mathsf{PSPACE}^L$. Next, let $M \in \mathsf{PSPACE}^L$, and we aim to show $M \in \mathsf{P}^L$. Since $L \in \mathsf{PSPACE}$ and PSPACE is low for itself, we know $M \in \mathsf{PSPACE}$. As such, we know there is a polynomial-time reduction f from M to L. Hence, we can compute M by running f on the input and then testing if that output is in L (using the oracle). Hence, $M \in \mathsf{P}^L$ and thus $\mathsf{P}^L = \mathsf{PSPACE}^L$. □

Lemma 3.7.3 ([4, Lemma 6.2])**.** *Let L be a PSPACE-robust language, with corresponding oracle A. Let $\tilde{A}$ be the unique multilinear extension oracle of A. Then the language*

$$\tilde{L} = \bigcup_{n\in\mathbb{N}} \{(x_1, \dots, x_n, z) \in \mathbb{F}^{n+1} \mid \tilde{A}(x_1, \dots, x_n) = z\} \tag{3.12}$$

is polynomially-equivalent to L; that is, $\tilde{L} \in \mathsf{P}^L$ and $L \in \mathsf{P}^{\tilde{L}}$.

To improve clarity, we present a novel proof of this statement; this differs substantially from that originally presented in [4].

Proof. First, we provide a polynomial-time reduction from L to $\tilde{L}$. Since for all $x \in \{0,1\}^n$, $\tilde{A}(x) = 1$ if and only if $x \in L$, it follows that

$$\begin{aligned} f\colon \Sigma^* &\to \Sigma^* \\ x &\mapsto (x, 1) \end{aligned} \tag{3.13}$$

is a polynomial-time reduction from L to L'.

```
Input: $(x_1, \dots, x_n, z) \in \mathbb{F}^{n+1}$
Output: Whether $\tilde{A}(x_1, \dots, x_n) = z$
$z' \leftarrow 0$;
for $k \in \{0,1\}^n$ do
    Simulate $L$ on input $k$;
    if $k \in L$ then
        // Compute $d_k(x)$
        $d \leftarrow 1$;
        for $i$ from 1 to $n$ do
            if $k_i = 1$ then
                $d \leftarrow d \cdot x_i$;
            else
                $d \leftarrow d \cdot (1 - x_i)$;
            end
        end
        $z' \leftarrow z' + d$;
    end
end
return whether $z = z'$;
```

Algorithm 3.2: Determiner for $\tilde{L}$

Next, consider Algorithm 3.2. This algorithm simply calculates the value of $\tilde{A}(x_1, \dots, x_n)$ directly, from the explicit definition we gave in Corollary 2.3.8, and then compares it to the value of z. As such, this is a determiner for L.

We now demonstrate that Algorithm 3.2 runs in P^L. From the definition of PSPACE-robustness, we know that we only need to show that the algorithm runs in PSPACE^L, a much weaker bound. The inner for-loop runs in polynomial *time*, hence it must run in polynomial space. The outer for-loop runs for 2^n iterations, so determining that it is in P^L is non-trivial. Beyond the inner loop (which we have already discussed), the only thing we do in the outer loop is simulate L, which can be done in one step with access to an oracle for L.

The only memory we need to simulate this oracle (beyond that for the input) is space for d and z'. We have already shown d needs polynomial space, so what remains is z'. Since $A(x_1, \dots, x_n) \in \{0, 1\}$, each term in the sum in Equation (2.7) is bounded above by $\delta_\beta(x)$. This means that the value of z' that we compute is bounded above by

$$2^n \max_{k \in \{0,1\}^n} \delta_k(x). \tag{3.14}$$

Since each $\delta_k(x)$ can be written in polynomial space, and 2^n can be *written* in polynomial space, it follows that z' can as well. Hence, Algorithm 3.2 is in PSPACE^L, and thus is in P^L.

Next, we show that Algorithm 3.2 determines $\tilde{L}$. As mentioned earlier, our algorithm computes $\tilde{A}$ directly through the equations given in Corollary 2.3.8. First, we show the inner loop (beginning on line 6) computes $\delta_k(x)$. We compute δ directly, through the formula described at Equation (2.8). We do this by simply iterating through each i and then multiplying d by either x_i or $1 - x_i$, as appropriate.

Second, in this case Equation (2.7) simplifies to

$$\tilde{A}_n(x_1, \dots, x_n) = \sum_{\beta \in L} \delta_\beta(x_1, \dots, x_n). \tag{3.15}$$

This is exactly what our outer loop does: computes the sum directly through iteration. Hence, the only thing the above algorithm does is calculate $\tilde{A}_n(x_1, \dots, x_n)$ and then compares it to the value we were given. As such, it determines $\tilde{L}$.

Since there is a reduction from L to $\tilde{L}$, we know that L is no harder than $\tilde{L}$, and Algorithm 3.2 demonstrates that $\tilde{L} \in \mathsf{PSPACE}$. Hence, $\tilde{L}$ is PSPACE-complete. □

With that as a base, we can now move on to the main theorem. As before, the more straightforward proof is the oracle where $\mathsf{P}^{\tilde{A}} = \mathsf{NP}^A$, so we begin with that.

Theorem 3.7.4 ([1, Theorem 5.1])**.** *There exist A, $\tilde{A}$ such that $\mathsf{NP}^A = \mathsf{P}^{\tilde{A}}$.*

Proof. For this theorem, we use the same technique we did in our proof of Theorem 3.3.1: find a PSPACE-complete language A and work from there. If we let $\tilde{A}$ be the unique multilinear extension of A, Lemma 3.7.3 tells us $\tilde{A}$ is PSPACE-complete. Hence, as mentioned before, we have $\mathsf{NP}^{\tilde{A}} \subseteq \mathsf{NP}^{\mathsf{PSPACE}} \subseteq \mathsf{NPSPACE}$, and since $\mathsf{NPSPACE} = \mathsf{PSPACE}$ and we know from Theorem 3.3.1 that $\mathsf{PSPACE} \subseteq \mathsf{P}^A$, it follows

$$\mathsf{NP}^{\tilde{A}} = \mathsf{NP}^{\mathsf{PSPACE}} = \mathsf{PSPACE} = \mathsf{P}^A.$$

□

Now it is time for the other case.

Theorem 3.7.5 ([1, Theorem 5.3]). *There exist A, $\tilde{A}$ such that* $\mathsf{NP}^A \neq \mathsf{P}^{\tilde{A}}$.

Proof. Like in Theorem 3.3.4, we aim to "diagonalize": iterate over all $\mathsf{P}^{\tilde{A}}$ machines to construct a language that none of them can recognize. Also like before, we will do this by constructing an oracle extension $\tilde{A}$ such that $L(A) \notin \mathsf{P}^{\tilde{A}}$. Since we only give an algebraic extension to P and not NP, we can reuse the result from Lemma 3.3.3 that $L(A) \in \mathsf{NP}^A$. We shall construct $\tilde{A}$ using the following algorithm: As before, we will

```
    Input: A sequence of P oracle machines $\{P_i\}_{i=1}^{\infty}$
    Output: An extension oracle $\tilde{A}$ such that $L(A) \notin \mathsf{P}^{\tilde{A}}$
1   $\tilde{A} \leftarrow \varnothing$;
2   $n_0 \leftarrow 0$;
3   for $i$ starting at 1 do
4       Let $n > n_i$ be large enough that $p_i(n) < 2^n$;
5       Run $P_i^{\tilde{A}}$ on input $0^n$;
6       if $P_i^{B(i-1)}$ rejects $0^n$ then
7           Let $\mathcal{Y}_{\mathbb{F}}$ be the set of all $y \in \mathbb{F}^{n_i}$ queried during the above computation;
            // See Lemma 2.3.12 for why we can do this
8           Let $w \in \{0,1\}^n$ such that the following works;
9           for all $\mathbb{F}$ do
10              Set $\tilde{A}_{n_i,\mathbb{F}}$ to be a multiquadratic polynomial such that $\tilde{A}_{n_i,\mathbb{F}}(w) = 1$
                 and $\tilde{A}_{n_i,\mathbb{F}}(y) = 0$ for all $y \in \mathcal{Y}_{\mathbb{F}} \cup (\{0,1\}^{n_i} \setminus \{w\})$;
11          end
12      else
13          Set $\tilde{A}_{n_i,\mathbb{F}} = 0$ for all $\mathbb{F}$;
14      end
15      $n_{i+1} \leftarrow 2^n$;
16  end
17  $B \leftarrow \bigcup_i B(i)$;
```

Algorithm 3.3: An algorithm for constructing $\tilde{A}$

start by demonstrating soundness and then move on to why the constructed oracle provides the separation we seek.

Perhaps the least intuitive section of the above algorithm is the section beginning at line 8. We want to leverage Lemma 2.3.12 to show that such a solution exists. We know that $p_i(n) < 2^n$, and since $p_i(n)$ is an upper bound on the number of total queries, this tells us that there is at least one $w \in \{0,1\}^{n_i}$ not queried. From the definition of $\mathcal{Y}_{\mathbb{F}}$, we also therefore know that $\sum_{\mathbb{F}} \mathcal{Y}_{\mathbb{F}} < 2^n$. Further setting up this lemma, we will let f be the zero function and $p_{\mathbb{F}}$ be the zero polynomial.

From the lemma, we know that there is some $B \in \{0,1\}^n$ with $|B| < 2^n$ such that for all f' agreeing with f there exists a series of $p'_{\mathbb{F}}$ extending f' and agreeing with $p_{\mathbb{F}}$ on $\mathcal{Y}_{\mathbb{F}}$. As such, if we pick any $w \in \{0,1\}^n \setminus B$, then the function $f'(x) = [x = w]$ agrees with f on B, and thus we know that there exists a series of $p'_{\mathbb{F}}$ that agree with the zero polynomial on $\mathcal{Y}_{\mathbb{F}}$ and each non-w Boolean point.

Now, we know that such a solution exists, and Equation (2.12) gives us an explicit formula for our $A_{n_i,\mathbb{F}}$; thus, we know that this is in fact computable. Since this algorithm is simply for *constructing* the language, we do not care about time or space complexity, so the fact that it is computable is enough. In terms of finding the w we need, we can simply iterate try the construction for each $w \in \{0,1\}^n$ and stop as soon as we are able to construct each polynomial.

The other component of soundness is determining how we can run P_i with the extension oracle $\tilde{A}$ when $\tilde{A}$ is not yet fully constructed. What we do is when simulating P_i, we assume that any $\tilde{A}_{n_i,\mathbb{F}}$ that we have not yet queried returns zero on all queried inputs. We then make sure that any time we set an $\tilde{A}_{n_i,\mathbb{F}}$, it also returns zero on any point that we queried. Further, we ensure that each n_i is large enough that no previous machine would have queried any string of length n_i on its respective input; ergo modifying these polynomials would not have any affect on their output.

Next, we show that $L(A)$ is not in $\mathsf{P}^{\tilde{A}}$. As we did in Theorem 3.3.4, the idea is that for each polynomial-time machine P_i, that machine will return the incorrect result on the string 0^{n_i}. We do this in Algorithm 3.3 by simulating P_i on the input, and then adjusting $\tilde{A}$ based on its output. We separate this into two cases: the case where $P_i^{\tilde{A}}$ rejects 0^{n_i}, and the case where it accepts. We shall begin with the case where it accepts.

When $P_i^{\tilde{A}}$ accepts, we want to ensure that no strings of length n_i are in A. The unique low-degree extension of the zero function is the zero polynomial; hence, we set $\tilde{A}_{n_i,\mathbb{F}}$ to be 0 for all $\mathbb{F}$. This ensures $\tilde{A}(x) = 0$ for all $x \in \{0,1\}^{n_i}$, and thus $A \cap \{0,1\}^{n_i} = \varnothing$. This means $0^{n_i} \notin L(A)$ and thus $P_i^{\tilde{A}}$ is incorrect.

When $P_i^{\tilde{A}}$ accepts, we want to make sure that there is at least some string $w \in A \cap \{0,1\}^{n_i}$, but also to make sure that any polynomials we add have their values align with what $P_i^{\tilde{A}}$ already saw. As we mentioned earlier, we know that such a polynomial exists, and thus we construct it. Since our constructed polynomials tell us that $w \in A$, it follows that $0^{n_i} \in L(A)$ and hence $P_i^{\tilde{A}}$ is incorrect there as well.

Since our argument earlier told us that none of the P_i machines would have their output affected by any of the polynomials modified outside of the corresponding iteration i, it follows that no machine P_i could recognize $L(A)$. Since P_i includes every machine recognizing a $\mathsf{P}^{\tilde{A}}$ language, it follows that $L(A) \notin \mathsf{P}^{\tilde{A}}$. □

Chapter 4

Interactive proof systems

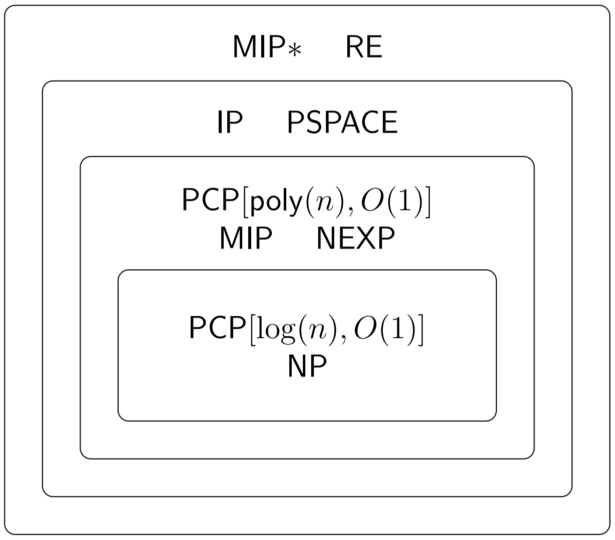

Figure 4.1: The interactive-proof classes and their relationships

Interactive proof systems are models of computation that involve multiple Turing machines exchanging messages between each other. In general, the machines are split into two categories: those that are computationally unbounded but untrustworthy (the provers), and those that are bounded but trustworthy (the verifiers). The "goal" of the system is to convince the verifier of whether or not the string is in the language. These systems almost always use randomness as part of their design: for this reason, almost all of the bounds are "with high probability" bounds and not complete mathematical certainty.

Interactive proof systems turn out to be surprisingly powerful—while the verifier only runs in polynomial time, it turns out that the interaction with the untrustworthy computer is still enough to boost the power significantly. The "classic" interactive proof model involves exactly two computers (one prover, one verifier), but many variants exist, all with distinct and interesting complexity-theoretic characteristics. In this chapter, we will introduce a good number of these variants and a few of their interesting properties; the goal of the remaining chapters will be to prove several of the more modern interesting results involving these systems.

4.1 Interactive Turing machines

The central mechanism underlying all of the interactive proof systems we will work with is the interactive Turing machine. This machine is a variant of a standard Turing machine, but it has the ability to communicate with another machine as part of its work. When multiple interactive machines work together, they can produce a joint computation in the same way that a single non-interactive machine can. From there, an interactive proof is just a pair of interactive machines working together, with some particular constraints on what they are allowed to do.

Definition 4.1.1 ([15, Def. 4.2.1]). An *interactive Turing machine* is a deterministic multi-tape Turing machine with the following tapes:

- Input tape (read-only)
- Output tape (write-only)
- Two communication tapes (one read-only, one write-only)
- One-cell switch tape (read-write)
- Work tape (read-write)

In addition to these tapes, an interactive TM has a single bit $\sigma \in \{0, 1\}$ associated with it, called its *identity*. When the content of the switch tape is not equal to the machine's identity, the machine performs no computation and is called *idle*.

In most cases, we will also give the interactive machines a source of randomness as well that they can read from. Since this is so common we will treat it as the default; if we ever want a machine to not have a source of randomness we will explicitly state as such.

On its own, a single interactive Turing machine is not worth much: in order to do work with these we need to define how a pair of them interact. The chief mechanism of interacting Turing machines is that of *shared tapes*. Shared tapes are tapes where any modifications can be seen by both Turing machines immediately. While the tapes themselves are shared, the *heads* are not: the two machines are perfectly capable of looking at different entries at the same time.

Definition 4.1.2 ([15, Def. 4.2.2]). A pair of interactive Turing machines (M, N) are *linked* if the following are true:

1. The identity of M is distinct from the identity of N.
2. The switch tapes of M and N coincide (i.e., writing to one affects the value in both).

3. The read-only communication tape of M coincides with the write-only communication tape of N.

4. The read-only communication tape of N coincides with the write-only communication tape of M.

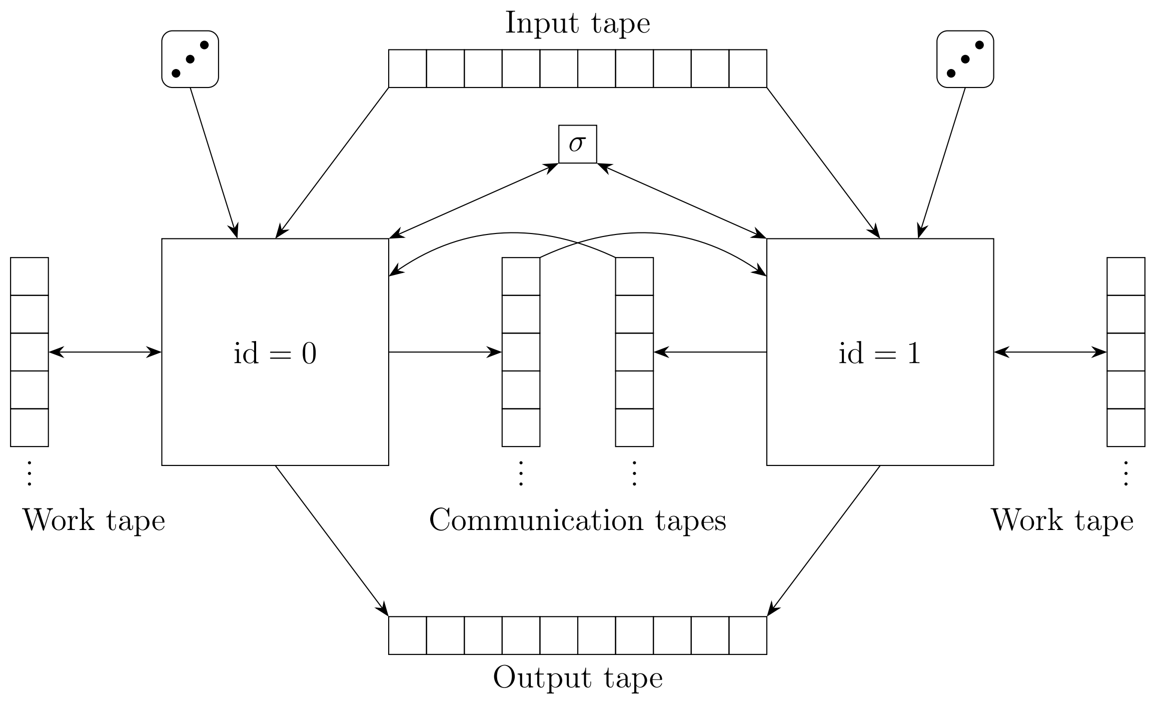

Figure 4.2: A linked pair of interactive Turing machines

We include a diagram of how a linked pair of Turing machines interact and share tape as Figure 4.2. The arrows point in the direction data is able to flow: read-only tapes have an arrow pointing from them and write-only tapes have an arrow pointing to them.

Definition 4.1.3 ([15, Def. 4.2.2])**.** The *joint computation* of a linked pair of interactive Turing machines (M, N) is, on a common input string x, the series of computation states for both M and N when each is given x as its initial input tape and when the initial value of the shared switch tape is 0. The joint computation halts when either machine halts and the halting machine is not idle.

We will denote the joint computation of machines M and N on input x by $\langle M, N\rangle(x)$. Since this output is not deterministic (it will depend on the values of the random bits read by P and V), it is important to note that this is a random variable, not an individual value.

Finally, we need the concept of a "view". A view is in essence a record of what a machine in a given interaction sees: it is an ordered list of everything the machine

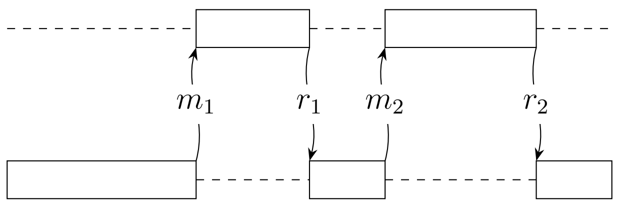

Figure 4.3: The flow of a joint computation of two interactive TMs

reads in sequence. We will care quite a bit about views throughout this thesis since views are a record of the "public" information of a proof: when we begin to work with zero knowledge, we will be using the view of an interaction in order to show that no information is leaked.

Definition 4.1.4. The *view* of an interactive Turing machine M in a joint computation on input string x is the sequence $(x, r, m_1, \dots, m_n)$, where x is the input string, r is the sequence of random bits seen by M, and m_i are the random messages received by M from N. We will denote the view of M by $\text{View}_M^N(x)$.

4.2 Single-prover systems

Now that we have a model for letting two machines talk to each other, we can define the requirements for an interactive proof. We will begin with the simplest form—that where there is only one prover and one verifier. To make sure our proof system is useful, we need three restrictions on the machines: to restrict the complexity of the verifier (lest it simply compute the problem itself without communication), to require the verifier to generally accept whenever the input is in the language, and to require the verifier to generally reject whenever the input is not in the language.

Definition 4.2.1 ([15, Def. 4.2.4])**.** An *interactive proof system* is a pair of interactive machines (P, V) such that V is polynomial-time and the following holds:

- *Completeness*: For every $x \in L$,

$$\mathbb{P}[\langle P, V\rangle(x) = 1] \geq \frac{2}{3}.$$

- *Soundness*: For every $x \notin L$ and every interactive machine B,

$$\mathbb{P}[\langle B, V\rangle(x) = 1] \leq \frac{1}{3}.$$

While we require our system to be correct at least 2/3 of the time, our choice of probability is actually somewhat arbitrary, so long as it is at least 50%. This is because with a greater than 50% chance of success, we can simply run the checker multiple times and take the majority vote, which will allow us to get the probability

	Honest	Dishonest
$x \in L$	$\mathbb{P} \geq \frac{2}{3}$	¯_(ツ)_/¯
$x \notin L$	$\mathbb{P} \leq \frac{1}{3}$	$\mathbb{P} \leq \frac{1}{3}$

Figure 4.4: Probability matrix for IP acceptance given prover and input string

arbitrarily high. Since this iteration is for a fixed number of times, it will only linearly scale the runtime and thus it does not affect whether our algorithm is an interactive proof system.

For the soundness clause, note that we require the inequality to hold for *any* interactive machine B, and not just our chosen machine P. This is important—it says that our verifier cannot be fooled reliably by a dishonest machine, so long as x is not in the language L. In practice, what this means is that if the verifier has reason to believe that the machine it is interacting with is not P, then it should always reject immediately, as we do not care what happens with an arbitrary machine when $x \in L$. A consequence of this is that if V ever receives back improperly-formatted or nonsense input from its prover, it will reject immediately. Similarly to what we do for ordinary Turing machines parsing their input, we will not explicitly write out that V should reject if it receives a poorly-formatted response, as it serves little but to provide clutter.

We also do not care how V fares if $x \in L$ and P is not the correct verifier. This is because neither insisting the protocol fail or insisting the protocol succeed will be a reasonable restriction. Since $x \in L$, an alternative P' could give messages arbitrarily close to the correct P (and in some cases even send identical messages, which would be impossible to distinguish), so we cannot insist L reject, even with high probability. However, if V could accept even in the face of an arbitrarily malicious prover, or a prover that sends no useful information whatsoever, it would mean that V would have some means of computing the problem on its own; hence we would only ever be able to accept languages in BPP (since L is a BPP machine).

As with all of our interactive-proof variants, we will also define a complexity class corresponding to the set of languages with the given proof. Once we have a complexity class, we will be able to work with it in the same way we have been all the "standard" classes like P or NSPACE.

Definition 4.2.2 ([15, Def. 4.2.5])**.** The class IP is the class of all languages that have an interactive proof system.

Now that we have seen the formal definition of an interactive proof, let us illustrate the formality with an example. To do so, consider the language of non-isomorphic graphs:

Definition 4.2.3. The language GI (for *graph isomorphism*) is the set

$$\mathsf{GI} = \{(G_0, G_1) \mid G_0, G_1 \text{ graphs and } G_0 \cong G_1\}.$$

This language is interesting for reasons beyond the scope of this paper, especially in that it is known to be in NP but is believed to be neither in P nor NP-complete.

Definition 4.2.4. The language GNI (for *graph non-isomorphism*) is the set

$$\mathsf{GNI} = \{(G_0, G_1) \mid G_0, G_1 \text{ graphs and } G_0 \ncong G_1\}.$$

Here, we will demonstrate that GNI has an interactive proof.

Theorem 4.2.5. *The language* GNI *is in* IP*.*

```
Input: Two $n$-vertex graphs $G_0$ and $G_1$, and a security parameter $s$
Output: Whether $G_0 \ncong G_1$
if $|V(G_0)| \neq |V(G_1)|$ or $|E(G_0)| \neq |E(G_1)|$ then
    accept;
end
for $i \in [s]$ do
    $V$: Pick a random $\sigma_i \in S^n$;
    $V$: Pick a random $b_i \in \{0, 1\}$;
    $V$: Compute $H_i \leftarrow \sigma_i \cdot G_{b_i}$;
    $V$: Send $H_i$ to $P$;
    $r_i \leftarrow$ if $H_i \cong G_0$ then
        $P$: Send 0 to $S$;
    else
        $P$: Send 1 to $S$;
    end
    if $r_i \neq b_i$ then
        reject;
    end
end
accept;
```

Algorithm 4.1: An interactive proof for the language GNI

Proof. We present an interactive protocol for GNI in Algorithm 4.1. In addition to the two graphs, we also give this algorithm one metaparameter s: the *security parameter*. This parameter does not need to be dynamic; adjusting it only affects the number of rounds of the protocol and correspondingly the probability of outputting the correct value.

First, we show V runs in polynomial time. Picking a random permutation and single bit can be done in polynomial time, and computing the action of a permutation on a graph is also polynomial.

Next, if $G_0 \ncong G_1$ and P is the honest prover, we show V will accept with probability $\geq 2/3$. Since $G_0 \ncong G_1$, it follows that $\sigma \cdot G_0 \ncong \sigma' \cdot G_1$ for any permutations σ and σ'. Hence, the honest prover will always answer with the correct $r_i = b_i$, and thus V will always accept.

If $G_0 \cong G_1$, we show V will reject with probability $\geq 2/3$, regardless of the prover. For any permutations σ and σ', we have that $\sigma \cdot G_0 \cong \sigma' \cdot G_1$, by transitivity of isomorphisms. Further, we have that S_n is exactly the class of isomorphisms on n-vertex labeled graphs, so for any isomorphic graph G, there exists exactly one σ_0 and σ_1 such that $\sigma_0 \cdot G_0 \cong G \cong \sigma_1 \cdot G_1$. Hence, the odds of b_i being 0 are 1 are equal for any given G. Thus, in each round the odds of P guessing correctly are no more than $1/2$. Further, in each round the random picks are statistically independent; hence the overall odds of P fooling V are no more than 2^{-s}. For any $s > 1$, $2^{-s} < 1/3$; hence V will reject with probability $\geq 2/3$. □

Once we have a complexity class, the question arises of how it relates to other complexity classes. For IP, Adi Shamir proved in 1992 [30] that a language has a standard interactive protocol if and only if it is in PSPACE.

Theorem 4.2.6 ([30])**.** IP = PSPACE.

4.3 Multi-prover systems

We have now seen quite a bit of single-prover interactive proofs. A natural extension of the standard interactive proof format is to add more machines to the interaction. Since our verifiers are trusted, increasing the number of verifiers is not useful since any pair of verifiers could simply be simulated with a single verifier working twice as hard (which would keep it polynomial). However, increasing the number of provers to two turns out to give us more power than we would get with a single prover.

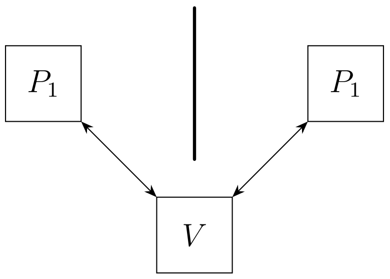

Figure 4.5: A multi-prover interactive system

Definition 4.3.1 ([15, Def. 4.11.2])**.** A *multi-prover interactive proof system* is a triplet of interactive machines (P_1, P_2, V) such that P_1 and P_2 cannot communicate, V is probabilistic polynomial-time, and the following hold.

- *Completeness*: For every $x \in L$,
$$\mathbb{P}[\langle P_1, P_2, V\rangle(x) = 1] \geq \frac{2}{3}.$$
- *Soundness*: For every $x \notin L$ and every pair of interactive machines B_1 and B_2,
$$\mathbb{P}[\langle B_1, B_2, V\rangle(x) = 1] \leq \frac{1}{3}.$$

The above definition should look rather similar to Definition 4.2.1; the only difference is now we have two provers instead of just one. The fact that the two provers cannot communicate is important: if they could, they would be able to "strategize"; that is, agree on a joint plan to make sure that their responses agree with each other. Since our provers are not required to be computationally bounded, if they could communicate it would be no different than simply having one prover. However, since the two provers cannot communicate, we gain some information from times where they lie in *different* ways: where each prover individually could say something plausible, but in combination, the provers' responses contradict.

Definition 4.3.2. The class MIP is the class of languages that have a multi-prover interactive proof system.

The first question about a class like MIP is how it relates to other complexity classes we have already seen. First, if we have a single-prover system we can always convert it into a multi-prover system by simply having the verifier ignore P_2 completely; thus we get the following.

Lemma 4.3.3. $\mathsf{IP} \subseteq \mathsf{MIP}$.

Proof. Let (P, V) be an interactive proof system. Consider the system (P_1, P_2, V'), where $P_1 = P$, and V' simulates V and sends all its messages to P_1. If $x \in L$, then $\langle P_1, P_2, V'\rangle(x)$ will accept with exactly the same probability as $\langle P, V\rangle(x)$. Since (P, V) is an interactive proof system, it follows that it will accept with probability at least $2/3$. If $x \notin L$, then $\langle P_1^*, P_2^*, V\rangle$ will reject with exactly the same probability as $\langle P_1^*, V\rangle$ regardless of the value of P_2^*, since the interaction is exactly the same and since we ignore the second prover completely. Again, since (P, V) is an interactive proof system, it follows that it will reject with probability at least $2/3$ in this case. Hence, (P_1, P_2, V') is a MIP system. $\square$

A groundbreaking result by Babai, Fortnow, and Lund [4] is that MIP is exactly equal to NEXP. Since it is known that $\mathsf{NP} \neq \mathsf{NEXP}$ [11], this tells us that adding multiple provers gives an actual boost in computational power over just having one.

Theorem 4.3.4 ([4]). $\mathsf{MIP} = \mathsf{NEXP}$.

Having seen how much more powerful systems become with two provers, one might wonder what would happen if we were to add a third. Unfortunately, it turns out that a third prover is no more powerful than just having two. We formalize this below; because we do not get any benefit from three provers we will not work with three-prover systems at all in this paper beyond this proof.

Theorem 4.3.5 ([7, Theorem 4])**.** *If we redefine* MIP *to have k provers instead of* 2*, the class is unchanged.*

```
1  $\hat{V}$: Generate all random bits and send results to $\hat{P}_1$;
2  $\hat{P}_1$: Send transaction log between $(P_1, \ldots, P_k, V)$ with the chosen randomness;
3  if The simulated log is longer than the worst-case runtime of $V$ then
4      reject;
5  end
6  $\hat{V}$: Choose random $i \in [k]$;
7  $\hat{V}$ and $\hat{P}_2$: Simulate conversation between $V$ and $P_i$ given coin tosses;
8  if the simulated conversation does not match the result of $\hat{P}_1$ then
9      reject;
10 else
11     accept if and only if $V$ would accept with the given transcript;
12 end
```

Algorithm 4.2: A 2-prover MIP simulating a k-prover MIP

Proof. Let $(P_1, \ldots, P_k, V)$ be a MIP with k provers. We show a simulator for $(P_1, \ldots, P_k, V)$ in Algorithm 4.2. Since V runs in polynomial time and all $\hat{V}$ does is look at the simulated transaction log of V (twice), it follows that $\hat{V}$ runs in polynomial time.

Let $x \in L$. By definition, we know that

$$\mathbb{P}[\langle P_1, \ldots, P_k, V\rangle(x) = 1] \geq \frac{2}{3}. \tag{4.1}$$

If $\hat{P}_1$ and $\hat{P}_2$ are honest, then the simulated conversations will match; hence $\hat{V}$ will reject if and only if P would have. Thus, $\mathbb{P}[\langle \hat{P}_1, \hat{P}_2, \hat{V}\rangle = 1] \geq 2/3$.

Let $x \notin L$, and let $\hat{P}_1^*$ and $\hat{P}_2^*$ be arbitrary verifiers. From the definition of MIP, for at least $2/3$ of the choices of randomness, the generated transcript would result in V rejecting; hence $\hat{P}_1$ must deviate from it somewhere. Since $\hat{P}_1$ deviates from the protocol at least once, it follows there is at least a $1/k$ chance that the simulated conversation between $\hat{V}$ and $\hat{P}_2$ is different from what $\hat{V}$ received from $\hat{P}_1$. Hence, the probability of correctly rejecting here is at least $2/3k$.

Note that if the two provers ever disagree, it must be that at least one of them is a cheating prover. Thus, if we run it at least k^2 times we will have at least one run where we catch the cheating with probability $2k^2/3k = 2/3$. Thus, we will correctly reject with probability at least $2/3$ regardless of the cheating prover. Hence, (P_1, P_2, V) is a valid MIP system. □

Similarly to how we defined views for single-prover systems, a verifier in a multi-prover system also has a view. This works similarly to the view of a single-prover system, but with the main difference being that we need a bit to signal which machine any given response came from.

Definition 4.3.6. The *view* of the verifier in a multi-prover interactive system is the tuple

$$(x, (b_1, m_1), \ldots, (b_n, m_n)),$$

where x is the input, m_i is the ith message received, and b_i is either 0 or 1, depending on which machine the message came from.

In some respects, we can think of the verifier's view as being the "leaked" information from an interaction. The framework behind this is the scenario where we think of multiple computers communicating over some sort of public, unencrypted link (for example, the internet).

4.4 Zero-knowledge proofs

Zero-knowledge proofs are a variant of interactive proofs that have certain cryptographic requirements. What we care about is the idea that zero-knowledge proofs transmit *no knowledge* other than precisely the statement trying to be proved. As an example, if the statement that you are trying to prove is "I have an instance of X", the conceptually-easiest way to prove it would be to produce the aforementioned instance. However, this would not be zero-knowledge since it also transmits the knowledge of exactly what your instance of X is.

The way we mathematically define zero-knowledge is a little tricky. The way we demonstrate that the proof is zero-knowledge is by creating a simulator S_V for each possible verifier V: a machine in P that *by itself* can reproduce the entire message log between P and V for any input. This definition shows that no knowledge has been released because we are able to reproduce all the public information of the proof with relatively little work. If non-trivial knowledge were released by the proof, we would not be able to recreate the interaction faithfully without access to the prover.[1]

Having said that, it is not particularly obvious that there are any languages that are outside of P with perfect zero-knowledge proofs.[2] It turns out, however, that these languages do in fact exist (and are reasonably common). Abstractly, the idea behind why many of these work is that the verifier can perform a transformation on some random value, such that undoing the transformation and reliably recovering the original value is only possible with knowledge of the language. However, a simulator would have access to the randomly-chosen value, and thus it could construct a response immediately with no reference to the problem to be solved.

Definition 4.4.1 ([15, Def. 4.3.1]). A proof system (P, V) for a language L is *perfect zero-knowledge* if for each probabilistic polynomial-time interactive machine V^* there exists a probabilistic polynomial-time ordinary machine M^* such that for every $x \in L$ we have the following conditions hold:

[1]Exception: if $L \in \mathsf{P}$ then we can trivially recreate the interaction no matter what, but that case is not particularly interesting for the purpose of zero-knowledge proofs.

[2]To some extent, showing that there are languages *truly* outside of P would require a proof that $\mathsf{P} \neq \mathsf{NP}$ (which is unfortunately beyond the scope of this paper), but there are lots of languages strongly believed to be outside of P with zero-knowledge proofs.

1. With probability at most $1/2$, on input x, machine M^* outputs a special symbol denoted $\perp$ (i.e. $\mathbb{P}[M^*(x) = \perp] \leq 1/2$).
2. Let $m^*(x)$ be the random variable such that

$$\mathbb{P}[m^*(x) = \alpha] = \mathbb{P}[M^*(x) = \alpha \mid M^*(x) \neq \perp] \tag{4.2}$$

for all α. That is, let $m^*(x)$ be the distribution of non-$\perp$ values of M^*. Then $\langle P, V^*\rangle(x)$ and $m^*(x)$ are identically distributed for all $x \in L$.

In this case, we say the machine M^* is a *perfect simulator* for the interaction of V^* with P.

Looking at this definition, one might wonder why we would want the ability for M^* to output $\perp$. This is a reasonable thing to wonder, because we do not actually want this. However, we do not know of any non-trivial proof systems that do not actually output $\perp$ at least some of the time, although [16] has made significant inroads on this problem.[3] Thankfully, we can make $\mathbb{P}[\perp]$ arbitrarily small (bounded above by $2^{-\mathsf{poly}(|n|)}$), but we cannot make it truly perfect. We can do this by simply re-running the simulator every time we get a $\perp$ until we get a valid answer.

Also note that while the definition of interactive proof systems focus on cheating *provers*, the definition of zero-knowledge focuses on cheating *verifiers*. This is because we can think of the two as being resilient to different threat models. For interactive proofs, we care about verifier efficiency: our goal is to show it is possible for some computer with unbounded resources to easily convince a verifier of something, and to show that no other equally-strong computer could lie.

For zero-knowledge, our main goal is to ensure that it is impossible for anybody except for the honest verifier to extract any information from the honest prover. The main way we do this is by ensuring that it is impossible for anybody snooping on the transaction to gain any information (hence the simulator), but we also want to ensure that the prover cannot be tricked into revealing information by a dishonest verifier. We do not care about what happens with a dishonest prover in this case because alternate provers could in theory reveal anything—there is always a prover that just dumps any relevant information to the interaction tape, for example.

As with other interactive proof systems, zero-knowledge proofs are probabilistic; in particular this means they do *not* function as proofs in the mathematical sense.

Definition 4.4.2 ([15, Def. 4.3.5])**.** The class PZK is the class of all languages with a perfect zero-knowledge proof system.

To demonstrate perfect zero-knowledge, we now show an example. In Section 4.2, we demonstrated a non-zero-knowledge interactive proof for the language GNI of non-isomorphic graphs. Here, we modify that algorithm to not reveal anything beyond simply whether G_0 and G_1 are isomorphic.

Theorem 4.4.3. *The language* GI *has a perfect zero-knowledge interactive proof.*

```
   Input: Two $n$-vertex graphs $G_0$ and $G_1$
   Output: Whether $G_0 \cong G_1$
 1 for $i \in [s]$ do
 2     $P$: pick a random $\sigma \in S^n$;
 3     $P$: pick a random $b \in \{0,1\}$;
 4     $P$: send $\sigma \cdot G_b$ to $V$;
 5     $V$: pick a random $b' \in \{0,1\}$;
 6     $V$: send $b'$ to $P$;
 7     $P$: compute $\sigma' \in S^n$ such that $\sigma' \cdot G_{b'} = \sigma \cdot G_b$;
 8     $P$: send $\sigma'$ to $V$;
 9     if $\sigma' \cdot G_{b'} \neq \sigma \cdot G_b$ then
10         reject;
11     end
12 end
13 accept;
```

Algorithm 4.3: A perfect zero-knowledge IP for GI

```
 1 $c \leftarrow 0$;
 2 $L \leftarrow [\,]$;
 3 for $i \in [2s]$ do
 4     if $c \geq s$ then
 5         return $L$;
 6     end
 7     Choose random $\sigma \in S_n$;
 8     Choose random $b \in \{0,1\}$;
 9     $H \leftarrow \sigma \cdot G_b$;
10     Add $H$ to $L$;
11     Simulate $V^*$ on input $(G_0, G_1)$ and received message $H$ until it sends a
         message $\sigma'$;
12     if $b = b'$ then
13         $c \leftarrow c + 1$;
14         Add $\sigma$ to $L$;
15     end
16 end
17 return $\perp$;
```

Algorithm 4.4: A simulator for Algorithm 4.3

Proof. We present the algorithm as Algorithm 4.3. Our proof will consist of two stages: first, we will prove that the algorithm is a functional interactive proof for GI, and then we will show that it does not leak any knowledge.

First, if $G_0 \cong G_1$ and P is honest, then regardless of what parameters we pick, there will always exist a σ' we can compute in line 7. Hence, the condition in line 9 will never be true and hence we will always accept.

If $G_0 \ncong G_1$, then whenever $b' \neq b$ there exists no σ' such that $\sigma' \cdot G_{b'} = \sigma \cdot G_b$. Hence, if $b' \neq b$ (which happens with probability $1/2$) then regardless of what σ' is sent to V the check in line 9 will fail and hence V will reject. Since b' and b are re-rolled in each round, the probability of them being equal in all s rounds is 2^{-s}. Hence, V will accept with probability no more than 2^{-s}.

Now, we show zero-knowledge. To do this we show a simulator in Algorithm 4.4. To summarize, it attempts to simulate a single round of the interaction repeatedly until it has s successes; if it cannot do this in $2s$ tries, it outputs $\perp$.

First, we show Algorithm 4.4 runs in polynomial time. We know V^* runs in polynomial time by our hypothesis; hence line 11 runs in polynomial time. All the rest of the lines are either random choice of items, computing permutations, or simple arithmetic; all of these are also doable in polynomial time. Hence the interior of the loop runs in polynomial time, and since we iterate no more than $2s$ times, it follows that the whole algorithm is polynomial time.

Note that an individual round of this simulator can only succeed when $b = b'$: if it could output a correct response σ' when $b \neq b'$, then it would have $\sigma' \circ \sigma^{-1}$ as an isomorphism from G_0 to G_1, and thus Algorithm 4.4 would be a probabilistic polynomial-time determiner for GI. Since we do not know whether or not $\mathsf{GI} \in \mathsf{BPP}$, we do not yet know how to construct any verifier that would do this.

Since V^* never receives b and b is randomly generated, if G_0 is isomorphic to G_1, for any value of G it is just as likely that $b = 0$ as it is that $b = 1$. More formally, for all G,

$$\mathbb{P}[G \mid b = 0] = \mathbb{P}[G \mid b = 1].$$

As such, for any $(G_0, G_1) \in \mathsf{GI}$, it is impossible for any V^* to send $b' \neq b$ with probability more than $1/2$.

Since, each individual round succeeds with probability at least $1/2$, the binomial theorem tells us that the probability of exactly k successes in $2s$ tries is $\binom{2s}{k}/2^{2s}$. Hence, the probability of at least s successes in $2s$ tries is

$$\frac{1}{2^{2s}} \sum_{i=s}^{2s} \binom{2s}{i}. \tag{4.3}$$

Since $\binom{2s}{k} = \binom{2s}{2s-k}$ and the sum of $\binom{2s}{k}$ over all k is $2s$, it follows that Equation (4.3)

[3]More specifically, they have shown that all of NP has a zero-knowledge proof without use of $\perp$, which includes nontrivial problems in the (generally expected) case that $\mathsf{NP} \neq \mathsf{BPP}$.

is equal to

$$\frac{\sum_{i=s}^{2s}\binom{2s}{i}}{\sum_{i=1}^{2s}\binom{2s}{i}} = \frac{s + \sum_{i=s+1}^{2s}\binom{2s}{i}}{2\sum_{i=s+1}^{2s}\binom{2s}{i}} \geq \frac{1}{2}. \tag{4.4}$$

Hence, the algorithm will output $\perp$ with probability no more than 1/2.

Next, we show Algorithm 4.4 outputs an identically-distributed view to what V^* sees for all $x \in \mathsf{GI}$, regardless of what V^* is. We show that it outputs an identical view for a single round; since the simulator will simulate s rounds it follows that the total result will be identical exactly when an individual round is.

An individual round of Algorithm 4.3 has a total of three messages sent: two from P and one from V. The view of a verifier only consists of the messages from P, so after each round we should be adding two items to the log. Our honest prover sends two messages: first, a random graph $H \cong G_b$ and second, an isomorphism σ that maps $G_{b'}$ to H.

The simulator picks a random b and H in exactly the same way as P; thus H will be identically distributed in the simulator as it is in the original algorithm. Next, remember that we only care about the views being identical in the case where $(G_1, G_2) \in \mathsf{GI}$, that is, where $G_1 \cong G_2$. In this case, picking a random isomorphic copy of G_1 will give you an identically-distributed random variable to picking a random isomorphic copy of G_2. Similarly, $H \cong G_1 \cong G_2$ by construction, so picking a random isomorphic copy of H will also give you an identical distribution. As such, σ is a random isomorphism and hence the transaction (H, σ) is distributed identically to the originally-generated transcript. $\square$

4.4.1 Commitment schemes

It should not come as too much of a surprise to learn that most intuitive proofs for a given problem are not in fact zero-knowledge.[4] As such, we will want the assistance of a few techniques that, once understood, will allow us to build zero-knowledge proofs more easily. The first of these is a *bit-commitment scheme*.

Abstractly, a bit-commitment scheme allows a machine to "commit to" a given single bit, with the intent of revealing it later on to a verifier. To do this, we need two important things: first, that the bit is not revealed to the verifier at commitment time, and second, that if the revealed bit is not equal to the committed bit, the verifier will be able to know (that is, the committer will not be able to change its choice once it has committed).

Before we can define a bit-commitment scheme formally, we do need a few preliminaries. First, we define what it means for an interaction to look like a commitment from the receiver's perspective; since the receiver needs to be convinced of the commitment, we need a definition that only considers its perspective.

[4]As an example, I would certainly hope that the proofs in this text have, in fact, imparted at least some knowledge on the reader.

Definition 4.4.4. Let $\sigma \in \{0,1\}$. A receiver's view of an interaction $(x, r, m_1, \ldots, m_n)$ is a *possible σ-commitment* if there exists a string s such that m_i describes the messages received by R when R uses local coins r and interacts with machine S that uses local coins s and has input $(\sigma, 1^n)$.

The above definition does not preclude a series of messages looking like it could be both a 0-commitment and a 1-commitment; consider the case of a machine S that completely ignores its input and sends the same series of strings. In this case, the record of that interaction would be a possible commitment for any input, since the input is ignored completely.

Definition 4.4.5. A receiver's view is *ambiguous* if it is both a possible 0-commitment and a possible 1-commitment.

Now, we can define a bit-commitment scheme. In brief, a bit-commitment scheme is a scheme such that there are no ambiguous views and yet the total publicly-released information is ambiguous.

Definition 4.4.6 ([15, Def. 4.4.1])**.** A *bit-commitment scheme* is a pair of interactive probabilistic polynomial-time machines (S, R), such that

1. Both machines receive an integer n in unary,
2. S receives a single bit v,
3. For any PPT machine R', the output of $\langle S(0), R' \rangle(1^n)$ and $\langle S(1), R' \rangle(1^n)$ are computationally indistinguishable over all inputs n, and
4. For almost all coin tosses of R, there exists no sequence of messages from S such that the view of R is ambiguous.

The simplest examples of bit-commitment schemes involve one-way functions. A *one-way function* is a function that is computable in polynomial time, but whose inverse is *not* computable in polynomial time. These functions are not known to exist: in particular their existence implies $\mathsf{P} \neq \mathsf{NP}$, but they are still widely believed to exist.

Not only do the simplest examples of bit-commitment schemes involve one-way functions, but Goldreich and Levin showed that bit-commitment schemes can only exist if one-way functions do [14]. Referring back to our discussion of Impagliazzo's five worlds in Section 2.2.5, this means that bit-commitment schemes exist only if we live in cryptomania. The importance of bit-commitment schemes to zero-knowledge proofs in general is one of the major reasons why this paper assumes the existence of one-way functions.

4.5 Probabilistically-checkable proofs

So far, all of our computational proofs have focused on the interaction between two computers, but there exist non-interactive models as well. Probabilistically-checkable

```
Input: A one-way function $G\colon \{0,1\}^* \to \{0,1\}^*$ such that $|G(s)| = 3|s|$ for
       all $s \in \{0,1\}^*$
/* Commit                                                    */
R: Pick uniform $r \in \{0,1\}^{3n}$ and send to $S$;
S: Let $v \in \{0,1\}$ be the committed-to bit;
if $v = 0$ then
    S: Send $\alpha = G(s)$ to $R$;
else
    S: Send $\alpha = G(s) \oplus r$ to $R$;
end
/* Reveal                                                    */
S: Send $s$ to $R$;
if $G(s) = \alpha$ then
    R: accept $v = 0$;
else if $G(s) \oplus r = \alpha$ then
    R: accept $v = 1$;
else
    R: reject;
```
Algorithm 4.5: A bit-commitment scheme based on a one-way function f

proofs do not use interactive Turing machines, but instead have access to a "proof" that their input is in the given language. The nontriviality is that the number of bits of the proof we can access is bounded—simply reading the entire proof will not suffice. For any string in the language, we ensure there exists a correct proof, which our algorithm must always recognize accurately. Further, for any string *not* in the language, the algorithm must reliably (but not necessarily always) reject.

In general, query-based machines (i.e. those where the *number* of queries is an important part of the complexity) come in two flavors: adaptive and non-adaptive. Adaptive-query machines allow the machine to change what locations it queries based on what it has already seen; non-adaptive machines do not allow that. In general, adaptive queries are more powerful—it turns out that any adaptive machine can be simulated by a non-adaptive machine using 2^q queries. Most interesting results about PCPs can be proven even with weaker non-adaptive machines, so that is what we will focus on for the rest of this paper.

Definition 4.5.1. Let M be a Turing machine with query access to some string $\pi \in \{0,1\}^*$. The queries by M are *non-adaptive* if the locations queried depend only on the contents of the input tape and random generator. If the queries are dependent on previous query results, then M is *adaptive*.

Now, it is time to define a probabilistically-checkable proof. Unlike an interactive proof system, our formal definition only defines the verifier, and not the entire system itself. This is because, in an interactive proof, both the prover and verifier are themselves Turing machines, with all the attendant parameters to be defined therein. On the other hand, with a PCP, only the verifier is actually a Turing machine, and

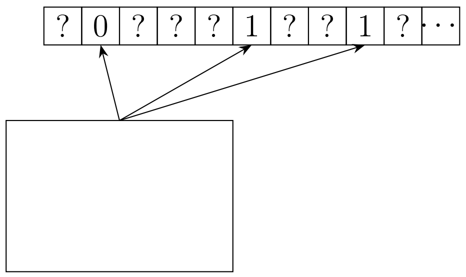

Figure 4.6: A probabilistically-checkable proof verifier

the proof itself is simply a particular string satisfying some properties.

Definition 4.5.2 ([2, Def. 18.1])**.** Let $L \subseteq \{0,1\}^n$ be a language and $q, r \colon \mathbb{N} \to \mathbb{N}$. A $(r(n), q(n))$-*verifier* for L is a polynomial-time probabilistic algorithm V such that

1. When given an input string $x \in \{0,1\}^n$ and random access to a string $\pi \in \{0,1\}^*$, V uses at most $r(n)$ random coins and makes at most $q(n)$ non-adaptive queries to locations of π before either accepting or rejecting.

2. If $x \in L$ then there exists a $\pi_x \in \{0,1\}^*$ (which we call the *proof* of x) such that V will always accept when given input x and random string π_x.

3. If $x \notin L$ then V will reject with probability $\geq 1/2$ for *all* random strings π.

We denote the output of V on input x and proof π by $V^\pi(x)$.

The first piece of this definition to notice is that if $x \in L$ we require L to accept unconditionally for x's corresponding proof, despite the fact that L has access to randomness. We want this because we want to construct verifiers that look for inconsistencies in the purported proof and reject if they find them—a correct proof should not contain any inconsistency at all so we should accept every time.

On the other hand, while an incorrect proof should still be noticeable *most* of the time (otherwise this would not be a particularly useful definition), if we are only querying a limited number of bits from a proof it is impossible to prevent scenarios where we only query bits that are identical to the original proof. In this case, it would be impossible for any algorithm to distinguish correct proofs from incorrect ones (outside of ignoring the proof completely and just solving the problem itself) and as such we do not require V to always correctly reject.

So far, all of our proof-complexity classes have just had a single class for all languages with the proof regardless of internal complexity, but for probabilistically-checkable proofs we actually stratify the class further. This is for multiple reasons: first, we can actually get astonishingly tight bounds on the parameters for PCPs (as we will see in Theorem 5.0.1), and second, because these are "access" complexity (i.e. we measure the number of preexisting bits actually read by the algorithm), they are actually independent of computational model, so the need for polynomial equivalence is negated.

In addition, this means PCPs become susceptible to the alphabet the proof is written in. Since individual bits carry more data when a larger alphabet is used, a larger alphabet will necessarily require fewer queries to transmit the same information. As such, while we will by default still work over the alphabet $\{0, 1\}$, there are times where we will need to be a little more specific about the alphabet we use.

Definition 4.5.3 ([2, Def. 18.1])**.** For any $q, r\colon \mathbb{N} \to \mathbb{N}$, the class $\mathsf{PCP}_\Sigma[q(n), r(n)]$ is the class of all languages with a $(cq(n), dr(n))$-verifier for some $c, d \in \mathbb{N}$, where the proof is written over the alphabet Σ. When $\Sigma = \{0, 1\}$, we sometimes omit it.

A small note on notation: The text "PCP" can be used as both a complexity class and an abbreviation: in this paper when it is written PCP, we are referring to the complexity class (and therefore the set of *languages* with a probabilistically-checkable proof); when it is written PCP, we are referring to the proofs themselves.

Next, we give an example of a nontrivial probabilistically-checkable proof. As with our previous examples, the language GNI provides a good example for us, as it is a relatively simple language that is still not known to be in BPP.

Theorem 4.5.4. $\mathsf{GNI} \in \mathsf{PCP}[\mathsf{poly}(n), 1]$.

```
Input: Two $n$-vertex graphs $G_1$ and $G_2$
Output: Whether $G_1 \ncong G_2$
/* Proof:                                                  */
for $H$ a graph with $n$ nodes do
    if $H \cong G_0$ then
        $\pi[H] \leftarrow 0$;
    else
        $\pi[H] \leftarrow 1$;
    end
end
return $\pi$;
/* Verifier:                                               */
Pick random $b \in \{0, 1\}$;
Pick random $\sigma \in S_n$;
Apply $\sigma$ to the vertices of $G_b$;
Accept if and only if $\pi[\sigma \cdot G_b] = b$;
```

Algorithm 4.6: A PCP for GNI

Proof. We describe such a PCP in Algorithm 4.6. Next, we show that this algorithm has the properties we seek.

The verifier runs in polynomial time: both picking and computing an n-bit permutation are known to be in polynomial time with respect to n. Further, picking a random n-bit permutation is doable in $\mathsf{poly}(n)$ bits, and picking a random bit is doable in 1 bit; hence V uses $\mathsf{poly}(n)$ bits of randomness. Lastly, we only make a single query to π, in line 12.

If $G_0 \ncong G_1$, we show V always accepts when given π as input. Since isomorphisms are transitive, we know there is no H with both $H \cong G_0$ and $H \cong G_1$. Hence, for all H, if $H \cong G_0$ then $\pi[H] = 0$ and if $H \cong G_1$ then $\pi[H] = 1$. Hence, since $\sigma \cdot G_b \cong G_b$ regardless of our choice of σ and b, we have that $\pi[\sigma \cdot G_b] = b$.

Next, if $G_0 \cong G_1$ then V rejects with probability at least $1/2$, regardless of choice of π. Since $G_0 \cong G_1$, it follows that if $H \cong G_0$ then $H \cong G_1$, and vice versa. Hence, for any graph $H = \sigma \cdot G_0$, there exists a $\sigma' \in S_n$ with $H = \sigma' \cdot G_1$. For any n-vertex graph H, this means that we are equally likely to query $\pi[H]$ with $b = 0$ and σ as we are with $b = 1$ and σ'. Since we know $\pi[H]$ can only be 0 or 1, it must be the case that $\pi[H]$ is incorrect at least half the time. Hence, V will reject with probability at least $1/2$ for any proof π. □

It is reasonable to be surprised about the fact that we only need one query to determine this problem to the constraints imposed by a PCP. This is our first clue that PCPs are surprisingly powerful: in Chapter 5 and again in Chapter 7 we will explore the extremes of the power of PCPs.

4.5.1 PCPs of proximity

Since a PCP will always only check a limited proportion of any given proof, for any strings in the language, the verifier will still accept any proof that is close to the official proof with very high probability. While this is interesting in and of itself, it can also lead to the question of how PCPs perform on *values* that are close to strings in our language. This is the notion behind PCPs of proximity: what if we weaken a PCP to only require it to reject strings that are not sufficiently close to strings in the given language?

We discussed Hamming distance, a measure for proximity, in Definition 2.5.1 earlier. This is the definition we will be using when we refer to the distance of strings. in a PCPP.

Unlike a PCP, which is defined for any language L, a PCPP is only defined for *pair languages*, languages that consist entirely of ordered pairs of two objects. This is because we need our language to consist of pairs so that we can have a notion of distance between the elements. The good news is this will not affect us too much—lots of the languages we care about are pair languages already.

The main difference between a PCP and a PCPP is that we relax the rejection condition to only require consistent rejection for pairs (x, y) where there is no close y' where (x, y') is in the accepted language. There are many pair languages that we can think of as being made of function-value pairs (f, y) with the property that there is some x such that $f(x) = y$. When we have a language like this, we have a more intuitive explanation of PCPPs: here, a PCPP will accept any (f, y) pair where $f(x)$ is δ-close to y.

Definition 4.5.5 ([17, Def. 2.2])**.** For $\delta\colon \mathbb{N} \to [0, 1]$, a *probabilistically-checkable proof of proximity* for a language L consisting of ordered pairs (x, y) with proximity parameter δ consists of a prover P and verifier V such that the following holds for all (x, y):

1. When given an input string $x \in \{0,1\}^n$ and random access to a string $\pi \in \{0,1\}^*$, V uses at most $r(n)$ random coins and makes at most $q(n)$ non-adaptive queries to locations of π and y before either accepting or rejecting.

2. If $(x, y) \in L$, there exists a proof $\pi_{(x,y)}$ such that V will always accept when given input x and oracle access to y and $\pi_{(x,y)}$.

3. If y is δ-far from the set $L(x) = \{y \mid (x, y) \in L\}$, then for every oracle π^*, V will reject on input x and oracles y and π^* with probability $\geq 1/2$.

One important thing to note here about the definition of a PCPP is that V does *not* get complete access to the input string y, only oracle access.[5] As such, unlike every other proof system we have seen, it is not immediate that a PCPP can solve every problem in P. With every other proof system, we have gotten unfettered access to the entire input string; thus if the language is in P our verifier could simply compute the answer itself without actually engaging in any part of what makes the proof system interesting. However, since a PCPP only gets oracle access to one of the two elements of the pair, it must make at least some queries to y in order to determine what the entire input is. This point will be salient to remember in Section 5.1, when we talk about a PCPP for a language in P.

4.5.2 Robustness

One important property of PCPs is their ability to reliably reject when given a proof of a false input. While often this requirement is enough to make PCPs useful, one particular place where it falls apart is in the notion of PCP composition. Here, we introduce the notion of *robust* PCPs, which strengthens the original requirements of a PCP in order to facilitate composition. In this case, we strengthen the original soundness requirement of PCPs to that of *robust soundness*: instead of simply requiring that a PCP reliably reject when given a proof of a false input, we also require that the set of public messages is reliably far from any set of accepting messages.

Definition 4.5.6. Let V be a non-adaptive PCP verifier that makes q queries. The set of *accepting views* of V for some input x and randomness μ, denoted $\mathrm{Acc}(V(x;\mu))$, is the set of all elements $a \in \Sigma^q$ such that V accepts when given input x, randomness μ, and has its queries answered by the sequential elements of a.

Definition 4.5.7 ([8, Def. 2.6])**.** Let $s, \rho \colon \mathbb{N} \to [0,1]$. A PCP verifier V has *robust-soundness error* s with *robustness parameter* ρ if for all $x \notin L$, the bits read by V are ρ-close to being accepted with probability strictly less than s. More formally,

$$\mathbb{P}[\Delta(\pi^*|_{Q(x)}, \mathrm{Acc}(V(x;\mu))) \leq \rho] \leq s. \tag{4.5}$$

[5]In particular, this means that access to y is subject to the limits imposed by our query bounds.

We will denote robust PCP classes with

$$\mathsf{PCP}\left[\begin{array}{rl} \text{query complexity:} & q(n) \\ \text{random complexity:} & r(n) \\ \text{robustness parameter:} & s(n) \\ \text{robust-soundness error:} & \rho(n) \end{array}\right].$$

Note the additional two parameters to our class compared to the original definition of a PCP class.

Having done that, adding two additional parameters to the notion of a PCP is a lot to take in, especially as it adds not one, but two new things to prove whenever we demonstrate any theorem. Thankfully, these two are relatively closely related, and so much of the time we will be able to roll them up into a single statement, which has the added benefit of being much easier to prove. The idea is that instead of putting a bound on the probability of our proofs being a certain distance away from acceptable ones, we instead simply look at the expected distance of proofs from acceptable ones.

Definition 4.5.8. Let $\rho : \mathbb{N} \to [0, 1]$. A PCP for some language L has *expected robustness* ρ if for all $x \notin L$, we have for every oracle π^*, the expected distance between any accepting view and the set of actually-seen elements is no more than $\rho(x)$. More formally,

$$\mathop{\mathbb{E}}_{\mu}\big[\Delta(\pi^*|_{Q(x)}, \mathrm{Acc}(V(x; \mu)))\big] \geq \rho(x). \tag{4.6}$$

So far, we have looked exclusively at PCPs, but these definitions are no less valid for PCPs of proximity. The only change here is since our soundness guarantee only applies to inputs that are δ-far from valid ones, similarly we will only require expected robustness on those same inputs.

Definition 4.5.9. Let $\rho : \mathbb{N} \to [0, 1]$. A PCPP for some pair language L has *expected robustness* ρ if for all (x, y) where y is δ-far from $L[x]$, we have for every oracle π^*, the expected distance between any accepting view and the set of actually-seen elements is no more than $\rho(x)$. More formally,

$$\mathop{\mathbb{E}}_{\mu}\big[\Delta(\pi^*|_{Q(x)}, \mathrm{Acc}(V(x; \mu)))\big] \geq \rho(x). \tag{4.7}$$

Now, it is important to define the relationship between expected robustness and robust soundness. The idea is that expected robustness provides a *relationship* between robust-soundness and robustness. In general, for any expected robustness ρ, the difference between robust-soundness error and robustness parameter is $1 - \rho$. Within that constraint, we can pick any ε to increase one with respect to the other; the only real bound is that both robust-soundness error and robustness parameter must be between 0 and 1.

Lemma 4.5.10 ([8, Proposition 2.10])**.** *If a PCPP has expected robustness ρ, then for all $\varepsilon \leq \rho$, it has robust-soundness error $1 - \varepsilon$ with robustness parameter $\rho - \varepsilon$.*

The original paper did not present an explicit proof of this; hence we present our own proof here.

Proof. We prove this by contradiction. Let V be a PCPP verifier with expected robustness $\rho(n)$, and let $\varepsilon(n)$ be between 0 and $\rho(n)$. Let $x \notin L$ such that V does not have robust-soundness error $1-\varepsilon$ with robustness parameter $\rho - \varepsilon$. By definition, this means

$$\mathbb{P}_{\mu}[\Delta(\pi^*|_{Q(x)}, \mathrm{Acc}(V(x;\mu))) \leq \rho - \varepsilon] > 1 - \varepsilon. \tag{4.8}$$

Pulling out the formal definition of probability, we get

$$\frac{|\{\mu \in \{0,1\}^{|\mu|} \mid \Delta(\pi^*|_{Q(x)}, \mathrm{Acc}(V(x;\mu))) \leq \rho - \varepsilon\}|}{2^{|\mu|}} > 1 - \varepsilon. \tag{4.9}$$

Define S to be the set in the numerator of Equation (4.9), and let S^C be its complement. Based on the definition of S, we get

$$\sum_{\mu \in S} \Delta(\pi^*|_{Q(x)}, \mathrm{Acc}(V(x;\mu))) \leq |S|(\rho - \varepsilon). \tag{4.10}$$

Similarly, since Hamming distance is bounded above by 1, we get

$$\sum_{\mu \notin S} \Delta(\pi^*|_{Q(x)}, \mathrm{Acc}(V(x;\mu))) \leq |S^C|. \tag{4.11}$$

Combining Equations (4.10) and (4.11), we get

$$\sum_{\mu \in \{0,1\}^{|\mu|}} \Delta(\pi^*|_{Q(x)}, \mathrm{Acc}(V(x;\mu))) \leq |S|(\rho - \varepsilon) + |S^C|. \tag{4.12}$$

From Equation (4.9), we get $|S| > 2^{|\mu|}(1-\varepsilon)$. Hence, it follows that $|S^C| < 2^{|\mu|}\varepsilon$. Thus,

$$\sum_{\mu \in 2^{|\mu|}} \Delta(\pi^*|_{Q(x)}, \mathrm{Acc}(V(x;\mu))) < |S|(\rho - \varepsilon) + 2^{|\mu|}\varepsilon. \tag{4.13}$$

Further, $|S| \leq 2^{|\mu|}$ (since $S \subseteq \{0,1\}^{|\mu|}$), so we have

$$\sum_{\mu \in 2^{|\mu|}} \Delta(\pi^*|_{Q(x)}, \mathrm{Acc}(V(x;\mu))) < 2^{|\mu|}(\rho - \varepsilon) + 2^{|\mu|}\varepsilon = 2^{|\mu|}\rho. \tag{4.14}$$

By the definition of expected value, the above equation implies

$$\mathbb{E}_{\mu}[\Delta(\pi^*|_{Q(x)}, \mathrm{Acc}(V(x;\mu))] < \rho, \tag{4.15}$$

a contradiction. □

This lemma will allow us to prove statements in terms of expected robustness instead of normal robustness. In general, we will find expected robustness much easier to prove; hence why we will be using it more often.

4.6 Zero-knowledge probabilistically-checkable proofs

A zero-knowledge probabilistically-checkable proof is a combination of the ideas of zero-knowledge proofs (as seen in Section 4.4) and probabilistically-checkable proofs (as seen in Section 4.5). Since we can model a PCP as an interaction between between a verifier and a proof (instead of a prover), we can model this interaction as being zero-knowledge as well.

Unlike interactive proofs, we cannot achieve *arbitrary* zero-knowledge guarantees: since the proof is non-interactive, we have no recourse against an attacker who simply reads the entire proof end-to-end. As such, we introduce a *query bound*: we limit our verifier to a certain number of queries, under which we retain the standard zero-knowledge restrictions.

Definition 4.6.1 ([18, Def. 8.6])**.** A probabilistically-checkable proof system is *zero-knowledge* with query bound q if for any verifier V' such that V' makes no more than $O(q(n))$ adaptive queries on an input of length n, there exists a probabilistic polynomial-time simulator S such that on input x, S can simulate every interaction of V' with the associated proof of x.[6]

The first thing to notice here is that for *validity* of PCPs, we parameterize over the proof (i.e. the one verifier should remain valid for all proofs π), for *zero-knowledge* we parameterize over the verifier V. This is because for zero-knowledge proofs we care about not revealing any information from the proof, no matter how clever we are about asking questions with the verifier. In that way, outside of the "happy path" (where $x \in L$ and the given string is the proof of x), the two notions are somewhat orthogonal: a PCP cares about how we react when $x \notin L$ but V is trusted, while zero-knowledge cares about what happens when V is not trusted, but the proof is.

Definition 4.6.2. The class PZK-PCP is the class of all languages that have a perfect zero-knowledge probabilistically-checkable proof.

4.7 Interactive probabilistically-checkable proofs

Interactive probabilistically-checkable proofs are a combination of the concepts of an interactive protocol and a probabilistically-checkable proof. The broad idea is our proof proceeds in two phases: first, the prover sends a purported proof to the verifier, after which they engage in an interactive protocol, during which the verifier can access the proof as an oracle.

Definition 4.7.1 ([24, §1.1])**.** Let L be a language, let $p, q, l\colon \mathbb{N} \to \mathbb{N}$, and let $c, s \in [0, 1]$. An *interactive probabilistically-checkable proof* for L is an interactive protocol as follows:

[6]To clarify, S does *not* have access to the proof of x, just x itself.

```
    Input: To both $P$ and $V$: a string $x$ of length $n$
    Input: To $P$ alone: A string $w$
    Output: Whether $x \in L$
1   $P$: Send an oracle $R$ to $V$;
2   $V^R$: Engage in an interactive protocol with $P$;
```

Algorithm 4.7: The IPCP protocol

Next, we need to define a few properties of IPCPs. In general we will care about IPCPs with specific bounds on these properties; later on we will spend some time working on optimizing bounds on some of these properties at the expense of others.

The first two complexities we will look at come from the interactive-proof portion of the IPCP. As a reminder, two important parameters we care about with regard to interactive proofs are the number of communication rounds (i.e., the number of times the switch tape flips) and the total amount of information sent between the two machines. Since these are supposed to relate to just the IP portion of the protocol, we need to modify the definitions slightly to exclude the information exchanged during the PCP phase.

Definition 4.7.2. The *round complexity* of an IPCP is the number of rounds in the second portion of the protocol.

Definition 4.7.3. The *communication complexity* of an IPCP is the total number of bits exchanged between P and V *except* for the message that contains R.

The third complexity parameter we care about comes from the PCP portion of the IPCP.

Definition 4.7.4. The *query complexity* of an IPCP is the total number of queries that V makes to the PCP oracle R.

As with all of our other proof-system definitions, with a new type of machine comes a corresponding class.

Definition 4.7.5. The class IPCP is the class of all languages with an interactive PCP.

Next, we give a few reasonable class inclusions regarding IPCP.

Theorem 4.7.6. $\mathsf{PCP} \subseteq \mathsf{IPCP}$ *and* $\mathsf{IP} \subseteq \mathsf{IPCP}$.

Proof. Both of these come from the definition of an interactive PCP: a regular PCP is simply the first half of the IPCP protocol where the interactive portion is useless, and an interactive proof is simply the second half of the protocol where the oracle is useless. □

The tuple-notation we used when talking about the class PCP (see Definition 4.5.3) is rather hard to read when we have this many parameters, and as such we will us the following clearer notation when talking about the various bounds on IPCP algorithms:

$$L \in \mathsf{IPCP}\begin{bmatrix} \text{round complexity:} & r \\ \text{PCP length:} & \ell \\ \text{comm. complexity:} & c \\ \text{query complexity:} & q \\ \text{soundness error:} & \varepsilon \end{bmatrix}$$

to mean the language L is a member of IPCP with the listed restrictions.

Definition 4.7.7. Let $\mathbb{F}$ be a field, and $d, m \in \mathbb{N}$. A *low-degree IPCP* is an IPCP instance with the following two properties:

1. The oracle sent by the honest prover P is an m-variable $\mathbb{F}$-polynomial Q with multidegree no more than d (i.e. $Q \in \mathbb{F}[X_{1,\dots,m}^{\leq d}]$)

2. Soundness is only required to hold against provers that send oracles that are polynomials in $\mathbb{F}[X_{1,\dots,m}^{\leq d}]$.

Similarly to what we did with normal IPCP oracles, we will use the following notation to talk about low-degree IPCP instances:

$$L \in \mathsf{IPCP}\begin{bmatrix} \text{round complexity:} & r \\ \text{PCP length:} & \ell \\ \text{comm. complexity:} & c \\ \text{query complexity:} & q \\ \text{oracle:} & \mathbb{F}[X_{1,\dots,m}^{\leq d}] \\ \text{soundness error:} & \varepsilon \end{bmatrix}.$$

We combine all the information about the degree of the oracle into one line because we have an efficient notation for multidegree-bounded polynomials, and so that we do not wind up with an exorbitant number of lines in our notation.[7]

Definition 4.7.8. The *view* of an IPCP (P, V) on input x is the random variable

$$(x, r, s_1, \dots, s_n, t_1, \dots, t_m)$$

where r is the random bits used by V, s_i are the messages from P, and t_i are the answers to V's queries to the oracle sent by P.

We denote the view of P and V on input x by $\text{View}_V^P(x)$.

4.8 Zero-knowledge IPCPs

At this point, the astute reader may have noticed a trend: after each new interactive-proof variant we introduce, we then describe how to make it zero-knowledge. We will continue this trend by showing how an interactive PCP can be made zero-knowledge.

[7]Having said that, there are still a lot of lines in this notation, but this is the best we can do.

Definition 4.8.1 ([10, §5.2])**.** An interactive PCP is *perfect zero-knowledge* with query bound b when there exists a polynomial-time simulator algorithm S such that for every interactive Turing machine $\tilde{V}$ that makes no more than b queries, $S^{\tilde{V}}(x)$ and $\mathrm{View}\langle P(x), \tilde{V}(x)\rangle$ are identically distributed.

Definition 4.8.2. The class $\mathsf{PZK\text{-}IPCP}$ is the class of all languages with a perfect zero-knowledge IPCP.

As with our other notations, we will write

$$L \in \mathsf{PZK\text{-}IPCP}\begin{bmatrix} \text{round complexity:} & r \\ \text{PCP length:} & \ell \\ \text{comm. complexity:} & c \\ \text{query complexity:} & q \\ \text{query bound:} & b \\ \text{soundness error:} & \varepsilon \end{bmatrix}$$

to show L has a perfect zero-knowledge IPCP with the listed restrictions.

Chapter 5

The PCP theorem

The PCP theorem is one of the most important and stunning results involving probabilistically-checkable proofs. In brief, it tells us how NP slots in to the hierarchy of PCP classes (based on query and randomness complexity). The reason it is so stunning is specifically how low on the hierarchy it falls: as we will see, a PCP for any problem in NP only needs to read 3 bits off of its proof and it will be able to solve the problem with high probability. Our goal with this chapter is to provide a proof of this theorem, with the secondary goal of designing the proof in such a way that it is an easy jumping-off point for the zero-knowledge variant we will be proving in Chapter 7. We start by providing a statement of the PCP theorem.

Theorem 5.0.1 (PCP theorem, [3])**.** *Any problem in* NP *has a probabilistically-checkable proof of constant query complexity and using a maximum of $O(\log n)$ random bits, and vice versa. Equivalently,* $\mathsf{NP} = \mathsf{PCP}[\log n, 1]$.

5.1 Algebraic circuits

Before we can get started on the main proof of the PCP theorem, we need to define a little more background. For much of this proof, we will be working in an algebraic landscape, and thus we would like an algebraically-focused NP-complete problem, as opposed to SAT, which is based on boolean formulae. We will find such a problem in the domain of algebraic circuits.

Definition 5.1.1 ([2, §14.1])**.** An *algebraic circuit* is a directed acyclic graph such that

1. each leaf (called an *input node*) takes values in some field $\mathbb{F}$,
2. each internal node (called a *gate*) is labeled with either $+$ or $\cdot$ (the two field operations),
3. there is one output node, and
4. each gate has in-degree no more than 2.

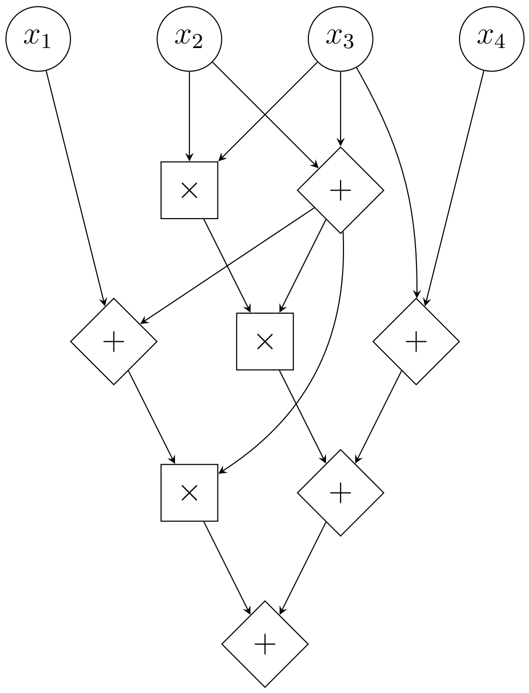

Figure 5.1: An algebraic circuit

Optionally, there may be input nodes labeled 1 and -1 as well.

We give an example of an algebraic circuit in Figure 5.1. Notice in particular that unlike the in-degree, the out-degree is unbounded (the rightmost node in the second row has out-degree three, for example). In many ways, we will see a parallel between how we think about algebraic circuits and how we think about boolean formulae. We will see this parallel both with the terminology we are about to define, and later on when we show how algebraic circuits can be used to define another NP-complete problem.

Definition 5.1.2. An *assignment* to an algebraic circuit is a labeling of its input nodes. The *result* of the assignment is the value in the output node, where the value of any $+$ gate is $a + b$ and any $\cdot$ gate is $a \times b$, where a and b are the values of the two in-nodes.

Not all assignments are created equal for our purposes. In parallel with our work on Boolean circuits, we would like to assign some sort of truthiness value to our assignments, which will give us the ability to turn this into a decision problem. As such, we define a "truthy" circuit to be one that evaluates to 1, and everything else is not.

Definition 5.1.3. An algebraic circuit has a *satisfying assignment* when there exists a labeling of its input nodes such that the value computed by the output node is 1.

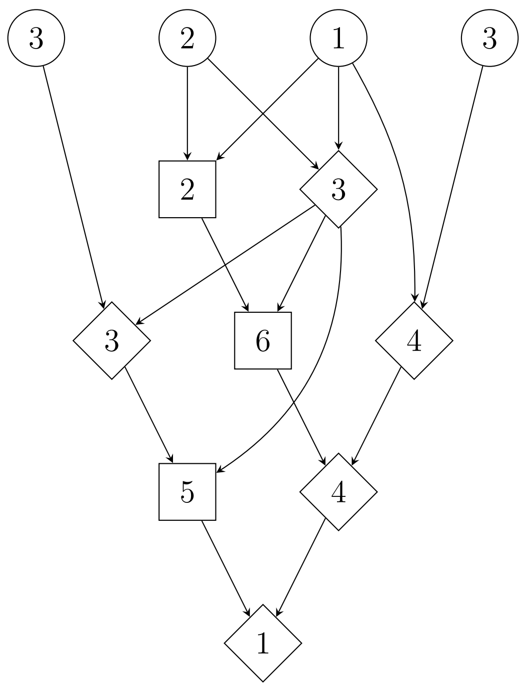

Figure 5.2: A satisfying assignment to Figure 5.1 over $\mathbb{Z}/7\mathbb{Z}$

We give a satisfying assignment to Figure 5.1 as an example in Figure 5.2. Notice that we have kept the shapes of the nodes from the original in order to communicate the underlying operation.

Satisfying assignments will form the root of our circuit classes: the problem we care about computing is whether or not a circuit has a satisfying assignment. We find this a useful formulation because it works well with arbitrary algebraic formulas: since we will be working in a primarily algebraic setting for these proofs, having an example of a NP-complete problem that is itself algebraic will make for relatively easy transformations.

Earlier, we talked about the language SAT consisting of all boolean formulae that have a satisfying assignment. In the same way that SAT relates to boolean formulae, the language we introduce, CktSAT, corresponds to algebraic circuits.

Definition 5.1.4. The language CktSAT is the language of all algebraic circuits with a satisfying assignment.

Theorem 5.1.5. CktSAT *is* NP*-complete.*

Proof. First, we need that $\mathsf{CktSAT} \in \mathsf{NP}$. Evaluating a circuit can be done in polynomial time relative to its size, so a certificate that is simply the values of each input node in the satisfying assignment will suffice.

Next, we need that there is a reduction from every problem in NP to CktSAT. We will use the fact that we already know normal SAT is NP-complete for this. First, consider the *boolean circuit problem*: similar to algebraic circuits but where the operators are boolean formulae instead of the field operators. We can construct a boolean circuit from a boolean formula by replacing every variable with an input node and every boolean operator with a gate corresponding to its formula.

From there, we can transform a boolean circuit into an algebraic circuit over the field $\mathbb{Z}/2\mathbb{Z}$. Multiplication in $\mathbb{Z}/2\mathbb{Z}$ is exactly an AND gate, so that is simply a one-to-one replacement. OR gates are slightly trickier, however. We would like the replacement to be as simple as multiplication: mapping $x \vee y$ to $x + y$. However, $T \vee T = T$ but $1 + 1 = 0$ in $\mathbb{Z}/2\mathbb{Z}$. Instead, we map $x \vee y$ to $(x + y) + (x \times y)$, which gives the answer we seek. Lastly, we map NOT gates to $x + 1$.

We have shown $\mathsf{CktSAT} \in \mathsf{NP}$ and we have shown a polynomial-time reduction from SAT to CktSAT; hence CktSAT is NP-complete. $\square$

Corresponding to CktSAT is a the pair language consisting of each element $x \in \mathsf{CktSAT}$ paired with its corresponding certificate. While often we do not think about this language explicitly, in this case the language is in fact reasonably interesting on its own. This is because a valid certificate for a circuit in CktSAT is the explicit values of a satisfying assignment, something which we will be working with later on. Further, we would like to define this because we now have a Turing machine variant that only works with pair languages, and so we cannot have it solve CktSAT directly.

Definition 5.1.6. The language CktVal is the language of all pairs (C, w), where C is an algebraic circuit and w is a satisfying assignment (i.e., $C(w) = 1$).

Theorem 5.1.7. CktVal *is in* P.

Proof. We are given a circuit and an assignment; computing the resultant value of a circuit is possible in polynomial time relative to its length. Hence, we can simply compute $C(w)$ directly and check if the answer is 1. $\square$

Lemma 5.1.8 ([8, Prop. 2.4])**.** *If* CktVal *has a PCPP, then* CktSAT *has a PCP with identical parameters.*

Proof. Let $C \in \mathsf{CktSAT}$. Then, let w be a satisfying assignment to C, and let π' be a PCPP that $(C, w) \in \mathsf{CktVal}$. Consider the oracle $\pi = (\pi', w)$: this proof can be verified using the PCPP verifier, forwarding all its queries to the new proof. $\square$

Theorem 5.1.9 ([8, Theorem 3.3])**.** *For any* $\varepsilon > 0$, CktVal *has a PCPP* (P, V) *where* P *is deterministic and polynomial-time, such that*

$$\mathsf{CktVal} \in \mathsf{PCPP}\begin{bmatrix} \textit{rand. complexity:} & \log(n) + O(\log^{\varepsilon}(n)) \\ \textit{query complexity:} & O(1/\varepsilon) \\ \textit{prox. param.:} & \Theta(\varepsilon) \\ \textit{soundness error:} & 1/2 \end{bmatrix}.$$

While important, the proof of this is somewhat involved and unfortunately outside the scope of this project. Hence, we will defer the proof of this particular theorem to the original paper [8].

5.2 The composition theorem

We now introduce a theorem that will allow us to compose robust PCPs with robust PCPPs to make a new (non-robust) PCP with improved bounds.

Theorem 5.2.1 ([8, Theorem 2.7])**.** *Let*

$$r_{\text{out}}, r_{\text{in}}, d_{\text{out}}, d_{\text{in}}, q_{\text{in}} : \mathbb{N} \to \mathbb{N}$$
$$\varepsilon_{\text{out}}, \varepsilon_{\text{in}}, \rho_{\text{out}}, \delta_{\text{in}} : \mathbb{N} \to [0,1]$$

be functions such that the following holds:

1. *The language L has a robust PCP verifier V_{out} with randomness complexity r_{out}, decision complexity d_{out}, robust-soundness error $1 - \varepsilon_{\text{out}}$, and robustness parameter ρ_{out}.*

2. CktVal *has a PCPP verifier V_{in} with randomness complexity r_{in}, query complexity q_{in}, decision complexity d_{in}, proximity parameter δ_{in}, and soundness error $1-\varepsilon_{\text{in}}$.*

3. *For every $n \in \mathbb{N}$, $\delta_{\text{in}}(d_{\text{out}}(n)) \leq \rho_{\text{out}}(n)$.*

Then L has a standard PCP V_{comp} with

a. *randomness complexity $r_{\text{out}}(n) + r_{\text{in}}(d_{\text{out}}(n))$,*

b. *query complexity $q_{\text{in}}(d_{\text{out}}(n))$,*

c. *decision complexity $d_{\text{in}}(d_{\text{out}}(n))$, and*

d. *soundness error $1 - \varepsilon_{\text{out}}(n)\varepsilon_{\text{in}}(d_{\text{out}}(n))$.*

We present the algorithm here, in Algorithm 5.1, but leave the full proof of completeness and soundness to the original construction in [8].

Corollary 5.2.2 ([17, Corollary 3.11])**.**

$$\mathsf{PCP}\begin{bmatrix} \textit{query complexity:} & r \\ \textit{random complexity:} & q \\ \textit{robustness parameter:} & s \\ \textit{robust-soundness error:} & \Omega(1) \end{bmatrix} \subseteq \mathsf{PCP}\begin{bmatrix} \textit{query complexity:} & r + \log(n) \\ \textit{random complexity:} & 1 \\ \textit{robustness parameter:} & s \\ \textit{robust-soundness error:} & \Omega(1) \end{bmatrix}. \tag{5.1}$$

Proof. Let

$$L \in \mathsf{PCP}\begin{bmatrix} \text{query complexity:} & r \\ \text{random complexity:} & q \\ \text{robustness parameter:} & s \\ \text{robust-soundness error:} & \Omega(1) \end{bmatrix}. \tag{5.2}$$

Then as per Theorem 5.1.9, we have

$$\mathsf{CktVal} \in \mathsf{PCPP}\begin{bmatrix} \text{rand. complexity:} & \log(n) + O(\log^{\varepsilon}(n)) \\ \text{query complexity:} & O(1/\varepsilon) \\ \text{prox. param.:} & k\varepsilon \\ \text{soundness error:} & 1/2 \end{bmatrix}, \tag{5.3}$$

for some $\varepsilon > 0$ and constant k. Define $\varepsilon = \rho/k$. Then, as per Theorem 5.2.1, we have L as desired. □

```
1  Pick a random $R \in \{0,1\}^{r_{\text{out}}}$;
2  Simulate $V_{\text{out}}$ on $x$ and random coins $R$;
3  Let $I_{\text{out}} = (i_1, \dots, i_{q_{\text{out}}})$ be the query responses of $V_{\text{out}}$;
4  Let $D_{\text{out}}$ be the decision of $V_{\text{out}}$;
5  Run $V_{\text{in}}$ on input $D_{\text{out}}$ and random coin tosses;
6  Let $I_{\text{in}} = ((b_1, j_1), \dots (b_{q_{\text{in}}}, j_{q_{\text{in}}}))$ be the query responses and $D_{\text{in}}$ be the result
     of the previous line's simulation;
7  for $\ell \in [q_{\text{in}}]$ do
8      if $b_\ell = 0$ then
9          $k_\ell \leftarrow (\text{out}, i_{j_\ell})$;
10     else
11         $k_\ell \leftarrow (R, i_{j_\ell})$;
12     end
13 end
14 return $(k_1, \dots, k_{q_{\text{in}}})$ and $D_{\text{in}}$;
```

Algorithm 5.1: A composed PCP [8, Theorem 2.7]

5.3 Alphabet reduction

One major difference between PCPs and many other types of computational model is that the size of the alphabet proofs are written can have a non-trivial impact. This is because when we talk about query complexity, we are no longer simply looking at classes with the coarseness of e.g. P, but are instead looking at much finer complexities.

Despite this, we would like to be able to construct some PCPs over non-binary languages. Non-binary PCPs can be more intuitive in certain circumstances, and the complexity bounds can be more apparent. As such, we would like a theorem for how changing the language we work over affects our parameters.

Theorem 5.3.1 ([8, Lemma 2.13]). *Let L be a language with a PCP over the language $\{0,1\}^a$ such that*

$$L \in \mathsf{PCP}_{\{0,1\}^a}\left[\begin{array}{rl} \textit{query complexity:} & q \\ \textit{random complexity:} & r \\ \textit{robustness parameter:} & s \\ \textit{robust-soundness error:} & \rho \end{array}\right].$$

Then L has a PCP over the language $\{0,1\}$ such that

$$L \in \mathsf{PCP}_{\{0,1\}}\left[\begin{array}{rl} \textit{query complexity:} & O(aq) \\ \textit{random complexity:} & r \\ \textit{robustness parameter:} & s \\ \textit{robust-soundness error:} & \Omega(\rho) \end{array}\right].$$

Proof. We show that Algorithm 5.2 is a boolean reduction algorithm.

By definition, we know if $x \in L$, then V will always accept when given proof π; thus line 5 will always pass. Further, since $\tau(\gamma)$ is defined as being $\mathsf{ECC}(\pi(\gamma))$, the

```
Input: A PCP $(P, V)$ over the language $\{0,1\}^a$ for $L$, along with input $x$
Input: A error-correcting code $\mathsf{ECC} : \{0,1\}^a \to \{0,1\}^b$, where $b = O(a)$
Output: A PCP over the language $\{0,1\}$ for $L$
/* Proof                                                    */
1  Let $\pi$ be the original accepting proof of $(P, V)$;
2  Define the proof oracle $\tau$ by $\tau(\gamma) = \mathsf{ECC}(\pi(\gamma))$;
3  return $(\pi, \tau)$;
/* Verifier                                                 */
4  Compute the set of queries $Q$ of $V$ on input $x$;
5  Simulate $V$ on input $x$ and proof $\pi$;
6  if $V$ rejects then
7      reject;
8  end
9  for $\gamma \in Q$ do
10     Compute $\mathsf{ECC}(\pi_a(\gamma))$;
11     Query $\tau(\gamma)$;
12     if $\mathsf{ECC}(\pi_a(\gamma)) \neq \tau(\gamma)$ then
13         reject;
14     end
15 end
16 accept;
```

Algorithm 5.2: A boolean reduction of a PCP [17, Construction 3.6]

check in line 12 will always pass. Hence, when given an $x \in L$ and the correct proof, Algorithm 5.2 will always pass.

If $x \notin L$, then the nature of our error-correcting code is that so long as our input proof has at least expected distance ρ from any correct view, our new input will as well. The reason we need the error-correcting code here is that when we move to a smaller alphabet, we can now flip a single bit within each character and it will only count for $1/a$ of the distance that it did before, since in the larger alphabet it throws off the whole character, while here it only throws off the single bit we flipped. □

This is relatively good: it tells us that query complexity is not made much worse by changing around our language. It is also worth pointing out that one can always scale *up* the size of their language for free: simply zero-padding the extra bits will leave you with a functioning PCP for a larger language. This is particularly useful since later on we will be working with languages whose base is not $\{0,1\}$ but some finite field $\mathbb{F}$, and this way we can scale them to some $\{0,1\}^a$ where $2^a \geq |F|$ without worry.

5.4 Robust total-degree test

An important algorithm to have in our back pocket for the rest of this chapter will be a robust way for a PCP verifier to check if a given polynomial is in fact a low-total-degree polynomial or not. This will allow us to test for membership in $\mathrm{RM}[\mathbb{F}, m, d]$ relatively easily.

Theorem 5.4.1 ([27, Prop. 5.7])**.** *Let $\delta > 0$, and $k, m, d \in \mathbb{N}^+$. Let $\mathbb{F}$ be a finite field with $|\mathbb{F}| > 25k$. Then there exists a test that,*

1. *has oracle access to a function $F \colon \mathbb{F}^m \to \mathbb{F}^k$,*
2. *makes $|F|$ queries to F,*
3. *runs in time $\mathsf{poly}(|\mathbb{F}|, m, d, k)$,*
4. *accepts with probability 1 if $F \in \mathrm{RM}^k[\mathbb{F}, m, d]$, and*
5. *if $\Delta(F, \mathrm{RM}^k[\mathbb{F}, m, d]) > \varepsilon$, then the expected distance between the tester's view of F and any accepting view is in $\Omega(\varepsilon)$.*

```
Input: A function $F \colon \mathbb{F}^m \to \mathbb{F}^k$
Output: Whether $F \in \mathrm{RM}[\mathbb{F}, m, d]$
1 Sample a uniformly random line $L \in \mathbb{F}^m$;
2 for $\ell \in L$ do
3     $x_i \leftarrow$ query $F$ at $\ell$;
4 end
5 if $F|_{\{x_i | i \in |L|\}}$ agrees with a polynomial $p \in \mathbb{F}^{\leq d}[X_{1,\dots,m}]$ then
6     accept;
7 else
8     reject;
9 end
```

Algorithm 5.3: A robust low-degree test [27, Prop. 5.7]

The algorithm in the previous proof is described in Algorithm 5.3. The complexity of the proof (in particular, the proof of robustness) is complex enough that we will leave it in full generality to [27]. However, we will note that by definition, if $F \in \mathrm{RM}[\mathbb{F}, m, d]$, then by definition the restriction of F to a subset L will always agree with a polynomial p; in particular it will agree with $p = F$.

5.5 PCPPs for polynomial summation

Definition 5.5.1 ([17, Def. 4.1])**.** The language Sum is the set of all ordered pairs $((\mathbb{F}, 1^m, 1^d, H, \gamma), F)$, where

- $\mathbb{F}$ is a finite field,
- $m, d \in \mathbb{N}$,
- $H \subseteq \mathbb{F}$,

- $\gamma \in F$, and
- $F \in \mathbb{F}^{\leq d}[X_{1,\dots,m}]$ with
$$\sum_{b \in H^m} F(b) = \gamma.$$

While the general language Sum can be very useful, it can be a little unwieldy, especially since it includes polynomials over all finite fields (so determining the language can be a little strange). As such, we define a subset of Sum where we restrict the possible values of everything except F to a fixed constant. This is much easier to work with, and it will still be very useful to us later on.

Definition 5.5.2. For a fixed finite field $\mathbb{F}$, $m, d \in \mathbb{N}$, $H \subseteq \mathbb{F}$, and $\gamma \in \mathbb{F}$, the language $\mathsf{Sum}[\mathbb{F}, m, d, H, \gamma]$ is the subset of Sum where the co-named parameters in Definition 5.5.1 are equal to their fixed values.

For simplicity, we will often say $F \in \mathsf{Sum}[\mathbb{F}, m, d, H, \gamma]$ in cases where we mean $((\mathbb{F}, m, d, H, \gamma), F) \in \mathsf{Sum}[\mathbb{F}, m, d, H, \gamma]$ to reduce redundancy, since every parameter save F is fixed in this definition. Now that we have defined Sum, we demonstrate a PCPP for it.

Finally, it is time to introduce a verifier for Sum.

Theorem 5.5.3 ([17, Lemma 4.2])**.** *Let $\delta > 0$, $\mathbb{F}$ be a finite field, $H \subseteq \mathbb{F}$, $\gamma \in \mathbb{F}$, and $m, d \in \mathbb{N}$ such that $\frac{md}{|\mathbb{F}|} < \delta$ and $d > |H| + 1$. Then there exists a PCP of proximity for $\mathsf{Sum}[\mathbb{F}, m, d, H, \gamma]$ over the alphabet $\mathbb{F}^{m+1}$ with proximity parameter δ and robustness parameter $\rho = \Omega(\delta)$. Further, the verifier makes $O(|\mathbb{F}|)$ queries to F and π and the proof length is $O(|\mathbb{F}|^m)$. Alternately,*

$$\mathsf{Sum}[\mathbb{F}, m, d, H, \gamma] \in \mathsf{PCPP}_{\mathbb{F}^{m+1}} \left[\begin{array}{rl} \textit{rand. complexity:} & \mathsf{poly}(n) \\ \textit{query complexity:} & |\mathbb{F}| \\ \textit{prox. param.:} & \delta \\ \textit{soundness error:} & \varepsilon \\ \textit{RS error:} & s \\ \textit{robustness param:} & \Omega(\delta) \end{array} \right].$$

Proof. We construct such an algorithm as Algorithm 5.4. To show this is a PCPP, we will first show that the verifier will always accept when given a valid input and proof; following that we will show it will correctly reject for all inputs not near any polynomial in the language.

Let $F \in \mathsf{Sum}[\mathbb{F}, m, d, H, \gamma]$ and let π be the honest proof. In this case, our definition of g_i is as

$$g_i(x_1, \dots, x_i) = \sum_{b \in H^{m-i}} F(x_1, \dots, x_i, b_1, \dots, b_{m-i}). \tag{5.4}$$

```
proof
    for $i \in \{1, \dots, m-1\}$ do
        $g_i(X) \leftarrow \sum_{b \in H^{m-i}} F(X, b)$;
    end
    Define $\pi \colon \mathbb{F}^{m-1} \to \mathbb{F}^{m-1}$ by

        $$\pi(c_1, \dots, c_{m-2}, \alpha) = (g_1(\alpha), g_2(c_1, \alpha), \dots, g_{m-1}(c_1, \dots, c_{m-2}, \alpha))$$

     for each $(c_1, \dots, c_{m-2}, \alpha) \in \mathbb{F}^{m-1}$;
    return $\pi$;
end
verifier
    Sample $c \in \mathbb{F}^{m-1}$ at random;
    for $\alpha \in \mathbb{F}$ do
        Query $\pi(c_1, \dots, c_{m-2}, \alpha)$;
        Query $F(c_1, \dots, c_{m-1}, \alpha)$;
    end
    for $i \in \{1, \dots, m-1\}$ do
        if $g_i \notin \mathbb{F}[X^{\leq d}_{1,\dots,i}]$ then
            reject;
        end
    end
    if $\sum_{b \in H} g_1(b) \neq \gamma$ then
        reject;
    end
    for $i \in \{1, \dots, m-2\}$ do
        Check
            $$\sum_{b \in H} g_{i+1}(c_1, \dots, c_i, b) = g_i(c_1, \dots, c_i);$$
    end
    Check
        $$\sum_{b \in H} F(c, b) = g_{m-1}(c);$$
    Run Algorithm 5.3 on $F$, with proximity parameter $\delta_R = \min(\delta, 1/5)$;
    accept if and only if the prior test passes;
end
```

Algorithm 5.4: A robust PCPP for Sum [17, Construction 4.3]

In particular, this means

$$\begin{aligned}\sum_{b\in H} g_1(b) &= \sum_{b\in H}\sum_{c\in H^{m-1}} F(b, c_1, \ldots, c_{m-1}) \\ &= \sum_{b\in H^m} F(b_1, \ldots, b_m) \\ &= \gamma.\end{aligned}$$

Hence, the check in line 19 will always pass.

Further, we have that

$$\begin{aligned}g_i(x_1, \ldots, x_i) &= \sum_{b\in H^{m-i}} F(x_1, \ldots, x_i, b_1, \ldots, b_{m-i}) \\ &= \sum_{a\in H}\sum_{b\in H^{m-i-1}} F(x_1, \ldots, x_i, a, b_1, \ldots, b_{m-i-1}) \\ &= \sum_{a\in H} g_{i+1}(x_1, \ldots, x_i, a).\end{aligned}$$

Hence, the check in line 23 will always pass.

The check in line 25 follows immediately from the definition of g_{m-1}, and thus it will pass. Since F is in our language, it follows that it has degree at most d and therefore the check in line 26 will pass. Since all the checks pass, it follows that Algorithm 5.4 will always accept when given an F in the language and an honest proof.

Next, let F be δ-far from $\mathsf{Sum}[\mathbb{F}, m, d, H, \gamma]$. We break this into two cases: those where F is δ_{RM}-far from $\mathrm{RM}[\mathbb{F}, m, d]$ and those where F is δ_{RM}-close to $\mathrm{RM}[\mathbb{F}, m, d]$.

Before that, note that the simulation of the low-degree test in line 26 makes $O(\mathbb{F})$ queries (as per Theorem 5.4.1), and the rest of the algorithm makes $2|\mathbb{F}|$ queries, all in lines 11 and 12. Both of these are independent of the length of the input, meaning the proportion of the queries made by each of the two halves of the verifier are constant.

If F is δ_{RM}-far from $\mathrm{RM}[\mathbb{F}, m, d]$, then Algorithm 5.3 will always have its view at least $\Omega(\delta)$ from any accepting view as per Theorem 5.4.1. Since the low-degree test is a constant fraction of the total number of queries, this means that the expected distance between the low-degree verifier's view and any accepting view is also $\Omega(\delta)$. Thus, V has expected robustness $\Omega(\delta)$ in this case.

If F is δ_{RM}-close to $\mathrm{RM}[\mathbb{F}, m, d]$, then we need to introduce a new lemma.

Lemma 5.5.4 ([17, Lemma 4.4]). *Let $\tilde{F}\colon \mathbb{F}^m \to \mathbb{F}$ be δ_Σ-far from $\mathsf{Sum}[\mathbb{F}, m, d, H, \gamma]$, but δ_{RM}-close to $\mathrm{RM}[\mathbb{F}, m, d]$ for some $\frac{md}{|\mathbb{F}|} < \delta_{\mathrm{RM}} \le \delta_\Sigma$ and $d \ge |H| + 1$. Then, for all proofs π^*,*

$$\mathop{\mathbb{E}}_{c\leftarrow\mathbb{F}^{m-1}}\left[\Delta\Big((\pi^*(c_1, \ldots, c_{m-2}, \alpha)_{\alpha\in\mathbb{F}}, \tilde{F}(c_1, \ldots, c_{m-1}, \alpha)_{\alpha\in\mathbb{F}}), \mathrm{Acc}(V)\Big)\right] \ge \frac{\min(\delta_{\mathrm{RM}}, 1 - 4\delta_{\mathrm{RM}})}{2}. \quad (5.5)$$

We leave this lemma unproven, as the proof is mostly nasty algebra. In brief, it gives us a specific bound on the distance between the accepting views of V and the set of polynomials given by the verifier.

By Lemma 5.5.4, we know that the view of the verifier is at least $\min(\delta_{\mathrm{RM}}, 1-4\delta_{\mathrm{RM}})/2$ from any accepting view. Since we defined $\delta_{\mathrm{RM}} = \min(\delta, 1/5)$, this means the distance is at least $\min(\delta/2, 1/10)$, which is in $\Omega(\delta)$. Since a constant proportion of Algorithm 5.4's queries occur outside of line 26, this means the total distance between the view of V and any accepting view is still $\Omega(\delta)$. Thus, V has expected robustness $\Omega(\delta)$ in this case as well.

Since V has expected robustness $\Omega(\delta)$ regardless of whether or not F is δ_{RM}-close to $\mathrm{RM}[\mathbb{F}, m, d]$, this means that V has expected robustness $\Omega(\delta)$ overall. Hence, Algorithm 5.4 is a robust PCPP for the language Sum. □

5.6 Properties of O3SAT

To begin defining a zero-knowledge PCP for NP and NEXP, we must first define a few convenience polynomials.

Definition 5.6.1. Let B be an instance of $\mathsf{O3SAT}$ and let $\hat{B}$ be the low-degree extension of $1 - B$. Let $\hat{A}\colon \mathbb{F}^{m_2} \to \mathbb{F}$. We define $g_{\hat{A}}\colon \mathbb{F}^{m_1+3m_2+3} \to \mathbb{F}$ to be

$$\begin{aligned} &g_{\hat{A}}(z, b_1, b_2, b_3, a_1, a_2, a_3) \\ &\qquad = \hat{B}(\gamma_1(z), \gamma_2(b_1), \gamma_2(b_2), \gamma_2(b_3), a_1, a_2, a_3)\prod_{i=1}^{3}(\hat{A}(b_i) + a_i - 1). \end{aligned} \tag{5.6}$$

Definition 5.6.2. Let B be an instance of $\mathsf{O3SAT}$ and let $\hat{B}$ be the low-degree extension of $1 - B$. Let $\hat{C}\colon \mathbb{F}^{m_2+k} \to \mathbb{F}$. We define $h_{\hat{C}}\colon \mathbb{F}^{m_1+3m_2+3+3k} \to \mathbb{F}$ to be

$$\begin{aligned} &h_{\hat{C}}(z, b_1, b_2, b_3, a_1, a_2, a_3, c_1, c_2, c_3) \\ &\qquad = \hat{B}(\gamma_1(z), \gamma_2(b_1), \gamma_2(b_2), \gamma_2(b_3), a_1, a_2, a_3)\prod_{i=1}^{3}(\hat{C}(b_i, c_i) + (a_i - 1)\delta_{0^k}(c_i)). \end{aligned} \tag{5.7}$$

As a reminder, δ is the low-multidegree polynomial defined in Equation (2.5) that for all $x, y \in H^n$, $\delta_x(y) = [x = y]$.

From the above lemma, we get two equivalent statements to whether $B \in \mathsf{O3SAT}$. These equivalent statements will play an important role in our proof of Theorem 7.5.2.

From here, we will prove a nice relationship between these two polynomials. This correspondence will come in handy when proving the next lemma, which will eventually give us some nice equivalent statements to $\mathsf{O3SAT}$.

Lemma 5.6.3. *If* $\hat{A}(X) = \sum_{c \in H^k} \hat{C}(X, c)$*, then*

$$\sum_{c_1, c_2, c_3 \in H^k} h_{\hat{C}}(z, b_1, b_2, b_3, a_1, a_2, a_3, c_1, c_2, c_3) = g_{\hat{A}}(z, b_1, b_2, b_3, a_1, a_2, a_3). \tag{5.8}$$

We will use this lemma to prove that two important conditions are equivalent to B being an element of O3SAT. Our upcoming PZK-PCP for O3SAT will leverage these conditions heavily in order to prove its correctness and soundness.

Lemma 5.6.4 ([17, Claim 6.5])**.** *Let $B\colon \{0,1\}^m \to \{0,1\}$. Then the following statements are equivalent:*

1. *$B \in \mathsf{O3SAT}$;*
2. *there exists a polynomial $\hat{A}\colon \mathbb{F}^m \to \mathbb{F}$ such that for all $z \in H^{m_1}$, $b_1, b_2, b_3 \in H^{m_2}$, and $a_1, a_2, a_3 \in \{0,1\}$,*
$$g_{\hat{A}}(z, b_1, b_2, b_3, a_1, a_2, a_3) = 0; \tag{5.9}$$
3. *there exists a polynomial $\hat{C}\colon \mathbb{F}^{m_2+k} \to \mathbb{F}$ such that for all $z \in H^{m_1}$, $b_1, b_2, b_3 \in H^{m_2}$, and $a_1, a_2, a_3 \in \{0,1\}$,*
$$\sum_{c_1,c_2,c_3 \in H^k} h_{\hat{C}}(z, b_1, b_2, b_3, a_1, a_2, a_3, c_1, c_2, c_3) = 0. \tag{5.10}$$

Proof. Lemma 5.6.3 tells us that Items 2 and 3 are equivalent. Hence, all we need to show is that Items 1 and 2 are equivalent.

($\Rightarrow$) Let $B \in \mathsf{O3SAT}$, and let $\hat{A}$ be the low-degree extension of a satisfying assignment A. Further fix $z, b_1, b_2, b_3, a_1, a_2, a_3 \in \{0,1\}$. In the case where $B(z, b_1, b_2, b_3, a_1, a_2, a_3) = 1$, it follows $\hat{B} = 1 - B = 0$, so we are done. In the other case, Otherwise, we know A is a satisfying assignment; thus for some $i \in \{1,2,3\}$ $A(b_i) \neq a_i$, since otherwise B must by definition evaluate to 1. Hence, by the definition of $\hat{A}$, $\hat{A}(b_i) = 1 - a_i$. Hence, the product on the right-hand side of Equation (5.6) evaluates to 0 and thus $g_{\hat{A}}$ is zero for all parameters.

($\Leftarrow$) Conversely, let $\hat{A}$ be a polynomial such that $g_{\hat{A}}$ is zero on all inputs. Define A to be the function

$$\begin{aligned} A : \{0,1\}^s &\to \{0,1\} \\ b &\mapsto [\hat{A}(b) = 1]. \end{aligned}$$

Let $z \in \{0,1\}^r$ and $b_1, b_2, b_3 \in \{0,1\}^s$ be arbitrary. Further, let $a_1, a_2, a_3 \in \{0,1\}$ such that $B(z, b_1, b_2, b_3, a_1, a_2, a_3) = 0$. If no such a_i exist, then we know whatever the output of A, B will evaluate to 1; hence we are done. Otherwise, it must be that there exists some $i \in \{1,2,3\}$ such that $\hat{A}(b_i) = 1 - a_i$ and thus $A(b_i) = \hat{A}(b_i)$ by definition. Thus, it follows that $A(b_i) \neq a_i$. By contradiction, therefore, it must be the case that if $A(b_i) = a_i$ for all i, then B evaluates to 1, and thus $A \in \mathsf{O3SAT}$. □

5.7 A PCP for NP

Finally, it is time to construct our PCP for all of NP. We will start by constructing a PCP that does not have particularly nice parameters, and then use several of the theorems we have shown earlier in this chapter to show that there exists a PCP with the parameters we would like. We do this so that our constructed algorithm is reasonably intuitive and explainable.

Theorem 5.7.1 ([17, Theorem 6.3])**.**

$$\mathsf{NEXP} \subseteq \mathsf{PCP}_{\Sigma(n)}\left[\begin{array}{rl} \textit{query complexity:} & \mathsf{poly}(n) \\ \textit{random complexity:} & \mathsf{poly}(n) \\ \textit{robustness parameter:} & s \\ \textit{robust-soundness error:} & \Omega(1) \end{array}\right].$$

where $\Sigma(n)$ *is any alphabet with* $|\Sigma(n)| \in \mathsf{poly}(n,q)$*.*

Proof. We construct a PZK-PCP for $\mathsf{O3SAT}$, a NEXP-complete language, in Algorithm 5.5. First, note that the combined effect of the three "repeat" loops is to ensure that the number of queries taken up by each of the two inner portions is no more than 1/3 of the total number of queries. This will be useful to us later on when we try to prove soundness.

We know Algorithm 5.3 runs in polynomial time, as does the verifier for Algorithm 5.4. Computing the value of a known polynomial is also polynomial time, as is querying values; hence the entire algorithm runs in polynomial time.

Next, we show Algorithm 5.5 will accept with probability at least 2/3 when given a $B \in \mathsf{O3SAT}$ and the honest proof. As per Lemma 5.6.4, the claim in line 5 is true. Hence, line 13 will always accept. From there, Algorithm 5.4 will always succeed, as per its own soundness guarantee. Further, the claim it outputs will always be true, so line 28's test will always succeed. Hence, Algorithm 5.5 will always accept given an honest prover and a $B \in \mathsf{O3SAT}$.

Next, we show Algorithm 5.5 will reject with probability at least 2/3 when given a $B \notin \mathsf{O3SAT}$. We aim to show this in two cases. The first is where π_C is ε-close to $\mathrm{RM}[\mathbb{F}, m_2+k, m_2+d]$, in which case we will show line 13 will reliably fail. The second is where π_C is ε-far, in which case we will show line 21 will reliably fail.

Define $\varepsilon = 1/100$. If π_C is ε-close to $\mathrm{RM}[\mathbb{F}, m_2+k, (m_2+k)d]$, then let $\hat{C} \in \mathrm{RM}[\mathbb{F}, m_2+k, (m_2+k)d]$ be the closest element to π_C. By the definition of distance and since π_C is ε-close, this means $\Delta(\hat{C}, \pi_C) \leq \varepsilon$. Next, note that a random evaluation of h_f depends on three random evaluations of f; as such it must be that $\Delta(h_{\hat{C}}, h_{\pi_C}) \leq 3\varepsilon$.

By Lemma 5.6.4, there exists $z \in H^{m_1}$, $b_1, b_2, b_3 \in H^{m_2}$, and $a_1, a_2, a_3 \in \{0,1\}$ such that

$$\sum_{\substack{z \in H^{m_1} \\ b_1,b_2,b_3 \in H^{m_2}}} \sum_{a_1,a_2,a_3 \in \{0,1\}} \delta_{(z,b,a)}(X) \sum_{c_1,c_2,c_3 \in H^k} h_{\hat{C}}(X, c_1, c_2, c_3) \tag{5.11}$$

is a nonzero polynomial in $\mathbb{F}[X_{1,\dots,m_1+m_2+m_3}^{O(d_B|H|)}]$. Hence, the claim our verifier makes in line 21 is false with probability $1 - O((m_1+m_2+m_3)d_b|H|/|\mathbb{F}|)$.

If π_C is ε-far from $\mathrm{RM}[\mathbb{F}, m_2+k, (m_2+k)d]$, then the verifier in line 21's view is $\Omega(\varepsilon)$-far from accepting, as per Theorem 5.4.1.

What we have now shown is that regardless of the distance of π_C from $\mathrm{RM}[\mathbb{F}, m_2+k, m_2+d]$, at least one of the two portions of our view is $\Omega(\varepsilon)$-far from an accepting

```
    Input: A 3-CNF $B\colon \{0,1\}^{r+3s+3} \to \{0,1\}$
    Output: Whether $B$ is implicitly satisfiable
 1  proof
 2      Let $A\colon \{0,1\}^n \to \{0,1\}$ be a satisfying assignment for $B$;
 3      Choose $\hat{C} \in \mathbb{F}[X_{m_2+k}^{\leq 2(|H|-1)}]$ randomly such that $\sum_{c \in H^k} \hat{C}(b,c) = A(\gamma_2, b)$ for
         all $b \in H^{m_2}$;
 4      for $\tau \in \mathbb{F}^{m_1+3m_2+3}$ do
 5          Let $\pi_\tau$ be a PZK-PCPP for the claim
                $$\sum_{\substack{z \in H^{m_1} \\ b_1,b_2,b_3 \in H^{m_2}}} \sum_{\substack{a \in \{0,1\}^3 \\ c_1,c_2,c_3 \in H^k}} \delta_{(z,b,a)}(\tau) h_{\hat{C}}(\tau, c_1, c_2, c_3) = 0;$$
 6      end
 7      return $(\pi_C, (\pi_\tau)_{\tau \in \mathbb{F}^{m_1+3m_2+3}})$;
 8  end
 9  verifier
10      $q_a, q_b \leftarrow 0$;
11      repeat
12          repeat
13              Run Algorithm 5.3 on the proof, with $\varepsilon = 1/100$ and $\delta = 1/2$;
14              Add the total number of queries in the last line to $q_a$;
15              if the above rejects then
16                  reject;
17              end
18          until $2q_a > q_b$;
19          repeat
20              Choose $\tau \in \mathbb{F}^{m_1} \times (\mathbb{F}^{m_2})^3 \times \mathbb{F}^3$ at random;
21              Simulate Algorithm 5.4 with proof $\pi_\tau$;
22              Add the total number of queries in the last line to $q_b$;
23              for $i \in \{1,2,3\}$ do
24                  Query $\hat{C}$ at $(\nu_i, \eta_i)$;
25                  Add 1 to $q_b$;
26              end
27              Compute $h_{\hat{C}}(\tau, \eta_1, \eta_2, \eta_3)$;
28              if the claim in line 21 is false then
29                  reject;
30              end
31          until $2q_b > q_a$;
32      until $2q_a > q_b$ and $2q_b > q_a$;
33      accept;
34  end
```

Algorithm 5.5: A PZK-PCP for O3SAT [17, Construction 6.4]

portion of the view. Since we know that these portions take up at least 1/3 of the view,[1] it means that the total distance of the verifier's view from an accepting view is at least $\Omega(\varepsilon/3) = \Omega(\varepsilon)$. Hence, Algorithm 5.5 has expected robustness $\Omega(\varepsilon)$. □

Corollary 5.7.2.

$$\mathsf{NP} \subseteq \mathsf{PCP}_{\Sigma(n)}\left[\begin{array}{rl} \textit{query complexity:} & \log(n) \\ \textit{random complexity:} & \mathsf{poly}(\log(n)) \\ \textit{robustness parameter:} & s \\ \textit{robust-soundness error:} & \Omega(1) \end{array}\right]. \tag{5.12}$$

Proof. For this, we leverage our work from Theorem 5.7.1. In that, we used the fact that O3SAT is a NEXP-complete problem. However, the Cook-Levin variant we proved in Theorem 2.2.30 is even more general than that: in particular it showed that log-length O3SAT is in fact NP-complete. Hence, if we adjust the length of our input to be $\log(n)$, the inclusion here follows. □

At last, it is time for our proof of the main PCP theorem.

Theorem 5.7.3 (Theorem 5.0.1, restated)**.** $\mathsf{NP} \subseteq \mathsf{PCP}[\log(n), 1]$.

Proof. To show this, we will take the PCP for NP that we showed in Corollary 5.7.2, which has relatively weak bounds and an arbitrary alphabet, and transform it into one with the exact parameters we seek. We do this first through an alphabet reduction (as per Theorem 5.3.1) and then through proof-composing it with the algorithm for CktVal with Corollary 5.2.2, we can get a constant query-complexity PCP. Since several of these class inclusions involve modifying a small number of parameters in a relatively complex class, we will be highlighting the changed parameters in each inclusion statement.

Let $q^*(n) \leq 2^{\mathsf{poly}(n)}$ be arbitrary. This will be the query complexity of our final PCP after we do all the class inclusions. As per Corollary 5.7.2, we know that

$$\mathsf{NP} \subseteq \mathsf{PCP}_{\Sigma(n)}\left[\begin{array}{rl} \text{query complexity:} & \log(n) \\ \text{random complexity:} & \mathsf{poly}(\log(n)) \\ \text{robustness parameter:} & s \\ \text{robust-soundness error:} & \Omega(1) \end{array}\right]. \tag{5.13}$$

where $|\Sigma(n)| \in \mathsf{poly}(n)$. Corollary 5.7.2 only guarantees this inclusion for alphabets of $\{0,1\}^a$, however a PCP over $\Sigma(n)$ is equivalent to a PCP over $\{0,1\}^{\log_2(|\Sigma(n)|)}$ by a simple relabeling of alphabet items, so this inclusion still holds.

Next, we perform an alphabet reduction: by Theorem 5.3.1,

$$\begin{aligned} &\mathsf{PCP}_{\Sigma(n)}\left[\begin{array}{rl} \text{query complexity:} & \log(n) \\ \text{random complexity:} & \mathsf{poly}(\log(n)) \\ \text{robustness parameter:} & s \\ \text{robust-soundness error:} & \Omega(1) \end{array}\right] \\ \subseteq\ &\mathsf{PCP}_{\{0,1\}}\left[\begin{array}{rl} \text{query complexity:} & \log(n) \\ \text{random complexity:} & \mathsf{poly}(\log(n)) \\ \text{robustness parameter:} & s \\ \text{robust-soundness error:} & \Omega(1) \end{array}\right]. \end{aligned} \tag{5.14}$$

[1] We note that the fraction 1/3 is actually arbitrary here, so long is it is a constant nonzero fraction; 1/3 is just a relatively easy number to deal with and it will make convergence faster.

Next, we perform proof composition. By Corollary 5.2.2, we have that

$$\mathsf{PCP}_{\{0,1\}}\left[\begin{array}{rl} \text{query complexity:} & \log(n) \\ \text{random complexity:} & \mathsf{poly}(\log(n))) \\ \text{robustness parameter:} & s \\ \text{robust-soundness error:} & \Omega(1) \end{array}\right] \subseteq \mathsf{PCP}_{\{0,1\}}[\log(n), 1]. \tag{5.15}$$

By combining the inclusions in Equations (5.13) to (5.15), we get that $\mathsf{NP} \subseteq \mathsf{PCP}[\log(n), 1]$, as desired. □

This has shown that there *exists* a constant-query PCP for any language in NP, but it would be nice of us to know what that query count is. First, note that there does exist a finite upper bound on the number of constant queries: since we can always perform a polynomial-time reduction and retain a functioning PCP, the necessary number of queries for any NP-complete problem is sufficient to prove any problem in NP. Johan Håstad showed that we can solve the SAT problem with only three queries, so it follows that any language in NP can be solved by querying only three bits.

Theorem 5.7.4 ([20]). *Any language in* NP *has a* PCP *that queries a maximum of* 3 *bits of the proof and uses* $O(\log n)$ *random bits.*

Chapter 6

Low-degree zero-knowledge IPCPs

Now that we can construct a zero-knowledge $\mathsf{MIP}*$ instance from a zero-knowledge IPCP, all that remains is to show that $\mathsf{NEXP} \subseteq \mathsf{PZK\text{-}IPCP}$. From there, we will be able to leverage [10, Lemma 9.1] to demonstrate $\mathsf{NEXP} \subseteq \mathsf{PZK\text{-}MIP}*$.

6.1 AQC of polynomial summation

We first need to define the *polynomial summation problem*. We will want a lower bound on the algebraic query complexity of this problem, similar to the examples we saw in Section 3.6.

Definition 6.1.1. The *polynomial summation problem* is the following:

> Let $\mathbb{F}$ be a field with $G \subseteq \mathbb{F}$. Let $m, k, d, d' \in \mathbb{N}$ and let $Z \in \mathbb{F}[X_{1,\dots,m}^{\leq d}, Y_{1,\dots,k}^{\leq d'}]$. What is the value of the polynomial
>
> $$R(X) = \sum_{\beta \in G^k} Z(X, \beta) \in \mathbb{F}[X_{1,\dots,m}^{\leq d}]?$$

We will need a lower bound on the algebraic query complexity of this problem (where Z functions as the oracle) later on in order to help demonstrate zero-knowledge. We will do this with one additional restriction—we will also need d' to be sufficiently large relative to G, but this will not hamper us in practice. In brief, the lower bound will tell us that so long as we limit our total queries, we will not receive *any* information about $R(X)$.

Lemma 6.1.2 ([10, Lemma 12.1]). *Let $\mathbb{F}$ be a field, $m, k, d, d' \in \mathbb{N}$, and G, K, L be finite subsets of $\mathbb{F}$ such that $K \subseteq L$, $d' \geq |G| - 2$, and $|K| = d + 1$. If $S \subseteq \mathbb{F}^{m+k}$ is such that there exist matrices $C \in M_{L^m,\ell}(\mathbb{F})$ and $D \in M_{S,\ell}(\mathbb{F})$ such that for all $Z \in \mathbb{F}[X_{1,\dots,m}^{\leq d}, Y_{1,\dots,k}^{\leq d'}]$ and all $i \in \{1, \dots, \ell\}$*

$$\sum_{\alpha \in L^m} C_{\alpha,i} \sum_{y \in G^k} Z(\alpha, y) = \sum_{q \in S} D_{q,i} Z(q), \tag{6.1}$$

then $|S| \geq \operatorname{rank}(BC)(\min(d' - |G| + 2, |G|))^k$, where $B \in M_{K^m,L^m}(\mathbb{F})$ is such that the column of B indexed by α represents $Z(\alpha)$ in the basis $\{Z(\beta) \mid \beta \in K^m\}$.

Proof. First, if $d' = |G| - 2$, then $d' - |G| + 2 = 0$; hence our bound simplifies to

$$\begin{aligned} |S| &\geq \operatorname{rank}(BC)\min(0, |G|)^k \\ &\geq 0\operatorname{rank}(BC) \\ &\geq 0, \end{aligned}$$

which is true regardless of S.

Otherwise, we can rewrite the left-hand side of Equation (6.1) as follows:

$$\sum_{\alpha\in L^m} C_{\alpha,i} \sum_{y\in G^k} Z(\alpha, y) = \sum_{\alpha\in L^m} C_{m,i} \sum_{\beta\in K^m} b_{\beta,\alpha} \sum_{y\in G^k} Z(\beta, y). \tag{6.2}$$

Then, define $B \in M_{K^m,L^m}(\mathbb{F})$ to be the matrix whose (i, j)-entry is $\beta_{i,j}$, and define $C' = BC \in M_{K^m,\ell}(\mathbb{F})$. From that, Equation (6.2) simplifies to

$$\sum_{\alpha\in L^m} C_{m,i} \sum_{\beta\in K^m} b_{\beta,\alpha} \sum_{y\in G^k} Z(\beta, y) = \sum_{\beta\in K^m} C'_{\beta,i} \sum_{y\in G^k} Z(\beta, y). \tag{6.3}$$

Next, define $H \subseteq G$ such that $|H| = \min(d' - |G| + 2, |G|)$. Further, let

$$P_0 = \{p \in \mathbb{F}[X^{\leq d}_{1,\ldots,m}, Y^{\leq|H|-1}_{1,\ldots,k}] \mid p(q) = 0 \text{ for all } q \in S\}.$$

We can write P_0 as the kernel of the linear function $F_S\colon \mathbb{F}[X^{\leq d}_{1,\ldots,m}, Y^{\leq|H|-1}_{1,\ldots,k}] \to \mathbb{F}^S$ defined by $(F_S(p))_s = p(s)$. By the rank-nullity theorem, this means

$$\dim(P_0) \geq (d + 1)^m|H|^m - |S|.$$

Let $A_0 \in M_{n,(K^m\times H^k)}(\mathbb{F})$ (for some arbitrary n) be a matrix whose rows form a basis for

$$\hat{P}_0 = \{(p(\alpha, y))_{\alpha\in K^m,y\in H^k} \mid p \in P_0\}.$$

The dimension of $\hat{P}_0$ is exactly the dimension of P_0; since two distinct polynomials of degree n can agree at no more than n points, polynomials in P have maximum degree $d^m(|H| - 1)^k$, and each vector in $\hat{P}_0$ has $|K^m \times H^k| = (d + 1)^m|H|^k$ points, it follows that the map from p to its corresponding vector in $\hat{P}_0$ is a bijection and thus dimension is preserved.

For any $y_0 \in H^k$, let $A_{y_0} \in M_{n,K^m}(\mathbb{F})$ be the submatrix of A_0 consisting only of rows where $y = y_0$. Since A_0's columns form a basis, it has full rank. Further, the set of all A_{y_0}s span the row space of A_0; hence

$$n = \operatorname{rank}(A_0) \leq \sum_{y_0\in H^k} \operatorname{rank}(A_{y_0}).$$

Hence, there exists a y_0 such that

$$\dim(A_{y_0}) \geq \frac{\operatorname{rank}(A_m)}{|H^k|} \geq \frac{(d + 1)^m|H^k| - |S|}{|H^k|} = (d + 1)^m - \frac{|S|}{|H^k|}.$$

Let $q \in \mathbb{F}[Y_{1,\dots,k}^{\geq |G|-1}]$ be the polynomial such that $q(y_0) = 1$ and $q(y) = 0$ for all $y \in G^k \setminus \{y_0\}$ (as per Theorem 2.3.7). Then, for all $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, k\}$ it holds that

$$(A_{y_0}C')_{ij} = \sum_{\beta \in K^m} C'_{\beta,j} p_i(\beta, y_0) \tag{6.4}$$

where $p_i(\alpha, \beta)$ is the element of A_0 in row i and column (α, β). This comes from the definition of matrix multiplication and A_{y_0}. Next, the definition of q gives us

$$\sum_{\beta \in K^m} C'_{\beta,j} p_i(\beta, y_0) = \sum_{\beta \in K^m} C'_{\beta,j} \sum_{y \in G^k} q(y) p_i(\beta, y). \tag{6.5}$$

Now, since p_i is a row of A_0 and the rows of A_0 form a basis for the space of all outputs of polynomials in P_0, we can simply treat p_i as a polynomial in P_0.

Note that $qp_i \in \mathbb{F}[X_{1,\dots,k}^{\geq d}, Y_{1,\dots,k}^{\geq d'}]$ (from the definitions of q and p_i). Hence, Equation (6.3) tells us that

$$\sum_{\beta \in K^m} C'_{\beta,j} \sum_{y \in G^k} q(y) p_i(\beta, y) = \sum_{s \in S} D_{s,i} (qp_i)(s). \tag{6.6}$$

Since $s \in Q$, the definition of P_0 tells us that $p_i(s) = 0$; hence the entire sum in Equation (6.6) is equal to zero and thus $A_{y_0}C' = 0$.

Lastly, we can apply Sylvester's rank inequality:

$$\begin{aligned} \operatorname{rank}(A_{y_0}) + \operatorname{rank}(C') - (d+1)^m &\leq \operatorname{rank}(0) \\ (d+1)^m - \operatorname{rank}(C') &\geq \operatorname{rank}(A_{y_0}) \\ (d+1)^m - \operatorname{rank}(C') &\geq (d+1)^m - |S|/|H|^k \\ |S| &\geq \operatorname{rank}(C')|H|^k. \end{aligned}$$

From the definition of H, this means

$$|S| \geq \operatorname{rank}(C')(\min(d' - |G| + 2, |G|))^k,$$

as desired. □

Corollary 6.1.3 ([10, Corollary 12.2])**.** *Let $\mathbb{F}$ be a finite field, $G \subseteq \mathbb{F}$, and $d, d' \in \mathbb{N}$ with $d' \geq 2(|G|-1)$. If $S \subseteq \mathbb{F}^{m+k}$ is such that there exist $(c_\alpha)_{\alpha \in \mathbb{F}^m}$ and $(d_\beta)_{\beta \in \mathbb{F}^{m+k}}$ such that*

1. *for all $Z \in \mathbb{F}[X_{1,\dots,m}^{\leq d}, Y_{1,\dots,k}^{\leq d}]$ it holds that*

$$\sum_{\alpha \in \mathbb{F}^m} c_\alpha \sum_{y \in G^k} Z(\alpha, y) = \sum_{q \in S} d_q Z(q), \tag{6.7}$$

and

2. *there exists $Z' \in \mathbb{F}[X_{1,\dots,m}^{\leq d}, Y_{1,\dots,k}^{\leq d}]$ such that*

$$\sum_{\alpha \in \mathbb{F}^m} c_\alpha \sum_{y \in G^k} Z'(\alpha, y) = 0, \tag{6.8}$$

then $|S| \geq |G|^k$.

From here, we get that the algebraic query complexity of polynomial summation is at least $|G|^k$. That is, if we query Z no more than $|G|^k$ times, then we we will receive *no information* about the polynomial $R(X) = \sum_{\beta \in G^k} Z(X, \beta)$.

Corollary 6.1.4 ([10, Corollary 12.3])**.** *Let $\mathbb{F}$ be a finite field, $G \subseteq \mathbb{F}$, and $d, d' \in \mathbb{N}$ with $d' \geq 2(|G|-1)$. Let Q be a subset of $\mathbb{F}^{m+k}$ with $|Q| \leq |G|^k$, and let Z be uniformly random in $\mathbb{F}[X_{1,\ldots,m}^{\leq d}, Y_{1,\ldots,k}^{\leq d'}]$. Then, the random variables $(\sum_{y \in G^k} Z(\alpha, y))_{\alpha \in \mathbb{F}^m}$ and $(Z(q))_{q \in Q}$ are independent.*

Proof. We will leverage Theorem 2.4.9 that we proved earlier. Now, consider the vector space

$$V = \left\{ \left((Z(\gamma))_{\gamma \in \mathbb{F}^{m+k}}, \left(\sum_{y \in G^k} Z(\alpha, y) \right)_{\alpha \in \mathbb{F}^m} \right) \middle| \, Z \in \mathbb{F}[X_{1,\ldots,m}^{\leq d}, Y_{1,\ldots,k}^{\leq d'}] \right\} \tag{6.9}$$

This is a vector space over $\mathbb{F}$ with basis $\{e_i \mid i \in \mathbb{F}^{m+k} \sqcup \mathbb{F}^m\}$.

Consider the subsets indexed by $\mathbb{F}^m$ and $Q \subseteq \mathbb{F}^{m+k}$. As per Theorem 2.4.9, we want to show there does not exist a $c \in V|_{\mathbb{F}^m}$ and $d \in V|_Q$ such that for all $w \in V$, $c \cdot w|_{\mathbb{F}^m} = d \cdot w|_Q$ and for some $w \in V$, $c \cdot w|_{\mathbb{F}^m} = 0$. Expanding these equations out given our definition of V, we get exactly Equations (6.7) and (6.8). However, Corollary 6.1.3 tells us that this can only be true if $|Q| > |G|^k$. But we assumed $|Q| \leq |G|^k$, so there cannot exist any such c or d. Hence, these two random variables are statistically independent. □

6.2 The sumcheck problem

Central to our upcoming work will be a nice answer to the *sumcheck problem*, a computational problem about verifying whether or not the sum of a polynomial over some subset of a field is equal to a provided value. This will prove useful to us in arithmetizing our problems: if we can turn our boolean formulae into a question in the sumcheck format, then we can delegate to the protocol to complete our proof.

Definition 6.2.1 ([26])**.** The *sumcheck problem* is the following problem:

> Let H be a subset of a finite field $\mathbb{F}$, let $F \in \mathbb{F}[X_{1,\ldots,m}^{\leq d}]$ a polynomial over $\mathbb{F}$, and let $a \in \mathbb{F}$. Does $\sum_{x \in H^m} F(x) = a$?

For this question, we give H and a to both the prover and verifier, but only the prover gets access to F as an algebraic oracle.

```
Input: A polynomial $F \in \mathbb{F}[X_{1,\ldots,n}^{\leq d}]$, subset $H \subseteq \mathbb{F}$, and number $a$
Output: Whether $\sum_{x \in H^m} F(x) = a$
1  $P$: send the polynomial $F_1(x_1) = \sum_{X \in \{0,1\}^{n-1}} F(x_1, X)$;
2  $V$: check if $a = \sum_{x \in H} F_1(x)$;
3  $V$: choose random $r_1 \in \mathbb{F}$ and send to $P$;
4  for $i$ from 2 to $n$ do
5      $P$: send the polynomial
           $F_i(x_i) = \sum_{X \in H^{m-i}} F(r_1, \ldots, r_{i-1}, x_i, X)$
       to $V$;
6      $V$: check $F_{i-1}(r_{i-1}) = \sum_{x \in H} F_i(x)$;
7      if $i \neq n$ then
8          $V$: choose random $r_i \in \mathbb{F}$ and send to $P$;
9      end
10 end
11 $V$: choose random $r_n \in \mathbb{F}$;
12 $V$: check $F_n(r_n) = F(r_1, \ldots, r_n)$;
```

Algorithm 6.1: The standard sumcheck protocol [26, Thm. 1]

6.2.1 A non-zero-knowledge sumcheck protocol

We begin with an interactive protocol for sumcheck that does not have any zero-knowledge characteristics. While this protocol itself does not preserve anything, it will form an essential building block for the zero-knowledge protocols we will construct later.

Theorem 6.2.2. *The sumcheck problem is in* IP.

Proof. We show this by implementing an interactive protocol for sumcheck in Algorithm 6.1. We need to start by showing that this algorithm will correctly answer the sumcheck question. First, if $\sum_{x \in H^m} F(x) = a$ and P is honest, then Vs check in line 2 will succeed if and only if the question is correct. Again assuming an honest P, from the definition of each F_i,

$$F_{i-1}(r_{i-1}) = \sum_{X \in H^{m-i+1}} F(r_1, \ldots, r_{i-2}, r_{i-1}, X)$$

and

$$\sum_{x \in H} \sum_{x \in H^{m-i}} F(r_1, \ldots, r_{i-1}, x, X) = \sum_{X \in H^{m-i+1}} F(r_1, \ldots, r_{i-1}, X).$$

Hence, line 6 will always succeed with an honest P. Finally, if P has been honest in the last iteration, line 12 will always succeed since the sum in the equation in line 5 is now over a single object.

If P is dishonest, then we show soundness by induction on n. If $n = 1$ then there is only one message sent. Two distinct n-variable polynomials of degree d can be equal at most d^n points, as such in the $n = 1$ case the probability of incorrectly passing the check in line 12 is at most $d/|\mathbb{F}|$.

Next, assume the $(n-1)$-variable case has soundness error at most $(n-1)d/|\mathbb{F}|$. Let

$$G_1(x_1) = \sum_{X \in H^{n-1}} F(x_1, X),$$

i.e. the "correct" value of F_1 (were P not to lie). If $F_1 \neq G_1$, then as we saw before, $F_1(r_1) \neq G_1(r_1)$ with probability $1 - d/|\mathbb{F}|$. If this is the case, then in the rest of the loop, the system is attempting to prove the claim

$$F_1(r_1) = \sum_{X \in H^{n-1}} F(r_1, X),$$

which we know to be false. $F(r_1, \cdot)$ is an $(n-1)$-variable polynomial of multidegree d; by induction this has soundness error $(r-1)d/|\mathbb{F}|$. Thus, V will reject with probability

$$1 - \mathbb{P}[F_1(r_1) \neq G_1(r_1)] - \mathbb{P}[V \text{ rejects in round } j > 1 \mid F_1(r_1) \neq G_1(r_1)]$$

which, by substituting in our definitions, gives us probability at least

$$1 - \frac{d}{|\mathbb{F}|} - \frac{d(n-1)}{\mathbb{F}} = 1 - \frac{dn}{|\mathbb{F}|}.$$

The only other step is to show that V runs in polynomial time. Since each loop iteration only requires checking a sum over H, we only need to compute the various F_is (which are themselves only polynomially larger than the original F) a total of nH times; computing F_i is also in polynomial time. Thus, V overall runs in polynomial time. □

6.2.2 Making the sumcheck protocol zero-knowledge

Theorem 6.2.3 ([10, Theorem 13.3]). *There exists a zero-knowledge variant of Algorithm 6.1.*

To avoid an overly long-winded proof, we will split the above theorem into two lemmata. The first will show that Algorithm 6.2 is correct, and the second will show that it is in fact zero-knowledge.

Lemma 6.2.4. *Algorithm 6.2 is a MIP* algorithm for sumcheck.*

Proof. If P is honest in Algorithm 6.2, then both sumcheck protocols will pass if and

```
Input: An instance $(H, a)$ to both $P$ and $V$
Input: A polynomial $F \in \mathbb{F}[X_{1,\dots,m}^{\leq d}]$ as an oracle to $P$
Output: Whether $\sum_{x \in H^m} F(x) = a$
1  $P$: draw random $Z \in \mathbb{F}[X_{1,\dots,m}^{\leq d}, Y_{1,\dots,m}^{\leq 2\lambda}]$;
2  $P$: draw random $A \in \mathbb{F}[Y_{1,\dots,k}^{\leq 2\lambda}]$;
3  $P$: send the polynomial
       $$O(W, X, Y) = W \cdot Z(X, Y) + (1 - W) \cdot A(Y)$$
    to $V$;
   // Note that $Z(x) = O(1, x)$ and $A(x) = O(0, 0, x)$, so $V$ can use both
      $Z$ and $A$ later
4  $P$: send $z = \sum_{\alpha \in H^m} \sum_{\beta \in G^k} Z(\alpha, \beta)$ to $V$;
5  $V$: draw random $\rho_1 \in \mathbb{F}^\times$;
6  $V$: send $\rho_1$ to $P$;
7  Run the standard sumcheck IP (Algorithm 6.1) on the statement
    $\sum_{\alpha \in H^m} Q(\alpha) = \rho_1 a + z$, where
       $$Q(X_1, \dots, X_m) = \rho_1 F(X_1, \dots, X_m) + \sum_{\beta \in G^k} Z(X_1, \dots, X_m, \beta).$$
    We have $P$ play the prover and $V$ the verifier, with the following modification:
    For $i = 1, \dots, m$, in the $i$th round, $V$ samples its random element $r_i$ from the
    set $I$ instead of from all of $\mathbb{F}$; if $P$ ever receives $r_i \in \mathbb{F} \setminus I$, it immediately
    aborts. In particular, in the $m$th (i.e., the final) round, $P$ sends a polynomial
       $$g_m(X_m) = \rho_1 F(c_1, \dots, c_{m-1}, X_m) + \sum_{\beta \in G^k} Z(c_1, \dots, c_{m-1}, X_m, \beta)$$
    for some $c_1, \dots, c_{m-1} \in I$;
8  $V$: send $c_m \in I$ to $P$;
9  $P$: send $w = \sum_{\beta \in G^k} Z(c, \beta)$ to $V$, where $c = (c_1, \dots, c_m)$;
10 $z' \leftarrow \sum_{\alpha \in H^m} A(\alpha)$;
11 $P$: send $z'$ to $V$;
12 $V$: draw random $\rho_2 \in \mathbb{F}$;
13 $V$: send $\rho_2$ to $P$;
14 $Q'(x) \leftarrow \rho_2 Z(c, x) + A(x)$;
15 Both: run Algorithm 6.1 on the statement $\sum_{\alpha \in H^m} Q'(\alpha) = \rho_2 w + z'$;
16 $V$: output the claim $F(c) = \frac{g_m(c_m) - w}{\rho_1}$;
```

Algorithm 6.2: Strong zero-knowledge sumcheck [10, Construction 3]

only if (F, H, a) is valid. From the various definitions in the program, we have

$$\begin{aligned}
\sum_{\alpha \in H^m} Q(a) &= \rho_1 a + z \\
\sum_{\alpha \in H^m} \left(\rho_1 F(\alpha) + \sum_{\beta \in G^k} Z(\alpha, \beta) \right) &= \rho_1 a + \sum_{\alpha \in H^m} \sum_{\beta \in G^k} Z(\alpha, \beta) \\
\sum_{\alpha \in H^m} \rho_1 F(a) + \sum_{a \in H^m} \sum_{\beta \in G^k} Z(\alpha, \beta) &= \rho_1 a + \sum_{\alpha \in H^m} \sum_{\beta \in G^k} Z(\alpha, \beta) \\
\rho_1 \sum_{a \in H^m} F(a) &= \rho_1 a \\
\sum_{a \in H^m} F(a) &= a.
\end{aligned}$$

Since $\rho_1 \neq 0$, all of these transformations are biconditionally true; hence the sumcheck protocol in line 7 will pass if and only if (F, H, a) is valid. The modification does not affect this correctness since all it does is limit the set of elements we can randomly sample from—since we know this works for all $r_i \in \mathbb{F}$, it will also be true for all $r_i \in I$.

For the second sumcheck (in line 15), we have

$$\begin{aligned}
\sum_{\alpha \in H^m} Q'(\alpha) &= \rho w + z \\
\sum_{\alpha \in H^m} (\rho Z(c, \alpha) + A(\alpha)) &= \rho w + \sum_{\alpha \in H^m} A(\alpha) \\
\sum_{\alpha \in H^m} \rho F(\alpha) + \sum_{\alpha \in H^m} A(\alpha) &= \rho w + \sum_{\alpha \in H^m} A(\alpha) \\
\rho \sum_{\alpha \in H^m} F(\alpha) &= \rho w \\
\sum_{\alpha \in H^m} F(\alpha) &= w.
\end{aligned}$$

As before, these transformations are all biconditionally true; hence an honest P will always cause the sumcheck in line 15 to succeed if and only if (F, H, a) is valid.

If P is dishonest, things get trickier. First, note that we have described the relevant portions of the path through Algorithm 6.2 as a decision tree in Figure 6.1.

Fix some $F \in \mathbb{F}[X_{1,\ldots,m}^{\leq d}]$ such that $\sum_{\alpha \in H^m} F(\alpha) \neq a$, and fix some $Z \in \mathbb{F}[X_{1,\ldots,m+k}^{\leq d}]$. Define

$$\begin{aligned}
\hat{a} &= \sum_{\alpha \in H^m} F(\alpha) \\
\hat{z} &= \sum_{\alpha \in H^m} \sum_{\beta \in G^k} Z(\alpha, \beta).
\end{aligned}$$

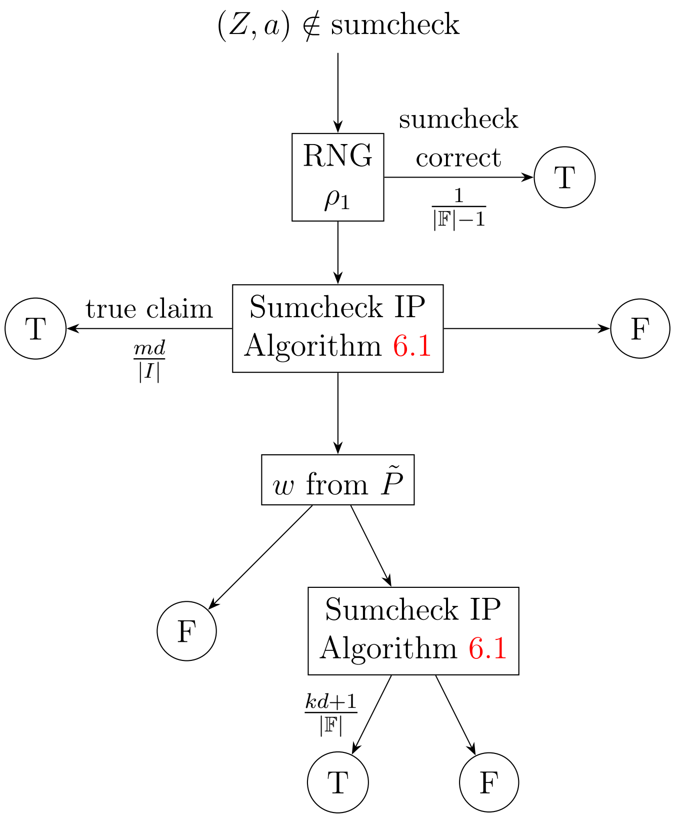

Figure 6.1: The decision tree for Algorithm 6.2 given a false input

We know $\hat{a} \neq a$, and we do not know whether $\hat{z} = z$ (it depends on whether or not P was honest in line 4). The sumcheck question we ask in line 7 is true if and only if $\rho_1 \hat{a} + \hat{z} = \rho_1 a + z$. Through some algebra, this simplifies to being true if and only if

$$\rho_1 = \frac{z - \hat{z}}{\hat{a} - a}. \tag{6.10}$$

We know ρ_1 is a random element from $\mathbb{F}^\times$; hence if $z \neq \hat{z}$,

$$\mathop{\mathbb{P}}_{\rho_1 \in \mathbb{F}^\times} [\rho_1 \hat{a} + \hat{z} = \rho_1 a + z] = \frac{1}{|\mathbb{F}| - 1} \tag{6.11}$$

$$\mathop{\mathbb{P}}_{\rho_1 \in \mathbb{F}^\times} \left[\sum_{\alpha \in H^m} Q(\alpha) = \rho_1 a + z \right] = \frac{1}{|\mathbb{F}| - 1} \tag{6.12}$$

and if $z = \hat{z}$ the above probabilities are 0.

Hence, by the soundness guarantee of Algorithm 6.1, if the presented sumcheck equation is incorrect then the subroutine will output a correct equation with probability at most $\frac{md}{|I|}$.

Next, we split into two cases: whether or not P sends $w = \sum_{\beta \in G^k} Z(c, \beta)$. In the case where P sends $w \neq \sum_{\beta \in G^k} Z(c, \beta)$, then as in the earlier sumcheck, we have that for a fixed A and Z, $\sum_{\alpha \in H^m} Q'(\alpha) = \rho_2 w + z'$ with probability at most $1/|\mathbb{F}|$ (In this case we *do* allow ρ_2 to be 0, since we will not be dividing by it in any later step). As we showed in Theorem 6.2.2, in the case when the sumcheck statement is false Algorithm 6.1 will incorrectly accept with probability $kd/|\mathbb{F}|$. Hence, line 15 will not reject with probability $\frac{kd+1}{|\mathbb{F}|}$.

In the case where P sends $w = \sum_{\beta \in G^k} Z(c, \beta)$, then it must be that $F(c) \neq \frac{g_m(c_m)-w}{\rho_1}$, since if the verifier did not reject then we have

$$\begin{aligned} \rho_1 F(c) + \sum_{\beta \in G^k} Z(c,\beta) &\neq g_m(c_m) \\ \rho_1 F(c) + w &\neq g_m(c_m) \\ F(c) &\neq \frac{g_m(c_m) - w}{\rho_1} \end{aligned}$$

regardless of ρ_1.

Since P is assumed to be malicious, we have to assume (in order to acquire an upper bound on when the input is incorrectly accepted) that it will therefore always send $w \neq \sum_{\beta \in G^k} Z(c, \beta)$, since that has a lower probability of being rejected.

In total, the probability that we accept given incorrect input is, based on the decision tree from Figure 6.1, bounded above by

$$\frac{1}{|\mathbb{F}|-1} + \left(1 - \frac{1}{|\mathbb{F}|-1}\right)\left(\frac{md}{|I|} + \left(1 - \frac{md}{|I|}\right)\left(\frac{kd+1}{|\mathbb{F}|}\right)\right). \tag{6.13}$$

Since we are looking for an upper bound, we can simplify both $(1 - \frac{1}{|\mathbb{F}|-1})$ and $(1 - \frac{md}{|I|})$ to 1, giving us that the probability is bounded above by

$$\frac{1}{|\mathbb{F}|-1} + \frac{md}{|I|} + \frac{kd+1}{|\mathbb{F}|}. \tag{6.14}$$

We can further increase the upper bound by decreasing the denominator of the third term by 1, allowing us to simplify to

$$\frac{md}{|I|} + \frac{kd+2}{|\mathbb{F}|}, \tag{6.15}$$

which is the claimed soundness bound. □

Lemma 6.2.5. *Algorithm 6.2 is zero-knowledge save for a single query to F.*

Proof. We construct a simulator for Algorithm 6.2 in two phases: first we construct a simulator that is slow but correct as Algorithm 6.3. Following that, we explain how to convert that simulator into an efficient version.

We will be showing that at each step, every random variable received by the (slow) simulator is identical to that of the prover in Algorithm 6.2. If every random variable is identical, then since Algorithm 6.3 is constructed similarly to the prover it will output the same result.

```
1  Pick random $Z_s \in \mathbb{F}[X^{\leq d}_{1,\dots,m}, Y^{\leq 2\lambda}_{1,\dots,k}]$;
2  Run the weak-ZK sumcheck simulator;
3  Begin simulating $\tilde{V}$, answering its oracle queries with $Z_s$ and the simulated $A$;
4  Send $z_s = \sum_{\alpha\in H^m}\sum_{\beta\in G^k} Z_s(\alpha,\beta)$;
5  Receive $\tilde{\rho}$ from the simulated $\tilde{V}$;
6  Draw $Q_s \in \mathbb{F}[X^{\leq d}_{1,\dots,m}]$ such that $\sum_{\alpha\in H^m} Q_s(\alpha) = \tilde{\rho}\alpha + z_s$;
7  Engage in the sumcheck protocol (Algorithm 6.1) on the claim
     $\sum_{\alpha\in H^m} Q_s(\alpha) = \tilde{\rho}\alpha + z_s$;
8  if $\tilde{V}$ sends $c_i \notin I$ as a challenge in the above protocol then
9      return $\perp$;
10 end
11 Let $c \in I^m$ be the point chosen by $\tilde{V}$;
12 Query $F(c)$;
13 $w_s \leftarrow Q_s(c) - \tilde{\rho}F(c)$;
14 Send $w_s$;
15 Draw $Z'_s \in \mathbb{F}[X^{\leq d}_{1,\dots,m}, Y^{\leq 2\lambda}_{1,\dots,k}]$ such that $\sum_{\beta\in G^k} Z'_s(c,\beta) = w_s$ and $Z'_s(\gamma) = Z_s(\gamma)$
     for all previous queries $\gamma$;
16 Use $S'$ to simulate the sumcheck protocol for the claim $\sum_{\beta\in G^k} Z'_s(c,\beta) = w_s$;
17 return the view of the simulated $\tilde{V}'$;
```

Algorithm 6.3: An inefficient simulator for Algorithm 6.2 [10, p. 15:33]

In the simulator, the random choice of Z_s in line 1, which is identical to the choice of Z in line 1 in the original algorithm. Similarly, our result z_s is identical to the z in the original protocol.

Next, note that Q_s is distributed identically to Q from the original statement, since the components defining Q are uniformly random. In particular, note that $\tilde{\rho}$ is a function constructed with fewer than λ^k queries to Z. Hence, due to Corollary 6.1.4, this means $\tilde{\rho}$ is independent of R so long as $\sum_{\alpha\in H^m} R(\alpha) = z$. Hence, so long as that formula holds, $\tilde{\rho}F$ is independent of R and thus $R + \tilde{\rho}F$ is a uniformly random polynomial subject to the condition that $\sum_{\alpha\in H^m} Q(\alpha) = \tilde{\rho}a + z$.

Next, we send $Q_s(c) - \tilde{\rho}F(c)$ to the verifier in our simulator. The original prover sends $Q(c) - \tilde{\rho}F(c)$. Since Q_s and Q are identically distributed, so too are these values.

After that, our simulator draws a new Z'_s that is random, but agrees with the already-seen Z_s everywhere we have already checked. If we define U to be the set of question-response pairs already seen, then Corollary 6.1.4 gives us the following:

$$\mathop{\mathbb{P}}_{Z'_s}\left[Z'_s(q) = a \;\middle|\; \begin{array}{c} Z'_s(\gamma) = b \quad \forall(\gamma,b)\in U \\ \sum_{\beta\in G^k} Z'_s(c,\beta) = w_s \end{array}\right] = \mathop{\mathbb{P}}_{Z}\left[Z(q) = a \;\middle|\; \begin{array}{c} Z(\gamma) = b \quad \forall(\gamma,b)\in U \\ \sum_{\beta\in G^k} Z'_s(X,\beta) = Q(X) - \tilde{\rho}F(X) \end{array}\right] \quad (6.16)$$

for any $q \in \mathbb{F}^{m+k}$ and $a \in \mathbb{F}$.

The left-hand side of this equation is the exactly the distribution of the answer to a query q made by S' to Z'_s. Similarly, the right-hand side describes the answer to a query to Z under the same constraints. Since these probabilities are identical, it follows that the results of the queries must too be identically distributed.

Lastly, we can transform Algorithm 6.3 into a polynomial-time algorithm through using a more efficient encoding system described in [9]; this allows us to maintain some state between simulations and thus improve the efficiency to polynomial-time. □

6.3 Extending the sumcheck algorithm to NEXP

Armed with our sumcheck algorithm, we are ready to take on the fight of the rest of NEXP. Like so many other theorems, we will do this with a NEXP-complete problem, and as before our problem of choice is O3SAT.

Theorem 6.3.1 ([10, Thm. 14.2]). *There exists a $c \in \mathbb{N}$ such that for any query-bound function $b(n)$, $d(n) \in \Omega(n^c)$, $m(n) \in O(n^c \log(b))$, and any sequence of fields $\mathbb{F}(n)$ that are field extensions of $\mathbb{F}_2$ with $|\mathbb{F}(n)| \in \Omega((n^c \log(b))^4)$,*

$$\mathsf{O3SAT} \in \mathsf{IPCP}\left[\begin{array}{rl} \textit{round complexity:} & O(n,b) \\ \textit{PCP length:} & \mathsf{poly}(2^n, b) \\ \textit{comm. complexity:} & \mathsf{poly}(n, \log(b)) \\ \textit{query complexity:} & \mathsf{poly}(n, \log(b)) \\ \textit{oracle:} & \mathbb{F}[X^{\le d}_{1,\dots,m}] \\ \textit{soundness error:} & 1/2 \end{array}\right],$$

which is zero-knowledge with query bound b.

Proof. We present an algorithm for O3SAT, which we have laid out in Algorithm 6.4. We will show this is zero-knowledge by implementing a simulator in Algorithm 6.5. For time reasons, the proof of correctness for this cannot be reproduced here, but it is in [10, Theorem 14.2]. □

Corollary 6.3.2. NEXP $\subseteq$ PZK-IPCP.

Proof. Since O3SAT is NEXP-complete as per Corollary 2.2.31, we can perform a polynomial reduction from any other language to O3SAT and then run Algorithm 6.4. □

6.4 Zero-knowledge MIP∗ for NEXP

Theorem 6.4.1 ([10, Lemma 9.1]). *Let L be a language, let $m, d, q \in \mathbb{N}$, and let $\mathbb{F}$ be a finite field of size* $\mathsf{poly}(m, d, q)$ *sufficiently large. Then, there exists a transformation*

$$T : \mathsf{IPCP}\left[\begin{array}{rl} \textit{round complexity:} & r \\ \textit{PCP length:} & \ell \\ \textit{comm. complexity:} & c \\ \textit{query complexity:} & q \\ \textit{oracle:} & \mathbb{F}[X^{\le d}_{1,\dots,m}] \\ \textit{soundness error:} & \varepsilon \end{array}\right] \to \mathsf{MIP}* \left[\begin{array}{rl} \textit{number of provers:} & 2 \\ \textit{round complexity:} & r+1 \\ \textit{comm. complexity:} & c \\ \textit{soundness error:} & 1 - \frac{1}{\mathsf{poly}(m,d)} \end{array}\right].$$

```
repeat
    $P$: Draw a random $Z \in \mathbb{F}[X_{1,\ldots,m}^{\leq |H|+2}, Y_{1,\ldots,m}^{\leq 2|H|}]$;
until $\sum_{\beta \in G^k} Z(\alpha, \beta) = A(\gamma_2(\alpha))$ for all $\alpha \in H^{m_2}$;
$P$: Generate oracle $\pi_0$ for Algorithm 6.2 on input $(\mathbb{F}, m_1 + 3m_2, d, H, 0)$;
$P$: Generate oracles $\pi_1, \pi_2, \pi_3$ for Algorithm 6.2 on input $(\mathbb{F}, k, 2|H|, |H|, \cdot)$;
$P$: Send $(\pi_0, \pi_1, \pi_2, \pi_3)$;
$V$: Choose $x, y \in \mathbb{F}^{r+3s}$ at random;
$V$: Send $x$ and $y$ to $P$;
Both: Simulate Algorithm 6.2 over the claim $F(x, y) = 0$ with $I = \mathbb{F} \setminus H$ and
  oracle $\pi_1$;
for $i \in \{1, 2, 3\}$ do
    $P$: Send $h_i = A(\gamma_2(c_i'))$ to $V$
end
$V$: Substitute the $h_i$ into the evaluation of $f$;
if the claims do not hold then
    reject;
end
for $i \in \{1, 2, 3\}$ do
    $P$ and $V$ implement Algorithm 6.1 on the claim $\sum_{\beta \in H^k} Z(c_i', \beta) = h_i$;
end
```

Algorithm 6.4: A low-degree IPCP for O3SAT [10, p. 15:36]

```
Draw a random polynomial $Z_s \in \mathbb{F}[X_{1,\ldots,m_2}^{\leq |H|+2}, Y_{1,\ldots,k}^{\leq 2|H|}]$;
Let $S_0$ be a simulated copy of Algorithm 6.2 on input
  $(\mathbb{F}, m_1 + 3m_2, \deg(f), H, 0)$;
for $i \in \{1, 2, 3\}$ do
    Let $S_i$ be a simulated copy of Algorithm 6.2 on input $(\mathbb{F}, k, 2|H|, |H|, \cdot)$;
end
Simulate $\tilde{V}$ to receive $x, y \in \mathbb{F}^{r+3s}$;
Simulate Algorithm 6.2 on the claim $F(x, y) = 0$. To answer the single query
  it makes at $c \in (\mathbb{F} \setminus H)^{r+3s}$, reply with $f(x, y, c)$ where $c = (c_0, c_1, c_2, c_3)$. To
  compute the values $A(\gamma(c_i))$, substitute with an $h_s^i \in \mathbb{F}$ drawn at random;
for $i \in \{1, 2, 3\}$ do
    Simulate Algorithm 6.2 with the claim $\sum_{\beta \in H^k} Z(\alpha, \beta) = h_s^i$, answering
      queries with $Z_s$;
end
```

Algorithm 6.5: A simulator for Algorithm 6.4 [10, p. 15.37]

such that (P', V') and $T(P', V')$ recognize the same language.

Further, if the IPCP (P', V') is zero-knowledge with query bound $b \geq 2(q+1)md+3$, then the $\mathsf{MIP}*$ *(P_1, P_2, V) is zero-knowledge.*

The above theorem demonstrates that we can take any IPCP and lift it into a MIP* while preserving zero-knowledge. Since Algorithm 6.4 is a zero-knowledge MIP* algorithm for $\mathsf{O3SAT}$ and $\mathsf{O3SAT}$ is NEXP-complete, it follows that $\mathsf{NEXP} \subseteq \mathsf{PZK\text{-}MIP}$.

Chapter 7

A zero-knowledge PCP theorem

In Chapter 5, we discussed and proved the PCP theorem. Given we have seen how widespread zero-knowledge PCPs can be, the question arises of whether or not the PCP theorem can be recreated entirely with zero-knowledge PCPs. Not only can it be recreated, but we can also show a PCP-theorem equivalent for NEXP.

Our proof will proceed in broadly the same manner as our original proof of the PCP theorem, and it will reuse broad portions of the same machinery. In order to reuse this, though, we will need a new notion of closeness for our proofs. This notion of closeness is important because we can show that it preserves zero-knowledge: that is, a machine that is close to a zero-knowledge machine must itself be zero-knowledge. This will allow us to more easily prove that we are constructing zero-knowledge proofs, without going through the hassle of constructing a simulator every time (and proving that it is in fact a simulator).

7.1 Locally-computable proofs

Locally-computable proofs are a notion of closeness for Turing machines. Abstractly, a Turing machine is ℓ-locally computable from another if it can be simulated in polynomial time using no more than ℓ queries about the behavior of the source machine.

Definition 7.1.1 ([17, Def. 3.1]). Let A and A_0 be randomized Turing machines, and let $\ell\colon \mathbb{N} \to \mathbb{N}$. Then A is *ℓ-locally computable* from A_0 on a subset $C \subseteq \{0,1\}^*$ if there exists an oracle Turing machine f that runs in polynomial time and makes no more than $\ell(n)$ queries to its oracle such that for every $x \in C$, the distribution of $A(x)$ is identical to the distribution of f with oracle $\pi_0 = A_0(x)$.

While this is not strictly a metric in the formal, metric-space definition (in particular, there exist distinct machines that are 0-locally computable from each other[1]), it still obeys the triangle inequality and any machine is 0-locally computable from itself; hence we can think of it as being metric-*like*.

[1]Since our simulators are allowed to be polynomial-time, any machine in P is 0-locally computable from any other machine, as an example.

For local-computation to be useful to us, we will need to rethink how we have been perceiving PCPs up to this point. So far, we have just thought of a PCP proof as being a static oracle that gets queried by our verifier. However, in the real world oracles do not spring in the world fully formed, they must be generated. Hence, here we will think of the proof as being generated by another Turing machine.

For us, the most important piece of locally-computable proofs is that they preserve zero knowledge. This will give us an easier way to prove that a given algorithm is zero-knowledge, since we can simply show it is locally-computable from another algorithm we have already shown. We will find this particularly useful when we are dealing with *transformations* of zero-knowledge PCPs; if we are starting from a zero-knowledge base, the idea of local computation will allow us to easily show that our transformed PCP is also zero-knowledge.

Theorem 7.1.2 ([17, Lemma 3.2]). *Let (P_0, V_0) be a PZK-PCP for some language L with query bound q^*, and let (P, V) be a PCP for a language M such that P is ℓ-locally computable from P_0 on M. Then (P, V) is perfect zero-knowledge with query bound q^*/ℓ.*

```
Input: A string $x$ and random coins $r$
Oracle: A function $\pi_0$
Output: The interaction transcript of $(P, V^*)$ on input $x$
1 $T \leftarrow []$;
2 Run $V^*$ on random coins $r$;
3 for each query $\alpha$ that $V^*$ makes do
4     $\beta \leftarrow f^{\pi_0}(\alpha)$;
5     Push $(\alpha, \beta)$ onto $T$;
6 end
7 return $(r, T)$;
```
Algorithm 7.1: A hybrid simulator for a locally-computable PCP [17, Construction 3.3]

```
Input: A string $x$
Output: The interaction transcript of $(P, V^*)$ on input $x$
1 Run $\overline{\text{Sim}}_{A_{V^*}}(x)$ to obtain $T_0$ with random coins $r$;
2 Run $A_{V^*}(x, r)$ (Algorithm 7.1) using $T_0$ to answer its questions;
3 return $(r, T)$;
```
Algorithm 7.2: A PZK simulator for a locally-computable PCP [17, Construction 3.4]

Proof. We construct a simulator for (P, V) in Algorithm 7.2.

Let V^* be a malicious verifier for (P, V), and let $\pi \leftarrow P$ and $\pi_0 \leftarrow P_0$ be random variables. Since P is ℓ-locally computable with the function f, by definition we have that π is identically-distributed to $(f^{\pi_0}(\alpha))_{\alpha\in\mathrm{dom}(\pi_0)}$. Since all Algorithm 7.1 does is compute $f^{\pi_0}(\alpha)$ for each query α, it will reproduce the same transcript as (P, V^*) would.

Next, since (P_0, V_0) is a PZK-PCP, and Algorithm 7.1 is a verifier for π_0, by definition there exists a simulator $\overline{\mathrm{Sim}}_{A_{V^*}}$ whose output is identically-distributed to $\mathrm{View}_{A_{V^*},P_0}$, so long as A_{V^*} makes no more than q^* queries to π_0. Since P is ℓ-locally computable from P_0, it follows that Algorithm 7.1 makes no more than ℓ queries to π_0. Hence, Algorithm 7.2 makes no more than q^* queries to π_0 so long as V^* makes no more than q^*/ℓ queries to π. Thus, the output of Algorithm 7.2 is identical to the view of Algorithm 7.1. □

Corollary 7.1.3. *Let (P_0, V_0) be a PZK-PCPP for some language L with query bound q^* and proximity parameter δ, and let (P, V) be a PCPP for a language M with proximity parameter δ such that P is ℓ-locally computable from V on M. Then (P, V) is perfect zero-knowledge with query bound q^*/ℓ.*

Proof. By replacing every instance of PZK-PCP in that proof with PZK-PCPP, every piece of the proof still holds, since zero-knowledge is identical for PCPs as it is for PCPPs. □

7.2 Zero-knowledge proof composition

Essential to our theorems will be the ability to combine proofs in a way that preserves zero knowledge. In Theorem 5.2.1, we showed that composing robust PCPs and PCPPs can result in further PCPs; here we show that that combination also preserves zero knowledge. To do this, we will leverage the notion of local computation we defined in the last section. More specifically, Algorithm 5.1 is locally-computable from our robust pcp V_{out}, *regardless* of whether or not the PCPP V_{in} is zero-knowledge.

Theorem 7.2.1 ([17, Theorem 3.7])**.** *The construction in Theorem 5.2.1 is perfect zero-knowledge with query bound q^*/q_{out} if V_{out} is perfect zero-knowledge with query bound q^*.*

```
  Input: A string $r \in \{0,1\}^{r_{\text{out}}}$
  Output: The function $\pi_r$
1 $I_{\text{out}} \leftarrow Q_{\text{out}}(x, r)$;
2 Compile $D_{\text{out}}$ (the decision algorithm of $V_{\text{out}}$) on input $r$ into a circuit
    $C_{\text{out}}\colon \{0,1\}^n \times \{0,1\}^{\ell_{\text{out}}} \to \{0,1\}$;
3 Run $P_{\text{in}}(C_{\text{out}}, \pi_{\text{out}}|_{I_{\text{out}}})$ to get $\pi_r$;
4 return $\pi_r$;
```

Algorithm 7.3: An algorithm for π_r from π_0

Proof. All that is needed to prove this theorem is to show that Algorithm 5.1 preserves zero-knowledge, since we have already showed that it satisfies the conditions in Theorem 5.2.1. We do this by showing P_{comp} is q_{out}-locally computable from P_{out}.

We need to show that for any input $x \in L$, the distributions of $P(x)$ and $f^{\pi_0}(x)$ are identically distributed. Consider the function

$$f(O,r) = \begin{cases} \pi_0(r) & O = b_0 \\ \text{Algorithm 7.3} & O = b_r. \end{cases} \tag{7.1}$$

If $O = b_0$, then we make no more than one query to π_0. If $O = b_r$, then Algorithm 7.3 makes no more than $|I_{\text{out}}|$ queries to π_{out} (since that is the size of the domain of the restricted function); regardless of r we have that $|I_{\text{out}}| \leq q_{\text{out}}$ since I_{out} is a set of queries and q_{out} is the maximum number of queries that V_{out} makes. Hence f makes no more than q_{out} queries.

Lastly, so long as V_{out} runs in polynomial time, so must D_{out} and Q_{out}; compiling into a circuit is also known to take polynomial time. Lastly, the problem statement tells us P_{in} is guaranteed to run in polynomial time; it follows that f runs in polynomial time. Lastly, since Algorithm 7.3 uses its input r as randomness to all the algorithms it calls, it follows that f itself is deterministic (since all randomness comes from the choice of r). Hence, P_{comp} is q_{out}-locally computable from P_{out}.

Since P_{comp} is q_{out}-locally computable from P_{out}, by Theorem 7.1.2 we have that P_{comp} is perfect zero-knowledge with query bound q^*/q_{out}. □

This ability to compose PCPs gives us some useful properties of the class PZK-PCP. In particular, it shows us the ability to reduce a PZK-PCP to a *single* query in a zero-knowledge manner, simply with a corresponding worsening of our randomness complexity and zero-knowledge query bounds.

Corollary 7.2.2 ([17, Corollary 3.11])**.**

$$\mathsf{PZK\text{-}PCP}\begin{bmatrix} \textit{rand. complexity:} & r \\ \textit{query complexity:} & q \\ \textit{query bound:} & q^* \\ \textit{soundness error:} & \varepsilon \\ \textit{RS error:} & s \\ \textit{robustness param:} & \Omega(1) \end{bmatrix} \subseteq \mathsf{PZK\text{-}PCP}\begin{bmatrix} \textit{rand. complexity:} & r+\log(n) \\ \textit{query complexity:} & 1 \\ \textit{query bound:} & q^*/q \\ \textit{soundness error:} & \varepsilon \\ \textit{RS error:} & s \\ \textit{robustness param:} & \Omega(1) \end{bmatrix}. \tag{7.2}$$

Proof. Let

$$L \in \mathsf{PZK\text{-}PCP}_{\{0,1\}}\begin{bmatrix} \text{rand. complexity:} & r \\ \text{query complexity:} & q \\ \text{query bound:} & q^* \\ \text{soundness error:} & \varepsilon \\ \text{RS error:} & s \\ \text{robustness param:} & \Omega(1) \end{bmatrix}. \tag{7.3}$$

Then as per Theorem 5.1.9, we have

$$\mathsf{CktVal} \in \mathsf{PCPP}\begin{bmatrix} \text{rand. complexity:} & \log(n) + O(\log^{\varepsilon}(n)) \\ \text{query complexity:} & O(1/\varepsilon) \\ \text{prox. param.:} & k\varepsilon \\ \text{soundness error:} & 1/2 \end{bmatrix}, \tag{7.4}$$

for some $\varepsilon > 0$ and constant k. Define $\varepsilon = \rho/k$. Then, as per Theorem 7.2.1, we have L as desired. □

7.3 Zero-knowledge alphabet reduction

Back in Section 5.3, we talked about the importance of alphabet reduction in our proofs. As before, alphabet reduction will be important, but here we need to show that an alphabet reduction also preserves zero knowledge. Luckily for us, it does. The idea behind our alphabet reduction is to use error-correcting codes: since error-correcting codes are designed with the idea that correct values are far from incorrect ones, so too can they help guard against incorrect proofs here.

Theorem 7.3.1 ([8, Lemma 2.13])**.** *Let L be a language with a PZK-PCP over the language $\{0,1\}^a$ such that*

$$L \in \mathsf{PZK\text{-}PCP}_{\{0,1\}^a}\left[\begin{array}{rl} \textit{rand. complexity:} & r \\ \textit{query complexity:} & q \\ \textit{query bound:} & q^* \\ \textit{soundness error:} & \varepsilon \\ \textit{RS error:} & s \\ \textit{robustness param:} & \rho \end{array}\right].$$

Then L has a PZK-PCP over the language $\{0,1\}$ such that

$$L \in \mathsf{PZK\text{-}PCP}_{\{0,1\}}\left[\begin{array}{rl} \textit{rand. complexity:} & r \\ \textit{query complexity:} & O(aq) \\ \textit{query bound:} & q^* \\ \textit{soundness error:} & \varepsilon \\ \textit{RS error:} & s \\ \textit{robustness param:} & \Omega(\rho) \end{array}\right].$$

Proof. We proved the non-zero-knowledge version of this as Theorem 5.3.1, so all that remains is to prove that Algorithm 5.2 preserves zero-knowledge.

First, we show P is 1-locally computable from P_a. Consider the function f (with oracle π_a) defined by

$$f^{\pi_a}\colon \{b_\pi, b_\tau\} \times \mathrm{dom}(\pi_a) \times [n] \to \{0,1\}$$
$$f^{\pi_a}(O,\alpha,i) \mapsto \begin{cases} \pi_a(\alpha) & O = b_\pi \\ \mathsf{ECC}(\pi_a(\alpha))_i & O = b_\tau. \end{cases} \tag{7.5}$$

We show the distribution of P is the same as the distribution of f with oracle from P_a for all x.

The domain for our function is three values: O, a marker for either π_a or τ; α, the value we query; and i, the bit of the error-correcting code we wish to query. This is equivalent to a way to access the output of P from Algorithm 5.2; since it returns an ordered pair of two functions (π_a, τ), whenever we query it we need first to choose which of the two functions to query, the value we are querying them on, and then since we can only look at one bit, if we are querying τ we need to also determine which specific bit we are asking for. Hence, for any (O, α, i), the definition of f means that $\pi(O, \alpha, i) = f^{\pi_0}(O, \alpha, i)$. Hence the two are identically distributed and thus P is 1-locally computable from P_a.

Since P is 1-locally computable from P_a, it follows from Theorem 7.1.2 that P is perfect zero-knowledge with the same query bound as P_a. □

7.4 Zero-knowledge PCPPs for Sum

In Section 5.5, we described a PCPP for the language Sum. Since we are now working with zero-knowledge proofs, the question arises of whether this PCPP can be made zero knowledge. It can, although it requires an algorithm with a little more complexity than the non-zero knowledge Algorithm 5.4.

Theorem 7.4.1 ([17, Lemma 5.1])**.** *Let $\delta > 0$, $\mathbb{F}$ be a finite field, $H \subseteq \mathbb{F}$, $\gamma \in \mathbb{F}$, and $m, d \in \mathbb{N}$ such that $\frac{md}{|\mathbb{F}|} < \delta$ and $d > |H| + 1$. Then there exists a perfect zero-knowledge PCP of proximity for $\mathsf{Sum}[\mathbb{F}, m, d, H, \gamma]$ over the alphabet $\mathbb{F}^{m+1}$ with proximity parameter δ and robustness parameter $\rho = \Omega(\delta)$. Further, the verifier makes $O(|\mathbb{F}|)$ queries to F and π and the proof length is $O(|\mathbb{F}|^m)$.*

Proof. We construct such an algorithm as Algorithm 7.4. First, we will show that the verifier runs in polynomial time. Then, we will show that this is a PCP of proximity by showing correctness in both acceptance and rejection (with high probability), and then we will demonstrate zero knowledge.

We know from Theorem 5.5.3 that Algorithm 5.4 runs in polynomial time. Since we also do nontrivial work for each query, we need to show that that work is itself polynomial—since the sum is over m terms, it is. As per Theorem 5.4.1, Algorithm 5.3 is also computable in polynomial time. The rest of the algorithm is just running Algorithm 5.3 twice; hence in total the verifier in Algorithm 7.4 runs in polynomial time.

Let $F \in \mathsf{Sum}[\mathbb{F}, m, d, H, \gamma]$, and let (π_Σ, π_P) be the honest proof. In this case, as we showed in Theorem 5.5.3, the simulation of Algorithm 5.4 we do on line 14 will always succeed. To show that the equation in line 14 is true, note that from the definition of π_P, $(\pi_P(\alpha))_1 = Q(x)$ and $(\pi_P(\alpha))_{i+1} = T_i(\alpha)$ for all $i \geq 1$. Doing these substitutions, the right hand side of the equation is exactly $F(\alpha)$ plus the definition of R from line 9. Since F is defined in the problem statement to be a polynomial of degree d, the check in line 15 will always succeed. Further, Q and each T_i are defined to be multidegree-d m-variable polynomials and thus their total degree will be less than md. Hence, the check in line 19 will pass. As such, when given valid inputs Algorithm 7.4 will always succeed.

Let F be δ-far from $\mathsf{Sum}[\mathbb{F}, m, d, H, \gamma]$. We split into two cases: where F is δ_{RM}-close to $\mathrm{RM}[\mathbb{F}, m, d]$ and where F is δ_{RM}-far from $\mathrm{RM}[\mathbb{F}, m, d]$. Similarly to what we did in Theorem 5.5.3, we will need to introduce a technical lemma to prove some of

```
proof
    Sample $Q \in \mathbb{F}[X_{1,\dots,m}^{\leq d}]$ uniformly;
    for $i \in \{1, \dots, m\}$ do
        Sample $T_i \in \mathbb{F}[X_{1,\dots,i-1}^{\leq d}, X_i^{\leq d-|H|}, X_{i+1,\dots,m}^{\leq d}]$ uniformly;
    end
    Define $\pi_P(x) = (Q(x), T_1(x), \dots, T_m(x))$;
    Define $Z_H(X) = \prod_{a \in H}(X - a)$;
    Define $Q_{\mathrm{rev}}(X) = Q(X_{\mathrm{rev}})$;
    Define $R(X) = Q(X) - Q_{\mathrm{rev}}(X) + \sum_{i=1}^{m} Z_H(X_i)T_i(X)$;
    Compute the proof $\pi_\Sigma$ for Algorithm 5.4 with explicit input
     $(\mathbb{F}, m, d, H, \gamma, \delta)$ and implicit input $F + R$;
    return $(\pi_\Sigma, \pi_P)$;
end
verifier
    Emulate Algorithm 5.4 on input $F + R$ and proof $\pi_\Sigma$. To query $F + R$ at
     some $\alpha \in \mathbb{F}^m$, query $F(\alpha)$, $\pi_P(\alpha)$, and $\pi_P(\alpha_{\mathrm{rev}})$, then compute

        $(F + R)(\alpha) = F(\alpha) + (\pi_P(\alpha))_1 - (\pi_P(\alpha_{\mathrm{rev}}))_1 + \sum_{i=1}^{m} Z_H(\alpha_i)(\pi_P(\alpha))_{i+1}$;

    Perform Algorithm 5.3 on $F$ with proximity parameter $\delta_{RM} = \min(d, 1/5)$;
    if Algorithm 5.3 fails then
        reject;
    end
    Perform Algorithm 5.3 on $\pi_P$ with proximity parameter $\varepsilon_P = \delta_R/8$ and
     degree parameter $d_P = md$;
    if Algorithm 5.3 fails then
        reject;
    else
        accept;
    end
end
```

Algorithm 7.4: A zero-knowledge robust PCPP for Sum [17, Construction 5.2]

this. Like before, we will leave this unproven, as its proof is highly technical and not particularly interesting.

Lemma 7.4.2 ([17, Lemma 5.3])**.** *Let $\tilde{F}\colon \mathbb{F}^m \to \mathbb{F}$ be δ_Σ-far from* $\mathsf{Sum}[\mathbb{F}, m, d, H, \gamma]$*, but δ_{RM}-close to* $\mathrm{RM}[\mathbb{F}, m, d]$ *for $\delta_\Sigma > \delta_{\mathrm{RM}} \geq \frac{md}{|\mathbb{F}|}$, $\delta_{\mathrm{RM}} < 1/5$, and $d \geq |H| + 1$. Then, for all proofs (π^*_Σ, π^*_P),*

$$\mathop{\mathbb{E}}_{c \leftarrow \mathbb{F}^{m-1}}\left[\Delta\Big((\pi^*_\Sigma(c_{m-2}, \alpha)_{\alpha\in\mathbb{F}}, \pi^*_P(c, \alpha)_{\alpha\in\mathbb{F}}, \pi^*_P(\alpha, c_{\mathrm{rev}})_{\alpha\in\mathbb{F}}, \tilde{F}(c, \alpha)_{\alpha\in\mathbb{F}}), \mathrm{Acc}(V)\Big)\right] \in \Omega(\delta_{\mathrm{RM}}). \quad (7.6)$$

If F is δ_{RM}-close to $\mathrm{RM}[\mathbb{F}, m, d]$, then by Lemma 7.4.2, we have that the expected view of the verifier (more specifically, the non-low-degree-test portions thereof) is $\Omega(\delta_{\mathrm{RM}}) = \Omega(\delta)$-far from an accepting view. Like we saw in Theorem 5.5.3, the total proportion of the queries made by this portion of the proof is constant. Hence, regardless of the failure rate of the low-degree test, the expected distance of any accepting view from the verifier's view is $\Omega(\delta)$.

If F is δ_{RM}-far from $\mathrm{RM}[\mathbb{F}, m, d]$, then since Algorithm 5.3 is robust, we know that the distance between any accepting view and the verifier's view is in $\Omega(\delta_{\mathrm{RM}}) = \Omega(\delta)$. Like in the other case, this forms a constant fraction of the overall queries; hence the expected distance is $\Omega(\delta)$. □

7.5 A zero-knowledge PCP for NP and NEXP

The next step in our journey is to define any sort of zero-knowledge PCP for both NP and NEXP. We will not be defining these to have any of the query properties we want (they will not even be in the language $\{0, 1\}$ for the time being), but that is okay for now. For now, it is more important to show that *any* zero-knowledge PCPs exist, and then in the next section we will show that these are reducible to zero-knowledge PCPs with the query properties that we want.

Lemma 7.5.1 ([9, Corollary 4.10])**.** *There exists a probabilistic algorithm* PolySim *such that, for every*

1. *$\mathbb{F}$ a finite field,*
2. *$m, d \in \mathbb{N}$,*
3. *$S = \{(\alpha_1, \beta_1), \ldots, (\alpha_\ell, \beta_\ell)\} \subseteq \mathbb{F}^m \times \mathbb{F}$,*
4. *and $(\alpha, \beta) \in \mathbb{F}^m \times \mathbb{F}$,*

then

$$\mathbb{P}[\mathsf{PolySim}(\mathbb{F}, m, d, S, \alpha) = \beta] = \mathop{\mathbb{P}}_{Q\in\mathbb{F}^{\leq d}[X_{1,\ldots,m}]}\left[Q(\alpha) = \beta \;\middle|\; \begin{matrix} Q(\alpha_1) = \beta_1 \\ \vdots \\ Q(\alpha_\ell) = \beta_\ell \end{matrix}\right]. \quad (7.7)$$

Finally, it is time to show an inclusion that involves NEXP! First, note that this is not remotely our desired inclusion: the language is wrong, and the $\mathsf{poly}(n)$ query complexity is not very close to the $O(1)$ we actually want. However, we are still in the home stretch—after this, we have proven every component we need to reduce this bound down into the one we desire, and we will do so in the next section.

Additionally, one might notice that we are starting with NEXP, instead of the much smaller class NP. This is because we are going to be working with the class O3SAT for *both* inclusions, and since O3SAT is NEXP-complete, the inclusions are much more straightforward for NEXP. Once we have done that, however, the reductions necessary to make this work for NP will not be too challenging.

Theorem 7.5.2 ([17, Theorem 6.3])**.** *For any query bound $q^*(n) \le 2^{\mathsf{poly}(n)}$,*

$$\mathsf{NEXP} \subseteq \mathsf{PZK\text{-}PCP}_{\Sigma(n)} \left[\begin{array}{rl} \textit{rand. complexity:} & \mathsf{poly}(n) + \log(q^*(n)) \\ \textit{query complexity:} & \mathsf{poly}(n) \\ \textit{query bound:} & q^*(n) \\ \textit{soundness error:} & \varepsilon \\ \textit{RS error:} & s \\ \textit{robustness param:} & \Omega(1) \end{array} \right].$$

where $\Sigma(n)$ is any alphabet with $|\Sigma(n)| \in \mathsf{poly}(n, q)$.

Proof. We construct a PZK-PCP for O3SAT, a NEXP-complete language, in Algorithm 7.5. First, note that the combined effect of the three "repeat" loops is to ensure that the number of queries taken up by each of the two inner portions is no more than 1/3 of the total number of queries. This will be useful to us later on when we try to prove soundness.

We note that Algorithm 7.5 is exactly identical to Algorithm 5.5, but where the sumcheck call has been replaced with the zero-knowledge sumcheck PCPP we constructed in Algorithm 7.4. Since the zero-knowledge algorithm has the same completeness and soundness properties as the nonzero-knowledge algorithm, completeness and soundness hold by the same argument we made in Theorem 5.7.1.

Hence, all we need is to show Algorithm 7.6 is a simulator for Algorithm 7.5 and thus, that it is zero-knowledge. For this, we will use a hybrid argument: we will start by constructing a simulator that uses an external oracle (and thus is not technically a valid simulator for our purposes) and reduce it until we get to a simulator that does not query an oracle, proving an equivalence at each step along the way. In the intermediate steps, our oracle Z replaces the queries to π_C.

Our sequence of algorithms is as follows:

1. H_0: $\mathrm{View}_{V^*,P}(x)$
2. H_1: $\overline{\mathrm{Sim}}^{V^*,Z}(x)$, where $Z\colon \mathbb{F}^{m_2+k}$ is sampled as $\hat{C}$ in line 3 in Algorithm 7.5
3. H_2: $\overline{\mathrm{Sim}}^{V^*,Z}(x)$, where $Z\colon \mathbb{F}^{m_2+k}$ is a uniformly random polynomial of multidegree $|H| - 1$

```
Input: A 3-CNF $B\colon \{0,1\}^{r+3s+3} \to \{0,1\}$
Output: Whether $B$ is implicitly satisfiable
proof
    Let $A\colon \{0,1\}^n \to \{0,1\}$ be a satisfying assignment for $B$;
    Choose $\hat{C} \in \mathbb{F}[X_{m_2+k}^{\leq 2(|H|-1)}]$ randomly such that $\sum_{c \in H^k} \hat{C}(b,c) = A(\gamma_2, b)$ for
     all $b \in H^{m_2}$;
    for $\tau \in \mathbb{F}^{m_1+3m_2+3}$ do
        Let $\pi_\tau$ be a PZK-PCPP for the claim
            $$\sum_{\substack{z \in H^{m_1} \\ b_1,b_2,b_3 \in H^{m_2}}} \sum_{\substack{a \in \{0,1\}^3 \\ c_1,c_2,c_3 \in H^k}} \delta_{(z,b,a)}(\tau) h_{\hat{C}}(\tau, c_1, c_2, c_3) = 0;$$
    end
    return $(\pi_C, (\pi_\tau)_{\tau \in \mathbb{F}^{m_1+3m_2+3}})$;
end
verifier
    $q_a, q_b \leftarrow 0$;
    repeat
        repeat
            Run Algorithm 5.3 on the proof, with $\varepsilon = 1/100$ and $\delta = 1/2$;
            Add the total number of queries in the last line to $q_a$;
            if the above rejects then
                reject;
            end
        until $2q_a > q_b$;
        repeat
            Choose $\tau \in \mathbb{F}^{m_1} \times (\mathbb{F}^{m_2})^3 \times \mathbb{F}^3$ at random;
            Simulate Algorithm 7.4 with proof $\pi_\tau$;
            Add the total number of queries in the last line to $q_b$;
            for $i \in \{1,2,3\}$ do
                Query $\hat{C}$ at $(\nu_i, \eta_i)$;
                Add 1 to $q_b$;
            end
            Compute $h_{\hat{C}}(\tau, \eta_1, \eta_2, \eta_3)$;
            if the claim in line 21 is false then
                reject;
            end
        until $2q_b > q_a$;
    until $2q_a > q_b$ and $2q_b > q_a$;
    accept;
end
```

Algorithm 7.5: A PZK-PCP for O3SAT [17, Construction 6.4]

```
Input: A query $\alpha$ to an oracle $O$, either $\pi_C$ or $\pi_\tau$
Output: The result of the query
1 $S \leftarrow \varnothing$;            // $S$ is the set of all previous queries to $\pi_C$
2 $T \leftarrow \varnothing$;            // $T$ is the set of all started $\mathrm{Sim}'_\tau$ instances
3 if The query request is to $\pi_C$ then
4     Sample $\beta \leftarrow \mathsf{PolySim}(\mathbb{F}, m+k, d, S, q)$;
5     Add $(\alpha, \beta)$ to $S$;
6     Respond with $\beta$;
7 else
8     if $\tau \notin T$ then          // When we have not queried this $\tau$ before
9         Add $\tau$ to $T$;
10        Start an instance $\mathrm{Sim}'_\tau$ of a simulator for Algorithm 7.4;
11    end
12    Use $\mathrm{Sim}'_\tau$ to answer $\alpha$: it may make queries to $\tau$ by asking $\pi_C$ (answered as
       in line 4);
13 end
```

Algorithm 7.6: A simulator for Algorithm 7.5 [17, Construction 6.7]

4. H_4: $\mathrm{Sim}^{V^*}(x)$ (Algorithm 7.6)

To show $H_0 \equiv H_1$, we use the zero-knowledge guarantee of Algorithm 7.4. As per Theorem 7.4.1, Algorithm 7.4 has an efficient simulator that outputs an identical distribution to the view of the verifier. Hence, our Sim'_τ will output a response statistically identical to the verifier's query of τ.

To show $H_1 \equiv H_2$, we rely on Corollary 6.1.4 proven earlier. From earlier, we know V^* makes at most q^* queries to the proof, and thus $\overline{\mathrm{Sim}}$ makes at most $p(q^*)$ queries to Z. Since $p(q^*) \leq |H|^k$, these query answers will be independent from $\left(\sum_{y \in H^k} Z(\alpha, y)\right)_{\alpha \in \mathbb{F}^m}$ as per Corollary 6.1.4. As such, for all v,

$$\begin{aligned}
\mathbb{P}[v \leftarrow H_2(x)] &= \mathbb{P}_Z\left[v \leftarrow \overline{\mathrm{Sim}}^{V^*, Z}(x) \,\middle|\, \forall b \in H^{m_2}, \sum_{c \in H^k} \hat{C}(b, c) = A(\gamma_2(b)) \right] \\
&= \mathbb{P}_Z\left[v \leftarrow \overline{\mathrm{Sim}}^{V^*, Z}(x) \right] \\
&= \mathbb{P}[v \leftarrow H_1(x)].
\end{aligned}$$

To show $H_2 \equiv H_3$, we use the property of $\mathsf{PolySim}$: the output of the algorithm is distributed identically to output of a random polynomial Z conditional on all the previous results being true about Z. Hence, the output of H_2 must be identically distributed to H_3.

Since equivalence is transitive, it follows therefore that $H_0 \equiv H_3$ and thus the view of V^* is always identical to the output of Algorithm 7.6. □

Corollary 7.5.3 ([17, Theorem 6.3]). *For any query bound $q^*(n) \leq 2^{\mathsf{poly}(n)}$,*

$$\mathsf{NP} \subseteq \mathsf{PZK\text{-}PCP}_{\Sigma(n)} \left[\begin{array}{rl} \textit{rand. complexity:} & \log(n) + \log(q^*(n)) \\ \textit{query complexity:} & \mathsf{poly}(\log(n) + \log(q^*(n))) \\ \textit{query bound:} & q^*(n) \\ \textit{soundness error:} & \varepsilon \\ \textit{RS error:} & s \\ \textit{robustness param:} & \Omega(1) \end{array} \right].$$

where $|\Sigma(n)| = \mathsf{poly}(n, q)$*.*

Proof. For this, we leverage our work from Theorem 7.5.2. In that, we used the fact that $\mathsf{O3SAT}$ is a NEXP-complete problem. However, the Cook-Levin variant we proved in Theorem 2.2.30 is even more general than that: in particular it showed that log-length $\mathsf{O3SAT}$ is in fact NP-complete. Hence, if we adjust the length of our input, the inclusion here follows. □

7.5.1 Reducing query complexity to constant

We are now finally able to present our analogues to the classical PCP theorem. Through our work with $\mathsf{O3SAT}$, we do not only have a version of the standard PCP theorem (the one involving NP), but another stronger one involving PCP's relationship with NEXP.

Theorem 7.5.4 ([17, Theorem 2]). $\mathsf{NP} \subseteq \mathsf{PZK\text{-}PCP}[\log(n), 1]$.

Proof. To show this, we will take the PCP for NP that we showed in Corollary 7.5.3, which has relatively weak bounds and an arbitrary alphabet, and transform it into one with the exact parameters we seek. We do this first through an alphabet reduction (as per Theorem 5.3.1) and then through proof-composing it with the algorithm for CktVal with Corollary 7.2.2, we can get a constant query-complexity PCP. Since several of these class inclusions involve modifying a small number of parameters in a relatively complex class, we will be highlighting the changed parameters in each inclusion statement.

Let $q^*(n) \leq 2^{\mathsf{poly}(n)}$ be arbitrary. This will be the query complexity of our final PCP after we do all the class inclusions. As per Corollary 7.5.3, we know that

$$\mathsf{NP} \subseteq \mathsf{PZK\text{-}PCP}_{\Sigma(n)} \left[\begin{array}{rl} \text{rand. complexity:} & \log(n) + \log(\tilde{q}(n)) \\ \text{query complexity:} & \mathsf{poly}(\log(n) + \log(\tilde{q}(n))) \\ \text{query bound:} & \tilde{q}(n) \\ \text{soundness error:} & \varepsilon \\ \text{RS error:} & s \\ \text{robustness param:} & \Omega(1) \end{array} \right]. \tag{7.8}$$

for any $\tilde{q} \leq 2^{\mathsf{poly}(n)}$ and where $|\Sigma(n)| \in \mathsf{poly}(n, \tilde{q})$. Corollary 7.5.3 only guarantees this inclusion for alphabets of $\{0,1\}^a$, however a PCP over $\Sigma(n)$ is equivalent to a PCP over $\{0,1\}^{\log_2(|\Sigma(n)|)}$ by a simple relabeling of alphabet items, so this inclusion still holds.

Next, we perform an alphabet reduction: by Theorem 7.3.1,

$$\begin{aligned}&\mathsf{PZK\text{-}PCP}_{\Sigma(n)}\left[\begin{array}{rl}\text{rand. complexity:} & \log(n)+\log(\tilde{q}(n))\\ \text{query complexity:} & \mathsf{poly}(\log(n)+\log(\tilde{q}(n)))\\ \text{query bound:} & \tilde{q}(n)\\ \text{soundness error:} & \varepsilon\\ \text{RS error:} & s\\ \text{robustness param:} & \Omega(1)\end{array}\right]\\ &\subseteq \mathsf{PZK\text{-}PCP}_{\{0,1\}}\left[\begin{array}{rl}\text{rand. complexity:} & \log(n)+\log(\tilde{q}(n))\\ \text{query complexity:} & \mathsf{poly}(\log(n)+\log(\tilde{q}(n)))\\ \text{query bound:} & \tilde{q}(n)\\ \text{soundness error:} & \varepsilon\\ \text{RS error:} & s\\ \text{robustness param:} & \Omega(1)\end{array}\right].\end{aligned} \tag{7.9}$$

Next, we perform proof composition. For any language $L \in \mathsf{NP}$, let $Q(n) \in \mathsf{poly}(\log(n)+\log(\tilde{q}(n)))$ be the query complexity of the PZK-PCP within the parameters of the right-hand side of Equation (7.9) that recognizes L. Since $\tilde{q}$ was arbitrary, we can define it to be whatever we like; hence let $\tilde{q}(n)$ be a polynomial in q^* and n large enough that for all $n \in \mathbb{N}$, $\tilde{q}(n)/Q(n) \geq q^*(n)$. By Corollary 7.2.2, we have that

$$\begin{aligned}&\mathsf{PZK\text{-}PCP}_{\{0,1\}}\left[\begin{array}{rl}\text{rand. complexity:} & \log(n)+\log(\tilde{q}(n))\\ \text{query complexity:} & \mathsf{poly}(\log(n)+\log(\tilde{q}(n)))\\ \text{query bound:} & \tilde{q}(n)\\ \text{soundness error:} & \varepsilon\\ \text{RS error:} & s\\ \text{robustness param:} & \Omega(1)\end{array}\right]\\ &\subseteq \mathsf{PZK\text{-}PCP}_{\{0,1\}}\left[\begin{array}{rl}\text{rand. complexity:} & \log(n)+\log(q^*(n))\\ \text{query complexity:} & 1\\ \text{query bound:} & q^*(n)\\ \text{soundness error:} & \varepsilon\end{array}\right].\end{aligned} \tag{7.10}$$

At the beginning of the proof, we said q^* was arbitrary; hence we can set $q^* \in O(1)$. Thus, we get

$$\mathsf{PZK\text{-}PCP}_{\{0,1\}}\left[\begin{array}{rl}\text{rand. complexity:} & \log(n)+\log(q^*(n))\\ \text{query complexity:} & 1\\ \text{query bound:} & q^*(n)\\ \text{soundness error:} & \varepsilon\end{array}\right] \subseteq \mathsf{PZK\text{-}PCP}[\log(n), 1]. \tag{7.11}$$

By combining the inclusions in Equations (7.8) to (7.11), we get that $\mathsf{NP} \subseteq \mathsf{PZK\text{-}PCP}[\log(n), 1]$, as desired. □

Theorem 7.5.5 ([17, Theorem 7.1])**.** $\mathsf{NEXP} \subseteq \mathsf{PZK\text{-}PCP}[\mathsf{poly}(n), 1]$.

Proof. Broadly speaking, this proof will proceed in the same style as the proof for Theorem 7.5.4. Let $q^* \leq 2^{\mathsf{poly}(n)}$ be arbitrary. This will be the query complexity of our final PCP after we do all the class inclusions. As per Theorem 7.5.2, we know that

$$\mathsf{NEXP} \subseteq \mathsf{PZK\text{-}PCP}_{\Sigma(n)}\left[\begin{array}{rl}\text{rand. complexity:} & \mathsf{poly}(n)+\log(\tilde{q}(n))\\ \text{query complexity:} & \mathsf{poly}(n)\\ \text{query bound:} & \tilde{q}(n)\\ \text{soundness error:} & \varepsilon\\ \text{RS error:} & s\\ \text{robustness param:} & \Omega(1)\end{array}\right]. \tag{7.12}$$

for any $\tilde{q} \leq 2^{\mathsf{poly}(n)}$ and where $|\Sigma(n)| \in \mathsf{poly}(n, \tilde{q})$. As before, Corollary 7.5.3 only guarantees this inclusion for alphabets of $\{0,1\}^a$, however a PCP over $\Sigma(n)$ is equivalent to a PCP over $\{0,1\}^{\log_2(|\Sigma(n)|)}$ by a simple relabeling of alphabet items, so this inclusion still holds.

Next, we perform an alphabet reduction: by Theorem 7.3.1,

$$\begin{aligned}&\mathsf{PZK\text{-}PCP}_{\Sigma(n)}\begin{bmatrix}\text{rand. complexity:} & \mathsf{poly}(n)+\log(\tilde{q}(n))\\ \text{query complexity:} & \mathsf{poly}(n)\\ \text{query bound:} & \tilde{q}(n)\\ \text{soundness error:} & \varepsilon\\ \text{RS error:} & s\\ \text{robustness param:} & \Omega(1)\end{bmatrix}\\ \subseteq\ &\mathsf{PZK\text{-}PCP}_{\{0,1\}}\begin{bmatrix}\text{rand. complexity:} & \mathsf{poly}(n)+\log(\tilde{q}(n))\\ \text{query complexity:} & \mathsf{poly}(n,\log(\tilde{q}(n)))\\ \text{query bound:} & \tilde{q}(n)\\ \text{soundness error:} & \varepsilon\\ \text{RS error:} & s\\ \text{robustness param:} & \Omega(1)\end{bmatrix}.\end{aligned} \tag{7.13}$$

Next, we perform proof composition. For any language $L \in \mathsf{NP}$, let $Q(n) \in \mathsf{poly}(n)$ be the query complexity of the PZK-PCP within the parameters of the right-hand side of Equation (7.9) that recognizes L. Since $\tilde{q}$ was arbitrary, we can define it to be whatever we like; hence let $\tilde{q}(n) = q^*(n) \cdot Q(n)$ for all $n \in \mathbb{N}$. By Corollary 7.2.2, we have that

$$\begin{aligned}&\mathsf{PZK\text{-}PCP}_{\{0,1\}}\begin{bmatrix}\text{rand. complexity:} & \mathsf{poly}(n)\\ \text{query complexity:} & \mathsf{poly}(n,\log(\tilde{q}(n)))\\ \text{query bound:} & \tilde{q}(n)\\ \text{soundness error:} & \varepsilon\\ \text{RS error:} & s\\ \text{robustness param:} & \Omega(1)\end{bmatrix}\\ \subseteq\ &\mathsf{PZK\text{-}PCP}_{\{0,1\}}\begin{bmatrix}\text{rand. complexity:} & \mathsf{poly}(n)+\log(q^*(n))\\ \text{query complexity:} & 1\\ \text{query bound:} & q^*(n)\\ \text{soundness error:} & \varepsilon\end{bmatrix}.\end{aligned} \tag{7.14}$$

At the beginning of the proof, we said q^* was arbitrary; hence we can set $q^* \in O(1)$. Thus, we get

$$\mathsf{PZK\text{-}PCP}_{\{0,1\}}\begin{bmatrix}\text{rand. complexity:} & \mathsf{poly}(n)+\log(q^*(n))\\ \text{query complexity:} & 1\\ \text{query bound:} & q^*(n)\\ \text{soundness error:} & \varepsilon\end{bmatrix} \subseteq \mathsf{PZK\text{-}PCP}[\mathsf{poly}(n), 1]. \tag{7.15}$$

By combining the inclusions in Equations (7.12) to (7.15), we get that $\mathsf{NEXP} \subseteq \mathsf{PZK\text{-}PCP}[\mathsf{poly}(n), 1]$, as desired. □

Appendix A

More on extension polynomials

In this appendix, we will work through some of the algebra we mentioned but did not go into detail about in Section 2.3.

A.1 A proof of Equation (2.6)

Our goal is to demonstrate the following:

$$[x = y] = \prod_{i=1}^{m} \left(\sum_{\omega \in H} \left(\prod_{\gamma \in H \setminus \{\omega\}} \frac{(x_i - \gamma)(y_i - \gamma)}{(\omega - \gamma)^2} \right) \right) \tag{A.1}$$

for all $x, y \in H^n$. We will do this in two cases: one where $x = y$ and one where $x \neq y$.

First, assume $x \neq y$ (so we want to show $\delta_y(x) = 0$). In this case, there exists at least one i where $x_i \neq y_i$. For this i, for each ω there exists some $\gamma \in H \setminus \{\omega\}$ such that either $x_i = \gamma$ or $y_i = \gamma$.[1] As such, it follows that either $(x_i - \gamma) = 0$ or $(\gamma_i - \gamma) = 0$. Hence, for this i the sum will be entirely over zero terms (since there will be at least one zero term in the product for each ω). As such, this means that the ith term of our outermost product is 0, and hence the entire product is 0, as desired.

When $x = y$ (and so we want to show $\delta_y(x) = 1$), the above equation simplifies to

$$[x = y] = \prod_{i=1}^{m} \left(\sum_{\omega \in H} \left(\prod_{\gamma \in H \setminus \{\omega\}} \frac{(x_i - \gamma)^2}{(\omega - \gamma)^2} \right) \right) \tag{A.2}$$

Whenever $\omega \neq x_i$, the innermost product becomes 0 since there will be a term where $\gamma = x_i$. Hence, we can simplify this further to

$$[x = y] = \prod_{i=1}^{m} \left(\prod_{\gamma \in H \setminus \{\omega\}} \frac{(x_i - \gamma)^2}{(x_i - \gamma)^2} \right). \tag{A.3}$$

Since $\gamma \neq x_i$, we can simplify the fraction to 1; since we have two nested products it follows that the equation as a whole simplifies to 1.

[1]This piece fails in the case where $x_i = y_i$, since if $\omega = x_i = y_i$ neither of the terms will ever be zero.

A.2 Algebra behind Equation (2.8)

Our goal is to show that the equation in Equation (2.5) simplifies to that of Equation (2.8) when $H = \{0,1\}^n$.

As a refresher, our starting equation has the form

$$\delta_y(x) = \prod_{i=1}^{m} \left(\sum_{\omega \in H} \left(\prod_{\gamma \in H \setminus \{\omega\}} \frac{(x_i - \gamma)(y_i - \gamma)}{(\omega - \gamma)^2} \right) \right). \tag{A.4}$$

We start by manually substituting the outer sum:

$$\delta_y(x) = \prod_{i=1}^{m} \left(\left(\prod_{\gamma \in \{0,1\} \setminus \{0\}} \frac{(x_i - \gamma)(y_i - \gamma)}{(\omega - \gamma)^2} \right) + \left(\prod_{\gamma \in \{0,1\} \setminus \{1\}} \frac{(x_i - \gamma)(y_i - \gamma)}{(\omega - \gamma)^2} \right) \right). \tag{A.5}$$

Next, notice that the inner products are actually each over one term, so we can manually substitute there:

$$\delta_y(x) = \prod_{i=1}^{m} \left(\frac{(x_i - 1)(y_i - 1)}{(0-1)^2} + \frac{(x_i - 0)(y_i - 0)}{(1-0)^2} \right). \tag{A.6}$$

Next, we simplify, taking note that the denominator of both fractions is 1:

$$\delta_y(x) = \prod_{i=1}^{m} ((x_i - 1)(y_i - 1) + x_i y_i). \tag{A.7}$$

From here, we take advantage of the fact that $y \in \{0,1\}^n$; here we split our product into two smaller products: one where $y_i = 0$ and one where $y_i = 1$.

$$\delta_y(x) = \left(\prod_{i:y_i=0} (x_i - 1)(0-1) + 0x_i \right) \left(\prod_{i:y_i=1} (x_i - 1)(1-1) + x_i 1 \right). \tag{A.8}$$

Finally, we simplify, bringing us to Equation (2.8).

$$\delta_y(x) = \left(\prod_{i:y_i=0} (1 - x_i) \right) \left(\prod_{i:y_i=1} x_i \right). \tag{A.9}$$

Bibliography

[1] Scott Aaronson and Avi Wigderson. “Algebrization: A New Barrier in Complexity Theory”. In: *ACM Trans. Comput. Theory* 1.1 (Feb. 2009). ISSN: 1942-3454. DOI: 10.1145/1490270.1490272.

[2] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. 1st. USA: Cambridge University Press, 2009. ISBN: 978-0-521-42426-4. DOI: 10.5555/1540612.

[3] Sanjeev Arora and Shmuel Safra. “Probabilistic checking of proofs: a new characterization of NP”. In: *J. ACM* 45.1 (Jan. 1998), pp. 70–122. ISSN: 0004-5411. DOI: 10.1145/273865.273901. URL: https://doi.org/10.1145/273865.273901.

[4] L. Babai, L. Fortnow, and C. Lund. “Nondeterministic exponential time has two-prover interactive protocols”. In: *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*. 1990, 16–25 vol.1. DOI: 10.1109/FSCS.1990.89520.

[5] László Babai and Lance Fortnow. “Arithmetization: A new method in structural complexity theory”. In: *computational complexity* 1.1 (Mar. 1991), pp. 41–66. ISSN: 1420-8954. DOI: 10.1007/BF01200057. URL: https://link.springer.com/article/10.1007/BF01200057.

[6] Theodore Baker, John Gill, and Robert Solovay. “Relativizations of the $\mathcal{P} \overset{?}{=} \mathcal{NP}$ Question”. In: *SIAM Journal on Computing* 4.4 (1975), pp. 431–442. DOI: 10.1137/0204037. eprint: https://doi.org/10.1137/0204037. URL: https://doi.org/10.1137/0204037.

[7] Michael Ben-Or et al. “Multi-prover interactive proofs: how to remove intractability assumptions”. In: *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. STOC ’88. Chicago, Illinois, USA: Association for Computing Machinery, 1988, pp. 113–131. ISBN: 0897912640. DOI: 10.1145/62212.62223. URL: https://doi.org/10.1145/62212.62223.

[8] Eli Ben-Sasson et al. “Robust PCPs of proximity, shorter PCPs and applications to coding”. In: *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*. STOC ’04. Chicago, IL, USA: Association for Computing Machinery, 2004, pp. 1–10. ISBN: 1581138520. DOI: 10.1145/1007352.1007361. URL: https://doi.org/10.1145/1007352.1007361.

[9] Eli Ben-Sasson et al. "Zero Knowledge Protocols from Succinct Constraint Detection". In: *Theory of Cryptography*. Ed. by Yael Kalai and Leonid Reyzin. Cham: Springer International Publishing, 2017, pp. 172–206. ISBN: 978-3-319-70503-3.

[10] Alessandro Chiesa et al. "Spatial Isolation Implies Zero Knowledge Even in a Quantum World". In: *J. ACM* 69.2 (Jan. 2022). ISSN: 0004-5411. DOI: 10.1145/3511100. URL: https://doi.org/10.1145/3511100.

[11] Stephen A. Cook. "A hierarchy for nondeterministic time complexity". In: *Journal of Computer and System Sciences* 7.4 (1973), pp. 343–353. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(73)80028-5. URL: https://www.sciencedirect.com/science/article/pii/S0022000073800285.

[12] Stephen A. Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[13] Thomas H. Cormen et al. *Introduction to Algorithms*. 4th ed. MIT Press, 2022. ISBN: 978-0-262-04630-5. URL: https://mitpress.mit.edu/9780262046305/introduction-to-algorithms.

[14] O. Goldreich and L. A. Levin. "A hard-core predicate for all one-way functions". In: *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*. STOC '89. Seattle, Washington, USA: Association for Computing Machinery, 1989, pp. 25–32. ISBN: 0897913078. DOI: 10.1145/73007.73010. URL: https://doi.org/10.1145/73007.73010.

[15] Oded Goldreich. *Foundations of Cryptography*. Vol. 1. 1st ed. Cambridge University Press, 2001. 372 pp. ISBN: 978-0-511-54689-1. DOI: 10.1017/CBO9780511546891.

[16] Oded Goldreich and Liav Teichner. "Super-Perfect Zero-Knowledge Proofs". In: *Computational Complexity and Property Testing: On the Interplay Between Randomness and Computation*. Ed. by Oded Goldreich. Cham: Springer International Publishing, 2020, pp. 119–140. ISBN: 978-3-030-43662-9. DOI: 10.1007/978-3-030-43662-9_8. URL: https://doi.org/10.1007/978-3-030-43662-9_8.

[17] Tom Gur, Jack O'Connor, and Nicholas Spooner. *A Zero-Knowledge PCP Theorem*. 2024. arXiv: 2411.07972 [cs.CC]. URL: https://arxiv.org/abs/2411.07972.

[18] Tom Gur, Jack O'Connor, and Nicholas Spooner. "Perfect Zero-Knowledge PCPs for #P". In: *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*. STOC 2024. Vancouver, BC, Canada: Association for Computing Machinery, 2024, pp. 1724–1730. ISBN: 9798400703836. DOI: 10.1145/3618260.3649698. URL: https://doi.org/10.1145/3618260.3649698.

[19] R. W. Hamming. "Error detecting and error correcting codes". In: *The Bell System Technical Journal* 29.2 (1950), pp. 147–160. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[20] Johan Håstad. “Some optimal inapproximability results”. In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*. STOC ’97. El Paso, Texas, USA: Association for Computing Machinery, 1997, pp. 1–10. ISBN: 0897918886. DOI: 10.1145/258533.258536. URL: https://doi.org/10.1145/258533.258536.

[21] R. Impagliazzo. “A personal view of average-case complexity”. In: *Proceedings of Structure in Complexity Theory. Tenth Annual IEEE Conference*. 1995, pp. 134–147. DOI: 10.1109/SCT.1995.514853.

[22] Kenneth E. Iverson. *A Programming Language*. 1st ed. John Wiley and Sons, Inc., 1962. 286 pp. ISBN: 978-0471430148. URL: https://dl.acm.org/doi/10.5555/1098666.

[23] Ali Juma et al. “The Black-Box Query Complexity of Polynomial Summation”. In: *Comput. Complex.* 18.1 (Apr. 2009), pp. 59–79. ISSN: 1016-3328. DOI: 10.1007/s00037-009-0263-7. URL: https://doi.org/10.1007/s00037-009-0263-7.

[24] Yael Tauman Kalai and Ran Raz. “Interactive PCP”. In: *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II*. ICALP ’08. Reykjavik, Iceland: Springer-Verlag, 2008, pp. 536–547. ISBN: 9783540705826. DOI: 10.1007/978-3-540-70583-3_44. URL: https://doi.org/10.1007/978-3-540-70583-3_44.

[25] Donald E. Knuth. “Two Notes on Notation”. In: *The American Mathematical Monthly* 99.5 (1992), pp. 403–422. ISSN: 00029890, 19300972. URL: http://www.jstor.org/stable/2325085 (visited on 11/19/2024).

[26] Carsten Lund et al. “Algebraic methods for interactive proof systems”. In: *J. ACM* 39.4 (Oct. 1992), pp. 859–868. ISSN: 0004-5411. DOI: 10.1145/146585.146605. URL: https://doi.org/10.1145/146585.146605.

[27] Orr Paradise. “Smooth and Strong PCPs”. In: *Computational Complexity* 30.1 (Jan. 2021). ISSN: 1420-8954. DOI: 10.1007/s00037-020-00199-3. URL: https://link.springer.com/article/10.1007/s00037-020-00199-3.

[28] Walter Rudin. *Principles of Mathematical Analysis*. 3rd ed. McGraw-Hill, 1976. 342 pp. ISBN: 978-0-07-085613-4.

[29] Walter J. Savitch. “Relationships between nondeterministic and deterministic tape complexities”. In: *Journal of Computer and System Sciences* 4.2 (1970), pp. 177–192. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(70)80006-X. URL: https://www.sciencedirect.com/science/article/pii/S002200007080006X.

[30] Adi Shamir. “IP = PSPACE”. In: *J. ACM* 39.4 (Oct. 1992), pp. 869–877. ISSN: 0004-5411. DOI: 10.1145/146585.146609. URL: https://doi.org/10.1145/146585.146609.

[31] Michael Sipser. *Introduction to the Theory of Computation*. 1st ed. International Thomson Publishing, 1996. 396 pp. ISBN: 978-0-534-94728-6. DOI: 10.1145/230514.571645.

[32] Irena Swanson. *Introduction to Analysis with Complex Numbers*. World Scientific Publishing, 2021. ISBN: 978-9811225857.

[33] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of The London Mathematical Society* 42.1 (1936), pp. 230–265. DOI: `10.1112/PLMS/S2-42.1.230`.

Index

Acc, 78
adaptive, 74
algebraic circuit, 85
algebrization, 51
alphabet, 20
alphabet reduction, 90
ambiguous view, 73
arithmetization, 50
assignment
    algebraic circuit, 86

Boole's inequality, 38
Boolean formula, 23
BPP, 29

certificate, 23
CktSAT, 87
CktVal, 88
commitment scheme, 73
communication complexity
    IPCP, 82
complexity class, 20
Cook-Levin theorem, 27

diagonalization, 49
DSPACE, 26
DTIME, 22

event, 37
EXP, 24
expected robustness
    PCP, 79
    PCPP, 79
expected value, 38
extension oracle, 51
extension polynomial, 31

far, 40
field, 30

GI, 64
GNI, 64

Hamming distance, 40

interactive PCP, 81
    zero-knowledge, 84
interactive proof
    multi-prover, 65
    single-prover, 62
IP, 63
IPCP, 82
Iverson bracket, 30

Jensen's inequality, 38
joint computation, 61

Kronecker delta function, 30

$L(X)$, 48
language, 20
locally-computable proof, 117
low, 45
low-degree extension, 32
low-degree IPCP, 83

MIP, 66
multidegree, 31
multilinear, 31
multiquadratic, 31

NEXP, 24
NEXP-complete, 27
NP, 22
NP-complete, 27
NPSPACE, 26

NSPACE, 26
NTIME, 22

O-notation, 21
O3SAT, 24
one-way function, 73
OR, 47
oracle, 43

P, 22
pair language, 77
PCP, 76
PCP theorem, 85
PCP theorem
    zero-knowledge, 128
PCPP, 77
perfect simulator, 68
perfect zero-knowledge, 68
perfect-zero knowledge PCP, 81
poly(n), 21
polynomial summation problem, 103
polynomial time, 22
polynomial-time reduction, 26
PolySim, 124
possible commitment, 73
probabilistically-checkable proof, 75
probability, 37
probability space, 37
PSPACE, 26
PSPACE-complete, 28
PSPACE-robust, 53
PZK, 69
PZK-IPCP, 84
PZK-PCP, 81

query bound, 81
query complexity
    algebraic, 52
    deterministic, 46
    randomized, 47

random variable, 37
RE, 20
recognize, 20
Reed-Muller code, 41
relativization, 45
restriction
    of a vector space, 38
RM, 41
robust verifier, 78
round complexity
    IPCP, 82

SAT, 23
satisfying assignment
    algebraic circuit, 86
Savitch's theorem, 26
security parameter, 64
shared tape, 60
space complexity, 25
statistical independence, 37
Sum, 92
sumcheck problem, 106

time complexity, 21
total degree, 31
Turing machine, 17
    identity, 60
    interactive, 60
    linked pair, 60
    multitape, 19
    nondeterministic, 18
    probabilistic, 19
    with oracle, 44
Turing-recognizable language, 20

verifier, 23
view, 62
    IPCP, 83
    MIP, 68

zero-knowledge proof
    for sumcheck, 108
    perfect, 68