This is a repository copy of *State of the Art Report : Verified Computation*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/202978/

Preprint:

Woodcock, Jim orcid.org/0000-0001-7955-2702, Aranha, Diego F., Hallerstede, Stefan et al. (7 more authors) (2023) State of the Art Report : Verified Computation. [Preprint]

State of the Art Report: Verified Computation

Jim Woodcock,

Mikkel Schimdt Andersen, Diego F. Aranha, Stefan Hallerstede,
Simon Thrane Hansen, Nikolaj Kühne Jakobsen, Tomas Kulik,
Peter Gorm Larsen, Hugo Daniel Macedo, Carlos Ignacio Isasa Martín,
Victor Alexander Mtsimbe Norrild

Aarhus University

31st October 2022

Executive Summary

This report describes the state of the art in verifiable computation. The problem being solved is the following:

> **The Verifiable Computation Problem (VCP)** Suppose we have two computing agents. The first agent is the verifier, and the second agent is the prover. The verifier wants the prover to perform a computation. The verifier sends a description of the computation to the prover. Once the prover has completed the task, the prover returns the output to the verifier. The output will contain proof. The verifier can use this proof to check if the prover computed the output correctly. The check is not required to verify the algorithm used in the computation. Instead, it is a check that the prover computed the output using the computation specified by the verifier. The effort required for the check should be much less than that required to perform the computation.

The problem is visualised in Fig. 1. This is a classic problem with many applications:
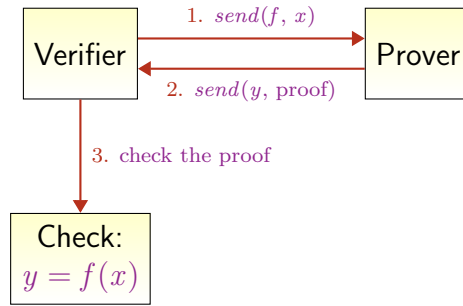

Figure 1: Verifiable Computation.

1. **Delegation** Verifiable computation can be used for delegating computation. Suppose that we have an honest prover that is efficient and runs in polynomial time (that is, the time taken is a simple function of the length of the input). Suppose also that the verifier is super-efficient and checks run in nearly linear time (that is, the time taken is directly proportional to the length of the input). The prover computes a result for the verifier. The prover then interactively proves the correctness of the result. This means the verifier can check the result's correctness nearly linearly instead of running the entire computation. The verifier does not need the computational power possessed by the prover. The prover's resources can be shared between many client verifiers.

2. **Cloud computing** CC Suppose we have a significant distributed computation on petabytes of data. The verifier outsources the computation and its massive dataset to a prover. The prover completes the computation and sends the results back to the verifier. The verifier wants to know that the prover executed the distributed computation correctly. The prover worries about faults that are sources of incorrect execution. These might include data corruption, communication errors, and hardware failures. The prover sends a proof that the results are free from faults.

3. **Information retrieval** A verifier wants to make a query on a remote database. A prover acts as the remote database server. The verifier wants assurance that the prover has performed the query correctly.

4. **Hardware supply chains** Hardware Trojans involve third-party threats of injecting malicious circuits into chip designs. We cannot trust hardware supply chains where this threat exists. For example, an adversarial facility might be manufacturing the hardware under contract. We want a verifier to check assurances provided by the hardware efficiently. This guarantees that the hardware is executing correctly on this input. This is a specific instance of a more general problem: verifying assertions where a third party supplies an untrusted execution substrate.

Although we describe a three-stage protocol in Fig. 1, the proof might be delivered over several rounds of interaction.

The verifiable computation problem complements the program verification problem (PVP). Verification relies on helpful redundancy. We need two descriptions of the same thing and then compare one against the other. Program verification establishes that we have expressed a given computation correctly. We make the judgement by comparing it with a higher-level specification. In the verifiable computation problem, the computation f is given. We are not verifying f against a specification. Instead, we want to know whether the execution performed by the prover is consistent with the expression of f.

The literature surveyed in this state-of-the-art report presents the theory of probabilistic proofs. A central result in this area is the Probabilistically Checkable Proof Theorem (PCPT). PCP has a necessary consequence. For any valid mathematical assertion, it is possible to encode the proof of that assertion. PCP shows that we can use this encoding to check the assertion's validity by inspecting only a constant number of points in the proof carried out elsewhere.

The practical consequence of PCP is in its application to the protocol in Fig. 1. Consider the computation f, input x, and supposed output y. There is a proof and a randomised way of inspecting it that guarantees the following. The verifier will accept the proof if $y = f(x)$ is correct. If $y \neq f(x)$, the verifier will almost always reject the proof. The proof might need interaction between the prover and the verifier. The fact that the verifier rejects such proofs *almost always* encodes an error bound. It means that, with some probability bounded in the analysis, the verifier will incorrectly view a wrong answer as correct.

The verifiable computation protocol in Fig. 1 does not explicitly check the result y. It does less work than that. If the verifier were to check the result $y = f(x)$, then it would need to re-do the computation. That contradicts the problem statement and is not the intention.

So, PCPs allow a randomised verifier, with access to a purported proof, to probabilistically verify an input statement of the form $y = f(x)$ by querying only a few proof bits. Zero-Knowledge PCPs (ZK-PCPs) enhance standard PCPs. In a zero-knowledge proof (ZK), one party can prove to another that a given statement is true. It does this while avoiding giving any additional information apart from the fact that the statement is indeed true.

There is a large body of literature devoted to probabilistically checkable proof protocols. The original naive implementations of the PCP theory were very slow. Since then, there have been orders of magnitude improvements in performance. Early tools used low-level representations of computations. Newer tools compile programs in a high-level language for these low-level protocol entities. Some publications report efficient verifiers that could tackle real-world problems. But it appears that these systems are limited to smaller executions, mainly due to the expense of the prover. Our initial impression is that these systems are limited to special-purpose applications.

This state-of-the-art report surveys 128 papers from the literature comprising more than 4,000 pages. Other papers and books were surveyed but were omitted. The papers surveyed were overwhelmingly mathematical. We have summarised the major concepts that form the foundations for verifiable computation. The report contains two main sections. The first, larger section covers the theoretical foundations for probabilistically checkable and zero-knowledge proofs. The second section contains a description of the current practice in verifiable compu-

tation. Two further reports will cover (i) military applications of verifiable computation and (ii) a collection of technical demonstrators. The first of these is intended to be read by those who want to know what applications are enabled by the current state of the art in verifiable computation. The second is for those who want to see practical tools and conduct experiments themselves.

Contents

Acronyms

AIR Algebraic Intermediate Representation

ARM Advanced RISC Machine

BCS Ben-Sasson-Chiesa-Spooner

BFS Byzantine Fault Tolerant NFS File System

BFT Byzantine Fault Tolerance

FFT Fast Fourier Transform

FHE Fully Homomorphic Encryption

FRI Fast Reed-Solomon Interactive Oracle Proof of Proximity

IOP Interactive Oracle Protocol

IOPP Interactive Oracle Protocol of Proximity

IP Interactive Proof

LPCP Linear Probabilistically Checkable Proof

MIP Multi-Prover Interactive Proof

NFS Network File System

NP Nondeterministic Polynomial time

PCP Probabilistically Checkable Proof

PCPP Probabilistically Checkable Proof of Proximity

PLONK Permutations over Lagrange-bases for Oecumenical Noninteractive Arguments of Knowledge

PSPACE Polynomial Space

QAP Quadratic Arithmetic Programs

R1CS Rank-1 Constraint System

RCAS Reduced Cyber-Attack Surface

RISC-V Reduced Instruction Set Computer (RISC) 5

ROM Random Oracle Model

RS Reed-Solomon

RSA Rivest–Shamir–Adleman

SAT Circuit/Boolean Satisfiability

SLA Service Level Agreements

SNARG Succinct Non-Interactive Argument

SNARK Succinct Non-Interactive Argument of Knowledge

STARK Zero-Knowledge Scalable Transparent Argument of Knowledge

TLS Transport Layer Security

TPM Trusted Platform Modules

UAV Unmanned Air Vehicle

VCP Verifiable Computation Problem

ZK Zero Knowledge

Glossary

AM Arthur-Merlin. The set of decision problems that can be decided in polynomial time by an Arthur–Merlin protocol

BPP Bounded-error Probabilistic Polynomial-time . The class of decision problems that probabilistic Turing machines can solve in polynomial time with an error probability bounded by ⅓ for all instances of the problem

CC Cloud Computing. The delivery over the internet of computing services, including data storage and computing power, on-demand with pay-as-you-go pricing (definition from NIST [90]; other interpretations exist)

CRH Collision-Resistant Hash function . A hash function with a very low probability of two different inputs hashing to the same output

DTIME Deterministic Time. Referring to a deterministic Turing machine's computational resource or computation time. Also TIME

Entropy Entropy of a random variable. See Shannon entropy

FF Finite Field. $\mathbb{F}$: A field that contains a finite number of elements. A field is a set with four basic operations: addition, subtraction, multiplication, and division, satisfying the rules of arithmetic. An example of a finite field is the integers modulo p, where p is a prime number

FME Fully Homomorphic Encryption. An encryption scheme that enables computations to be run directly on data encrypted by the scheme without first decrypting it

HWT Hardware Token. A peripheral hardware device used to provide access to a protected or restricted electronic source

IPT Interactive Polynomial Time. The class of problems solvable by an Interactive Proof system in polynomial time

IP Interactive Proof/Interactive Proof System. An abstract machine that models proof as an exchange of messages between a possibly untrustworthy Prover and an honest Verifier

MT Merkel Tree. A data structure in the form of a tree usually with a branching factor of 2 in which each internal node has a hash of all the information in its child nodes

MIP Multi-prover Interactive Proof. An Interactive proof system distributed across multiple provers

NAV Non-Adaptive Verifiers. A PCP proof in which all the verifier queries are predefined

NP Nondeterministic Polynomial-time. The class of decision problems that have proofs which are verifiable in polynomial time by nondeterministic Turing machines

Oracle An abstract machine used in complexity theory and computability to study decision problems. Equivalent to a black box addition to a Turing machine that can solve certain problems in a single step

PCP Probabilistically Checkable Proof. A proof statement that can be checked using a randomised verification algorithm to within a high probability of accuracy by examining only a bounded number of letters of the proof

PCPT Probabilistically Checkable Proof Theorem. A theorem stating that each decision problem of NP complexity can be rewritten as a probabilistically checkable proof

PVP Program Verification Problem. The problem of verifying that a computer program always achieves the intended result as given by a higher level specification

PSPACE Polynomial Space. The class of all decision problems that can be solved by a deterministic Turing machine using space which is polynomial in the size of the input

PTM Probabilistic Turing Machine. A nondeterministic Turing machine that uses a probability distribution to decide between alternative transitions

RO Random or Randomised Oracle/Random Oracle model. An oracle that responds to every unique query with a random response chosen uniformly from its output domain. The response to any particular query will be the same each time the query is submitted

SH Shannon entropy. The Shannon entropy of a given stochastic source is the average rate at which the source produces information. The higher the Shannon entropy, the bigger the information gained from a new value in the process

SNARG Succinct Non-interactive Argument. A proof construction that satisfies succinctness, meaning that the proof size is asymptotically smaller than the statement size and the witness size

TM Turing Machine. An abstract computing device which provides a model for reasoning about computability and its limits

VCP Verifiable Computing Problem. A computational task problem involving two agents, a relatively weak machine called the Verifier (or Client), and a more powerful Prover (or Worker). The problem involves the Verifier delegating computational tasks to the Prover. In return, the Verifier can expect the result of the task plus a proof by which it can verify the result with less computational effort than would be needed to perform the task from scratch

ZK Zero-Knowledge Proof. A proof construction in which the Prover can prove to another verifier that a statement is true without the Prover having to impart any information apart from the fact that the statement is true

ZK-PCP Zero-Knowledge PCP. A PCP with an additional Zero-Knowledge guarantee between the Prover and the Verifier

ZK-SNARK Zero-Knowledge Succinct Non-Interactive Argument of Knowledge. A Zero-Knowledge proof construction which does not involve any interaction between the Prover and Verifier

1 Introduction & Overview

In this report, we review the state of the art in verified computation:

> *How can a single computer check computations carried out by other computers with untrusted software and hardware?*

We start by putting research in verified computation into context: where would it be useful?[1] A typical application is in trusted cloud services. The use of cloud computing is now widespread. The cloud is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [90]. Cloud providers offer computational power and data storage facilities that significantly extend the capabilities of weaker devices. Clouds are large, complex systems that are usually provided without convincing reasons why we should trust them. In practice, they suffer from software errors, configuration issues, data corruption, hardware problems, and malicious actors. As Walfish and Blum [122] point out, this raises two important research questions: (i) How can we trust results computed by a third party? (ii) How can we assure data integrity stored by a third party? We consider several answers to these questions, suggested by Walfish and Blum [122]. The answers are provided by replication, trusted hardware, and remote attestation.

Replication The most obvious answer to both questions is replicating computations and data over several cloud servers. For example, Canetti et al. [35] discuss practical delegation of computation using multiple servers as an efficient and general solution to the problem. Their protocol guarantees the correct answer produced by a replicated, efficiently computable function. The guarantee relies on at least one honest server, but we do not know which is the honest server and which is the right answer. The protocol requires logarithmically many rounds based on any collision-resistant hash CRH family. (Collision resistance means that it is hard to find two inputs that hash to the same output.) The protocol uses Turing Machines (TMs) but can be adapted to other computational models. The protocol must be deterministic. This requires, in turn, deterministic versions of system calls, such as `malloc()` and `free()`. The construction is not based on probabilistically checkable proofs (PCP, see Sect. 3.3) or fully homomorphic encryption (FHE FME, see Gentry [61]). Canetti et al.'s protocol, does not rely on trusted hardware, nor does it require a complex transformation of a Turing Machine program to a boolean circuit. The faults that must be guarded against in replicated computation are Byzantine (see Lamport et al. [83]): they leave imperfect evidence of the fault's occurrence. Castro and Liskov [36] propose a novel replication algorithm, BFT, designed to handle Byzantine faults. BFT is used to implement real services, performs well, is safe in asynchronous environments like the Internet, incorporates mechanisms to defend against Byzantine-faulty clients, and can recover replicas. The recovery mechanism tolerates any number of faults over the system's lifetime, provided fewer than one-third of the replicas become faulty within a short time interval. Their Byzantine-fault-tolerant NFS file system,[2] BFS has a 24% performance penalty compared with production implementations of the NFS protocol without replication. A major drawback of replication algorithms is that they assume failures are unrelated. This assumption is invalid for cloud services, where hardware and software platforms are often homogeneous.

---

[1] A future deliverable in the RCAS project describes some example military applications.

[2] Network File System (NFS) is a distributed file system protocol originally developed by Sun Microsystems in 1984, allowing a user on a client computer to access files over a computer network much like local storage is accessed.

Trusted hardware   Another solution is to use trusted hardware under the cloud provider's control. Sadeghi et al. [106] show how to combine a trusted hardware token (HWT) with secure function evaluation to compute arbitrary functions on encrypted data where the computation leaks no information and is verifiable. Their work reduces the computation latency usually experienced by pure cryptographic solutions based on fully homomorphic and verifiable encryption.

Remote attestation   Yet another solution is remote attestation (see Feng [53]), where one system makes reliable statements about the software it is running on another system. The remote system can then make authorisation decisions based on that information. Many remote attestation schemes have been proposed for various computer architectures, including Intel, RISC-V, and ARM.

Walfish and Blum [122] give an overview of an alternative to replication, trusted hardware, and attestation: a technology where the cloud (or some other third party) returns the results of the computation with proof that the results have been computed correctly. This technology aims to make this proof inexpensive to check compared to the cost of redoing the computation. If this technology is feasible, it would not need the assumptions about faults required for replication, trusted hardware, or remote attestation. Either the proof is valid, or it is not. Walfish and Blum call this *proof-based verifiable computation.* They give four insights into the practicality of this research area:

1. Researchers have already built systems for a local computer to check the correctness of a remote execution efficiently. For a comprehensive discussion of the state-of-the-art and tool implementations, see Sect. 4.

2. There are diverse applications, such as computational complexity, cryptography and cryptocurrencies, secure exploitation of untrusted hardware supply chains, and trustworthy cloud computing. Additional applications, not suggested by Walfish and Blum, include computationally enhanced UAV swarms and automated reviewing systems for mathematical journals and conferences.

3. Important foundations include probabilistically checkable proofs (PCPs) [5, 6, 113], interactive proof systems [8, 69, 86], and argument systems [29], all of which have a sound and well-understood mathematical basis.

4. Engineering these theories for practical application is a new interdisciplinary research area.

We describe the underlying theories in detail in later sections of this report, but for now, here is an informal motivating example due to Mordechai Rorvig and published in *Quanta* magazine [105]:

> If a million computer scientists had dinner together, they'd rack up an enormous bill. And if one of them were feeling particularly thrifty and wanted to check if the bill was correct, the process would be straightforward, if tedious: they'd have to go through the bill and add everything up, one line at a time, to make sure that the sum was equal to the stated total.
>
> But in 1992, six computer scientists proved in two papers [both by Arora and his colleagues [6, 5]] that a radical shortcut was possible. There's always a way to reformat a bill—of any length—so that it can be checked with just a few queries. More importantly, they found that this is true for any computation or even any mathematical proof since both come with their own receipt, so to speak: a record of steps that a computer or a mathematician must take.

We give a very brief overview of PCP; more details are in Sect. 3.3. A PCP for a language consists of a probabilistic polynomial-time verifier with direct access to individual bits of a bit-string. This string (which acts as an Oracle) represents a proof and will be only partially accessed by the verifier. Queries to the oracle are locations on the bit-string and will be determined by the verifier's input and coin tosses (potentially, they might be determined by answers to previous queries). The verifier must decide whether a given input belongs to the language. The verifier will always accept an input that belongs to the language, given access to the oracle (the bit string). On the other hand, if the input does not belong to the language, then the verifier will reject with probability at least ½, no matter which oracle is used. One can view PCP systems in terms of interactive proof systems (IP). The oracle string is the proof, and the queries are the messages sent by the verifier. The prover is memoryless and cannot adjust answers based on previous queries.

The PCP theorem is important. Scheideler [107] observes that Arora's original proof of the PCP theorem [5] is one of the most complicated proofs in the theory of computation. It has been described by Wegener as the most important result in complexity theory since Cook's theorem [124][3] And by Goldreich as a culmination of impressive research rich in innovative ideas [65]. Boneh (quoted in Rorvig's *Quanta* article [105]) considers it very rare that such deep algebraic tools from mathematics have made it into practice.

Both the PCP theorem and its proof have been simplified. Arora et al.'s original proof was dramatically simplified by Dinur's PCP construction [48]. Radhakrishnan and Sudan [103] give an accessible commentary on Dinur's proof. Zimand [127] presents a weaker variant of the PCP Theorem that has a correspondingly simpler proof compared to Arora et al. In Zimand's simplification, the prover has only a limited time to compute each bit of the answer, in contrast to the original prover being all-powerful. Song's account of the theoretical setting for the PCP theorem [112] contains a simplified version of the theorem and an accompanying proof. The simplified theorem states that every NP statement with an exponentially long proof can be locally tested by looking at a constant number of bits. This is weaker than the original PCP theorem since the proofs validated by the original theorem may be much more significant. In Arora et al.'s work, the PCP verifier deals with proofs of polynomial size, whereas in Song's weaker theorem, the verifier deals with proofs of exponential size. Despite this, it is interesting that exponentially sized proofs can be verified by a constant number of queries. Ben-Sasson [13] presents two variants of the PCP theorem. The first achieves a nearly optimal trade-off between the amount of information read from the proof and the certainty of the proof. If the verifier is willing to tolerate an error probability of 2^{-k}, it suffices to inspect $O(k)$ bits of the proof. The second variant is very efficient in terms of the length of the proof.

1.1 Organisation of the Report

The report consists of the following sections.

1. The Executive Summary at the beginning of the report describes the Verifiable Computation Problem. It outlines several applications and gives a very high overview of the main foundation: the PCP theorem. A lecture given by Walfish inspires part of this description.[4] The example of delegating computations is taken from Goldwasser et al. [70].

---

[3]The Cook–Levin theorem, also known as Cook's theorem, proves that SAT, the Boolean satisfiability problem, is NP-complete. A problem is in NP when the correctness of each solution can be verified in polynomial time, and solutions can be found using brute-force search (formalised using a nondeterministic Turing machine). A problem is NP-complete if it can be used to simulate every other problem for which we can verify polynomially that a solution is correct. So Cook's theorem states that SAT is in the complexity class NP and any problem in NP can be reduced in polynomial time by a deterministic Turing machine to SAT.

[4]"Introduction and Overview of Verifiable Computation", a lecture by Prof. Michael Walfish during

2. In Sect.2, we focus on a statement of the problem we are trying to solve with verifiable computation.

3. The main part of this report is Sect. 3, where we comprehensively introduce the theory of probabilistically checkable and zero-knowledge proofs. We set the scene in Sect. 3.1 by describing security models for two-party computation. We then review interactive proof systems (Sect. 3.2), probabilistically checkable proofs (Sect. 3.3), and polynomial interactive oracle proofs (Sect. 3.4). In Sect. 3.5, we discuss verifiable computation as a set of polynomial constraints that can be verified efficiently. Zero-knowledge proof protocols are used to prove that a party possesses certain information without revealing it and without any interaction between the parties proving and verifying it. We discuss such protocols in Sects 3.6 and 3.7. Finally, in Sect. 3.8, we review a game theoretic approach.

4. The actual state of the art lies in the practice of verifiable computations embodied in usable tools. Usability includes the following considerations: the expressiveness of the language describing the computation; the efficiency of creating (and checking) the proof; and the (e.g., cryptographic) libraries on which the implementation relies. Addressing these aspects of usability poses different challenges. We review the landscape of implementations in Sect. 4. This is before the detailed report on the performance of practical tools on demonstrators and benchmarks.

5. We conclude the report in Sect. 5 with a high-level summary of the work.

6. The final section contains an extensive bibliography of papers and books on verifiable computation, all cited in the report.

2 Problem Statement

A verifier sends the specification of a computation P and input x to a prover. The specification of the computation might be a program text. The prover computes an output y and sends it back to the verifier V that either accepts y with $V(y) = 1$ or rejects y with $V(y) = 0$. If $y = P(x)$, then a correct prover should convince the verifier of this. The prover might answer some questions from the verifier or provide a certificate of correctness. If $y \neq P(x)$, the verifier should reject y with a certain probability. The problem is to provide a protocol to carry out this procedure. The protocol is subject to three requirements.

1. The verifier must benefit from using the protocol. It might be cheaper for the verifier to follow the protocol than to directly compute $P(x)$. The prover might be able to handle computations that the verifier cannot. It might have access to data inaccessible to the verifier.

2. We do not assume that the prover follows the protocol.

3. P should be general. We assume that the length of the input statically bounds the prover's running time.

Example. Suppose a device D needs to compute the matrix product $f(X) = X^T * X$ and D does not have the computation resources. Further, suppose another device S has those resources. Let us allocate the f computation to S. Now we have the following scenario: D sends the pair (f, A) to S, which computes $f(A)$ and returns the result B to D.

Instead of the trusted device S that computes f, device D could also rely on an untrusted, possibly adversarial, device S' to compute B'. Now, D needs to determine whether B' equals the expected B. To ensure this, D requires that S' sends a proof π along with the result B' so that it now receives a tuple (B', π). On reception, device D verifies π is a proof of $B' = f(A)$. Our main interest focuses on the two roles S' and D play in the interaction: device S' is tasked with producing proof and D with verifying it. We emphasise this by referring to S' as the *prover* and D as the *verifier*.

The protocol described in the preceding paragraph has a severe problem: the proof π can be prohibitively long, rendering the protocol infeasible. For the protocol to become practical, the proof needs to be shortened. Looking at only an excerpt of the proof π', we are no longer confident that $B' = f(A)$. We can obtain this result only up to some probability: $\exists\, \pi' \cdot Pr^{\pi'}[B' = f(A)] \geq c$. The value c is called *completeness*. Of course, there is also a probability that $B' \neq f(A)$. We can estimate this probability with a bound over all corresponding proofs $\forall\, \pi' \cdot Pr^{\pi'}[B' \neq f(A)] \leq s$. The value s is called *soundness*.

The challenge addressed by PCP is how to keep the shortened proofs π' to be considered as small as possible while maximising c and minimising s.

3 Theoretical Foundations

3.1 Security models for two-party computation

Security is essential for multiparty and outsourced computation. There are many strategies to outsource computation securely. A natural approach is to rely on cryptography, which provides strong security guarantees from formal analysis grounded in time-tested hardness assumptions. Techniques not depending on cryptography can be more convenient or efficient in some contexts, but they might leave room for attacks if the incentives are large enough. We introduce several concepts: trust, verifiability, and secure communication. We discuss some of the foundational works in provable outsourced computation. Our focus is on two-party computation. Here, one entity is the client with a computational problem. The other entity (a cloud server) executes the computation and provides results to the client.

Defining security requires an adversary model against which the security properties of a system must be enforced. Different types of adversaries capture the other threat models for two-party computation. In general, there are three adversary models:

- *Semi-honest adversaries* follow the protocol specification but try to learn from a protocol execution additional information held by other parties. This adversary model is also called honest-but-curious and limits itself to confidentiality properties.

- *Malicious adversaries* can deviate arbitrarily from the protocol specification. Protecting against these more powerful adversaries involves threats to integrity and authentication.

- *Covert adversaries* do not necessarily follow the protocol specification but try not to be caught deviating from it. This model is a middle-ground between the previous ones and captures realistic incentive structures supporting malicious behaviour.

The types of adversaries, non-adversarial (honest) parties, and the notion of honesty are further discussed in Section 3.1.3.

3.1.1 Trust in Multiparty Systems

Trust is one of the top considerations when choosing a cloud provider for computation outsourcing. There are several definitions of trust in cloud computing [76, 84, 100]. We define trust as

> *An expectation that the cloud carries out the computation without any malicious intent, leakage of information, or purposefully returning wrong results.*

Guaranteeing such a trusted relationship is complex. We present several ways trust is often established.

A common way of determining trust is by considering the cybersecurity standard [41] that the cloud provider follows. This might involve a third party's accreditation of the cloud provider based on periodic assessments. While this approach is often considered acceptable, there is no guarantee that the notion of trust holds during the execution of the client's request. In a particularly sensitive computation, such as critical utility, military, or healthcare use, cloud operators might be tempted by financial, ideological, or other incentives to temporarily violate trust.

Another approach considers service-level agreements (SLAs) [1] for implementing requested cloud services. This enables the client to monitor specific parameters defined within the SLA. These parameters could be the type of a machine that carries out the computation, versions of operating systems, used execution frameworks, etc. The client is informed about any changes to the agreed parameters, or the client might monitor the parameters. The cloud provider might be capable of spoofing some of the SLA parameters, or if the monitoring is random, the client might take a chance and relax the parameters. This could break the trust between the client and the cloud.

Another approach is reputation [25] using criteria defined by the client (often based on an SLA). The client then monitors these criteria over time, analysing feedback from the client and third parties. If the criteria metric (the cloud provider's reputation) drops below some threshold, the cloud provider is no longer considered trusted. One of the challenges of this approach is the possibility of the cloud provider utilising malicious third parties to increase its reputation. A server with an honestly gained reputation may still break trust if the reward is significant enough.

These challenges have led to specialised cloud deployments. These include private clouds [55], where the client's organisation is responsible for operating the cloud platform. There may be dedicated clusters with specific security hardware, such as custom Trusted Platform Modules (TPMs) [7]. Both examples are expensive for the client, who is responsible for platform management, and in the latter case, also hardware components.

Whilst trust might be established, it cannot be fully guaranteed if the stakes for the outsourced computation are high enough. This also applies to zero trust within cloud environments [88]. As well as cloud components not trusting external clients and devices, they may not trust each other. All access uses gateways handling authentication and authorisation. The zero-trust frameworks pose a barrier to malicious entities, including the cloud provider. But they do not guarantee the correctness of the result of the outsourced computation. This report provides an overview of techniques that provide proof of the correctness of computation and could be applied to cloud platforms without any prior establishment of trust.

3.1.2 Verifiability in Multiparty Systems

A potential solution to outsourced computation is verifiable computation [56]. The client dispatches the computational problem to the cloud and receives the result and proof of correctness of the result. Different approaches can be used with varying degrees of complexity. While the

technical part of this report primarily discusses probabilistic and cryptographic methods. As well as receiving a result with the proof, the data and the computed problem stay hidden from the cloud. We discuss several other approaches in this section.

The first approach is to provide *verifiability by multiple executions* [9, 37]. In this approach, the same problem is distributed to various executors (cloud providers), executed, and then the results are collected and compared. This approach poses several challenges. First, it requires multiple cloud providers to be able to compute the given problem. Second, in the case of using only two cloud providers, it is not possible to determine which cloud provides the correct result (if any) without independent re-execution. Finally, the approach does not offer the ability to detect possible collusion or a state when two cloud providers provide purposefully incorrect results that are equal without utilising more than two parties (an approach based on game theory has been proposed to resolve this and is discussed later in this report (Sect. 3.8)).

Another approach is *certifying the results by secure hardware elements* [99]. In this approach, the execution is carried out with trust bootstrapped using cryptography hardware such as TPM. Any computation result is signed by keys stored within this TPM, where the client can verify this signature. In this instance, there are several challenges. The first challenge is the inability to verify the result itself (without re-execution), using only its signature. Another, perhaps more critical challenge is that trust in the physical components, such as TPMs, needs to be guaranteed, as we discussed in Sect. 3.1.1; this is often difficult, and if there are high stakes, computing might provide a false sense of security.

Approaches based on cryptography and probability theory, such as interactive proof systems (Sect. 3.2), probabilistically checkable proofs (Sect. 3.3), interactive oracle proofs (Sect. 3.4), ZK-SNARK (Sect. 3.6), ZK-STARK (Sect. 3.7), and finally a game theory based approach (Section 3.8).

3.1.3 Honesty within Multiparty Computation

Within the principles of secure multiparty computation, several considerations exist for the parties' honesty. A frequent concern is the so-called *honest majority*, where most parties are assumed to have honest intentions. They do not deviate from the protocol or gather secret information. In this case, the honest majority can also be utilised to remove a dishonest party from the computation [63].

In the *semi-honest adversaries* case, the party does not try to deviate from the protocol specification. Instead, it tries to gather more information from the protocol execution than is allowed. This could be by studying the protocol trace or messages exchanged during protocol execution [58] and trying to compute additional information. Of course, there could be several semi-honest and honest parties.

In the worst-case scenario (the *malicious adversaries* case), the party would willingly use any attack vector to deviate from the protocol. This deviation could be changing the inputs and outputs of the protocol, as well as aborting the protocol at any time. Security can still be achieved, especially with an honest majority. The efficiency of secure multiparty computation with malicious parties still requires improvements to reach everyday practicality [58, 3]. The malicious party could act overtly, i.e., not trying to avoid detection or a covert (*covert adversaries case*) way, making it more difficult to determine that the party is indeed malicious as the honest majority might not have enough information to support the conclusion.

An important notion is the number of corrupted parties, which may be semi-honest or malicious. The most common considerations used within the research of protocols for secure multiparty computation are $t < n$, stating that the number of corrupted parties t could reach any number of parties within the computing environment, but less than n: the total number of parties involved within the multiparty computation. The other common model is $t < \frac{n}{2}$, where

there is an honest majority. A more robust model has a two-thirds honest majority: $t < 1/3$. In many protocol cases, an honest majority is a requirement [38]. However, protocols are aimed explicitly at cases with a dishonest majority [80]. In both cases, current research is focused on the efficiency of these protocols.

3.1.4 Secure Communication - a brief note

Secure communication is an essential part of any data exchange. This becomes especially important in cases where the data is being sent over the network to remote parties. A typical approach is to ensure that the traffic utilises network encryption; this could be by using public key cryptography such as the TLS protocol. While the context of this report is outsourcing high-stakes computation to potentially untrusted parties, some primary considerations shall be met. It could be expected that a remote party provides a valid TLS certificate to continue outsourcing the computation. This means that the certificate shall be signed by a trusted certificate authority, limiting the remote entity's potential to fake its identity. The TLS protocol suite supported by the remote entity should also contain support for recent cypher suites and consider the use of cryptography algorithms for post-quantum TLS [32]. This needs to be scrutinised every time the connection is established, requiring the client to drop support for older cypher suites as they become outdated, forcing the communication to utilise only the newer ones. While there could be other protocols based on custom encryption schemes, the approaches specified in the report can be compatible with a data exchange layer based on TLS.

3.2 Interactive Proof Systems

Our study of verified computation and proof systems starts with a precise definition of proof.

An alphabet Σ is typically expressed as field $\mathbb{F}$, which defines a context for us to define objects. The set Σ^* defines all finite sequences of elements in Σ, including the empty sequence denoted by ϵ. Elements in Σ^* define statements and proofs. The validity of a proof w, for a statement x, depends on whether (x, w) belongs to the relation $R \subseteq \Sigma^* \times \Sigma^*$. We define a language L_R to be the set of provable statements over an existing relation R:

$$L_R := \{(x, w) \in \Sigma^* \times \Sigma^* \mid (x, w) \in R\}$$

Complexity classes divide languages into different categories depending on specific properties. A language L belongs to the complexity class **NP** (the Nondeterministic Polynomial time complexity class) iff there exists a nondeterministic Turing machine (which we in our context call the verifier) V_L, that given access to an instance x of length n and a proof π of size $poly(n)$ either rejects $V_L(x, \pi) = 0$ or accepts $V_L(x, \pi) = 1$ the claim $x \in L$ with perfect soundness and completeness.

- **Perfect Completeness**: $x \in L \Rightarrow \exists \pi[V_L(x, \pi) = 1]$ (i.e., there exists a proof for all correct claims that will make the verifier accept).

- **Perfect Soundness**: $x \notin L \Rightarrow \forall \pi[V_L(x, \pi) = 0]$ (i.e., there exists no proof that can convince the verifier to accept a false claim).

A famous instance of **NP** is circuit/Boolean satisfiability (or simply SAT) that considers a Boolean circuit formula C, where we want to check if w is a satisfying assignment $C(w) = 1$. An attractive property of the Cook–Levin theorem [43] is that any **NP**-problem can be reduced in polynomial time by a deterministic Turing machine to SAT.

Classical proofs require the verifier to read the entire proof π to be convinced of a given claim $x \in L$, which in many situations becomes unattractive for a weak computational client.

A well-studied question has therefore been whether we can get away with having the verifier read fewer bits.

In the context of verifiable computation, we restrict ourselves to interactive proof systems (IP) between two parties. Interactive proof systems were introduced in [8, 69] for a function $f : \Delta \to \tau$ that is a protocol that allows a probabilistic *Verifier* to interact with a deterministic *Prover* to determine the validity of a statement $y = f(x)$ on a common input x based on an interaction of k rounds (by exchanging $2k$ messages). The *Verifier* and the *Prover* interact using a sequence of messages $t = (m_1, \pi_1, \ldots m_k, \pi_k)$, called the *transcript*. In the i-th round of interaction, the Verifier challenges the Prover by sending a uniformly random message m_i to the Prover. The Prover replies to the Verifier with a message π_i. The Verifier is probabilistic, meaning that any challenges (messages) m_i sent by the Verifier may depend on some internal randomness r and previous messages $m_0, \ldots, m_{i-1}$. The behaviour of the Verifier can be either adaptive or non-adaptive. The Verifier is non-adaptive (NAV) if its queries (challenges) depend only on the Verifier's inputs and its randomness. An adaptive verifier selects its challenges based on previous messages. At the end of the protocol, the *Verifier* either accepts or rejects the statement $y = f(x)$ based on the transcript and its internal randomness.

Definition 3.1 (Interactive Proof). Let $L \in \Sigma^*$ be any language. We say that (P, V) is an interactive proof system for the language L if there exists a k-message protocol between a polynomial-time verifier V and an unbounded prover P satisfying the following two properties:

- Completeness[5]: $Pr[\langle P, V_r \rangle (x) = 1 \mid x \in L] \geq \frac{2}{3}$

- Soundness against all unbounded malicious provers P^*: $Pr[\langle P^*, V_r \rangle (x) = 1 \mid x \notin L] \leq \frac{1}{3}$

The notation $\langle -, - \rangle$ denotes interaction, and the subscript notation denotes the number of times the verifier draws a random bit (or *coin*) to use during the verification. Figure 2 illustrates the interaction between a prover and a verifier. Instead of computing $C(x)$, a malicious prover could try to guess the proof for an invalid computation. The soundness condition expresses how hard it is to "find" an invalid proof that would pass the test by the verifier despite being invalid. For reasons of complexity, "finding" such a proof could only be achieved by resorting to randomised guessing. An example of an interactive proof system can be found in Sect. 3.2.1.

There exist two different branches of interactive proof systems: *public-coin* where the random choices (coin tosses) of the *Verifier* are made public, and *private-coin* where the random choices are kept secret. Any private-coin protocol can be transformed into a public-coin protocol using the technique from [71]. A public-coin interactive proof is often called an Arthur–Merlin AM game.

The complexity class *IP* denotes all languages for which an interactive proof system exists. A fundamental measure of the efficiency of an interactive proof protocol is the round complexity k. As illustrated in Fig. 2, the round complexity counts the number of interactions between the prover and verifier.

It was proved in 1992 by Shamir [111] that every language in **PSPACE** has an interactive proof system (**IP** = *PSPACE*) that extended the work in [86]. Argument systems are a relaxation of interactive proof systems, where soundness is reduced to *computational soundness*, meaning that soundness is only required to hold against a computationally bounded prover running in polynomial time [123].

Definition 3.2 (Argument System). An argument system (P, V) is defined similarly to an interactive proof system (P, V), with the following differences:

---

[5]The constants like $\frac{2}{3}$ used in the definition of IP stem from the original theoretical work. They are also sufficient for proving the main PCP results. For the later, more applied work on PCP improved bounds have been derived.
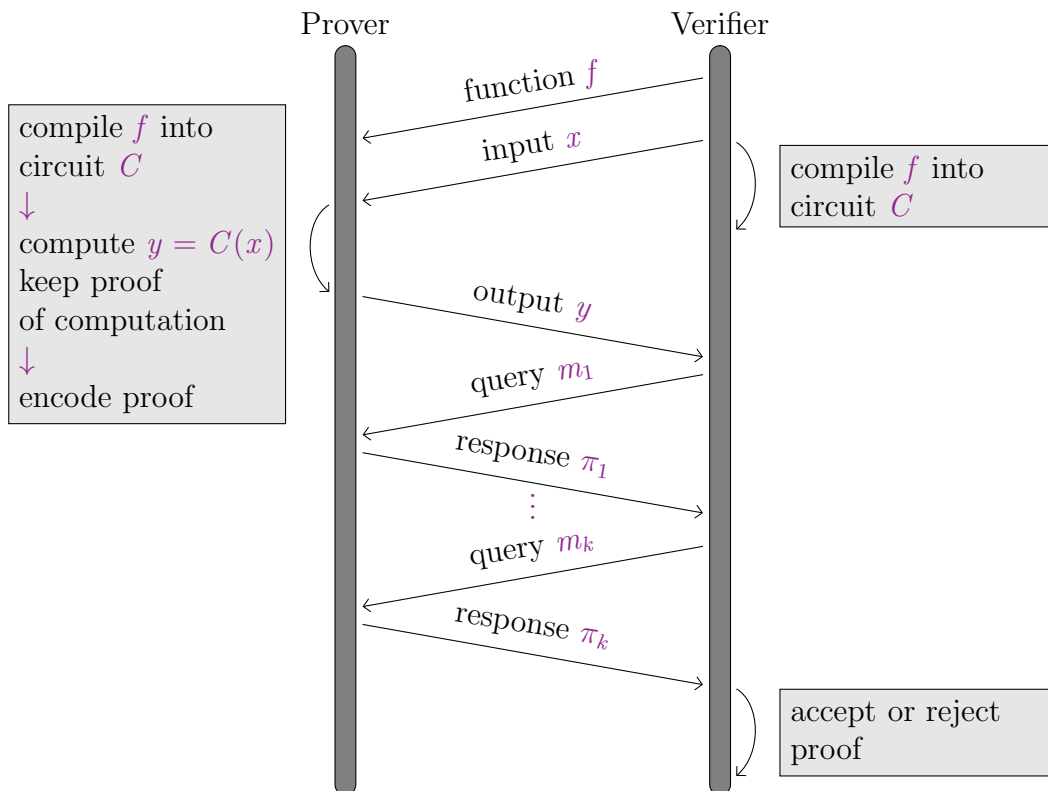
Figure 2: Interaction between prover and verifier

- The soundness condition is replaced by computational soundness: For every probabilistic polynomial time machine P^*, for all sufficiently long $x \notin L$, the verifier V rejects with probability at least ½.

Argument systems were introduced by Brassard et al. in 1986 [29] to obtain perfect zero-knowledge protocols (see Section 3.2.3) for **NP**. Kilian uses the relaxation to build a constant-round protocol with low communication complexity [81]. Limiting the prover's computational power seems necessary to attain soundness of argument systems [57, 66]. Argument systems use cryptographic primitives and other assumptions, which a super-polynomial time prover can break. Such a prover might still be "efficient enough" to generate invalid proof that might pass the test.[6] Cryptographic primitives have been used to improve the situation and achieve additional desirable properties that are unattainable for interactive proofs, such as re-usability (i.e., the ability for the verifier to reuse the same "secret state" to outsource many computations on the same input), and public verifiability. Some of this work might be relevant when studying specific properties of PCP. A general insight here is that some properties of interest require strong assumptions on provers that limit the usefulness of protocols supporting those properties.

3.2.1 The Sum-check Protocol

The Sum-Check protocol[7] is an interactive proof system introduced by Lund, Fortnow, Karloff, and Nisan in [86]. The sum-check protocol is used in a brief account of the original proof of the PCP theorem in Subsection 3.3.1. The protocol approach is very similar to how the classic

---

[6]Super-polynomial time can be "just" not polynomial. For instance, the Adleman–Pomerance–Rumely primality test has super-polynomial time complexity $(\log n)^{O(\log \log \log n)}$ which gets dominated by a polynomial only for very large n.

[7]See the tutorial at `semiotic.ai/articles/sumcheck-tutorial/` on an excellent account of the sum-check protocol.

television detective questions a suspect to detect whether they are lying. The detective starts by asking for the whole story before digging into more minor details to look for contradictions. The detective carefully selects the questions so that each communication round restricts the range of future valid answers; this means that a lying witness will eventually be caught in a contradiction. In the following, let $\mathbb{F}$ be a finite field FF.

The Verifier performs a similar interaction with the Prover. This is in the original setting of the sum-check protocol encoding the problem of interest as a sum-checking problem of values of a low-degree multivariate polynomial on an exponentially large hypercube.

The Prover takes as input an m-variate polynomial $g : \mathbb{F}^m \to \mathbb{F}$ of degree $\leq d$ in each variable, where $d \ll |\mathbb{F}|$. The goal of Prover is to convince a verifier that:

$$\beta = \sum_{x_1 \in \{0,1\}} \sum_{x_2 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} g(x_1, x_2, ..., x_n), \tag{1}$$

where $\beta \in \mathbb{F}$. The verifier has oracular access to the polynomial g and is given the summand β. It is the Verifier's job to determine whether β is a sum of the polynomials $g(x_1, x_2, \ldots, x_n)$ for each x_i.

The protocol proceeds in v rounds, where in each round i, the Prover sends a univariate polynomial g_i to the Verifier. In the first round $i = 1$ the Prover sends the polynomial $g_1(X_1)$ to the Verifier, with the claim that $g_1(X_1) = \sum_{x_2} \sum_{x_3} \cdots \sum_{x_v} g(X_1, x_2, \ldots, x_v)$. In following rounds $i > 1$, the Verifier selects a random value $r_{i-1} \in \mathbb{F}$ and sends it to the Prover. The Prover then sends a polynomial $g_i(X_i)$ to the Verifier, with the claim that $g_i(X_i) = \sum g(r_1, \ldots, r_{i-1}, X_i, x_{i+1} \ldots, x_v)$. The Verifier checks the newest claim by considering that $g_{i-1}(r_{i-1}) = g_i(0) + g_i(1)$. In the final round the Prover sends the polynomial $g_v(X_v)$ claimed to be equal to $g_v(X_v) = g(r_1, \ldots, r_{v-1}, X_v)$.

The sum-check protocol has, since its introduction, been refined into different efficient proof protocols [120, 125, 118, 115, 44, 68]. A significant advantage of the sum-check protocol is that implementing the prover can, in specific settings, avoid using costly operations such as the Fast Fourier Transform, which is common in other protocols. Examples include [115] that implements a linear prover, or [44] where the prover was a streaming algorithm.

The sum-check protocol has been generalised to cover univariate polynomials [19] and tensor codes [89].

3.2.2   The Fiat-Shamir Heuristic

The Fiat-Shamir heuristic [54] allows the transformation of any public-coin interactive proof protocol I into a non-interactive, publicly verifiable protocol Q in the random oracle model (RO).

The random oracle model presented in [11, 34] is an idealised cryptographic model that gives all parties (the *Prover* and the *Verifier*) access to an entirely random function $H : \{0,1\}^k \to \{0,1\}^k$ between constant sized bit strings, typically a hash function, since no such ideal function exists. The function H takes an input and produces a random output chosen uniformly from the output domain. Random oracles are, in theory, used to obtain practical, efficient, and secure protocols, while in practice are heuristically secure since no truly random hash functions can be used in practice. Examples of this include the work in [34]. They show that there are secure schemes in theory, but for which any implementation of the random oracle results in insecure schemes. The random oracle allows the *Prover* to predict the random queries (challenges) on behalf of the *Verifier*. This eliminates the need for the *Verifier* to send messages to the *Prover*, making the protocol inactive.

Kilian [81] shows that 4-message argument systems for all of **NP** can be established by combining any PCP with Merkle-hashing based on collision-resistant hash functions. Micali

[92] shows that applying the Fiat-Shamir transformation to Killan's 4-message argument system yields a succinct non-interactive argument (SNARG) in the random oracle model. The resulting Kilian-Micali construction uses a Merkle-Tree (MT) [91] as the basis for a commitment scheme that allows sending just a single hash value (i.e., the tree's root) as commitment. The leaves of the tree correspond to all the Prover's evaluation points. Commitment schemes are discussed in the context of polynomial commitment schemes in Section 3.5.2.

3.2.3 Proof of Knowledge and Zero-knowledge

An honest prover can always convince the verifier about a true statement in a traditional interactive proof system with perfect completeness and soundness. Still, it cannot persuade it about something false. These notions can be relaxed to *statistical* properties, where they hold except with negligible probability, or to *computational* ones, where we admit the unlikely possibility of the prover cheating the verifier if it is computationally infeasible to do so.

A variant of this concept is *proofs of knowledge* [69], where the prover claims to *know* (or can compute) a particular piece of secret information in a way that convinces the verifier. What it means precisely for knowledge is defined as an *extractor*. Since the prover cannot simply output the secret knowledge itself, a *knowledge extractor* with access to the prover can extract a *witness* of such knowledge [97].

Let x be a statement of a language L in **NP** as before, and $W(x)$ a set of witnesses for x that should be accepted in the proof. Define the relation

$$S = \{\, (x, w) : x \in L, w \in W(x) \,\}.$$

Proof of knowledge for relation S is a two-party protocol between a prover and verifier in which the security notions now hold over the *knowledge* of the secret value:

- **Knowledge Completeness**: If $(x, w) \in S$, then the prover P who knows the witness w succeeds in convincing the verifier V of his knowledge. More formally, $Pr[P(x, w) \Rightarrow (V(x) = 1)] = 1$, i.e. a prover can always convince the verifier, given their interaction.

- **Knowledge Soundness**: requires that the success probability of a knowledge extractor E in extracting the witness, after interacting with a possibly malicious prover, must be at least as high as the success probability of the prover P in convincing the verifier. This property guarantees that if some prover can convince the verifier using some strategy, the prover knows the secret information.

The notion of *zero-knowledge* can be seen as an additional property that an interactive proof system, an interactive argument, or a proof of knowledge can have. In this notion, there is another requirement that whatever strategy and *a priori* knowledge the verifier follows or may have, respectively, it learns nothing except that the statement claimed by the prover is true. This is achieved by requiring that the interaction between the prover and verifier can be efficiently simulated without interacting with the prover, assuming the prover's claim holds. *Perfect zero-knowledge* is a more robust notion of zero-knowledge that does not limit the verifier's power [57].

We can model a cheating verifier as a polynomial-time Turing machine V^* that gets an auxiliary input δ of length at most polynomial in the size of input x. The auxiliary input δ represents *a priori* information that the verifier may have collected from previous executions of the protocol. That is, allowing the collection of this information, the verifier may cheat.

Definition 3.3 (Zero-Knowledge [45]). An interactive proof or argument system (P, V) for language L is zero-knowledge if for every polynomial-time verifier V^* there is a simulator M running in expected probabilistic polynomial time such that the simulation is computationally indistinguishable (in polynomial time) from (P, V) in input $x \in L$ and arbitrary δ.

Similarly, as before, we can generalise the zero-knowledge notion as perfect (resp. statistical) zero-knowledge by replacing the requirement of computational indistinguishability with perfect indistinguishability (resp. except for negligible probability).

Remark. *Falsifiable* assumptions refer to the cryptographic assumptions that can be formulated in terms of an interactive game between a challenger C and an adversary A such that C can determine whether A won at the end of the game and an efficient A can only succeed with at most negligible probability. Intuitively, an efficient C can test whether an adversarial strategy breaks the assumption. That is why the majority of assumptions and constructions in cryptography are falsifiable. On the other hand, some examples of cryptographic assumptions cannot be modelled this way and are consequently non-falsifiable. Knowledge assumptions are a clear example of this phenomenon.

An example of a simple proof of knowledge protocol that also happens to be zero-knowledge is Schnorr's proof of knowledge of a discrete logarithm [108]. The protocol is defined for a cyclic group $\mathbb{G}$ or order q with a generator element $g \in \mathbb{G}$. The prover wants to prove knowledge of $x = \log_g y$ in a group where computing x given $(g, y = g^x)$ is computationally infeasible. This setting constitutes a group where *computing discrete logarithms is hard*. The prover interacts with the verifier as follows:

1. The prover commits to randomness r by sending $t = g^r$ to the verifier.

2. The verifier replies with a challenge c chosen randomly.

3. The prover receives c and responds by sending $s = r + cx \bmod q$.

4. The verifier accepts if $g^s = ty^c$.

The protocol can be made non-interactive using the Fiat-Shamir heuristic to hash $c = H(g, y, t)$ as described in Section 3.2.2 above. It is a valid proof of knowledge because it has a knowledge extractor that extracts x by interacting with the prover two times to obtain $s_1 = r + c_1 x$ and $s_2 = r + c_2 x$ for two distinct challenges and computing $x = (s_1 - s_2)/(c_1 - c_2)$.

3.2.4 Multi-Prover Interactive Proof Systems

The prover of an IP can be split into multiple entities (MIP) with the restriction that these entities cannot interact while interacting with the verifier to form a multi-prover interactive proof (MIP) [12]. The setup is reminiscent of the police procedure of isolating suspects and interrogating each of them separately [64]. The suspects are allowed to coordinate a strategy before they are separated. However, once they are separated, the suspects can no longer interact. The verifier tries like an interrogator to determine if the prover's stories are consistent with each other and the claim being asserted.

A multiple-prover proof system is more expressive than a regular IP because each prover P_i is unaware of all messages sent to a prover P_j where $j \neq i$. It has been proved that the two-prover systems are as robust and expressive as any multi-prover interactive proof system.

Many of the ideas of MIPs have been adapted into interactive oracle proofs and probabilistically checkable proofs where a polynomial commitment scheme replaces the second prover.

3.3 Probabilistically Checkable Proofs

The first proof system introduced for **NP** in section 3.2 has perfect soundness and completeness; however, the proof size is not constant and requires the verifier to work much harder than we what we would like to be efficient and practical. Luckily enough, research has shown that the

performance of the verifier can be significantly improved if we are willing to settle for less-than-perfect soundness using a probabilistic approach where the verifier queries a random subset of the proof.

The Probabilistically Checkable Proof (PCP) theorem [6, 5, 48] revolutionised the field of verifiable computation by asserting that all NP statements have a PCP, meaning that they can be written in a format that allows an efficient (poly-logarithmic) probabilistic verifier V with oracle access to the proof π to probabilistically verify claims such as "$x \in L$" (for an input x and an NP-language L) by querying only a few bits of the proof π with soundness error $\delta_s = \frac{1}{2}$. The soundness error can be reduced to $2^{-\sigma}$ by running the verifier σ times.

An essential aspect of verifiable computation is the probabilistic verifier, which we will describe next.

Definition 3.4 (Verifier)**.** The Verifier is a randomised Turing machine (PTM) restricted by the following functions: $l, q, r, t : \mathbb{N}^+ \to \mathbb{N}^+$. The Verifier has oracle access to a proof π of the statement $x \in L$ with $|x| = n$, where the length of the proof is restricted by $|\pi| \le l(n)$. The Verifier flips at most $r(n)$ coins, queries at most $q(n)$ locations of the proof to either accept ($V_r(x, \pi) = 1$) or reject ($V_r(x, \pi) = 0$) the statement $x \in L$ in time $\le t(n)$.

Given such a verifier, the complexity class of **PCP** can be constructed.

Definition 3.5 (PCP class)**.** The **PCP** class is defined:

$$
\mathbf{PCP} \begin{pmatrix} length & = l(n) \\ randomness & = r(n) \\ queries & = q(n) \\ time & = t(n) \end{pmatrix},
\tag{2}
$$

where l, q, r, t are defined as in definition 3.4. A language L belongs to the **PCP** complexity class if there exists a **PCP**$(l(n), r(n), q(n), t(n))$ with (perfect) completeness 1 and soundness ½ for L, where

- Perfect Completeness: $x \in L \Rightarrow \exists \pi \cdot Pr[V_r(x, \pi) = 1] = 1$

- Soundness: $x \notin L \Rightarrow \forall \pi \cdot Pr[V_r(x, \pi) = 1] \le$ ½

That is, an answer will always be returned. The soundness of the answer will be no worse than a half.

The **PCP** class is often described as **PCP**$(r(n), q(n))$, which only considers the query complexity $q(n)$ and the randomness $r(n)$. The verifier is described using these two functions.

Much research has studied the construction of PCPs, essentially looking for languages in the PCP class with minimal proof length, running time, randomness and query complexity while minimising the soundness error. The most significant costs in verifiable computing are typically the verifier's and prover's runtime, the number of queries $q(n)$, and the creation of the circuit.

Definition 3.6 (PCP Theorem)**.** The PCP theorem: $\mathbf{NP} = \mathbf{PCP}(O(\log n), O(1))$.

There have been various attempts to minimise the query complexity. Håstad [75] showed in 1997 that the query complexity in the PCP theorem can be reduced to 3 bits.

The PCP theorem is one of the most challenging theorems in theoretical computer science and is considered one of the most important results in complexity theory [124]. The proof of the PCP theorem is far beyond our scope here, but we point the reader to [48]. We will instead highlight the significant steps of the proof.

3.3.1 Proof Techniques and Insights

The early work on the PCP theorem was motivated by complexity-theoretic considerations [51, 52]. In contrast, later work such as [75] focused on applications where precise bounds on the performance of verifiers matter most. There has also been some work on simplifying assumptions to improve bounds and proofs of corresponding theorem variants.

The assumptions about the provers in the PCP theorem are weak, making the PCP theorem very strong. A prover can be very powerful, in particular, and have unlimited resources. Strengthening these assumptions, for instance, assuming that a prover has a (large but) limited amount of memory available, can be used to simplify proofs of corresponding variants of the PCP theorem [127].

Complexity Considerations. In [51, 52], the theorem is used to prove a result on the non-approximability of NP-complete problems. More precisely, it is shown that approximating maximal cliques in graphs is "almost" **NP**-complete. (The "almost" was later discarded by [6].) A *clique* of a graph G is a set of vertices C such that each pair of vertices n_1, n_2 contained in C is connected by an edge in G. A *maximal clique* M is a clique of G such that $|C| \leq |M|$ for all cliques C of G.[8]

To prove non-approximability, a variant of a **MIP** consisting of a polynomial-time Turing machine M, that checks whether $x \in L$ for some language L, and a memoryless oracle O. Given an input x (with length n), a maximum number of random bits $r(n)$ that M reads from x and a maximum number $c(n)$ of communication bits exchanged between M and the oracle O, one can construct a graph G_x whose vertices are accepting communications transcripts $\langle r, q_1, a_1, \ldots q_l, a_l \rangle$ where q_i is a query by M and a_i is the answer of the oracle. Furthermore, $r = r(n)$ and the length of $\langle q_1, a_1, \ldots q_l, a_l \rangle$ must be at most $c(n)$. A transcript is accepting if M accepts x with random string r and the history of communications $\langle q_1, a_1, \ldots q_l, a_l \rangle$. The edges of G_x are consistent transcripts where two transcripts are consistent if identical queries in either transcript yield identical answers. The size of G_x is exponential in $r(n) + c(n)$. Let $\omega(G_x)$ denote the size of the maximal clique of G_x. Noting that $Pr_r[M^O(x, r) \text{ accepts}]$ denotes the fraction of nodes for which M accepts and $r(n)$ the number of choices made in a transcript, we get the equation $max_O Pr_r[M^O(x, r) \text{ accepts}] \cdot 2^{r(n)} = \omega(G_x)$. Using the definitions of perfect completeness and soundness given above, we have: if $x \in L$, then $\omega(G_x) = 2^{r(n)}$; and if $x \notin L$, then $\omega(G_x) \leq 2^{r(n)}/2$. This construction relates a **PCP**$(r(n), c(n))$ problem directly to an NP-complete clique problem of size exponential in $r(n)$. It implies that solving the clique problem by approximation using M would also solve it. Using the exponentially-sized clique in the graph G_x, we determine that the running time must be exponential. The PCP theorem says that –reading a few bits from the input x witnessed by the communications $c(n)$ and tossing a few random coins $c(n)$, both logarithmic in the size of the input– a polynomial-time M, namely $n^{O(1)}$, can check whether $x \in L$ for some **NP** language L: M needs to be run $O(\log \log n)$ times; so, it takes $n^{O(\log \log n)}$ times, i.e., it is "almost" polynomial. Iterating the execution of M amplifies the approximation, yielding. If approximating $\omega(G_x)$ within a factor $2 \log^{1-\epsilon} n$ for some $\epsilon > 0$ is "almost" polynomial, then **NP** is contained in the class of "almost" polynomial problems. As stated earlier, the "almost" can be dropped. This says that the only way an approximation can exceed the bound is if $\mathbf{P} = \mathbf{NP}$.

For the applications of the PCP theorem, this insight is crucial because it implies that for the **NP**-complete problems submitted to the prover, there is no deterministic polynomial algorithm that could carry out computations that could "cheat" the verifier. The only way is to

---

[8]Cliques have many practical applications. For example, the vertices in a social network represent individuals, and the edges represent mutual acquaintance. A social network clique represents people who all know each other. Finding cliques locates groups of common friends. When Facebook makes a friend suggestion, it tries to join two cliques.

toss a random coin and guess among an exponential number of alternatives yielding a negligible chance of cheating the verifier. For this reason, in work on the PCP theorem, complexity is often mentioned.

Approximation Bounds. The concrete approximation bounds used in [51, 52] are not of great interest here, only that they establish that getting a "close" approximation is impossible. In later publications, variants of the PCP theorem provide better bounds that are important for practical applications. We give a brief account of the original proof of the PCP theorem. Any NP-complete language can be used as a basis for the theorem. The proof uses 3-SAT. 3-SAT formulas can be arithmetised straightforwardly, which plays a central role in formulating the procedure used in the proof.

The procedure to check whether an input formula f of length n is in 3-SAT follows the five following major steps:

1. Arithmetise f with satisfying assignment (to variables) A; the oracle knows A, but the oracle machine does not. The following steps permit the oracle machine to evaluate whether A is a satisfying assignment with high probability reading only logarithmically many bits m. The number m is chosen so that no smaller number satisfies $2^m = n$. So m is the bit width required to represent the clause indices of f. To reduce the communication with the oracle, function f is extended (multilinearly) over a finite field $\mathbb{F}$ such that the probability of detecting deviations by the order of the cardinality $|\mathbb{F}|$ of $\mathbb{F}$. To achieve a rejection probability below $\frac{1}{2}$ for unsatisfying formulas later on, the size of the field must be at least $20m$.

2. Test multilinearity of A. A function of m arguments is multilinear if it is linear in each argument. The assignment A produced in step 1 is multilinear, and this property is a precondition for step 4. Linearity in each argument can be verified by sampling three random points of A from the oracle. This is sufficient to see whether A is "almost" linear for each argument with high probability because deviations of multiple segments in an argument lead to a high proportion of non-linear points. And this proportion can be detected with high probability. The "almost" above is described by a threshold of deviations below which the function would be accepted as linear. The number of randomly generated triples can be reduced by two-point sampling where two points are randomly generated and the remaining $20m$ points deterministically computed. Here the size of the field $\mathbb{F}$ is important to achieve a rejection probability below ½.

3. Choose a random sequence R of m weights determining a sub-range of A to evaluate. This random sequence is communicated to the oracle to retrieve a portion of A and to evaluate f for A on the side of the oracle machine.

4. The sum-check protocol (see Subsection 3.2.1) is used to evaluate the transformed formula f for A at the sample points determined by R from step 3. In several rounds, the oracle machine receives m polynomials. The size of the field $\mathbb{F}$ increases the probability of discovering deviations from 0 because the fraction of points where a multilinear function is $(1 - \frac{1}{|\mathbb{F}|})^m$. (It is 0 for at most one value per argument. The fraction is just the ratio to the totality of points $|\mathbb{F}|^m$.)

5. The oracle machine requests the missing last three points of A from the sum-check protocol to complete the evaluation of f at A. For steps 4 and 5 to attain the necessary probability of rejection, the field $\mathbb{F}$ must have at least cardinality 100.

In the steps just listed, note that all quantities are expressed in terms of m and constant multiples of m and that $m \approx \log n$. Summing up over the different logarithmic terms, we obtain the version of the PCP theorem stated in the paragraph above discussing complexity: Any language $L \in \mathbf{NP}$ is accepted by a probabilistic oracle-machine with $r(n) + c(n) \leq \log n \cdot \log \log n$, and M's running time is $\mathbf{DTIME}(n^{O(1)})$ (DTIME), that is $\mathbf{NP} \subseteq \mathbf{PCP}(\log n \cdot \log \log n, \log n \cdot \log \log n)$ [52]. An alternative proof of the PCP theorem based on [48] is discussed in [103, 113].

Improved Bounds. Early work on PCP provided the result that there is a constant $C \geq 1$ that the optimal value opt of specific problems cannot be approximated with a polynomial algorithm beyond C. (Polynomial approximability is referred to as *efficient* compared to approximability in general.) More formally, an algorithm C-approximates a maximisation problem O if, for each input x, it outputs a value ω such that $\text{opt}/C \leq \omega \leq \text{opt}$ (and correspondingly for minimisation problems) [75]. Concerning PCP, we are interested in inapproximability. There are differences between problems' inapproximability. No efficient algorithm can find an *omege* with the bounds. For some, C is a constant. For others, it is an expression depending on the input's size, and others can be approximated arbitrarily tight [94]. The PCP theorem also made it possible to classify these problems and get tighter bounds as the mathematical tools used in its proof evolved.

Håstad [75] presents a collection of inapproximability results. These depend very much on specific NP-complete problems. Håstad conjectures that obtaining tighter bounds for PCP depends on specialising different NP-complete problems. The improvements achieved in [75] are due to long codes [10], their Fourier-transforms [74] and joining long-code and proof verification [75]. The approximation value is calculated from the expected value of the Fourier transform of the long-code encoding of a specific approximation problem. Long codes permit encoding the problem for transmission. They lift provers' answers into a function space such that the long code of an answer x is a mapping A_x such that $A_x(f) = f(x)$. It is exponential in the size of x. Applying the Fourier-transform t and its inversion to A_x we get an expression $B_x = t^{-1}(t(A_x))$ composed of sums and products with $A_x = B_x$ that permits easy calculation of expectations $E[B_x]$. Testing the long code consists in choosing two random functions f_0 and f_1 uniformly as well as one function g tossing a biased random coin (with probabilities ϵ, $1 - \epsilon$) for each value. For these the functions the product $A_x(f_1) \cdot A_x(f_2) \cdot A_x(f_2)$ with $f_2 = f_0 \cdot f_1 \cdot g$ is used as the acceptance condition. The expectation of the acceptance condition $E[A_x(f_1) \cdot A_x(f_2) \cdot A_x(f_2)]$ can be calculated algebraically as suggested above, yielding a small chance of rejecting a correct code –because of the choice of g– and a logarithmic bound on the size of the size of the part of the acceptance condition $A_x(f_1) \cdot A_x(f_2) \cdot A_x(f_2)$ to be evaluated. The random choice of g is essential to obtain the latter. The bound ensures that there can be an efficient algorithm for the verifier.

Håstad found that he can modify the algorithm for long-code testing so that it verifies Max-E3-Lin-2 and test the long code at the same time, conflating the codes to be transmitted for both so that only three values need to be requested from the prover. The completeness of the protocol is $1 - \epsilon$ and its soundness $1/2 + \epsilon$.

If instead of a two-valued domain as in Max-E3-Lin-2 a finite Abelian group of size n is used, then the number of bits to be transferred is multiplied by $\log n$. This corresponds to the number of bits required to encode the group's values.

Using a construction by Trevisan et al., [116] called *gadgets*, the bounds of Max-E3-Lin-2 can be used for a reduction in the proof of the PCP theorem [75] to derive tight bounds while reading three bits from the prover. He proves that Max-Cut cannot be $(\frac{17}{16} + \epsilon)$-approximated by an efficient prover.

3.3.2 Linear PCP

There exist multiple flavours of PCPs. In this and the following subsections, we discuss a selection of them. We begin with linear PCP, in short, LPCP.

In an LPCP the proof is a linear function λ, that is

$$\lambda(a * x + b * y) \;=\; a * \lambda(x) + b * \lambda(y) \;.$$

A language belongs to **LPCP** if there exists an $(r(n), q(n))$-LPCP-verifier V

- Perfect Completeness: $x \in L \Rightarrow \exists\,\lambda \cdot Pr[V_r(x, \lambda) = 1] = 1$

- Soundness: $x \notin L \Rightarrow \forall\,\lambda \cdot Pr[V_r(x, \lambda) = 1] \leq \delta$,

where λ is a linear function.

LPCP are interesting because the proof can be encoded with a homomorphic encryption scheme permitting efficient implementation. The polynomial communication to the prover and constant communication to the verifier [77]. These properties make LPCP a practical PCP method compared to more general protocols.

Recognising such practical and useful classes of PCP appears essential for use in implementations as more general protocols have theoretical limits that cannot be overcome.

3.3.3 PCPs of proximity

PCPs of proximity (PCPP), also called assignment testers, relax the notion of PCPs to a problem of proximity testing [22, 49], where the verifier only should test that the input is close to an element of the language instead of asserting membership.

The closeness of a language can be described using a variety of measures, but the most common one is the relative Hamming distance.

Definition 3.7 (Hamming Distance). Let $\mathbb{F}$ be a field, $n \in \mathbb{N}^+$, and two string $s, t \in \mathbb{F}^n$ of length n. The Hamming distance between x and y is $\Delta_n(s, t) = |\{i \in \{0, .., n-1\} \mid x_i \neq y_i\}|$. Their relative Hamming distance is $\Delta_n^*(s, t) = \Delta(s, t)/n$.

A string s is δ-far from the string t if $\Delta_n^*(s, t) > \delta$ and δ-close otherwise.

PCPPs refer to pair languages. A $L \subseteq \Sigma^* \times \Sigma^*$ is a "pair language" where each instance consists of pairs of strings of the form (x, w). The first input x is the *explicit* input, and the second input w is the *implicit* input. String x is supposed to be in a language (characterised by proximity), and w is a redundant encoding of a proof used as an additional oracle by the verifier. Whereas x is known to the verifier in its entirety, w is only known "implicitly" and can be queried by the verifier. The idea behind providing the (x, w) is to be able to say that some string y is close to a string $x \in L$ instead of determining directly whether $y \in L$. Pair languages can be defined in terms of their explicit input. Without loss of generality, we assume that the explicit input includes a specification of the length of the implicit input. Formally, the explicit input is of the form $x = (x', N)$, where $|x| = x'$ and $N = |y|$. For a pair language L and $x \in \Sigma^*$, $x = (x', N)$, let $L_x = \{y \in \Sigma^N \mid (x, y) \in L\}$.

The verifier is given the explicit input x and is tasked with the challenge to distinguish with high probability and with a small number of queries to the implicit input y to accept implicit inputs in L_x and reject implicit inputs that are far from being in the language. The complexity class PCPP is defined in definition 3.8.

Definition 3.8 (PCPP). For a family of relative distance measures

$$\Delta^* = \{\Delta_N^* \mid \Sigma^N \times \Sigma^N \to [0, 1]\}_{N \in \mathbb{N}^+}$$

and proximity parameter $\delta \in [0, 1]$ we define the complexity class **PCPP**:

$$\mathbf{PCPP} \begin{pmatrix} randomness & = r(n) \\ queries & = q(n) \\ distance & = \Delta_n^* \end{pmatrix}, \tag{3}$$

where the functions q, r are defined as in definition 3.4. A pair language L belongs to the **PCPP** class if there exists an $(r(n), q(n))$-PCPP-verifier V satisfying the following conditions:

- Perfect Completeness: $x \in L \Rightarrow \exists \pi \cdot Pr[V_r(x, y \circ \pi) = 1] = 1$

- Soundness: $x \notin L \Rightarrow \forall \pi \cdot Pr[V_r(x, y \circ \pi) = 1] \leq \frac{1}{2}$,

where $\circ$ denotes composition of proofs. This is used to treat the two oracles conceptually as one.

We do not define the PCPP-verifier explicitly since it is similar to the verifier in definition 3.4.

Proximity testing is often used in the context of locally testable codes. A locally testable code is a code that a randomised Turing machine with oracular access to the supposed codeword can distinguish with a high probability between words in the code and words far from it using a small number of queries [67]. An example of a locally testable code is the Reed-Solomon code [67], used in the FRI protocol, described in section 3.7.

The current state of the art in the PCPP model gives quasi-linear proofs of length $n \cdot \log(O(1) \cdot n)$ with constant query complexity [24, 48] and linear prover complexity $O(n)$; the verifier complexity is $poly \log n$ [22, 93].

3.4 Polynomial Interactive Oracle Proofs

Interactive Oracle Proof (IOP) [20, 104] mixes the powers of **PCP** with the interactive nature of an IP and is a generalisation of interactive PCP [78].

Informally it is a k-round interactive protocol with k rounds of interaction between the prover and the verifier with the twist that the verifier reads only a subset of each prover's message, referred to as oracles since the verifier has full query access.

Definition 3.9 (IOP). An IOP system is a k-round interactive oracle protocol defined by some relation R and some soundness error $\delta \in [0, 1)$ and two probabilistic algorithms (P, V) satisfying the following properties:

- Completeness: $Pr[\langle P, V \rangle (x) = 1 \mid x \in R] = 1$

- Soundness: $Pr[\langle P, V \rangle (x) = 1 \mid x \notin R] \leq \delta$

This means it is an interactive protocol where the verifier runs in time sub-linearly in the total proof's length. In each communication round, the verifier gave access/permission to a single query of the prover's message (proof).

It has been shown that any IOP can be transformed into a non-interactive argument based on the random oracle model by using the Fiat-Shamir heuristic and the transformations by Kilian and Micali [20].

Necessary measures of IOP protocols include the round complexity k, along with two additional steps, namely the proof length p, which is the total number of alphabet symbols in all of the prover's messages, and the query complexity q.

Polynomial IOP [19] extends the IOP protocol where each message the verifier checks is a set of low-degree polynomials. The idea of using an oracle is that rather than reading the entire list of coefficients describing the polynomial, the verifier queries these polynomials in a given point through an oracle interface, which is simulated using a polynomial commitment scheme [114].

3.4.1 IOPs of Proximity

IOPs of Proximity (IOPP) is a generalisation of **PCPP** to the IOP model introduced in [28].

3.5 Computations as Polynomial Constraints

An essential aspect of verifiable computation is the notion of computation. We want to express a computation as a set of constraints that can be efficiently verified. Efficiency in our context means we want to minimise the overhead associated with the interaction between the two parties and move computation to the Prover.

3.5.1 Polynomials

The main idea behind using polynomials is to use the mathematical foundation of the Schwartz-Zippel lemma [47], which can be interpreted as follows:

Two different polynomials p and q of degree d agree on at most d points, formally:

$$p \neq q \Rightarrow card(\{\, x \mid x \in \mathbb{F} \cdot p(x) = q(x) \,\}) \leq d,$$

where $\mathbb{F}$ is the finite field in which the polynomials are defined. Consequently, the probability of finding a different polynomial Q that agrees with P in an arbitrary point x is $d/card(\mathbb{F})$, which is negligibly tiny if the field is large enough. This means that polynomials are very useful for encoding the executing programs since they amplify errors made in the execution of the program, which is vital since computations are incredibly fragile—changing only a single bit in the program can change the result of the computation completely.

The probability can be further reduced by checking in more than one point, expanding the field, or reducing the degree of the polynomials.

Another fact about polynomials is that a set of n point-value pairs uniquely determines a polynomial of degree $n - 1$. This polynomial can be determined using Lagrange Interpolation [42].

It is well known that any polynomial $p(x)$ with n roots $(r_1, \dots, r_n)$ can be described using the product of two polynomials $t(x)$ and $h(x)$, formally:

$$p(x) = t(x) * h(x), \tag{4}$$

where $t(x) = \prod_{i=1}^{n} (x - r_i)$, a polynomial with the same roots as $p(x)$. The degree of polynomials resulting from the multiplication and addition of two polynomials is well understood. From two polynomials $p(x)$ and $q(x)$, where the degree of $p(x)$ is n and the degree of $q(x)$ is m, we can create an $n + m$ degree polynomial by $q(x) \times p(x)$ and a polynomial of degree $\max(m, n)$ $p(x) + q(x)$. Polynomials can be multiplied using the Fast Fourier Transformation in time $O(n \log n)$ [73].

3.5.2 Polynomial Commitment Schemes

Commitment schemes [96] are used to ensure that the prover cannot modify a proof during exchanges with the verifier. The verifier must *commit* to the result (of its computation) and the associated proof. Afterwards, some information is *revealed* to the verifier as it accesses information received from the prover. A commitment is an encoding by the prover that fixes the prover's answer while not revealing that answer. The verifier accesses bits of the solution that are revealed without seeing the whole answer. The use of commitment schemes is widespread in cryptography applications, and they are necessary for PCP applications for the reason just stated.

Polynomial commitment schemes [79] take advantage of the specific shapes of polynomials $p(x)$ and their algebraic properties to permit pointwise access to the values $p(i)$ of the polynomial at specific points i without revealing the polynomial itself.

3.5.3 Arithmetic Circuits

An arithmetic circuit naturally expresses the computation of a polynomial over a field $\mathbb{F}$, which is why they are used in verifiable computation. Any Turing machine can be unrolled into a circuit somewhat larger than the number of steps in the computation [59, 101]. Constructing an arithmetic circuit from a program is often referred to as *arithmetisation*.

An arithmetic circuit C of a finite field F is a directed graph where each vertex is either a multiplication or an addition gate with fan-in 2, and each edge is a connection between two gates representing a wire in the circuit. The degree of the expressed polynomial equals the number of multiplication gates in the circuit.

All gates in the same layer of a circuit C (viewed as a directed acyclic graph of gates) can be executed in parallel, meaning that the depth d of the circuit C represents the number of sequential steps in the program. The size S of an arithmetic circuit C is encoded as $S = \mid C \mid$: the number of gates in the circuit C.

An arithmetic circuit is always encoded as a constraint system in the context of verifiable computation. The most common encodings are described below.

3.5.4 Rank 1 Constraint Systems

A constraint system defines a collection of arithmetic constraints over a set of variables to express the satisfiability of an arithmetic circuit in terms of linear constraints.

The de facto standard for these constraint systems is the Rank-1 Constraint System (R1CS), specifying an NP-complete language. R1CS is an intermediate representation used by many tools to represent arithmetic circuits since it is easy to generate and verify.

Definition 3.10 (R1CS). An R1CS instance is a tuple $(\mathbb{F}, A, B, C, v, m, n)$ where A, B, C are $m \times m$ matrices over the field $\mathbb{F}$, v is a vector of maximally size m denoting the public inputs and outputs, and m is the number of variables.

An R1CS instance $x = (\mathbb{F}, A, B, C, v, m, n)$ is *satisfiable* $(SAT_{R1CS}(x, w) = 1)$ if there exists a witness w such that $(A \cdot s) + (B \cdot s) = C \cdot s$, where w are the values of the private variables/wires, sometimes referred to as the witness.

We refer to the variables v and w as the public and private variables since this technique allows the *Prover* to hide the private variables in the context of zero-knowledge proofs.

The matrices A, B and C represent the structure program/circuit of interest by representing the left input of a gate, the correct input of a gate, and the output of a gate. The vector s represents the values a wire/variable can take during execution on a given input.

The *Verifier* generates the R1CS from the circuit; since the matrices, A, B, and C are independent of the input values, the cost of the R1CS generation can be amortised over all future computations.

3.5.5 Quadratic Arithmetic Programs

Quadratic arithmetic programs (QAP) were introduced by Gennaro et al. in [59] to encode a program or arithmetic circuit in terms of a set of polynomials.

Definition 3.11 (QAP). A QAP Q over field $\mathbb{F}$ contains three sets of $m + 1$ polynomials $V = \{v_k(x)\}$, $W = \{w_k(x)\}$ and $Y = \{y_k(x)\}$ for $k \in \{0, \ldots, m\}$ and a target polynomial

$t(x)$ having the same degree and roots as the polynomial p described below. The constant m denotes the number of wires in the arithmetic circuit.

The QAP Q computes the function $F : \mathbb{F}^n \to \mathbb{F}^{n'}$ if there exist $N = n + n'$ field elements such that $(c_1, \ldots, c_N) \in \mathbb{F}^N$ is a valid assignment of F's inputs and outputs, if and only if there exist coefficients, i.e. values on internal wires, $(c_{n+1}, \ldots, c_m)$ such that $t(x)$ divides $p(x)$, where:

$$p(x) = \left(v_0(x) + \sum_{k=1}^{m} c_k * v_k(x) \right) * \left(w_0(x) + \sum_{k=1}^{m} c_k * w_k(x) \right) - \left(y_0(x) + \sum_{k=1}^{m} c_k * y_k(x) \right) \quad (5)$$

There must be a polynomial $h(x)$ such that $h(x) \cdot t(x) = p(x)$ for all x. It is a well-known fact that any polynomial $p(x)$ with n roots $(r_1, \ldots, r_n)$ can be described using the product of two polynomials $t(x)$ and $h(x)$, formally:

$$p(x) = t(x) \cdot h(x),$$

where $t(x) = \prod_{i=1}^{n} (x - r_i)$ (the target polynomial). The target polynomial is created by arbitrarily picking a root r_i for every multiplication gate g_i in the arithmetic circuit [98].

3.5.6 Algebraic Intermediate Representation

Another approach for expressing a program as polynomial constraints is the so-called algebraic intermediate representation (AIR) introduced in [15]. An AIR does not represent the computation as a circuit but as a trace showing how the registers and memory evolve through the execution of the program. A T steps algebraic execution trace of a program P is represented by a $\mathbb{F}^{w \times T}$ matrix. A row describes the state at a given step, and a column tracks the state of a register over the entire execution.

A valid trace is determined by a set of transition constraints defining algebraic relations between two (or more) rows of the trace, along with a set of boundary constraints that enforce equality between some cells of execution and a set of constant values.

3.6 zk-SNARKs

A succinct non-interactive argument (SNARG) is a triple of algorithms (G, P, V) where G is a probabilistic generator that, given the security parameter λ and a time bound T as inputs, outputs a reference string crs and a corresponding verification state $priv$. The honest prover P will, given a witness w, produces a proof $\pi \leftarrow P(crs, y, w)$ of the statement $y = (M, x, t)$ if $t \leq T$.[9] The validity of the proof π is determined by the verifier $V(priv, y, \pi)$. The SNARG is adaptive if the prover may choose the statement after seeing crs. Otherwise, it is non-adaptive.

Definition 3.12 (SNARG). (See [18].) A triple of algorithms (G, P, V) is a SNARG for an **NP** language L with a corresponding **NP** relation R, if it satisfies the following three properties:[10]

- Completeness: For all $\lambda \in \mathbb{N}^+$ and for all $(u, w) \in R$:

$$Pr \left[V(priv, u, \pi) = 1 \; \middle| \; \begin{array}{l} (crs, priv) \leftarrow G(1^\lambda) \\ \pi \leftarrow P(crs, u, w) \end{array} \right] = 1 \quad (6)$$

---

[9]We are using here the notation $\leftarrow$ to define the variable π.

[10]λ is the security parameter expressing the hardness of breaking the security scheme mentioned above. The notation 1^λ denotes the string $1 \ldots 1$ of λ 1s.

- Computational Soundness: For all probabilistic polynomial-time malicious provers P^* that outputs and adaptively chosen pair (u, π) such that there is no a for which $(u, a) \in R$,

$$Pr\left[\begin{array}{c} V(crs, u, \pi) = 1 \wedge \\ u \notin L \end{array} \;\middle|\; \begin{array}{c} (crs, priv) \leftarrow G(1^\lambda) \\ (u, \pi) \leftarrow P^*(1^\lambda, crs) \end{array}\right] = negl(\lambda), \qquad (7)$$

  where $negl(\cdot)$ denotes a negligible function of its input.

- Succinctness: The length of a proof is given by $|\pi| = poly(|x| + |w|)$.

A SNARG is publicly verifiable if the private verification state $priv = crs$, meaning that any party can verify the proofs since crs is publicly known [18]. On the other hand, a SNARG is designated unverifiable if $priv \neq crs$ since it is only that party that knows $priv$ can verify proofs and soundness only holds if $priv$ is kept private. This definition does not address the issue of reusing the crs for multiple proofs.

A SNARG where the condition of computational soundness is replaced by knowledge soundness is a SNARK (succinct non-interactive arguments of knowledge) [27, 26]. This means that every prover producing convincing proof must "know" a witness. We define the property of knowledge soundness using a so-called knowledge extractor that can efficiently extract the witness from the prover.

Definition 3.13 (SNARK). A triple of algorithms (G, P, V) is a succinct non-interactive argument of knowledge (SNARK) if (G, P, V) is a SNARG that comes together with a knowledge extractor ϵ as previously discussed.

A SNARK is considered a zero-knowledge SNARK if the argument does not leak information besides the statement's truth.

Definition 3.14 (zk-SNARK). A SNARK for an **NP** language L defined over the relation R is zero-knowledge if there exists a simulator (Sim_1, Sim_2) such that Sim_1 outputs a *simulated crs* and a trapdoor τ, whereas Sim_2 produces a simulated proof π given an crs, a statement x and a trapdoor τ. It must hold for all $\lambda \in \mathbb{N}^+$, for all $(u, w) \in R$ and for all probabilistic polynomial-time adversaries $\mathcal{A}$ (perfect zero-knowledge):

$$Pr\left[\begin{array}{c} (x, w) \in R \wedge \\ \mathcal{A}(\pi) = 1 \end{array} \;\middle|\; \begin{array}{c} crs \leftarrow G(1^\lambda) \\ (u, w) \leftarrow \mathcal{A}(crs) \\ \pi \leftarrow P(crs, u, w) \end{array}\right] = Pr\left[\begin{array}{c} (x, w) \in R \wedge \\ \mathcal{A}(\pi) = 1 \end{array} \;\middle|\; \begin{array}{c} (crs, \tau) \leftarrow Sim_1(1^\lambda) \\ (u, w) \leftarrow \mathcal{A}(crs) \\ \pi \leftarrow Sim_2(crs, u, \tau) \end{array}\right]$$

The simulator (Sim_1, Sim_2), using some trapdoor information τ, can find a proof without consulting the witness w, i.e., it does not require any knowledge about the witness.

SNARKs have been extensively studied and are used in many different contexts and applications. There are many different variants of SNARKs. Nevertheless, they all arise from an underlying characterisation of the complexity class **NP** as a specification of an NP-complete problem with specific advantageous properties. A popular starting point for many of the most influential SNARKs [59, 85, 46, 72, 60] has been the NP-complete decision problem circuit satisfiability (Circ-SAT). Many of the mentioned SNARKs for circuit satisfiability are based on the characterisation of quadratic span programs and quadratic arithmetic programs introduced by Gennaro et al. in [59].

Definition 3.15 (Circuit Sat). The Circ-SAT problem is the **NP**-complete decision problem of determining whether a given circuit has an assignment of its inputs that makes the output true. A circuit C is a function $C : \{0, 1\}^\ell \to \{0, 1\}$. The decision problem consists in finding a vector of constants $(a_1, \ldots, a_\ell)$ such that $C(a_1, \ldots, a_\ell) = 1$ if it exists.

3.6.1 Examples of constructions

Designing zk-SNARKs that produce short constant-sized proofs and efficient prover/verifier algorithms is a hot research topic. We provide an overview of some recent examples below. We note that all these constructions are based on specific results from group theory that make the computation of discrete logarithms hard. Consequently, the algorithms cannot be quantum-safe or offer long-term security against quantum adversaries. In practice, these groups are typically instantiated with cryptographically secure *elliptic curves* [4].

Groth16. It was the first practical zk-SNARK based on cryptographic pairings [72], and thus is used as a reference point for comparison and benchmarking. The circuits are represented as a QAP, and proofs are quite compact (only three group elements), much shorter than [98]. The main drawback with this construction was relying on a public reference string used by the prover and verifier that was created using a one-time trusted setup. The setup was circuit-specific, so it had to be repeated for any new circuit. If the setup is performed maliciously, soundness is broken, and undetectable fake proofs may be generated. A proposed way to mitigate this problem was to conduct complex multi-party computation for the setup. The most famous practical deployment of this zk-SNARK construction was the ZCash cryptocurrency [17].

Bulletproofs. It is a proof system introduced by Bünz et al. [31] that does not require a trusted setup or cryptographic pairings to be efficient, thus relying on weaker assumptions. However, it is not a zk-SNARK *per se* since proof sizes scale logarithmically and verification time scales linearly with the size of the circuit, even when batching is applied. Bulletproofs is still quite practical and competitive for simpler circuits, for example, (batched) range proofs to prove that integers lie in a certain interval. For this reason, it is a popular option for proof systems implementing confidential transactions in cryptocurrencies.

Sonic. It is an early and quite practical example of a polynomial IOP [87], as described in Section 3.4. It represents the circuit as three vectors of left, right and output wires, for which the consistency for multiplication and addition gates is reduced to one large polynomial equation. This equation is embedded into the constant term of a bivariate polynomial. The verifier checks that the bivariate polynomial is computed by the prover from the witness and circuit such that the constant term is cancelled out. Sonic proofs are small, the prover is efficient, and verification is fast when many proofs for the same circuit are verified simultaneously. This fits many applications, for example, checking a batch of financial transactions for consistency, where Sonic remains state-of-the-art. Sonic also enjoys an optional *helper* mode, in which a third party commits to a batch of proofs and provides a short and easy-to-verify proof of the correct calculation, making the final verification more efficient. Because of its custom constraint system, Sonic implementations typically offer an adaptor for circuits in R1CS.

Gates. A polynomial division check ensures that the prover knows satisfying inputs and outputs for each gate (witness), and a permutation argument provides correct wiring between the gate outputs and inputs. The constraint system is non-standard and richer than all previous constructions, supporting different operations and optimizations, which perform at some cost in complexity in the PLONK compiler. PLONK offers shorter proofs and proving times in comparison to Sonic.

Marlin. It is an improvement over Sonic with much faster prover and verification times for individual proofs [39]. While Sonic remains state-of-the-art for batched settings, Marlin shines when batching is impossible. Compared to PLONK, the proving cost and proof size are similar,

with the advantage that Marling uses an R1CS constraint system that is much more standard and widely supported by existing tooling.

3.7 zk-STARKs

The most crucial difference between a zero-knowledge scalable transparent argument of knowledge (zk-STARK) and a zk-SNARK is the transparency of the zk-STARK approach. As a consequence of this transparency, it is also post-quantum secure. The zk-STARK approach was introduced by Ben-Sasson et al. [15] and is based on the FRI protocol described below.

3.7.1 Fast Reed-Solomon IOP of Proximity

Fast Reed-Solomon Interactive Oracle Proof of Proximity (FRI) was introduced by Ben-Sasson et al. [14] and gradually improved concerning the soundness bound of FRI [23, 21, 16]. The protocol is an IOPP working over a pair language of Reed-Solomon (RS) codes.

Definition 3.16 (Reed-Solomon Codes)**.** The evaluation of a polynomial $P(z) = \sum_{i=0}^{d} a_i z^i$ over $S \subseteq \mathbb{F}$, with $\mathbb{F}$ a finite field and $|S| = n$, is the function $p : S \to \mathbb{F}$ defined by the function $p(s) = P(s)$ for all $s \in S$. The code $RS[\mathbb{F}, S, \rho]$ is the space of functions $f : S \to \mathbb{F}$ that are evaluations of polynomials of degree $d < \rho \cdot n$. Formally,

$$RS(\mathbb{F}, S, \rho) = \{f : S \to \mathbb{F} \mid deg(f) < \rho \cdot |S|\} \tag{8}$$

The pair language PAIR-RS is defined as follows. The direct input is the definition of the RS code $(\mathbb{F}, S, \rho)$, where $\mathbb{F}$ is a finite field, $S \subseteq \mathbb{F}$, and $\rho \in [0, 1]$ is the code rate. The code rate describes the amount of helpful information in a string: if a string has length n (as above) and the code rate ρ is a fraction of n, then the string contains $\rho \cdot n$ useful bits and $n - \rho \cdot n$ redundant bits. The verifier's task is to distinguish with high probability and with a small number of queries to the implicit function $f : S \to \mathbb{F}$, between the case that $f \in RS[F, S, \rho]$ and the case that f is δ-far from (all members of) $RS[F, S, \rho]$ in relative Hamming distance.

We define the distance of an element $e \in \Sigma^N$ and a set $S \subseteq \Sigma^N$ to be

$$\Delta^*(e, S) = \begin{cases} \min_{s \in S} \Delta^*(e, s) & \text{if } S \neq \emptyset \\ 1 & S = \emptyset \end{cases} \tag{9}$$

Checking if $f \in RS[F, S, \rho]$ is essentially a variant of low-degree testing, which can be naively performed using $d + 2$ queries ($d + 1$ queries to recover the polynomial and 1 query to verify), which is far worse than the logarithmic query complexity of the FRI protocol.

We will not describe the FRI protocol in detail but briefly overview its essential components. The FRI protocol is inspired by the Fast Fourier Transform (FFT), from where it inherits the word "fast" in its name and many properties. The FRI protocol uses the proof composition introduced for PCPs in [6] adapted for the use with RS to reduce the problem of proximity testing of a code word f^i to the RS code $RS[F, S, \rho]$ to the problem of testing the proximity of a linear combination of two code words f^{i+1} to $RS[F, S', \rho]$ where $|S'| = \frac{1}{2} \cdot |S|$, which effectively is a problem of half size involving low-degree testing of a polynomial of degree $< \rho \cdot |S| \cdot \frac{1}{2}$. Proof composition refers to encoding proofs that permit random access to different parts that can be composed instead of reading (and re-reading) larger parts of a proof. The PCP approach was adapted to PCPPs in [22, 49, 24]. An essential property of this proof composition is distance preservation meaning that if f^i is δ^i-far from all code words, $\Delta^*(f^i, RS[F, S, \rho]) \geq \delta^i$, then it must be the case that f^{i+1} is δ^{i+1}-far from all code words, $\Delta^*(f^{i+1}, RS[F, S, \rho]) \geq \delta^{i+1}$, where

the reduction in distance between rounds is negligible. The FRI protocol improves significantly on distance preservation compared to previous constructions for PCPPs and IOPPs based on the bivariate testing Theorem of Polischuk and Spielman [102], which suffers a constant multiplicative loss in the distance per round of proof composition ($\delta^{i+1} \leq \delta^i/2$).

Proof composition of the RS code $RS[F, S, \rho]$ to $RS[F, S', \rho]$ is called *folding* and can maximally be repeated until the degree of the polynomial is 0. The FRI protocol relies on the following three observations [28].

1. Decomposition of polynomials: Any polynomial f where $deg(f) = d$ can be split into two polynomials f_e and f_o where $deg(f_e, f_o) < \frac{d}{2}$ such that:

$$f(x) = f_e(x^2) + x * f_o(x^2) \tag{10}$$

This property is also used for using FFT to multiply polynomials where $f_e(x) = \frac{f(x)+f(-x)}{2}$ represent the even coefficients and $f_o(x) = \frac{f(x)-f(-x)}{2x}$ the odd coefficients. While f is a polynomial over the domain $S \subseteq \mathbb{F}$, the functions f_o and f_e have domain $S^2 = \{x^2 \mid x \in S\}$, where $S^2 \subseteq S$ since $x^2 = (-x)^2$.

2. Randomised folding/hashing: Let S be a multiplicative group of order 2^r with the generator ω. Let f be a polynomial over S with degree d. Then, define $S' = \langle \omega^2 \rangle = \{x^2 \mid x \in S\}$ such that $|S'| = |S|/2$. Let $\pi : \mathbb{F} \to \mathbb{F}$ be the map defined such that $\pi(S) = S'$. The structure of the evaluation domain allows the reduction of the proximity problem to one of half the size at each round of interaction.

Based on the decomposition property of polynomials, a folding operator $Fold[f, z] : \mathbb{F}^S \to \mathbb{F}^{S'}$ is defined. The operator maps for any $\alpha \in \mathbb{F}$ a code word in $RS[\mathbb{F}, S, \rho]$ to a code word in $RS[\mathbb{F}, S', \rho]$. The folding is defined as follows:

$$Fold[f(x), \alpha] = g(x^2) + z * h(x^2) \tag{11}$$

where $deg(g) < d/2$ and $deg(h) < d/2$ if $deg(f) = d$. Moreover we have the relationship $f(x) = g(x^2) + x * h(x^2)$.

3. Folding is Distance preserving: Except with small probability over α, the folding of a polynomial f is distance preserving. Formally,

$$\Delta^*(f, RS[\mathbb{F}, S, \rho]) \geq \delta \Rightarrow \Delta^*(Fold[f, z], RS[\mathbb{F}, S', \rho]) \geq (1 - o(1))\delta \tag{12}$$

FRI follows a simple repeating pattern of r rounds of interaction divided into two phases Commit and Query. The verifier sends in each round a random challenge $\alpha \in \mathbb{F}$ to detect if $f^i \in RS[\mathbb{F}, S^i, \rho]$. The prover answers by committing to a new code word/oracle function $f^{i+1} : S^{i+1} \to \mathbb{F}$, which it claims to be equal to $Fold[f^i, z] : S^{i+1} \to \mathbb{F}$. Each round reduces the problem by half, eventually leading to a function f^r evaluated over a small enough evaluation domain. This induces a sequence of Reed-Solomon codes of strictly decreasing length. Nevertheless, the code ρ and the relative minimum distance remain unchanged. Finally, the verifier ensures that f^r belongs to the last RS code.

The verifier checks the consistency between related code words f^i and f^{i+1}. The verifier queries f^{i+1} in a single point $z \in S^{i+1}$ and selects two corresponding points $z_+, z_- \in S^i$ such that $\pi(z_+) = \pi(z_-) = z$. The consistency check interpolates between the two points $(z_-, f^i(z_-))$ and $(z_+, f^i(z_+))$ to create the maximally one degree polynomial $p(x)$, that should pass through all three points $(z_-, f^i(z_-))$, $(z_+, f^i(z_+))$ and $(z, f^{i+1}(z))$. Efficient implementations save query complexity and boost soundness by reusing the oracle queries to test the consistency of f^{i-1} vs. f^i and consistency of f^i vs. f^{i+1}. Efficient implementations use the Fiat-Shamir transform to achieve a non-interactive protocol.

3.8 Game theory approach

In the search for lower-cost approaches to verifiable computation, an approach based on game theory and smart contracts has been proposed. Lowering costs is based on limiting (or avoiding) heavy cryptographic protocols. In cases that utilise this approach, a client outsources the computation to multiple parties while also creating distrust between the computing parties.

An approach that considered outsourcing computation to two execution parties while creating distrust between them by use of smart contracts [126] have shown promising results [50]. In general terms, the approach avoids cryptography to remove the possibility of collusion between computing entities. At the same time, it focuses on utilising economic means to create distrust between the computing entities, which leads to the prevention of collusion. Collusion prevention is essential in this approach as it is based on *replication*, meaning that it outsources the same problem to multiple computing entities and compares results from them. In case of collusion, several computing parties can collude to provide a wrong result, which could be undetected by the client that ordered the computation. The approach works by making collusion the least favourable option for computing entities.

The computing entities are asked to sign two contracts, a so-called *Prisoner contract* and a *Traitor contract*. The prisoner contract is created between the client ordering the computation (paying a fee cl_c) and two execution parties, where the execution parties take a fee c to carry out the computation. The incentive is based on a deposit that each cloud is asked to pay in advance that will be refunded in case of correct computation. In addition, the contract specifies cases where an honest party receives the deposit confiscated from the cheating party. This leads to a higher payoff for honesty than collusion and incentivises one party to try to get the other party to cheat while remaining honest. This causes instability for collusion since both parties are aware of the incentive structure. The parties further agree on specific deadlines T_i to enforce timeliness and ensure that the parties move forward in execution. If the dispute arises between the two parties, it needs to be settled by a trusted third party $\mathcal{T}$ that takes a fee $c(\mathcal{T})$. The computation problem $f()$, including the input x provided by the client to the executing parties, could or could not be hidden depending on the cost sensitivity of the ordering client. Each of the executing entities is asked to pay a deposit d where the game analysis has demonstrated that given that $d > c + c(\mathcal{T})$ the game reaches an equilibrium where both parties execute $f(x)$ with probability 1 and return a correct result while respecting T_i.

Before exploring the traitor contract, it is essential to explore a situation where executing entities create a so-called *Colluder's contract*. The Colluder's contract is introduced to create a position where the executing parties could make enforceable promises to each other and be able to subvert the prisoner contract, hence making collusion an attractive option again. This contract introduces a bribe $\mathcal{B}$ paid by the entity that has initiated the collusion and a collusion deposit $d(c)$ that both parties pay and the party that does not execute the collusion loses. The contract needs to be taken into effect before the solutions from the executing parties are provided back to the client. The game analysis has shown that upon satisfying conditions such as $\mathcal{B} < c \wedge cl_c - \mathcal{B} > cl_c - c$ then the collusion provides the highest payoff and hence it is often utilised.

Finally, to avoid the issue of colluding parties, the traitor contract provides a solution. This contract allows an execution entity to be exempt from paying a penalty for collusion specified within the prisoner contract, providing an incentive for reporting the collusion. If collusion is reported it is up to the $\mathcal{T}$ to determine who has cheated in computation. This contract allows for one party to betray the other party secretly, and the betrayal is risk-free. Hence, this contract destabilises the potential for collusion by creating distrust between execution entities. The game analysis based on including all three contracts has shown that execution entities eventually behave honestly. Most of the time, no execution party wants to initiate collusion.

A variation on this approach could be utilised in a setting with more than two execution

parties that can achieve strict Nash equilibrium given a correct set of incentives [82]. The client first *randomly* chooses different execution entities and then assigns a job to them. Once the job is completed, the client checks if the provided answers match. If this is the case, all execution entities are rewarded. If the answers do not match, the client dispatches the same problem to another randomly selected population of execution entities and repeats the step until a matching result is obtained from all of the execution entities. This is then further utilised to punish, i.e. fine the execution entities that returned wrong results. To ensure that the execution entity can pay a fine in case of a wrong result, the client first checks all of the execution entities it is about to hire for funds available to ensure this payment. One issue is in the case of colluding execution entities, where all entities collude to provide the same answer to the client. To do that, however, all the colluders would need to keep some budget for a fine; otherwise, they would not be hired. This means they would need to behave honestly, at least sometimes, to keep their budget from becoming negative. This led to a bound of the maximum damage colluders can cause to the client could be evaluated as extra work defined as $rgm/(r + f)$, where r is the reward for the execution entities, g is the amount of colluding entities, m is the total amount of execution entities, and f is fine for wrong computation.

While the approaches based on incentives and game theory could assure that the computation is carried out correctly, it often requires a certain level of re-execution (either local or with another set of remote execution entities). Hence, it is not clear of overhead. Another issue that this approach does not solve is hiding the function and the inputs being computed. This still requires the use of cryptographic methods described previously.

4 Practice in Verifiable Computations

The ultimate goal of the research in verifiable computation is to enable computation outsourcing to third parties and obtain the computation results at an asymptotically lower computational cost than performing the actual computation locally. Nevertheless, given the additional overhead the proof system adds, achieving this goal is a challenge to the theoretical foundations because devising an efficient proof system implementation is far from trivial, to the point that authors have characterised existing implementations as "near practical" [122]. In this section, we report on our findings after performing experiments to study the feasibility of using verifiable computations toolchains as a component off-the-shelf. More precisely, given a user-defined program, is it possible to use the toolchain to produce a prover, a verifier, and use a verifiable computation proof system without hacking its implementation?

4.1 Implementations: The Bottleneck

A proof system for verifiable computation translates the original computation (obtain the result y from running P on input x) into (abstractly[11]) three computation steps:

1. The verifier setup: producing a specification Σ.

2. The prover computation: producing $y' = \Sigma(x)$ and a certificate π.

3. The verifier check: checking the logical assertion: $\pi \Rightarrow y = y'$.

These steps have varying complexities depending on each approach but are still not generally applicable. According to the latest accounts [121], the prover computation (step 2) adds an overhead of at least $10^7 * T$ (where T is the time to compute y), the verifier adds at least a linear computation overhead on the input and output to check the proof π (step 3) and also

---

[11]Given the diversity of proof protocols there may be more or fewer steps involved.

incurs a cost proportional to running the program when transforming the original program P (step 1) into a specification Σ, a suitable description of P in the computational model supported by the prover. Step 1 is necessary, given that developers very seldom specify their programs using any of the computational models required by the proof systems, e.g., arithmetic circuits or polynomials. Usually, a program is first specified in a high-level programming languages like Java, C, or Python.

The implementation systems available as mature research contributions, and the ones we cover in our study, consist of toolchains transforming the program P into prover and verifier as toolchain elements that work as a "compiler", which typically accepts some subset or superset of C. The compilation process may be architecturally divided into front-end and back-end components. The front end compiles the program into one of the above-described models suitable for probabilistic checking. The technique for turning a computation into a constraint system involving native operations over a finite field is called *arithmetisation*. The back end is an interactive proof or argument system that is applied to verify the correctness of the statement.

4.2 Surveying Implementations: A Selection of Tools

We covered tools with a publicly available version-control repository. We selected an assortment that fits our project goal: to assess the usage of research projects as off-the-shelf in projects where verified computation would run in practice. In the following, we introduce the particulars of each tool we evaluate.

Pequin. This tool is the amalgamation of several implementations of the theoretical results in [109, 110, 117, 30, 119] produced over several years of research by the Pepper project, which began with the original Pepper tool. It is a refactored version of the Pepper library made more readable and easily accessible, featuring easy-to-use Docker files and tutorials. Pequin has the option to run with the implementation of five different implementations: **Ginger**, **Zataar** [117], **GGPR12**, **Pinocchio**, and **PinocchioZK**. These protocols can be set from inside the utility header file. This allows the user of the tool to quickly switch between the different protocols based on the requirements of the tool. It is very intuitive to work with Pequin since writing proofs is done via a simple *compute* function that takes some inputs and outputs that the user can define. Pequin is being maintained and updated. It has ongoing issues being handled, making it an active and maturing tool.

Cairo. This is a Turing-complete, production-ready language for creating zk-STARKs [62]. It was written and maintained by **StarkWare Industries**. It has a large and growing community around it. The main purpose of Cairo is to be the language for creating provable programs. Cairo cannot verify an application since it is only the language for creating proofs. It can be extended with the Cairo-SHARP functionality to send the proof to an external server, which will then verify the proof.

Libiop. This library provides zk-SNARK constructions that are transparent and post-quantum, which rely only on lightweight symmetric cryptography [19]. It uses IOPs based on designs from the tools Ligero [2], Aurora [19], and Fractal [40]. The tool creates zk-SNARKs with IOPs via a BCS transformation (Ben-Sasson-Chiesa-Spooner transformation [20]). The BCS transformation preserves proof of knowledge and zero knowledge of the proof based on the behaviour of the IOP. The library is labelled as being *not production ready* and having a somewhat slow issue-handling cycle. The latest activity is from April 2022, so it would suggest that still, some maintenance is happening in the library.

Bulletproofs. The Bulletproofs library implements the protocol in [31] and provides implementations of both single-party proofs of single or multiple ranges and online multi-party computation for range-proof aggregation. There is a programmable R1CS API for expressing, proving, and verifying proofs of arbitrary statements. The tool is not production ready, and the R1CS API feature is marked as experimental and unstable. The main benefit of Bulletproofs is the short proof size (logarithmic in witness size) without a trusted setup. Proof generation and verification times are linear.

The library provides the fastest implementation of the Bulletproof protocol that we could find and is written purely in Rust. The tool uses elliptic curves and relies on the discrete logarithm assumption making it vulnerable to quantum adversaries. The original paper's authors, however, propose a solution to this dilemma.

Zilch. This framework is for deploying transparent zero-knowledge proofs without trusted third parties [95]. The framework utilises zk-STARKs and relies on collision-resistant hash functions and the random oracle model, making it post-quantum resilient. The framework uses a custom abstract machine (zMIPS) adopted from the general-purpose MIPS processor architecture (Multi-party Interactive Proof System) to create arithmetic circuits and enable zero-knowledge proofs. Zilch uses a custom, high-level language called ZeroJava (a subset of Java) that supports object-oriented programming, albeit with a few limitations. A cross-compiler from ZeroJava to zMIPS and an assembly optimiser can be found in the repository.

LEGO-SNARK. This implementation is a library that works off the premise of having modular SNARK designs[33]. It runs off a general composition tool that creates zk-SNARKS. It has the option via an additional layer of *lifting tools* and *gadgets* that it uses to be modular in the way that it can exchange the way QAPs are made and operate akin to that of other known zk-SNARK's.

4.3 Experimental Evaluation of Tool Usability

To assess the usage of the tools as off-the-shelf, we tried all of them by installing the latest version on stock hardware and ran a batch of tests/checks to assess several criteria. We followed the steps prescribed for each installation in a Linux OS based on the Ubuntu distro (version 16.04 to 20.04). All tools were run from a docker container to provide a standard benchmark for easiness of installation and to keep the running environment controlled, increasing the possibility of reproducing the experiments. The stock hardware comprised an M1 chip laptop and a standard i7-based platform, which did not affect the experiences at this stage due to containerisation.

Evaluation Criteria. The criteria used to judge each tool is presented in the first column of Table 1. The first judgement was made on its *Compilation of Framework* and to what extent a user can run the framework (from impossible to use due to unmet dependencies to a push button container execution). The *Documentation*, *Ease of Entry*, *Community* and *Support* criteria provide an indicator to what degree the user will find help in overcoming the typical issues encountered when adopting a tool (e.g., finding troubleshooting instructions for an error). Typically, adopting a tool leads to the need to adapt it to the particular problem it is being applied to. As a result, the user needs to have a good grasp of the tool's inner workings to obtain a practical solution to that problem. Thus, judging the degree of *Complexity* of a tool allows a future user to estimate how easy it will be to adapt such tool examples to their particular computation scenario. The *Maturity* criteria were used to judge how the protocols had grown since their original publication. It clarified which tools were being maintained and updated and

which had not matured since the paper's initial publication. The *Dependencies* criteria provide insight into the coupling the codebase had. With a lot of external dependencies comes even more maintenance. Finally, the *Expressiveness* criterion judges the capabilities of the tools. Whether they support complex operations such as flow control and memory management or if it was simple compile-time operations only.

Rank	1	2	3	4	5
Compilation of Framework	It could not be compiled	Many problems when compiling. A lot of manual adjustments needed	Problems when compiling, and few to non adjustments to files needed	Minor problems, and few to none adjustments to files needed	Easy compilation. Potentially even a Dockerfile
Documentation	No documentation	Little and vague documentation	Documentation with sufficient descriptions	Wiki with a lot of documentation	Wiki and related guides. Continuously updated.
Ease of Entry	No clear instructions or examples on how to create new proofs	Vague instructions with no examples on how to create new proofs	Complex instructions with no examples on how to create new proofs	Somewhat simple instructions with examples on how to create new proofs	Easy examples with instructions on how to create new proofs
Community	No community	No active community	Small community with very limited activity	Some activity but it might take time to get help	Active community. Possible to get help and communicate ideas
Support	No support and not possible to contact developers		Minimal support, an associated email to contact is given		Easy to get in contact with. Active support with forums or discord
Complexity	Code base is very hard to understand	Hard to understand	Somewhat easy to understand	Easier to understand	Easy to understand
Maturity	No handling of issues and pull requests	Acknowledges issues in documentation	Must have issue handling	Issues, but have been acknowledged	Very few issues, and issues are actively being handled
Dependencies	Many required, external dependencies.	Some required external dependencies	Few required external dependencies	Optional dependencies or 1-3 required ones	No external dependencies
Expressiveness	Special purpose	Pure	Stateful	General control flow	OOP

Table 1: Criteria used to rank the usability of the different tools in terms of its COTS readiness

4.4 Results

Our hands-on experiments are in line with the accounts in the literature, where though full compiler toolchain implementations exist, they are somehow experimental and too brittle to be deemed general computation ready.[12] The breakdown of the results of our experiments with each of the tools can be consulted in Table 2, and in the following, we elaborate on the contents grouped by each line/tool. Unless mentioned directly in the separate paragraphs, all of the tools were labelled as either not production ready or for experimental use.

Pequin. The *compilation* of this tool could be left to a shell script that would use an existing *Dockerfile* to set up a container with the many dependencies required for the tool. A well-documented setup guide helped with the navigation of the library and the creation of new verifiable programs. This made the more essential parts of *Pequin* easier to understand immediately. A moderate-sized community is available should one encounter issues or have questions regarding the tool. Should this not suffice, the administrating team behind the pepper project encourages one to contact them at their mail address for further help with their tool. Many working examples in the library highlight that the tool supports many distinct high-complexity programs, even some requiring memory management.

Cairo. The tool can be obtained through Python's package manager *Pip*. To set up a working environment for *Cairo*, a *Dockerfile* was available in the GitHub repository, although we

---

[12]As of November 2022, a lot of the issues stem from the fundamental way the different libraries have been written: they are not modular and not easy to apply to other computational programs. Many libraries are based on LibSNARK, which provides good front and back ends (if both are chosen from libSNARK), but other implementations fall short because of this issue.

experienced some issues with it on non-UNIX systems. *Cairo* is still being developed, and the team behind it just recently posted about releasing its first stable production-ready environment. This means that a developer might encounter bugs along the way. Thankfully, *Cairo* has a large community that covers both a forum and a discord community with over 32.000 members. Besides these points, *Cairo* provides excellent documentation on how the language can be used to develop programmable proofs, with many examples in their playground. Verifiable programs are created via the *Cairo* compiler, which takes a *.cairo* file with some main functionality. It is worth noting that proof cannot be verified locally on the client side. The proof must be sent to an external server known as SHARP to verify a computation.

Libiop. This tool is an implementation that produces transparent, post-quantum and symmetrically encrypted proofs, with the use of Interactive oracle proofs based on three internal tools: *Ligero*, *Aurora* and *Fractal*. Installing the library had its initial issues. *Gtest*, which is Google's test framework, was from an older release that did not support the compilation of the release version of the library. The solution was to update *Gtest* to a newer release. Testing the individual tools was straightforward since several compiled tests for the different internal tools were available as bash scripts on the compilation. Though this did not give a clear insight into how the library was set up and if it even offered the option to create new proofs with the library.

Bulletproofs. A pure Rust implementation by Dalek cryptography was tested. Initially, there were a few versioning issues with the tool's setup. Attempting to run the given test suite produced errors such as "attributes are not yet allowed on if-expressions". A *fix* was to change the enforced nightly build in the *rust-toolchains* file to a current version. This, however, had some side effects, as certain macros and methods used in the library were now flagged as deprecated. The deprecated functionality was manually updated to modern standards in the required files. The tool is well-documented. The codebase is thoroughly commented on, and the developers have provided a wiki that describes structs, modules, etc., used in the implementation. Additionally, the mathematics behind the protocol is described in detail with a change in notation from the original Bulletproofs paper. The library does not have a lot of dependencies. Using rustup to install rustc, cargo, and other standard Rust tools was sufficient for us. Creating and running new programs heavily resembles the traditional Rust process.

Zilch. After installing the required dependencies, the tool was easy to start with as the Readme thoroughly describes how to use the ZeroJava compiler, the compilation process, and the zMIPS extension to the MIPS ISA. Unfortunately, the documentation related to the syntax of ZeroJava is not given the same level of attention to detail. Small things like not being able to add or multiply more than two integers without including parentheses, or being unable to add two array values together in a single line, are not mentioned in the documentation. Also, sometimes not defining new variables at the top of the program caused compilation to fail. Despite these inconsistencies, Zilch is one of the few tools that support object-oriented programming. However, ZeroJava does have a few limitations compared to modern object-oriented languages. For instance, it does not support interfaces or function overloading; it only has a single inheritance. Furthermore, constructs such as for-loops and 2D arrays are not supported. It is also worth noting that some of the tests in the repository might not work correctly. For instance, the algorithm used for matrix multiplication is flawed and failed to verify. The matrix multiplication failed to verify even after adjusting the algorithm so that the correct matrix product was computed.

LEGO-SNARK. The main problem with this tool was that the dependency on libSnark was from an older release that was no longer compatible with the implementation of the current state of the tool. Therefore it was required to manually go in and change the release version of libSnark to the newest release so that it would work with the tool. Running examples in the library was straightforward after the Makefile had run its magic, though few examples were present. The code itself did not give a very clear inside of what was happening in each example. Therefore recreating proofs or creating new ones is rather complicated. There was also an issue with a utility file. Due to a deprecated function from an external library, the **util** code had to be updated to be able to compile the examples. From the paper, LEGO-SNARK sounds smart since it has a modular SNARK design, but in practice, this is not intuitive to work with. We had a hard time understanding how to extend the library with new functions since the repository or the paper does not go into depth about how the original author got to the example proofs that the library holds.

4.5 Discussion

Overall, none of the works provides an off-the-shelf framework. For the surveyed tools, an adopter has to build its client-server architecture and reuse components to implement the proof system or derive the program specification Σ. Even in cases where the toolchain provides a Σ, we experienced that it is easy to find unsupported corner cases where the compiler cannot find a Σ for a simple C routine making additions, for example. Below we break down our assumptions and conclusions after using each tool.

Pequin. The idea of Pequin was that it was supposed to be an easier implementation of the original library titled Pepper. A docker file was available for easier implementation, together with several bash scripts to fetch dependencies and READMEs to help with both the installation and the actual testing of the tool. Pequin delivered an easy setup with a very user-friendly way to start making your proofs.

Cairo. The idea of Cairo was to create a new language from scratch, which would be utilised to make provable programs. It also had a connected verification server in the form of Cairo-SHARP, which would allow for verification of the provable programs.

Libiop. The idea of libiop was that it would utilise interactive oracular proofs. This would help make the libraries quantum-safe, transparent and lightweight. It consisted of 3 internal libraries (Ligero, Aurora and Fractal) with a test program. Working with libiop was not very informative as to how tests were made; it was obvious to see the tests run and the computation being proved and verified. When trying to create new proofs, it becomes much more complicated. It is not an intuitive setup to create new proofs and can therefore require a bit of further reading.

Bulletproofs. The primary goal of Bulletproofs was to allow for efficient range proofs on committed values – a type of problem that is not limited to but occurs frequently in blockchain-based cryptocurrencies. The proof protocol has short proofs that are only logarithmic in the witness size and supports multiparty computation. Beyond range proofs, the tool also computes inner product proof. The underlying protocol is only designed to do a limited number of things, so unlike other tested tools, Bulletproofs only allow for high-level implementations of range and inner product proofs. To prove and verify arbitrary statements, one must use the R1CS

API, which is flagged as experimental. However, creating and running programs requires little domain-specific knowledge, especially if one is familiar with Rust.

Zilch. The main idea of the Zilch tool is to abstract the encoding of computations as arithmetic circuits away from the user as debugging and optimising programs, as circuits are unfamiliar to most users and require domain-specific knowledge. Instead, Zilch is one of the few tools that uses a high-level language that supports an object-oriented programming style. The ZeroJava compiler in the repository creates and runs new programs similar to compiling and running a regular Java program. However, syntactically the parser was very strict and not all the behaviour is documented. Furthermore, not all the tests in the repository worked as intended – like the matrix multiplication. Specific programs are a hassle to write in ZeroJava due to the number of inconsistencies hinted at in the result section and the lack of constructs such as for-loops and 2D arrays.

LEGO-Snark The idea of LEGO-Snark is to have a modular design, which allows for interchangeable SNARK protocols. After compiling the library, a few examples were available for testing. Looking through the code of the examples made it clear that only three commands were used to produce the proof and the verification. Still, a guide on using these functions was not present.

Developer	Tool	Compilation	Documentation	Entry	Community	Support	Complexity	Maturity	Dependencies	Expressiveness	Overall
	Ginger	3	2	3	3	3	3	3	2	4	**2.89**
Pepper Project	Zaatar	3	2	3	3	3	3	3	2	4	**2.89**
	Pequin	5	4	5	4	3	5	4	2	4	**4.00**
StarkWare Industries	Cairo	5	5	5	5	5	4	3	3	4	**4.33**
Scipr-lab	Libiop	4	3	3	4	3	3	3	3	4	**3.33**
Dalek cryptography	Bulletproofs	3	4	4	4	3	4	4	4	4	**3.78**
Trustworthy Computing Group	Zilch	4	2	4	1	1	2	1	2	4.5	**2.39**
IMDEA-software	LEGO-SNARK	3	3	2	1	3	2	3	3	3	**2.56**

Table 2: Implementation scores for the evaluation criteria.

5   Conclusions

It is not easy to check a mathematical proof [113]. We may suspect an error but have no counterexample to use to reject the proof. Scanning through the proof, flipping the pages randomly and reading a line here and there would be ideal. But could we ever be confident that we could find the errors in such a superficial way? The usual presentation of proof is not like this. It is easy to write proof of a false assertion where the mistake is subtly hidden.

But we do not have to present proofs traditionally. The last 25 years have seen the development of new formats for writing mathematical proofs. These new formats rely on profound results in theoretical computer science. These results underpin probabilistic algorithms for validating the proofs of general assertions. A checker does not need to review an entire proof. Instead, they need to read only a constant number of bits to gain confidence in the proof. Someone who proves a theorem can rewrite the proof in the new format. They can then give it to the checker with the assurance that the reviewer will believe this new proof. A malicious person might claim a false assertion is a theorem. Since it is false, there can be no proof of its correctness. If they propose a pretend proof, the checker will discover an error with near certainty. It does not matter whether the so-called proof is in the new format.

This report has surveyed the state of the art and summarised the theoretical foundations that underlie this result. This includes the PCP theorem. It also provides for the notion

of zero-knowledge proofs used to shield clients working with PCP, with a summary of recent constructions for deploying zero-knowledge proof systems in practice. Remarkably, a statement can be proven correct in zero-knowledge with such short proofs and efficient prover and verifier algorithms. As Sudan points out [113], PCPs make verification more robust.

Our work also surveyed the practice in the field of verifiable computations. We tested several implementations of the state-of-the-art proof systems put forward in recent research papers and ranked them according to several criteria to evaluate whether a development team can use the implementation as a component off-the-shelf, i.e., without developing the implementation of the proof system itself. Overall, none of the implementations found provides an off-the-shelf framework. When using an implementation, the user has to build its client-server architecture. The most advanced implementations provide a toolchain compiling user-defined programs into a prover and verifier. Still, we experienced that finding unsupported corner cases where the toolchain cannot deal with a specific program is easy.

Our test results indicate that Pequin, a tool that refactored several tools [122] and packaged them into a single toolchain, is an excellent candidate to transform into a verifiable computation framework. In particular, we asserted that. Pequin is currently being developed and packaged in a container ready to use. Nevertheless, there are still some pieces to be put together if the codebase is to be a ready-to-use component. Another tool faring well in our experiments is Cairo. It is also being developed, and we consider it another good candidate for choice as an off-the-shelf component. Our survey covered some tools that are not being actively developed: Libiop, Zilch and Bulletproofs, but the research results they illustrate are interesting from a user perspective. For instance, Libiop reports highly efficient and transparent proofs that may interest our application domain. LEGO-SNARK was chosen because its modularity allows different implementations of the zk-SNARKS in the back end. Nevertheless, only Libiop and Bulletproofs fare well in our ranking.

6   References

[1] Mohammed Alhamad, Tharam Dillon, and Elizabeth Chang. SLA-based trust model for cloud computing. In *2010 13th International Conference on Network-Based Information Systems*, pages 321–324, 2010.

[2] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the 2017 acm sigsac conference on computer and communications security*, pages 2087–2104, 2017.

[3] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries—breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 843–862. IEEE, 2017.

[4] Diego F. Aranha, Youssef El Housni, and Aurore Guillevic. A survey of elliptic curves for proof systems. Cryptology ePrint Archive, Paper 2022/586, 2022. To appear in Designs, Codes and Cryptography. eprint.iacr.org/2022/586.

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

[6] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of NP. *J. ACM*, 45(1):70–122, 1998.

[7] Will Arthur and David Challener. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress, USA, 1st edition, 2015.

[8] László Babai. Trading group theory for randomness. In Robert Sedgewick, editor, *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 421–429. ACM, 1985.

[9] Mira Belenkiy, Melissa Chase, C. Chris Erway, John Jannotti, Alptekin Küpçü, and Anna Lysyanskaya. Incentivizing outsourced computation. Cryptology ePrint Archive, Paper 2013/156, 2013.

[10] Mihir Bellare, Oded Goldreich, and Madhu Sudan. Free bits, PCP and non-approximability — Towards tight results. *Electron. Colloquium Comput. Complex*, 2(24), 1995.

[11] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993*, pages 62–73. ACM, 1993.

[12] Michael Ben-Or, Shafi Goldwasser, Joe Kilian, and Avi Wigderson. Multi-prover interactive proofs: How to remove intractability assumptions. In *STOC*, pages 113–131. ACM, 1988.

[13] Eli Ben-Sasson. Course 236603: Probabilistically checkable proofs. Department of Computer Science, Technion IIT, eli.net.technion.ac.il/files/2013/03/notes_2007_Fall.pdf, Fall 2007–8.

[14] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast Reed-Solomon interactive oracle proofs of proximity. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPIcs*, pages 14:1–14:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[15] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, page 46, 2018.

[16] Eli Ben-Sasson, Dan Carmon, Yuval Ishai, Swastik Kopparty, and Shubhangi Saraf. Proximity gaps for Reed-Solomon codes. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 900–909. IEEE, 2020.

[17] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society, 2014.

[18] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 90–108. Springer, 2013.

[19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P Ward. Aurora: Transparent succinct arguments for R1CS. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 103–128. Springer, 2019.

[20] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *Theory of Cryptography Conference*, pages 31–60. Springer, 2016.

[21] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf. DEEP-FRI: Sampling outside the box improves soundness. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, volume 151 of *LIPIcs*, pages 5:1–5:32. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[22] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil P. Vadhan. Robust PCPs of proximity, shorter PCPs, and applications to coding. *SIAM J. Comput.*, 36(4):889–974, 2006.

[23] Eli Ben-Sasson, Swastik Kopparty, and Shubhangi Saraf. Worst-case to average case reductions for the distance to a code. *Electron. Colloquium Comput. Complex.*, page 90, 2018.

[24] Eli Ben-Sasson and Madhu Sudan. Short PCPs with Polylog query complexity. *SIAM Journal on Computing*, 38(2):551–607, 2008.

[25] Luís Felipe Bilecki and Adriano Fiorese. A trust reputation architecture for cloud computing environment. In *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*, pages 614–621, 2017.

[26] Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. The hunting of the SNARK. *IACR Cryptol. ePrint Arch.*, page 580, 2014.

[27] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, pages 326–349. ACM, 2012.

[28] Sarah Bordage, Mathieu Lhotel, Jade Nardi, and Hugues Randriam. Interactive oracle proofs of proximity to algebraic geometry codes. In Shachar Lovett, editor, *37th Computational Complexity Conference, CCC 2022, July 20-23, 2022, Philadelphia, PA, USA*, volume 234 of *LIPIcs*, pages 30:1–30:45. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[29] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *J. Comput. Syst. Sci.*, 37(2):156–189, 1988.

[30] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 341–357. ACM, 2013.

[31] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE symposium on security and privacy (SP)*, pages 315–334. IEEE, 2018.

[32] Kevin Bürstinghaus-Steinbach, Christoph Krauß, Ruben Niederhagen, and Michael Schneider. Post-quantum TLS on embedded systems: Integrating and evaluating Kyber and SPHINCS+ with Mbed TLS. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, page 841–852, New York, NY, USA, 2020. Association for Computing Machinery.

[33] Matteo Campanelli, Dario Fiore, and Anaïs Querol. LegoSNARK: Modular design and composition of succinct zero-knowledge proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2075–2092, 2019.

[34] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited (Preliminary Version). In *STOC*, pages 209–218. ACM, 1998.

[35] Ran Canetti, Ben Riva, and Guy N. Rothblum. Practical delegation of computation using multiple servers. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 445–454. ACM, 2011.

[36] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[37] Xiaofeng Chen, Jin Li, Jian Weng, Jianfeng Ma, and Wenjing Lou. Verifiable computation over large database with incremental updates. In Mirosław Kutyłowski and Jaideep Vaidya, editors, *Computer Security - ESORICS 2014*, pages 148–162, Cham, 2014. Springer International Publishing.

[38] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 34–64, Cham, 2018. Springer International Publishing.

[39] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *EUROCRYPT (1)*, volume 12105 of *Lecture Notes in Computer Science*, pages 738–768. Springer, 2020.

[40] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 769–793. Springer, 2020.

[41] Chung, Michaela Iorga, Wei Bao, Ang Li, Qinghua Li, and Antonios Gouglidis. General access control guidance for cloud systems, 2020-07-31 2020.

[42] Samuel Daniel Conte and Carl De Boor. *Elementary numerical analysis: An algorithmic approach*. Number 78 in Classics in applied mathematics. Society for Industrial and Applied Mathematics, Philadelphia, 3rd edition, 2018.

[43] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[44] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In Shafi Goldwasser, editor, *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 90–112. ACM, 2012.

[45] Ivan Damgård. Commitment schemes and zero-knowledge protocols. In *School organized by the European Educational Forum*, pages 63–86. Springer, 1998.

[46] George Danezis, Cédric Fournet, Jens Groth, and Markulf Kohlweiss. Square span programs with applications to succinct NIZK arguments. In *ASIACRYPT (1)*, volume 8873 of *LNCS*, pages 532–550. Springer, 2014.

[47] Richard A. DeMillo and Richard J. Lipton. A probabilistic remark on algebraic program testing. *Inf. Process. Lett.*, 7(4):193–195, 1978.

[48] Irit Dinur. The PCP theorem by gap amplification. In Jon M. Kleinberg, editor, *Proceedings of the 38th Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, May 21-23, 2006*, pages 241–250. ACM, 2006.

[49] Irit Dinur and Omer Reingold. Assignment testers: Towards a combinatorial proof of the PCP theorem. *SIAM J. Comput.*, 36(4):975–1024, 2006.

[50] Changyu Dong, Yilei Wang, Amjad Aldweesh, Patrick McCorry, and Aad van Moorsel. Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 211–227, New York, NY, USA, 2017. Association for Computing Machinery.

[51] Uriel Feige, Shafi Goldwasser, Laszlo Lovász, Shmuel Safra, and Mario Szegedy. Approximating the clique is almost NP-complete. In *Proc. 32nd IEEE Symp. on Foundations of Comp. Science*, pages 34–39. IEEE, 1991.

[52] Uriel Feige, Shafi Goldwasser, Laszlo Lovász, Shmuel Safra, and Mario Szegedy. Interactive proofs and the hardness of approximating cliques. *Journal of the ACM*, 43(2):268–292, 1996.

[53] Dengguo Feng. *Trusted Computing: Principles and Applications*. De Gruyter with Tsinghua University Press, 2018.

[54] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *LNCS*, pages 186–194. Springer, 1986.

[55] Aidan Finn, Hans Vredevoort, Patrick Lownds, and Damian Flynn. *Microsoft private cloud computing*. John Wiley & Sons, 2012.

[56] Dario Fiore, Rosario Gennaro, and Valerio Pastro. Efficiently verifiable computation on encrypted data. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 844–855, New York, NY, USA, 2014. Association for Computing Machinery.

[57] Lance Fortnow. The complexity of perfect zero-knowledge. *Adv. Comput. Res.*, 5:327–343, 1989.

[58] Jun Furukawa and Yehuda Lindell. Two-thirds honest-majority MPC for malicious adversaries at almost the cost of semi-honest. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1557–1571, New York, NY, USA, 2019. Association for Computing Machinery.

[59] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic Span Programs and Succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 626–645. Springer, 2013.

[60] Rosario Gennaro, Michele Minelli, Anca Nitulescu, and Michele Orrù. Lattice-based zk-SNARKS from square span programs. In *CCS*, pages 556–573. ACM, 2018.

[61] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178. ACM, 2009.

[62] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo—A Turing-complete STARK-friendly CPU architecture. *Cryptology ePrint Archive*, 2021.

[63] O. Goldreich, S. Micali, and A. Wigderson. How to play ANY mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, page 218–229, New York, NY, USA, 1987. Association for Computing Machinery.

[64] Oded Goldreich. A taxonomy of proof systems (part 2). *SIGACT News*, 25(1):22–30, 1994.

[65] Oded Goldreich. *Computational complexity — a conceptual perspective.* Cambridge University Press, 2008.

[66] Oded Goldreich and Johan Håstad. On the complexity of interactive proofs with bounded communication. *Inf. Process. Lett.*, 67(4):205–214, 1998.

[67] Oded Goldreich and Madhu Sudan. Locally testable codes and PCPs of almost-linear length. *J. ACM*, 53(4):558–655, 2006.

[68] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. *J. ACM*, 62(4):27:1–27:64, 2015.

[69] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In Oded Goldreich, editor, *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 203–225. ACM, 2019.

[70] Shafi Goldwasser, Guy N. Rothblum, and Yael Tauman Kalai. Delegating computation: Interactive proofs for muggles. *Electron. Colloquium Comput. Complex.*, page 108, 2017.

[71] Shafi Goldwasser and Michael Sipser. Private coins versus public coins in interactive proof systems. In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 59–68. ACM, 1986.

[72] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT (2)*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, 2016.

[73] David Harvey and Joris van der Hoeven. Polynomial multiplication over finite fields in time $O (n \log n)$. *J. ACM*, 69(2):12:1–12:40, 2022.

[74] Johan Håstad. Clique is hard to approximate within $n^1 - epsilon$. *Electron. Colloquium Comput. Complex*, 4(38), 1997.

[75] Johan Håstad. Some optimal inapproximability results. *J. ACM*, 48(4):798–859, 2001.

[76] Jingwei Huang and David M Nicol. Trust mechanisms for cloud computing. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):1–14, 2013.

[77] Yuval Ishai, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient arguments without short PCPs. In *22nd Annual IEEE Conference on Computational Complexity (CCC 2007), 13-16 June 2007, San Diego, California, USA*, pages 278–291. IEEE Computer Society, 2007.

[78] Yael Tauman Kalai and Ran Raz. Interactive PCP. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, volume 5126 of *LNCS*, pages 536–547. Springer, 2008.

[79] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2010.

[80] Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 549–560, New York, NY, USA, 2013. Association for Computing Machinery.

[81] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (Extended Abstract). In S. Rao Kosaraju, Mike Fellows, Avi Wigderson, and John A. Ellis, editors, *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 723–732. ACM, 1992.

[82] Alptekin Küpçü. Incentivized outsourced computation resistant to malicious contractors. *IEEE Transactions on Dependable and Secure Computing*, 14(6):633–649, 2017.

[83] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[84] Jens Lansing and Ali Sunyaev. Trust in cloud computing: Conceptual typology and trust-building antecedents. *SIGMIS Database*, 47(2):58–96, jun 2016.

[85] Helger Lipmaa. Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes. In *ASIACRYPT (1)*, volume 8269 of *LNCS*, pages 41–60. Springer, 2013.

[86] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 39(4):859–868, 1992.

[87] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-Knowledge SNARKs from linear-size universal and updatable structured reference strings. In *CCS*, pages 2111–2128. ACM, 2019.

[88] Saima Mehraj and M. Tariq Banday. Establishing a zero trust strategy in cloud computing environment. In *2020 International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–6, 2020.

[89] Or Meir. IP = PSPACE using error-correcting codes. *SIAM J. Comput.*, 42(1):380–403, 2013.

[90] Peter Mell and Timothy Grance. The NIST definition of cloud computing, 2011-09-28 2011.

[91] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, volume 293 of *LNCS*, pages 369–378. Springer, 1987.

[92] Silvio Micali. Computationally sound proofs. *SIAM J. Comput.*, 30(4):1253–1298, 2000.

[93] Thilo Mie. Short PCPPs verifiable in polylogarithmic time with $O(1)$ queries. *Ann. Math. Artif. Intell.*, 56(3-4):313–338, 2009.

[94] Dana Moshkovitz. The tale of the PCP theorem. *XRDS*, 18(3):23–26, 2012.

[95] Dimitris Mouris and Nektarios Georgios Tsoutsos. Zilch: A framework for deploying transparent zero-knowledge proofs. *IEEE Transactions on Information Forensics and Security*, 16:3269–3284, 2021.

[96] Moni Naor. Bit commitment using pseudorandomness. *Journal of Cryptology*, 4(2):151–158, 1991.

[97] Anca Nitulescu. zk-SNARKs: A gentle introduction. Technical report, Technical report, 2020.

[98] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. *Commun. ACM*, 59(2):103–112, 2016.

[99] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. Bootstrapping trust in modern computers. In *Springer Briefs in Computer Science*. Springer, 2011.

[100] Siani Pearson. Privacy, security and trust in cloud computing. In Siani Pearson and George Yee, editors, *Privacy and Security for Cloud Computing*, pages 3–42. Springer London, London, 2013.

[101] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.

[102] Alexander Polishchuk and Daniel A. Spielman. Nearly-linear size holographic proofs. In Frank Thomson Leighton and Michael T. Goodrich, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada*, pages 194–203. ACM, 1994.

[103] Jaikumar Radhakrishnan and Madhu Sudan. On Dinur's proof of the PCP Theorem. *Bulletin (New Series) of the American Mathematical Society*, 44(1):19–61, January 2007. Article electronically published on September 26, 2006.

[104] Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Constant-round interactive proofs for delegating computation. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 49–62. ACM, 2016.

[105] Mordechai Rorvig. How computer scientists learned to reinvent the proof. www.quantamagazine.org/how-computer-scientists-learned-to-reinvent-the-proof-20220 23rd May 2022. Accessed: 2022-08-01.

[106] Ahmad-Reza Sadeghi, Thomas Schneider, and Marcel Winandy. Token-based cloud computing. In Alessandro Acquisti, Sean W. Smith, and Ahmad-Reza Sadeghi, editors, *Trust and Trustworthy Computing, Third International Conference, TRUST 2010, Berlin, Germany, June 21-23, 2010. Proceedings*, volume 6101 of *Lecture Notes in Computer Science*, pages 417–429. Springer, 2010.

[107] Christian Scheideler. Course notes: Probabilistically Checkable Proofs. www.cs.jhu.edu/~scheideler/courses/600.471_S05/lecture_8.pdf. Accessed: 2022-08-01.

[108] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *Conference on the Theory and Application of Cryptology*, pages 239–252. Springer, 1989.

[109] Srinath T. V. Setty, Richard McPherson, Andrew J. Blumberg, and Michael Walfish. Making argument systems for outsourced computation practical (sometimes). In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012.

[110] Srinath T. V. Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 253–268. USENIX Association, 2012.

[111] Adi Shamir. IP = PSPACE. *J. ACM*, 39(4):869–877, 1992.

[112] Min Jae Song. An introduction to the PCP Theorem. Courant Institute of Mathematical Sciences, New York University. `math.uchicago.edu/~may/REU2013/REUPapers/Song.pdf`, accessed 3rd August 2022.

[113] Madhu Sudan. Probabilistically checkable proofs. *Commun. ACM*, 52(3):76–84, 2009.

[114] Alan Szepieniec and Yuncong Zhang. Polynomial IOPs for linear algebra relations. In Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe, editors, *Public-Key Cryptography - PKC 2022 - 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8-11, 2022, Proceedings, Part I*, volume 13177 of *LNCS*, pages 523–552. Springer, 2022.

[115] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *LNCS*, pages 71–89. Springer, 2013.

[116] Luca Trevisan, Gregory B. Sorkin, Madhu Sudan, and David P. Williamson. Gadgets, approximation, and linear programming. *SIAM Journal on Computing*, 29(6):2074–2097, 2000.

[117] Victor Vu, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 223–237. IEEE Computer Society, 2013.

[118] Riad S. Wahby, Ye Ji, Andrew J. Blumberg, Abhi Shelat, Justin Thaler, Michael Walfish, and Thomas Wies. Full accounting for verifiable outsourcing. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2071–2086. ACM, 2017.

[119] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[120] Riad S. Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 926–943. IEEE Computer Society, 2018.

[121] Riad Samir Wahby. *Concretely Efficient Interactive Proofs and Their Applications*. PhD thesis, Stanford University, 2021.

[122] Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them. *Commun. ACM*, 58(2):74–84, 2015.

[123] Hoeteck Wee. On round-efficient argument systems. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*, volume 3580 of *LNCS*, pages 140–152. Springer, 2005.

[124] Ingo Wegener. *Complexity theory — exploring the limits of efficient algorithms*. Springer, 2005.

[125] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. A zero-knowledge version of vSQL. *IACR Cryptol. ePrint Arch.*, page 1146, 2017.

[126] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475–491, 2020.

[127] Marius Zimand. Probabilistically checkable proofs the easy way. In Ricardo A. Baeza-Yates, Ugo Montanari, and Nicola Santoro, editors, *Foundations of Information Technology in the Era of Networking and Mobile Computing, IFIP 17th World Computer Congress — TC1 Stream / 2nd IFIP International Conference on Theoretical Computer Science (TCS 2002), August 25-30, 2002, Montréal, Québec, Canada*, volume 223 of *IFIP Conference Proceedings*, pages 337–351. Kluwer, 2002.