# Verifiable Computation in Practice: A Technical Report on Building a Secure Sudoku Solver with Zero-Knowledge Proofs

## The Principle of Verifiable Puzzles: An Introduction to Zero-Knowledge Proofs

The challenge of proving knowledge without revealing the knowledge itself is a cornerstone of modern cryptography. Zero-Knowledge Proofs (ZKPs) provide a formal framework for this seemingly paradoxical task. A ZKP is a cryptographic protocol in which one party, the Prover, can convince another party, the Verifier, that a given statement is true, while conveying no information apart from the validity of the statement itself.2 This capability has profound implications for privacy, security, and scalability in digital systems. The classic Sudoku puzzle serves as an excellent and intuitive model for understanding the core principles and mechanics of ZKP systems before delving into their complex mathematical underpinnings.

### Defining the Prover-Verifier Model in the Context of Sudoku

At its core, a Zero-Knowledge Proof operates within an interactive proof system involving two key roles: the Prover and the Verifier.3 When applied to a Sudoku puzzle, these roles are clearly delineated:

* **The Prover (Peggy):** This is the entity, whether a human user or a software algorithm, that claims to possess a valid solution to a given Sudoku puzzle. The Prover's objective is to generate a cryptographic proof that demonstrates their knowledge of this solution. Crucially, this proof must be constructed in such a way that it does not leak any of the numbers in the completed grid.3
* **The Verifier (Victor):** This entity receives the proof from the Prover. The Verifier's role is to check the validity of this proof. The Verifier has access to the public information—the initial, unsolved Sudoku grid—and the proof itself, but never the secret solution. If the proof is valid, the Verifier becomes convinced that the Prover indeed knows a correct solution.3

Within this model, the problem is framed in terms of a **Statement** and a **Witness**.3

* The **Statement** is the public claim being proven. In this case, it is: "A valid solution exists for this specific 9x9 Sudoku grid that is consistent with the provided clues."
* The **Witness** is the secret information that the Prover knows, which makes the statement true. For Sudoku, the witness is the complete 81-cell grid, with all empty cells filled in according to the game's rules.

This framing is significant because solving a Sudoku puzzle is a problem within the complexity class NP (Non-deterministic Polynomial time). This means that while finding a solution can be computationally difficult, verifying a given solution is easy and can be done in polynomial time. It is a well-established result in cryptography that zero-knowledge proofs exist for any problem in NP, provided that one-way functions exist. This theoretical foundation assures us that constructing a ZKP for a Sudoku solver is not just a theoretical curiosity but a practical possibility.

### The Three Pillars of a ZKP System

For any ZKP protocol to be considered secure and functional, it must satisfy three fundamental properties: Completeness, Soundness, and Zero-Knowledge. These properties are not independent features but a tightly coupled logical triad; a failure in one property fundamentally compromises the utility of the others.6 A system lacking soundness allows deception, rendering the proofs meaningless. A system lacking zero-knowledge violates the core privacy promise, making it no different from revealing the secret outright.

* **Completeness:** This property ensures that the system works for honest participants. If the Prover genuinely possesses a correct Sudoku solution and both the Prover and Verifier follow the protocol correctly, the Verifier will always be convinced and accept the proof.3 For a Sudoku solver, this means that a user with a valid solution will always be able to successfully generate a proof that the system recognizes as valid.
* **Soundness:** This property guarantees the security of the system against malicious or dishonest Provers. If the Prover's claimed solution is incorrect (e.g., a row has a duplicate number), they should not be able to generate a proof that convinces an honest Verifier, except with a probability so small it is considered negligible.4 Soundness ensures that the proof is a true "argument of knowledge"; one cannot construct it without possessing the actual witness. This prevents a user from cheating by claiming to have solved a puzzle they have not.
* **Zero-Knowledge:** This is the defining privacy property of the system. The Verifier (or any third party observing the interaction) learns nothing about the secret witness other than the fact that the statement is true.1 In the Sudoku context, the proof reveals that a valid solution exists and is known by the Prover, but it gives away zero information about the specific numbers placed in the empty cells. The formal method for demonstrating this property involves the concept of a "simulator." A simulator is a hypothetical algorithm that can produce a transcript of a proof-verifier interaction that is computationally indistinguishable from a real one, without ever having access to the secret witness. The existence of such a simulator implies that the real transcript contains no more information than a simulated one, and therefore, no knowledge has been leaked.1

### Conceptualizing the Proof: Interactive Analogies

Before examining the complex cryptographic machinery of zk-SNARKs and zk-STARKs, it is instructive to build intuition using physical, interactive analogies. These examples demonstrate how the properties of completeness, soundness, and zero-knowledge can be achieved through clever protocol design, providing a pedagogical bridge to the more abstract computational methods.

* **The Card-Based Protocol:** A well-known physical ZKP for Sudoku uses several decks of standard playing cards.9 The protocol proceeds as follows:
  1. The Prover takes a large 9x9 board. For the public clues, they place the corresponding cards face up. For each empty cell, they place a card corresponding to their secret solution face down. To prevent the Verifier from simply guessing, they actually place three identical cards in each cell (e.g., three "5" cards in a cell that should be 5).
  2. The Verifier then issues a challenge by randomly choosing to check either all 9 rows, all 9 columns, or all 9 3x3 sub-grids.
  3. Suppose the Verifier chooses to check the rows. For each of the 9 rows, the Verifier randomly selects one of the three face-down cards from each cell in that row.
  4. The Prover gathers the 9 selected cards for each row, shuffles each set of 9 cards, and hands them to the Verifier.
  5. The Verifier can now turn over the cards in each of the 9 shuffled packets and confirm that each packet contains exactly one of each card from 1 to 9.

This protocol works because if the Prover is honest (Completeness), every row, column, and block is valid, so any challenge will pass. If the Prover is cheating and has a duplicate number in a row (Soundness), there is a high probability the Verifier will choose to check that constraint (rows, columns, or blocks) and discover the fraud. The protocol is Zero-Knowledge because the Verifier only ever sees a shuffled, complete set of numbers for a given row, column, or block; they never learn which number was in which specific cell.9

* **The Permutation and Commitment Protocol:** Another approach introduces cryptographic concepts like commitment schemes.12
  1. The Prover begins with their valid Sudoku solution. They create a random permutation, or a secret code, for the numbers 1-9 (e.g., all 1s become 7s, all 2s become 4s, etc.). They apply this permutation to their entire solution grid.
  2. The Prover then "commits" to each cell in this permuted grid. A commitment is like putting a value in a locked, transparent box. One can see there's something inside but not what it is. This is done using a cryptographic hash function. For each cell, the Prover combines the permuted number with a secret random value (a "nonce") and hashes the result. They send all 81 of these hash commitments to the Verifier.
  3. The Verifier issues a random challenge, asking the Prover to "open" either a specific row, a specific column, a specific 3x3 sub-grid, or to prove consistency with the public clues.
  4. If challenged to reveal a row, the Prover provides the permuted numbers and the secret nonces for all 9 cells in that row. The Verifier can re-compute the hashes to ensure the Prover didn't change their mind, and they can check that the 9 revealed permuted numbers are all unique.
  5. To ensure the solution corresponds to the original puzzle, the Verifier can also challenge the Prover to reveal the permuted values for all cells corresponding to a specific public clue (e.g., "show me the permuted value for all cells that were a '5' in the original puzzle"). The Prover must reveal them, and the Verifier checks that they are all identical, confirming the permutation was applied consistently.12

This protocol is Zero-Knowledge because the Verifier only ever sees permuted, meaningless numbers. It is Sound because a cheating Prover cannot find a permutation that simultaneously hides their error and remains consistent with the public clues without being caught.12 This method brings us closer to the computational ZKPs, which replace physical actions with cryptographic functions.

## A Comparative Analysis of Modern ZKP Protocols: zk-SNARKs vs. zk-STARKs

While conceptual protocols illustrate the principles of zero-knowledge, practical implementations rely on sophisticated cryptographic systems. The two most prominent families of non-interactive ZKPs in use today are zk-SNARKs and zk-STARKs. The choice between them is not merely technical but a strategic one that reflects a project's long-term priorities regarding security, cost, and decentralization. A project choosing zk-SNARKs optimizes for current conditions like lower on-chain costs and a mature developer ecosystem, while accepting known future risks like quantum vulnerability. Conversely, a project choosing zk-STARKs is "future-proofing" against quantum threats and prioritizing transparency, while accepting higher immediate costs and a less mature toolchain. Understanding their distinct architectures and trade-offs is essential for determining the feasibility and design of a secure Sudoku solver.

### Architectural Overview of zk-SNARKs (Succinct Non-Interactive Argument of Knowledge)

zk-SNARKs represent a major breakthrough in making ZKPs practical for real-world applications, particularly in blockchain systems like Zcash and Ethereum scaling solutions.13 The acronym itself encapsulates its key properties 15:

* **Z**ero-**K**nowledge: The proof reveals nothing about the secret witness.
* **S**uccinct: The proofs are extremely small (often less than 300 bytes), and verification is computationally very fast. This succinctness is critical for applications where proofs must be stored or processed on-chain, as it minimizes storage and transaction costs.15
* **N**on-**I**nteractive: Unlike the conceptual protocols described earlier, a zk-SNARK proof is a single piece of data that the Prover sends to the Verifier. No back-and-forth communication is required, allowing the proof to be published and verified by anyone at any time.2
* **AR**gument of **K**nowledge: The proof is computationally sound, meaning a Prover without knowledge of the witness can only create a valid proof with negligible probability. It is an "argument" rather than a "proof" because its soundness relies on computational hardness assumptions, not absolute mathematical certainty.15

The core technology behind most zk-SNARKs involves advanced mathematics, including cryptographic pairings on elliptic curves (ECC) and polynomial commitment schemes.17 In essence, the computational problem (like Sudoku) is transformed into a system of polynomial equations. The proof demonstrates that the Prover knows a satisfying assignment to these polynomials without revealing the assignment itself. This reliance on ECC, however, is also their primary long-term vulnerability, as the underlying mathematical problems are believed to be solvable by sufficiently powerful quantum computers.17

Prominent zk-SNARK schemes include Groth16, which is highly efficient but requires a new trusted setup for every unique circuit, and PLONK, which uses a universal and updatable trusted setup, offering greater flexibility for developers.19

### Architectural Overview of zk-STARKs (Scalable Transparent Argument of Knowledge)

zk-STARKs were developed to address some of the perceived limitations of zk-SNARKs, particularly the need for a trusted setup and the vulnerability to quantum computers. The name highlights its distinguishing features 13:

* **S**calable: Both the Prover and Verifier's computation times scale very efficiently as the complexity of the problem grows. Prover time is quasilinear, and Verifier time is polylogarithmic, which is exceptionally efficient.15
* **T**ransparent: This is the most significant architectural difference. zk-STARKs do not require a trusted setup ceremony. Instead, they rely on publicly verifiable randomness derived from hash functions. This eliminates the "toxic waste" problem and the need to trust a group of ceremony participants, making the system more transparent and philosophically aligned with decentralized principles.13

The cryptographic foundation of zk-STARKs is also different. Instead of elliptic curves and pairings, they are built upon collision-resistant hash functions and algebraic techniques like FRI (Fast Reed-Solomon Interactive Oracle Proof of Proximity).17 Since well-established hash functions like SHA-256 are believed to be resistant to attacks from both classical and quantum computers, zk-STARKs are considered a plausibly post-quantum secure technology.17

### In-Depth Protocol Comparison: A Multi-Factor Analysis

To make an informed decision for the Sudoku solver project, a direct comparison across several key metrics is necessary. The following table synthesizes the characteristics of both protocols, highlighting the critical trade-offs a developer must consider.

| Feature | zk-SNARK (e.g., Groth16) | zk-STARK | Supporting Snippets |
| --- | --- | --- | --- |
| **Trusted Setup** | Required for most variants (e.g., Groth16, PLONK). Poses a security risk if "toxic waste" is not destroyed. | Not required ("Transparent"). Uses public randomness. | 13 |
| **Quantum Resistance** | Vulnerable. Relies on Elliptic Curve Cryptography (ECC), breakable by quantum computers. | Plausibly resistant. Relies on collision-resistant hash functions. | 17 |
| **Proof Size** | Succinct. Very small (e.g., <300 bytes), making them cheaper to store/verify on-chain. | Larger. Can be several kilobytes, leading to higher on-chain gas costs. | 17 |
| **Prover Complexity** | Linear (O(C)) with the circuit size C. | Quasilinear (O(ClogC)). Slower for smaller circuits but scales better. | 13 |
| **Verifier Complexity** | Linear (O(P)) with public input size P, but constant time for fixed P. | Polylogarithmic (O(log2C)). Extremely fast verification, scales well. | 13 |
| **Cryptographic Assumptions** | Stronger assumptions (e.g., knowledge of exponent, pairings on ECC). | Weaker, more standard assumptions (collision-resistant hash functions). | 17 |
| **Toolchain Maturity** | More mature. Wider adoption, more libraries (Circom, ZoKrates), and developer support. | Less mature. Newer technology, smaller but growing ecosystem (e.g., Cairo). | 15 |

### The "Trusted Setup" Dilemma: Security Implications and Mitigation

The "trusted setup" is one of the most contentious aspects of many zk-SNARK systems. This ceremony is a procedure required to generate a set of public parameters, known as a Common Reference String (CRS) or Structured Reference String (SRS), which are necessary for both proving and verifying.19

* **The "Toxic Waste" Problem:** The generation of the CRS involves creating secret random values. These secrets, often referred to as "toxic waste," must be securely destroyed after the ceremony is complete. If any entity were to retain these secrets, they would gain the ability to forge proofs for false statements. For example, in a cryptocurrency context, they could create money out of thin air. Because of the zero-knowledge property, such forgeries would be indistinguishable from valid proofs, completely breaking the soundness of the system.14
* **Mitigation via Multi-Party Computation (MPC):** To mitigate this single point of failure, modern trusted setups are conducted as large-scale Multi-Party Computation (MPC) ceremonies. The "Powers of Tau" ceremony is a well-known example.25 In an MPC setup, many independent participants each contribute their own piece of randomness to the final CRS. The security of the final parameters relies on the assumption that  
  *at least one* participant in the entire ceremony acts honestly and destroys their secret contribution. As long as this condition holds, the complete set of secrets is unknowable, and the CRS is secure. This distributes trust across a large, diverse group, making the security assumption far more palatable than trusting a single person or organization.25
* **Universal vs. Circuit-Specific Setups:** A further distinction exists between different zk-SNARK schemes. Early and highly efficient schemes like Groth16 require a circuit-specific setup. This means that if a developer makes even a minor change to their program's logic, a completely new, expensive, and time-consuming trusted setup ceremony must be performed. This severely hampers development agility. Newer systems like PLONK and Marlin utilize a universal and updatable SRS. A single, large MPC ceremony can be performed once, and its output can be used for any circuit up to a certain size. This dramatically improves the developer experience and removes a major barrier to adoption.19

## From Logic to Algebra: The Arithmetization of a Sudoku Puzzle

Zero-knowledge proof systems do not operate on high-level logical statements like "every number in this row must be unique." Instead, their cryptographic machinery is designed to work with systems of polynomial equations defined over finite fields.3 The process of converting a computational problem, with its set of logical rules, into this required mathematical format is known as

**arithmetization**. This is often the most challenging and intellectually creative part of developing a ZKP application, as the efficiency of the final proof system is directly dependent on the efficiency of this translation. The "art" of ZKP development lies in this process; a simple logical rule can explode into hundreds of constraints, and the developer's ability to find clever mathematical representations that minimize this count is a primary form of optimization. For most zk-SNARKs, the target of arithmetization is a **Rank-1 Constraint System (R1CS)**.

### The Necessity of Arithmetization

Arithmetization serves as the bridge between the world of computation (code and logic) and the world of abstract algebra (polynomials and finite fields) where ZKPs operate. A program is first broken down into a series of fundamental operations, creating an "arithmetic circuit." This circuit is then converted into a set of constraints that the proof system can understand.3 The goal is to create a system of equations that is satisfiable if and only if the original computation was performed correctly. For the Sudoku solver, this means creating a set of mathematical constraints that are only satisfied by a grid that adheres to all the rules of Sudoku.

### Introduction to Rank-1 Constraint System (R1CS)

R1CS is a format for representing a computation that is particularly well-suited for zk-SNARKs. An R1CS is defined by a set of three vectors (a, b, c) for each constraint. A solution to the system is a "witness" vector w that contains all the inputs, outputs, and intermediate values of the computation. The witness w is considered valid if, for every single constraint in the system, the following equation holds true 29:

(w⋅a)×(w⋅b)−(w⋅c)=0

Here, · represents the dot product. Each of the dot products (w · a), (w · b), and (w · c) results in a single value, which is a linear combination of the witness variables. The structure of the R1CS constraint, therefore, is (linear\_combination\_1) \* (linear\_combination\_2) = (linear\_combination\_3). This format allows for exactly one multiplication per constraint. Any more complex computation must be broken down, or "flattened," into a sequence of these simple steps, often requiring the introduction of new intermediate variables into the witness vector w.30

### Constructing the R1CS for a 9x9 Sudoku

To build a ZKP for our Sudoku solver, we must define a set of R1CS constraints that collectively enforce all the rules of the game. The witness vector w will contain, at a minimum, the 81 values of the solved grid cells, which we can denote as Cij​ for row i and column j.

* Constraint 1: Cell Value Range: Every cell Cij​ must contain an integer value from 1 to 9. A direct comparison like 1≤Cij​≤9 is not a valid R1CS constraint. A more clever algebraic approach is to enforce that each cell value must be a root of the polynomial (x−1)(x−2)...(x−9). Therefore, for each of the 81 cells, we must add a constraint that enforces:  
    
  (Cij​−1)×(Cij​−2)×⋯×(Cij​−9)=0  
    
  This single equation must be flattened into 8 separate R1CS constraints per cell, introducing 7 intermediate variables for each. For example, the first two steps would be:
  1. intermediate\_1 = (C\_ij - 1) \* (C\_ij - 2)
  2. intermediate\_2 = intermediate\_1 \* (C\_ij - 3)  
     ...and so on, until the final product is constrained to be zero.16
* Constraint 2: Row Uniqueness: For each of the 9 rows, we must prove that all cell values within that row are distinct. A direct and robust way to enforce this is through pairwise comparison. For any two distinct cells in the same row, Cij​ and Cik​ (where j=k), their values must not be equal. The constraint Cij​=Cik​ is equivalent to saying that their difference, (Cij​−Cik​), is non-zero. In a finite field, a non-zero value always has a multiplicative inverse. Thus, we can prove inequality by demonstrating the existence of a value inv such that:  
    
  (Cij​−Cik​)×inv=1  
    
  This forms a valid R1CS constraint. For each row, there are (29​)=36 pairs of cells that must be compared. Applied across all 9 rows, this amounts to 9×36=324 constraints just for row uniqueness.32
* **Constraint 3: Column Uniqueness:** The logic for column uniqueness is identical to that for row uniqueness. We must enforce the pairwise inequality constraint for all pairs of cells within each of the 9 columns. This adds another 324 constraints to our system.
* **Constraint 4: Sub-Grid Uniqueness:** Similarly, we must enforce the same pairwise inequality constraint for all pairs of cells within each of the 9 3x3 sub-grids. This adds a final 324 constraints for uniqueness.
* Constraint 5: Public Input Consistency: The Prover's secret solution must be consistent with the public, unsolved puzzle. Let the public puzzle be represented by a grid Pij​, where empty cells are 0. For every cell (i,j) where the puzzle provides a clue (Pij​>0), we must enforce that the Prover's solution matches this clue. This is a simple equality constraint:  
    
  Cij​=Pij​  
    
  This can be written in R1CS format as a linear constraint:  
    
  (Cij​−Pij​)×1=0  
    
  This constraint is only applied for the non-empty cells of the initial puzzle. If a puzzle has 25 clues, this adds 25 constraints to the system.32

### Circuit Complexity and Constraint Count Estimation

By summing the constraints from each rule, we can estimate the total size of our Sudoku circuit. This size is a primary determinant of the performance of the ZKP system, especially the time required for the Prover to generate a proof.

* **Range Checks:** 81 cells × 8 constraints/cell = 648 constraints.
* **Uniqueness Checks (Rows, Columns, Blocks):** 3 groups (rows, cols, blocks) × 9 instances/group × 36 pairs/instance = 972 constraints.
* **Public Input Consistency:** Varies with the puzzle, but typically between 17 and 30 constraints.

The total estimated number of constraints is approximately **1,650**, plus the number of clues. This is a non-trivial but computationally feasible circuit size for modern zk-SNARK frameworks.33 The process of arithmetization reveals a critical trade-off: developer effort and mathematical ingenuity versus final circuit performance. While the pairwise comparison method for uniqueness is straightforward to implement, it results in a large number of constraints. More advanced techniques, such as using polynomial representations of sets, could potentially reduce this count but would significantly increase the complexity of the circuit design.36 This highlights that the most significant optimization challenge in ZKP engineering often lies not in the choice of the cryptographic protocol itself, but in the clever and efficient representation of the problem as a mathematical circuit.

## A Practical Implementation Guide: Building the Solver with Circom and snarkjs

Translating the theoretical R1CS constraints into a working zero-knowledge proof system requires a specialized toolchain. The most mature and widely used stack for zk-SNARK development is **Circom** and **snarkjs**. Circom is a domain-specific language for writing arithmetic circuits, and snarkjs is a JavaScript library for handling the entire ZKP lifecycle, from trusted setup to proof generation and verification.37 This section provides a step-by-step guide to implementing the Sudoku solver using these tools.

The ZKP development workflow is not a simple "code-compile-run" cycle but a multi-stage pipeline. Each stage—Circuit Design, Compilation, Trusted Setup, Witness Generation, Proof Generation, and Verification—is a distinct process with its own specialized tools and cryptographic significance. An inefficiency or error in an early stage, such as a poorly optimized circuit, will have cascading negative effects on all subsequent stages, leading to longer proving times, higher verification costs, and potential security vulnerabilities.

### Setting Up the Development Environment

Before writing the circuit, the development environment must be properly configured. This involves installing the necessary compilers and libraries.

1. **Install Prerequisites:** A modern version of Node.js is required for snarkjs. The Circom compiler is written in Rust, so the Rust toolchain (including cargo) must also be installed.37
2. **Install Circom and snarkjs:** These can be installed globally via npm, the Node.js package manager:  
   Bash  
   npm install -g circom  
   npm install -g snarkjs  
     
   38
3. **Project Setup:** Create a new project directory. Inside this directory, create a circuits subdirectory to hold the Circom source files. Initialize a package.json file and install circomlib, a standard library of useful pre-built circuits and templates.32  
   Bash  
   mkdir sudoku\_zkp && cd sudoku\_zkp  
   mkdir circuits && cd circuits  
   npm init -y  
   npm install circomlib

### Writing the Sudoku Circuit in Circom

The core of the project is the sudoku.circom file, which defines the arithmetic circuit that enforces the Sudoku rules. Circom code consists of templates, signals (inputs, outputs, and intermediate variables), and constraints.39

The main circuit template will take two 9x9 arrays as input: unsolved (the public puzzle) and solved (the private solution).

Code snippet

pragma circom 2.0.0;  
  
include "node\_modules/circomlib/circuits/comparators.circom";  
  
template Sudoku() {  
 // Public input: the initial puzzle grid (0 for empty cells)  
 signal input unsolved;  
 // Private input: the prover's complete solution  
 signal input solved;  
  
 // --- CONSTRAINT 1: Cell values are between 1 and 9 ---  
 component gte\_one;  
 component lte\_nine;  
 for (var i = 0; i < 9; i++) {  
 for (var j = 0; j < 9; j++) {  
 var idx = i \* 9 + j;  
 // solved[i][j] >= 1  
 gte\_one[idx] = GreaterEqThan(32);  
 gte\_one[idx].in <== solved[i][j];  
 gte\_one[idx].in <== 1;  
 gte\_one[idx].out === 1;  
  
 // solved[i][j] <= 9  
 lte\_nine[idx] = LessEqThan(32);  
 lte\_nine[idx].in <== solved[i][j];  
 lte\_nine[idx].in <== 9;  
 lte\_nine[idx].out === 1;  
 }  
 }  
  
 // --- CONSTRAINT 2, 3, 4: Uniqueness in rows, columns, and blocks ---  
 // Helper component to check inequality  
 component is\_not\_equal[972 \* 3]; // 324 for rows, 324 for cols, 324 for blocks  
 var is\_not\_equal\_idx = 0;  
  
 // Check rows  
 for (var i = 0; i < 9; i++) {  
 for (var j = 0; j < 8; j++) {  
 for (var k = j + 1; k < 9; k++) {  
 is\_not\_equal[is\_not\_equal\_idx] = IsEqual();  
 is\_not\_equal[is\_not\_equal\_idx].in <== solved[i][j];  
 is\_not\_equal[is\_not\_equal\_idx].in <== solved[i][k];  
 is\_not\_equal[is\_not\_equal\_idx].out === 0; // Constrain to be not equal  
 is\_not\_equal\_idx++;  
 }  
 }  
 }  
 // (Similar loops for columns and 3x3 blocks would follow here)  
  
 // --- CONSTRAINT 5: Solution is consistent with the puzzle ---  
 component is\_clue\_equal;  
 component is\_clue\_zero;  
 for (var i = 0; i < 9; i++) {  
 for (var j = 0; j < 9; j++) {  
 var idx = i \* 9 + j;  
 // Check if unsolved[i][j] is zero  
 is\_clue\_zero[idx] = IsZero();  
 is\_clue\_zero[idx].in <== unsolved[i][j];  
  
 // Check if solved[i][j] equals unsolved[i][j]  
 is\_clue\_equal[idx] = IsEqual();  
 is\_clue\_equal[idx].in <== solved[i][j];  
 is\_clue\_equal[idx].in <== unsolved[i][j];  
  
 // If unsolved[i][j] is NOT zero, then solved[i][j] MUST equal it.  
 // This is enforced by: is\_clue\_zero.out \* (1 - is\_clue\_equal.out) === 0  
 // This means either the clue is zero, or the values are equal.  
 (1 - is\_clue\_zero[idx].out) \* (solved[i][j] - unsolved[i][j]) === 0;  
 }  
 }  
}  
  
component main { public [unsolved] } = Sudoku();

32

### Circuit Compilation and Setup

Once the circuit is written, the next steps are to compile it into R1CS and then perform the trusted setup for the chosen zk-SNARK scheme (e.g., Groth16).

1. **Compilation:** The circom compiler takes the .circom file and generates several outputs, most importantly the R1CS file (the mathematical representation of the constraints) and a WASM file (a program for calculating the witness).40  
   Bash  
   circom circuits/sudoku.circom --r1cs --wasm --sym -o build
2. **Trusted Setup (Groth16):** The Groth16 protocol requires a circuit-specific trusted setup. This is a two-phase process managed by snarkjs.
   * **Phase 1 (Powers of Tau):** This phase is circuit-independent. One can either participate in a public ceremony or download the output from a completed one. This generates a .ptau file.  
     Bash  
     snarkjs powersoftau new bn128 14 pot14\_0000.ptau -v  
     snarkjs powersoftau contribute pot14\_0000.ptau pot14\_0001.ptau --name="First contribution" -v
   * **Phase 2 (Circuit-Specific):** This phase uses the .ptau file and the compiled R1CS to generate the final proving and verification keys (.zkey and verification\_key.json).26  
     Bash  
     snarkjs powersoftau prepare phase2 pot14\_0001.ptau pot14\_final.ptau -v  
     snarkjs groth16 setup build/sudoku.r1cs pot14\_final.ptau sudoku\_0000.zkey  
     snarkjs zkey export verificationkey sudoku\_0000.zkey verification\_key.json

### The Proving Pipeline

With the setup complete, the Prover can now generate a proof for a specific Sudoku solution.

1. **Witness Computation:** The Prover needs an input.json file containing the public unsolved grid and the private solved grid. The snarkjs command uses the WASM code generated during compilation to execute the circuit's logic and compute all intermediate signal values, writing them to a witness.wtns file.40  
   Bash  
   node build/sudoku\_js/generate\_witness.js build/sudoku\_js/sudoku.wasm input.json witness.wtns
2. **Proof Generation:** The Prover uses their proving key (sudoku\_0000.zkey) and the computed witness to generate the final cryptographic proof. This process is computationally intensive. The output is two files: proof.json (the proof itself) and public.json (the public inputs and outputs of the circuit).27  
   Bash  
   snarkjs groth16 prove sudoku\_0000.zkey witness.wtns proof.json public.json

### The Verification Pipeline

The final step is for a Verifier to check the proof. This can be done off-chain for testing or on-chain for decentralized applications.

1. **Off-Chain Verification:** Using snarkjs, anyone can verify the proof with the verification key, the public inputs, and the proof itself. This is extremely fast.41  
   Bash  
   snarkjs groth16 verify verification\_key.json public.json proof.json  
     
   If the command outputs OK, the proof is valid.
2. **On-Chain Verification:** For blockchain applications, snarkjs can generate a Solidity smart contract that acts as the verifier. This contract contains the verification key embedded in its code.42  
   Bash  
   snarkjs zkey export solidityverifier sudoku\_0000.zkey verifier.sol  
     
   This verifier.sol contract can be deployed to an Ethereum-compatible blockchain. Other users or smart contracts can then call its verifyProof function, passing the proof and public inputs, to verify the Sudoku solution in a trustless and decentralized manner.44

## Feasibility and Adaptation for zk-STARKs

While the Circom/snarkjs stack provides a mature pathway for implementing a zk-SNARK-based Sudoku solver, it is also feasible to build one using zk-STARKs. However, this involves a different toolchain and a different conceptual approach to representing the computation. The term "scalability" in the context of STARKs is nuanced; while they offer superior computational scalability for massive problems, zk-SNARKs may offer better economic scalability for fixed-size problems on a blockchain due to their smaller proof sizes.

### Conceptual Differences in Computation Representation

zk-SNARK systems like Groth16 are typically designed around the **R1CS** model, which represents a computation as a static set of constraints that must all be satisfied simultaneously. zk-STARK systems, on the other hand, often use a model called **Algebraic Intermediate Representation (AIR)**. An AIR represents a computation as a sequence of states, or an "execution trace." The constraints in an AIR are typically "transition constraints" that define the valid evolution from one step of the trace to the next.

For a static problem like Sudoku, which does not have a natural step-by-step execution, the logic must be adapted to fit this model. The Sudoku validation checks would be performed sequentially, and the STARK proof would attest to the integrity of this entire computational trace. While the end goal is the same—proving the validity of the solution—the intermediate mathematical representation is fundamentally different.

### Overview of the STARK-Friendly Toolchain

The primary language and toolchain for building STARK-based provable programs is **Cairo**. Developed by StarkWare, Cairo is a Turing-complete programming language designed to generate STARK proofs for general computation.45

* **Cairo vs. Circom:** Unlike Circom, which is a declarative language for defining circuits, Cairo is an imperative language with Rust-like syntax.45 Developers write functions and logic much like in a traditional programming language. The Cairo compiler is responsible for converting this code into the AIR format that the STARK proving system can process. This can offer a more familiar development experience for those not accustomed to circuit-based thinking.
* **Implementing Sudoku in Cairo:** A Sudoku solver in Cairo would involve writing a main function that takes the unsolved and solved grids as input. This function would contain loops to iterate through rows, columns, and blocks, performing the necessary checks (value range, uniqueness, clue consistency). The STARK proof would then be generated for the execution of this specific function, proving that it ran correctly and returned true.

### Performance and Scalability Implications

Adapting the Sudoku solver for a zk-STARK system has significant performance implications that stem directly from the architectural differences discussed in Section 2.

* **Prover Time:** For a computation of Sudoku's size (a few thousand constraints), a STARK prover is likely to be slower than a highly optimized zk-SNARK prover. This is due to the asymptotic complexity—quasilinear (O(ClogC)) for STARKs versus linear (O(C)) for SNARKs. The logarithmic factor makes STARKs less efficient for smaller computations.23
* **Proof Size:** This is the most significant drawback of STARKs for many applications. A STARK proof for the Sudoku solver would likely be in the range of tens of kilobytes, whereas a Groth16 SNARK proof would be under 300 bytes. For on-chain applications where data storage translates directly to user fees (gas costs), this difference of two orders of magnitude can make STARKs prohibitively expensive.13
* **Verifier Time:** STARKs excel in verifier performance. The verification time grows polylogarithmically (O(log2C)) with the size of the computation. This means that even for extremely large computations, verification remains incredibly fast. For a fixed-size 9x9 Sudoku, the absolute verification time for both SNARKs and STARKs would be in the milliseconds, but the superior scaling of STARKs becomes a decisive advantage when proving much larger statements, such as verifying a batch of thousands of Sudoku solutions at once.

In this context, the choice depends heavily on the specific application's definition of "scalability." If scalability means handling increasingly complex computations while keeping verification cheap, STARKs are superior. However, if scalability means minimizing the on-chain footprint and cost for a fixed, moderately sized computation, SNARKs are currently more practical.

## In-Depth Analysis of Challenges, Security, and Performance

Building a secure and efficient zero-knowledge proof system for a Sudoku solver is feasible, but it is not without significant challenges. These challenges span the entire development lifecycle, from the logical design of the circuit to the cryptographic security of the underlying protocols and the performance characteristics of the final implementation. A comprehensive security model for a ZKP application must be bifurcated, addressing both the "cryptographic core" (the proof system itself) and the "application layer" (the circuit logic). A failure in either layer can lead to a total system compromise. This necessitates a rare combination of expertise in deep cryptographic protocol analysis and application-specific logic, making comprehensive security audits a major, often underestimated, cost and risk factor.

### Common Pitfalls in Circuit Design and Logic

The correctness of the arithmetic circuit is paramount to the security of the entire system. A flawed circuit can undermine the soundness property, allowing a malicious Prover to create valid proofs for invalid statements.

* **Under-constraining the Circuit:** This is the most critical and common vulnerability in ZKP development. If a logical rule is not fully captured by the constraints, a loophole is created. For example, in the Sudoku circuit, if a developer only constrains the sum of each row to be 45, a Prover could use negative numbers or duplicates (e.g., ``) to satisfy the sum constraint while violating the uniqueness rule. Every assumption about the inputs (e.g., that they are positive integers within a certain range) must be explicitly enforced with a constraint.17
* **Over-constraining the Circuit:** The opposite problem, while less of a security risk, breaks the system's functionality. If the constraints are too restrictive or contradictory, it becomes impossible to generate a valid witness, even for a correct Sudoku solution. This violates the Completeness property, rendering the system useless for honest Provers.
* **Finite Field Arithmetic Issues:** Circom and other circuit languages operate over large prime finite fields, not standard computer integers.27 This has several implications. For instance, division is defined as multiplication by a modular multiplicative inverse. A constraint that attempts to compute the inverse of zero will be unsatisfiable. Developers must carefully handle edge cases like division by zero to prevent their circuits from becoming unusable. Furthermore, large numbers involved in calculations (like the product of all numbers in a row) can wrap around the field modulus, leading to unexpected behavior if not managed properly.

### Performance Benchmarking: Analyzing Prover and Verifier Time

The asymmetric performance of ZKP systems is their defining characteristic and a key consideration in system design.

* **Prover Performance:** Proof generation is the computational bottleneck of any ZKP system. The process involves complex cryptographic operations, such as a large number of elliptic curve multiplications for zk-SNARKs. For a circuit with several thousand constraints, like the Sudoku solver, proof generation time on a standard consumer-grade CPU would likely be on the order of seconds.23 This makes ZKPs well-suited for off-chain computation tasks where a delay of a few seconds is acceptable, but generally unsuitable for applications requiring real-time, low-latency interactions.
* **Verifier Performance:** In contrast, verification is extremely efficient. For a zk-SNARK like Groth16, verification involves a small, constant number of cryptographic pairings and is typically completed in milliseconds.13 This efficiency is what makes ZKPs so valuable for blockchain scalability; a complex computation that takes a long time to execute and prove can be verified on-chain with minimal computational cost.16
* **Memory Consumption:** Both the trusted setup process and the proof generation can be memory-intensive, especially as the number of constraints in the circuit grows. For very large circuits, memory can become a more significant limitation than CPU time, requiring specialized hardware for the Prover.

### Security Considerations

The security of a ZKP-based application rests on a chain of trust that extends from the abstract mathematical assumptions to the concrete lines of code in the implementation.

* **Protocol-Level Security:** The soundness of the ZKP system relies on the hardness of the underlying cryptographic problems. For most zk-SNARKs, this is the discrete logarithm problem on elliptic curves. For zk-STARKs, it is the collision-resistance of the chosen hash function. A breakthrough in solving these problems (e.g., via a large-scale quantum computer for ECC) would break the security of the protocol.14
* **Implementation-Level Security:** Even if the protocol is theoretically sound and the circuit logic is correct, bugs in the software libraries (e.g., circom, snarkjs) or in the verifier smart contract can introduce critical vulnerabilities. For example, an error in the Solidity verifier contract could lead it to accept invalid proofs. Therefore, all components of the toolchain and the final application code must be subject to rigorous security audits.17
* **Trusted Setup Integrity:** As discussed previously, for zk-SNARKs that require a trusted setup, the integrity of this ceremony is a foundational security assumption. Users of the system must trust that the MPC was conducted correctly and that at least one participant was honest. Any doubt about the ceremony's integrity casts doubt on the security of every proof generated with its parameters.14

### Choosing the Right Protocol: A Decision Framework for the Sudoku Use Case

Given the analysis, the choice between zk-SNARKs and zk-STARKs for the Sudoku solver depends on the specific goals and constraints of the project.

* **For Minimal On-Chain Cost and Rapid Development:** If the primary application is an on-chain game or verification system where transaction fees are a major concern, a **zk-SNARK (specifically Groth16)** is the superior choice. Its incredibly small proof size directly translates to lower gas costs, and the mature Circom/snarkjs ecosystem provides a lower barrier to entry for developers.13
* **For Long-Term Security and Maximum Decentralization:** If the project prioritizes future-proofing against quantum computers and adheres to a strict "no trust" philosophy, then a **zk-STARK** is the better option. The elimination of the trusted setup and the reliance on quantum-resistant cryptography provide stronger long-term security guarantees, despite the higher immediate costs.15
* **Overall Recommendation:** For a project focused on learning ZKP development or building a proof-of-concept, the **zk-SNARK path with Circom and snarkjs is the more practical starting point.** The wealth of documentation, tutorials, and community support makes the development process significantly more accessible. The performance characteristics are more than adequate for the Sudoku problem, and the trusted setup risk can be mitigated by using parameters from a well-known public ceremony.

## Conclusion: The Feasibility and Future of Verifiable Computation

The exploration of building a secure Sudoku solver using zero-knowledge proofs provides a definitive answer to its feasibility while simultaneously opening a window into the transformative potential of verifiable computation. The successful implementation of such a system is not merely a cryptographic novelty; it serves as a tangible proof-of-concept for a new paradigm of trustless computation, demonstrating our ability to decouple the verification of a result from the need to trust the entity that produced it. This has profound implications for nearly every domain of digital interaction, from finance and gaming to identity and education.

### Summary of Findings: A Definitive Statement on Feasibility

Based on a thorough analysis of the theoretical foundations, available cryptographic protocols, and practical toolchains, it is **unequivocally feasible to build a secure Sudoku solver algorithm using zero-knowledge proof protocols like zk-SNARKs and zk-STARKs.** The problem of Sudoku validation is well-defined and falls squarely within the NP complexity class, for which ZKPs are expressly designed. The estimated circuit size of approximately 1,700 constraints is well within the capabilities of modern frameworks like Circom and Cairo.32

The primary decision point is not one of feasibility but of trade-offs. The choice between zk-SNARKs and zk-STARKs hinges on a project's core priorities:

* **zk-SNARKs** offer immediate practical advantages in the form of succinct proofs, leading to lower on-chain transaction costs, and a more mature, accessible developer ecosystem.
* **zk-STARKs** provide superior long-term security through quantum resistance and greater transparency by eliminating the need for a trusted setup, at the cost of larger proofs and a steeper learning curve.

### Final Recommendations on Protocol and Toolchain Selection

For the specific task of building a Sudoku solver, especially as an initial foray into the ZKP space, the following recommendations are made:

1. **Recommended Protocol:** **zk-SNARK (Groth16)**. Its unparalleled succinctness is a decisive advantage for any application with an on-chain component, and its performance is more than sufficient for this problem size.
2. **Recommended Toolchain:** **Circom and snarkjs**. This stack is the most mature, well-documented, and widely supported for zk-SNARK development, providing the most direct path from circuit design to a functional proof-of-concept.

For production systems where long-term security and decentralization are non-negotiable, and where on-chain costs are less of a concern, an investigation into the zk-STARK ecosystem via the Cairo language is strongly encouraged.

### Broader Applications and Future Research Directions

The Sudoku solver is a powerful microcosm of the broader field of verifiable computation, where a solution is hard to find but easy to check.16 The principles and techniques used to build it are directly applicable to a vast range of more impactful problems.

* **Gaming:** ZKPs are poised to revolutionize gaming by enabling provably fair and private gameplay. Applications include creating incomplete information games (e.g., poker, Battleship) where players' hands or strategies remain hidden, implementing verifiable randomness for loot boxes and other chance-based mechanics, and building robust anti-cheat systems that do not require invasive client-side software.51
* **Digital Identity and Credentials:** ZKPs allow for selective disclosure, enabling users to prove attributes about themselves without revealing sensitive data. A user could prove they are over 18 without revealing their exact birthdate, prove their income is within a certain range for a loan application without disclosing their salary, or prove citizenship without showing a passport.54
* **Education:** In the educational sector, ZKPs can be used to verify academic credentials, certifications, or disability status for accommodations without compromising student privacy.55

The successful implementation of a ZKP Sudoku solver validates a model that can be generalized to any computation y=F(x,w), where a proof can attest that the public output y was correctly computed from public input x and some private witness w. This model is the foundation for technologies like blockchain rollups, which bundle thousands of transactions off-chain and post a single proof on-chain, and privacy-preserving machine learning, where one can prove the result of a model's inference on private data.16

Ongoing research continues to push the boundaries of what is possible, focusing on developing more efficient proof systems, enabling recursive proofs for Incrementally Verifiable Computation (IVC) to handle unbounded computations, and creating higher-level programming languages that abstract away the immense complexity of arithmetization.24 The journey to build a simple, verifiable puzzle solver is, in essence, a hands-on entry point into one of the most transformative and promising technologies in modern computer science.

#### Works cited

1. Zero-Knowledge (a tutorial by Oded Goldreich), accessed September 4, 2025, <https://www.wisdom.weizmann.ac.il/~oded/zk-tut02.html>
2. Zero-knowledge proof - Wikipedia, accessed September 4, 2025, <https://en.wikipedia.org/wiki/Zero-knowledge_proof>
3. Zero-Knowledge Proofs: A Practical Deep Dive For Programmers - Cyfrin, accessed September 4, 2025, <https://www.cyfrin.io/blog/what-is-a-zero-knowledge-proof-a-practical-guide-for-programmers>
4. An Introduction to Zero Knowledge Proofs - Code the Change - Stanford, accessed September 4, 2025, <https://codethechange.stanford.edu/guides/guide_zk.html>
5. Zero-knowledge proofs explained in 3 examples - Circularise, accessed September 4, 2025, <https://www.circularise.com/blogs/zero-knowledge-proofs-explained-in-3-examples>
6. www.circularise.com, accessed September 4, 2025, <https://www.circularise.com/blogs/zero-knowledge-proofs-explained-in-3-examples#:~:text=Zero%2Dknowledge%20proofs%20must%20satisfy,and%20do%20not%20allow%20cheating.>
7. What are Zero Knowledge Proofs? - OpenMined, accessed September 4, 2025, <https://openmined.org/blog/zero-knowledge-proof/>
8. Lecture 15 - Zero Knowledge Proofs - cs.princeton.edu, accessed September 4, 2025, <https://www.cs.princeton.edu/courses/archive/fall07/cos433/lec15.pdf>
9. Demonstration of Zero-Knowledge Proof for Sudoku Using Standard Playing Cards, accessed September 4, 2025, <https://www.wisdom.weizmann.ac.il/~naor/PAPERS/SUDOKU_DEMO/>
10. Cryptographic and Physical Zero-Knowledge Proof Systems for Solutions of Sudoku Puzzles, accessed September 4, 2025, <https://www.wisdom.weizmann.ac.il/~naor/PAPERS/sudoku_abs.html>
11. Zero knowledge / proof of knowledge sudoku solution - Cryptography Stack Exchange, accessed September 4, 2025, <https://crypto.stackexchange.com/questions/25322/zero-knowledge-proof-of-knowledge-sudoku-solution>
12. Interactive Sudoku Zero-knowledge Proof - In Pursuit of Laziness - Manish Goregaokar, accessed September 4, 2025, <https://manishearth.github.io/blog/2016/08/10/interactive-sudoku-zero-knowledge-proof/>
13. zk-STARK vs zk-SNARK : An In-Depth Comparative Analysis, accessed September 4, 2025, <https://www.quillaudits.com/blog/ethereum/zk-starks-vs-zk-snarks>
14. zk-SNARK vs zkSTARK - Explained Simple - Chainlink, accessed September 4, 2025, <https://chain.link/education-hub/zk-snarks-vs-zk-starks>
15. SNARKs vs. STARKS vs. Recursive SNARKs - Alchemy, accessed September 4, 2025, <https://www.alchemy.com/overviews/snarks-vs-starks>
16. An approximate introduction to how zk-SNARKs are possible, accessed September 4, 2025, <https://vitalik.eth.limo/general/2021/01/26/snarks.html>
17. zk-SNARKs vs. zk-STARKS: Comparing the Main ZKPs in Blockchain, accessed September 4, 2025, <https://www.halborn.com/blog/post/zk-snarks-vs-zk-starks-comparing-the-main-zkps-in-blockchain>
18. Pinocchio: verifiable computation revisited - LambdaClass Blog, accessed September 4, 2025, <https://blog.lambdaclass.com/pinocchio-verifiable-computation-revisited/>
19. Full Guide to Understanding zk-SNARKs and zk-STARKS - Cyfrin, accessed September 4, 2025, <https://www.cyfrin.io/blog/a-full-comparison-what-are-zk-snarks-and-zk-starks>
20. Are ZK-Snark based layer 2s ultimately doomed? : r/ethereum - Reddit, accessed September 4, 2025, <https://www.reddit.com/r/ethereum/comments/vqcerp/are_zksnark_based_layer_2s_ultimately_doomed/>
21. Non-interactive zero-knowledge proof - Wikipedia, accessed September 4, 2025, <https://en.wikipedia.org/wiki/Non-interactive_zero-knowledge_proof>
22. Differences between zk-SNARKs and zk-STARKs - Extrimian, accessed September 4, 2025, <https://extrimian.io/differences-between-zk-snarks-and-zk-starks/>
23. Evaluating the Efficiency of zk-SNARK, zk-STARK, and Bulletproof in ..., accessed September 4, 2025, <https://www.mdpi.com/2078-2489/15/8/463>
24. Zero Knowledge Proofs, accessed September 4, 2025, <https://zk-learning.org/>
25. Trusted Setup | By RareSkills, accessed September 4, 2025, <https://rareskills.io/post/trusted-setup>
26. Trusted Setup - ZoKrates, accessed September 4, 2025, <https://zokrates.github.io/toolbox/trusted_setup.html>
27. A beginner's intro to coding zero-knowledge proofs - DEV Community, accessed September 4, 2025, <https://dev.to/spalladino/a-beginners-intro-to-coding-zero-knowledge-proofs-c56>
28. Zero-Knowledge Proof Frameworks: A Survey - arXiv, accessed September 4, 2025, <https://arxiv.org/html/2502.07063v1>
29. R1CS - The Zero Knowledge Blog, accessed September 4, 2025, <https://www.zeroknowledgeblog.com/index.php/the-pinocchio-protocol/r1cs>
30. Converting to rank one constraint system (R1CS) - Cryptography Stack Exchange, accessed September 4, 2025, <https://crypto.stackexchange.com/questions/55963/converting-to-rank-one-constraint-system-r1cs>
31. R1CS Explainer | ZK Learning Resources, accessed September 4, 2025, <https://learn.0xparc.org/materials/circom/additional-learning-resources/r1cs%20explainer/>
32. How to create a Zero Knowledge DApp: From zero to production - Vivian Plasencia, accessed September 4, 2025, <https://vivianblog.hashnode.dev/how-to-create-a-zero-knowledge-dapp-from-zero-to-production>
33. Complexity of a Sudoku - Ceremade, accessed September 4, 2025, <https://www.ceremade.dauphine.fr/~gontier/Presentations/2014_03_27_Cermics.pdf>
34. Circom - Sindri, accessed September 4, 2025, <https://sindri.app/docs/how-to-guides/frameworks/circom/>
35. CIRCOM: A Robust and Scalable Language for Building Complex Zero-Knowledge Circuits - UPCommons, accessed September 4, 2025, <https://upcommons.upc.edu/bitstreams/22002ca7-a4d6-404d-9ae3-f2c084312c53/download>
36. Private Sudoku solution verification from polynomial set representation - ResearchGate, accessed September 4, 2025, <https://www.researchgate.net/publication/353548394_Private_Sudoku_solution_verification_from_polynomial_set_representation>
37. Zero-Knowledge Proofs Using SnarkJS and Circom - Better Programming, accessed September 4, 2025, <https://betterprogramming.pub/zero-knowledge-proofs-using-snarkjs-and-circom-fac6c4d63202>
38. Tutorial — iden3 0.1 documentation - Read the Docs, accessed September 4, 2025, <https://iden3-docs.readthedocs.io/en/latest/iden3_repos/circom/TUTORIAL.html>
39. Writing circuits - Circom 2 Documentation, accessed September 4, 2025, <https://docs.circom.io/getting-started/writing-circuits/>
40. Compiling circuits - Circom 2 Documentation, accessed September 4, 2025, <https://docs.circom.io/getting-started/compiling-circuits/>
41. Getting Started - ZoKrates, accessed September 4, 2025, <https://zokrates.github.io/gettingstarted.html>
42. ZoKrates: Introduction, accessed September 4, 2025, <https://zokrates.github.io/>
43. ZoKrates tutorial with Truffle - by Extropy.IO - Medium, accessed September 4, 2025, <https://medium.com/extropy-io/zokrates-tutorial-with-truffle-41135a3fb754>
44. Tutorial: A SNARK Powered RNG - ZoKrates, accessed September 4, 2025, <https://zokrates.github.io/examples/rng_tutorial.html>
45. Tutorial - Cairo Language, accessed September 4, 2025, <https://www.cairo-lang.org/tutorial/>
46. STARK Technology | StarkWare, accessed September 4, 2025, <https://starkware.co/stark/>
47. Understanding Constraints in Circom | by Shreyansh Jain - Medium, accessed September 4, 2025, <https://sudoshreyansh.medium.com/understanding-constraints-in-circom-1749ae4c4515>
48. Evaluating the Efficiency of zk-SNARK, zk-STARK, and Bulletproof in Real-World Scenarios: A Benchmark Study - University of Twente Research Information, accessed September 4, 2025, <https://research.utwente.nl/en/publications/evaluating-the-efficiency-of-zk-snark-zk-stark-and-bulletproof-in>
49. Sudoku ZK-SNARK Example - Tomas Delclaux - Medium, accessed September 4, 2025, <https://dlxtomas.medium.com/sudoku-zk-snark-example-48ccbd5cf8a1>
50. Cryptography 101: Arithmetic Circuits | by Frank Mangone - Medium, accessed September 4, 2025, <https://medium.com/@francomangone18/cryptography-101-arithmetic-circuits-351ca87647a9>
51. Zero Knowledge Proofs for Games. Phil Kelly - o1Labs, accessed September 4, 2025, <https://blog.o1labs.org/zero-knowledge-proofs-for-games-f8b690a2c1ef>
52. medium.com, accessed September 4, 2025, <https://medium.com/@ingonyama/how-zero-knowledge-proofs-will-change-gaming-forever-1ed8ac6fe93f#:~:text=A%20Zero%2DKnowledge%20Proof%20can,their%20progress%20without%20disclosing%20solutions.>
53. Zero-Knowledge Proofs in Gaming: Unlocking Privacy, Fairness, and Trustless Innovation, accessed September 4, 2025, <https://tokenminds.co/blog/knowledge-base/zero-knowledge-proofs-in-gaming>
54. Zero-Knowledge Proofs: A Beginner's Guide - Dock Labs, accessed September 4, 2025, <https://www.dock.io/post/zero-knowledge-proofs>
55. Zero-knowledge proofs in education: a pathway to disability inclusion and equitable learning opportunities - ResearchGate, accessed September 4, 2025, <https://www.researchgate.net/publication/378009475_Zero-knowledge_proofs_in_education_a_pathway_to_disability_inclusion_and_equitable_learning_opportunities>
56. Zero-knowledge proofs in education: a pathway to disability inclusion and equitable learning opportunities - Consensus, accessed September 4, 2025, <https://www.consensus.app/papers/zeroknowledge-proofs-in-education-a-pathway-to-disability-xu/74bd310c347b54e899307fdfbedf415d>
57. VerifBFL: Leveraging zk-SNARKs for A Verifiable Blockchained Federated Learning - arXiv, accessed September 4, 2025, <https://arxiv.org/abs/2501.04319>
58. Recursive SNARKs and incrementally verifiable computation (IVC) | Smart contract audits from Veridise, accessed September 4, 2025, <https://veridise.com/blog/learn-blockchain/recursive-snarks-and-incrementally-verifiable-computation-ivc/>