Leonardo Ravani & Tobias Kistler

Zero-Knowledge Sudoku

Bachelor's Thesis

OST – Eastern Switzerland University of Applied Sciences
Campus Rapperswil-Jona

Supervision

Daniel Tschudi

2025

Abstract

Imagine sharing that you know a secret without revealing the secret itself. This is what zero-knowledge proofs (ZKPs) aim to do. In ZKPs there is a prover who claims knowledge of something and a verifier who checks this claim. The goal of this project is to further explore current technologies revolving around ZKPs and understand possible adaptations to an everyday application beyond blockchain use cases.

To explore the practical use of ZKPs, this project introduces a web puzzle application that keeps the solutions of individual users private using ZKPs. A user can solve a logic-based puzzle like Binairo or Sudoku and check the validity of their solution by sharing only a ZKP of the solution. That way, the solution never leaves the user's device. To further strengthen the security, the following two checks are implemented: (1) making sure that the solution matches the original puzzle, and (2) integrating the user's ID during the generation process. These checks prevent users from reusing a proof to "solve" other puzzles or for the proof to be stolen by another user. To implement this application, different ZKP frameworks are considered. Circom and snarkjs are selected because of their active development, clear documentation and good web development capabilities.

The final result is a secure application that demonstrates how ZKPs can be applied in a realistic and practical way. This highlights their broader potential in digital security. In most applications, the impact of ZKPs is intentionally hidden, as good cybersecurity aims to operate in the background. The ZKP-Puzzles application puts the ZKPs in the spotlight and visualizes how ZKPs work.

Keywords: Zero-knowledge proof, Cyber Security, Web Development.

Executive Summary

Problem Background

In most cases for a malicious actor in the IT-world, it is difficult to gain access to a system. However, communication over the Internet should be considered public. This means it is essential to offer increased protection in this area, e.g. the transition of data from one place to another. Zero-knowledge proof is a concept in IT security that protects a user's data during the transfer from their environment to another party. This is achieved by never sending the actual data but instead only sending a proof which consists mostly of mathematical equations based on the prover data. One can imagine it as showing someone a small cutout window from a large sheet of paper placed on a "Where's Waldo?" puzzle that only reveals Waldo. Anyone seeing this is now convinced that the prover knows where Waldo is. But when removing the paper and showing the entire image, the position of Waldo remains a secret. This way, the prover demonstrates that they know Waldo's location without revealing it. This proof should only reveal that the prover has knowledge of something. With this in mind and the mathematical basis of how the proof is calculated it is not possible to apply this to every situation.

This means that although ZKPs are widely studied and being used in blockchain related systems, their usage in everyday web applications is very limited. Therefore, this project aims to study such a use-case.

Approach

To generate a proof of knowledge of a solution for a puzzle, it was important to choose puzzles that can be mathematically verified. In the case of Sudoku it is only necessary to calculate the presence of each number 1 to 9. This calculation can then be applied to each row, column and box to verify that the entire solution is correct. Additionally, the verifier checks that the user's ID that has been added to the proof matches the ID sent with the request. Since the user's ID also influences the proof, it is impossible for a malicious actor to steal someone else's proof. Finally, the original unsolved puzzle is attached and compared to the submitted solution. This ensures that not only is the solution valid, but also corresponds to the original puzzle it was meant to solve. Otherwise users could resend the proof of one puzzle, falsely claiming it is another puzzle, which could give them an unfair advantage such as climbing up the leaderboard. In addition to the Sudoku the Binairo puzzle was also added, the rules here are equally simple, stating that every row and column require an equal amount of red and blue tiles while horizontally and vertically only a maximum of 2 tiles next to each other are allowed to be the same color. The entire application is kept dynamic in order to offer the possibility to expand on the puzzles and add more variations later on.

Result

The result is a web application that allows the solving of puzzles while always keeping the solution of each user safe. For demonstrational and educational purposes, additional features are implemented which allow an easier handling and deeper insight into how ZKPs are used in this application. It is possible to automatically fill out some of the puzzles using the "Auto Solve" button. This allows for faster testing. Further, the "Visualize" button in the top right corner allows for a more detailed animation to display and illustrate the internal process. The problem with demonstrating an application which uses ZKPs is that if it is implemented correctly, the user will not be able to tell that it is there, making it difficult to understand how this application is any different from any other puzzle solving website. When a puzzle is solved, the proof data (proof.JSON and public.JSON) are both stored along with who solved the puzzle and which puzzle it is. These entries are used to keep track of the scores of each user, allowing the creation of a leaderboard.

Contents

Chapter 1

Glossary

Binairo: A puzzle that has slightly different rules depending on the provider. The size can vary as well, but is always a square grid. The goal is to fill in the grid with two symbols where each symbol is represented equally in each column and row and each symbol does not border more than 1 of the same symbol horizontally and vertically each.

Blockchain: A decentralized, distributed, and immutable digital ledger system that records transactions across many computers.

C4: Context, Containers, Components, and Code

C4 model: A graphical notation technique for modeling software architecture. Each C represents a layer of abstraction.

Completeness: When an honest prover claims his proof, an honest verifier will always verify it as true.

Constraint: In context of a ZKP a constraint is a mathematic rule that must be enforced to generate a proof. Example: "x + y = 3". If a constraint is not met, the proof cannot be generated.

CRUD: Create, Read, Update, Delete which are operations on a database.

Docker: A platform that creates standardized units by packaging software and its dependencies into containers.

Encryption: The process of converting data to a code to prevent unauthorized access.

ERD - Entity Relationship Diagram: A diagram to show the fields, their types and the connections between the tables of a database.

Flowchart: A chart that depicts a process with actions and exchanges in a flow from the top to the bottom.

Framework: A set of libraries or tools that distribute a structured base for creating software applications.

GitLab: A web-app much like GitHub, offering tools for software development lifecycle, repository management, continuous integration and even issue tracking.

Hamiltonian Cycle: A cycle in a graph that visits each vertex exactly once before returning to the first vertex.

HTML Element: A building block for web applications. They define the structure of the web page.

Jira: An agile project management tool to plan, track, release and support software.

JSON - JavaScript Object Notation: JSON is a lightweight data format, here is an example "propertyName" : "propertyValue"

Low Fidelity Mockups: A simple visual representation of how the final application may look like and its core features.

Many to Many: In context of databases this means that two tables are logically connected in a way where both tables can have multiple entries in relation to the other table.

npx: A command-line tool that runs Node.js packages without globally installing them.

OAuth: Allows applications or websites to access the information of a user from a different application or website but only with the users permission of course.

One to Many: In context of databases this means that two tables are logically connected in a way where on one table can have multiple entries in relation to the other table and the other table can only have one. In this project a user has only one solution for a Sudoku but a Sudoku can be solved by multiple users.

Polynomial equation: An equation consisting of variables, coefficients, and exponents, where a polynomial expression is set equal to zero.

PostgreSQL: A free and open-source relational database management system emphasizing extensibility and SQL compliance

Prover: In the context of ZKPs the prover is the party that wants to prove their knowledge of something.

Quadratic equation: An equation that is either a variable squared or one variable multiplied with another.

Risk Matrix: A probability matrix that plots the damage potential to the change of a risk occuring.

Scrum: A project organisation framework that consists of sprints to divide a project into more manageable sections. This promotes agile software development.

Scrum+: An upgraded version of scrum that adds more practices and tools to enhance productivity.

Smart Contracts: The terms of smart contracts are written directly into the code that is running on a blockchain and executed automatically when the conditions are met.

Soundness: When a dishonest prover claims his proof, an honest verifier will always verify it as false.

Sprint: For each sprint the agreed upon features must be developed, each sprint has the same fixed time period. A meeting is held to plan the sprint and more meetings scattered throughout to ensure progress.

SQL - Structured Query Language: The default language used to perform actions on a database.

Sudoku: Sudoku is a puzzle that usually consists of a 9x9 grid where some numbers are already written into some cells. The goal is to have each number 1 to 9 only once in every row, column and 3x3 box.

Verifier: In the context of ZKP the verifier is the party that verifies a claim provided by a prover.

ZKP - Zero-knowledge proof: A proof generated to prove the knowledge of some information without revealing said information.

Chapter 2

Introduction

This chapter provides an overview of the project's background and objectives.

2.1 Initial Situation

A main issue with security in web applications is that data is transferred in what can be considered a public area — the internet. Nowadays there are plenty of strong algorithms and protocols to encrypt data and keep your data safe when it has to be transferred online. However sometimes, though rarely, vulnerabilities are uncovered or, even worse, malicious actors are gathering data and storing it. These malicious actors are waiting for an opportunity such as an advancement in decryption technology or the discovery of a a vulnerability that would allow them to access the data.

To prevent this zero-knowledge proofs (ZKPs) allows a user to transfer a proof of their knowledge of information without having to transfer that information to begin with. In this case it prevents all such issues as no actual data of value ever passes through the internet. Another area where ZKPs shine when transferring data online is when the data has to remain public which is why it is so commonly used in blockchain technology. Furthermore when data needs to be transferred but a user would prefer to stay anonymous or not to have that information stored on a server ZKPs are useful in such cases.

2.2 Study Thesis Objective

This project aims to engage with the fundamentals of zero-knowledge proofs, examine how they are applied in real-world scenarios, and expand ZKPs to a web application. The goal is to create a secure application that utilizes the advantages of ZKPs to keep the users secrets safe as they never leave the device. This prevents malicious actors to store the secrets on a server or otherwise exploit them.

The core of a ZKP is very mathematical which means it works most easy with numerical information. For that reason, given the time span of this project and the unfamiliarity with the technology it was imperative to apply the ZKP to an inherently numerically centered concept, which is why puzzles such as Binairo and Sudoku are good options.

Chapter 3

Requirements

This chapter describes the functional and non-functional requirements of the application. Functional requirements focus on the tasks that should be performed while the non-functional requirements define how the tasks should be performed and put more focus on usability and performance. By clearly defining these requirements, the application can be functional and user-friendly at the same time.

3.1 FR - Functional Requirements

The functional requirements describe what the application should do. They outline the features and capabilities that the system must provide to meet user needs. This includes the specific tasks the application must be able to perform, the interactions between the user and the system, and how the system should respond in different scenarios. These requirements serve as a base to the development of the application.

3.1.1 Actors

The only actor that will access the application is the user. The user wants to upload his Sudoku and see themselves on the leaderboard after proving the correctness of his solution. Any administrative work is managed directly in the back end of the application.

3.1.2 Use Cases

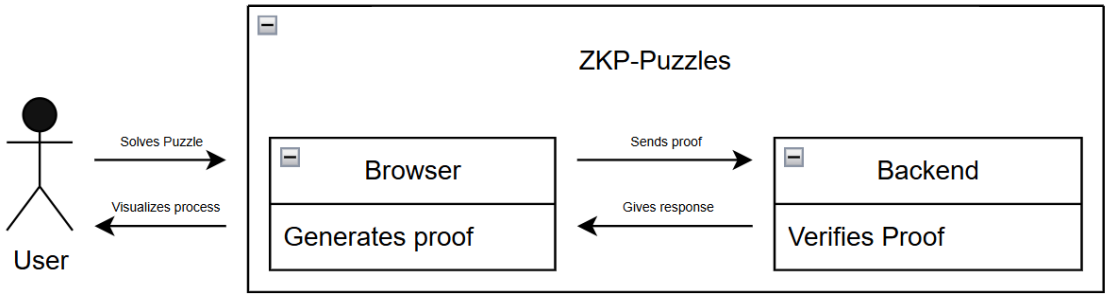

Figure 3.1: Use Case diagram

UC1: Access
The user wants to access the application in the browser. For this to be possible, the application needs to run on a server, or the user has to start the application locally.

UC2: Authentication
The user wants to log into the application and set a username that will be displayed on the leaderboard.

UC3: Solving a Sudoku
The user can solve a Sudoku on the webpage that has to be recognized by the application.

UC4: Verify the Sudoku solution
After solving a Sudoku, the user wants to prove the solution with zero-knowledge and see themselves on the leaderboard.

UC5: See the proving process
The user wants to see what is happening inside of the application during the proving process.

3.2 NFR - Non Functional Requirements

3.2.1 Details

ID	NFR-1
Subject	The app is usable on Firefox, Safari and Google Chrome
Requirement	Portability - Adaptability
Measures	The application must run on the three most common browsers in use today.
Priority	Medium
Creation Date	12.03.2025
Verification Date	Beta, Final Submission
Due Date	Final Submission

Table 3.1: NFR-1 Adaptability

ID	NFR-2
Subject	Errors are easily understood by a user
Requirement	Usability - User Error Protection
Measures	If any errors are displayed, they must be easily understandable to a user, even one who does not know anything about IT.
Priority	Medium
Creation Date	12.03.2025
Verification Date	Beta, Final Submission
Due Date	Final Submission

Table 3.2: NFR-2 Error Protection

ID	NFR-3
Subject	Solving and uploading a Sudoku is simple
Requirement	Usability - Operability
Measures	The process of solving and uploading the solution of a Sudoku must be understandable through the context of the app with no further knowledge about ZKPs.
Priority	Low
Creation Date	12.03.2025
Verification Date	Beta, Final Submission
Due Date	Final Submission

Table 3.3: NFR-3 Operability

ID	NFR-4
Subject	Proofs are verified correctly
Requirement	Security - Completeness
Measures	A proof must be verified as correct if a Sudoku is solved correctly - This is commonly referred to as completeness.
Priority	High
Creation Date	12.03.2025
Verification Date	Beta, Final Submission
Due Date	Final Submission

Table 3.4: NFR-4 Completeness

ID	NFR-5
Subject	Checking a solution is quick
Requirement	Performance - Time Behaviour
Measures	The duration of creating and verifying a proof should be kept below 10 seconds.
Priority	Low
Creation Date	12.03.2025
Verification Date	Beta, Final Submission
Due Date	Final Submission

Table 3.5: NFR-5 Performance

ID	NFR-6
Subject	Fake Proofs cannot be created
Requirement	Security - Soundness
Measures	A proof cannot be generated without knowledge of the solution - This is commonly referred to as soundness.
Priority	High
Creation Date	12.03.2025
Verification Date	Beta, Final Submission
Due Date	Final Submission

Table 3.6: NFR-6 Soundness

3.2.2 NFR Verification

Although the application was developed as a general ZKP puzzle solving web application, the Sudoku carries greater importance and is the only puzzle that will be tested for the NFRs. The NFRs are tested according to the following descriptions and the results can be found in section 7.3.

NFR-1

To verify this NFR the application is started on the Safari, Firefox and Google Chrome browser. A proof generation is started once with a correct solution and once with a false

solution. Each attempt has to be handled accordingly by the application and tested. If on all three browsers the false solutions are denied and the correct ones get accepted this NFR passes.

NFR-2

During the purposefully erroneous use of the application no error message should be unreadable to a person who is not educated in IT.

NFR-3

Although the backend is more complicated, the frontend should resemble a general Sudoku page which does not use ZKP. From login to testing the solution of a puzzle it must not take more than five clicks (excluding any clicks needed to fill in the puzzle solution).

NFR-4

Two different Sudokus must be correctly solved and then verified. If the verification succeeds on both this NFR passes.

NFR-5

One false and one correctly solved Sudoku must each be generated into a proof and verified within 10 seconds. If the entire process for either takes longer than 10 seconds this NFR is considered failed.

NFR-6

Many different cases must be tested and each must be denied by the verification or proof generation process:

1. Not filled out at all.
2. Filled out but with a false solution.
3. Filled out with a correct solution but for another puzzle than the one being solved.
4. Filled out with a correct solution but changing the userId in the publicJSON.
5. Filled out solution for puzzle A in puzzle B but changing the puzzle in the publicJSON to the puzzle A.

Chapter 4

Zero-Knowledge Proofs

Zero-knowledge proofs are not general knowledge. Therefore this section is dedicated to providing an overview on the topic. We begin by introducing the concept of a proof of knowledge and then go into further detail in the context of zero-knowledge.

4.1 Proof of Knowledge

Proof of knowledge is a proof system used in cryptography by which one party can prove to another that they know certain information. This is usually achieved with an interaction between two roles: the *prover* and the *verifier*.

- **Prover**: This is the person claiming to know the information. They have the goal to prove this claim.
- **Verifier**: The verifier is the person checking if the claim of the prover is actually true.

A proof of knowledge can have different properties. It is typically desirable for a proof of knowledge to be complete and sound.

- **Completeness**: The verifier always accepts the proof if the claim of the prover is true.
- **Soundness**: A dishonest prover is not able to trick the verifier with a false statement.

An example of a proof of knowledge is:

The prover starts with the statement: *"I know the factorization of the number 943."* The verifier does not just believe him and requests a proof. The prover responds by sending the values $p = 41$ and $q = 23$. The verifier checks the multiplication: $41 \times 23 = 943$. Since the equation is correct, the verifier accepts the proof.

4.2 Zero Knowledge

If a prover wants to convince a verifier that they know a secret, but they do not want to tell them the secret, they can use zero-knowledge. Zero-knowledge is a cryptographic process in which a proof of knowledge can be created without any information about the secret being revealed. To illustrate this idea, consider the following famous example called *The Alibaba Cave*.

In this example, there is a cave that splits up into two paths A and B which lead back together. However, where the paths meet, there is a locked door that can only be opened with a secret password. The prover, called Peggy, claims to know this password. To verify her claim, she walks into the cave and chooses one of the paths A or B. It is crucial that the verifier, Victor, does not see which path she took when entering the cave. Then Victor randomly chooses one of the paths and asks Peggy to return from the cave from this path. If Peggy has chosen the same path as Victor she can just walk out. Otherwise she needs to go through the locked door to come out of the correct path.

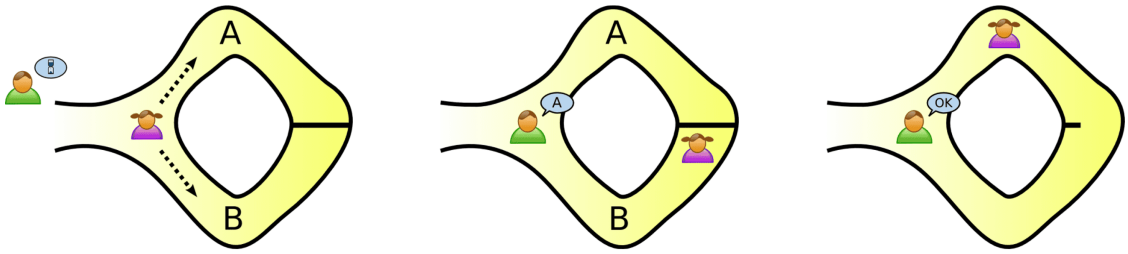

Figure 4.1: The Alibaba Cave [1]

If Peggy does not know the password, she can only return from the same path she entered. In that case, she has a $\frac{1}{2}$ chance of guessing correctly which path Victor will ask for. If Victor wants to be sure that Peggy knows the secret password, he has to repeat this sequence until he is satisfied. If he repeats it k times, the probability of Peggy guessing correctly each time would be $\frac{1}{2^k}$. For example, after only 20 repetitions, the probability becomes $\frac{1}{2^{20}}$, which is around 1 in a million, making it statistically negligible. This is called probabilistically sound and is a typical characteristic of interactive ZKPs. This means that the verifier's confidence in the prover's honesty increases with each repeated interaction, but is never absolute. This example is zero-knowledge since Victor is now convinced that Peggy knows the password, but he has gained no knowledge about the password and could not open the door by themselves.

4.2.1 Non-Interactive Proof

A zero-knowledge proof can be either interactive or non-interactive. In interactive proofs the verifier and the prover need to communicate with each other, like in the previous example of *The Alibaba Cave*. Another example of an interactive proof would be *Zero Knowledge using Hamiltonian Cycles*, where a prover claims to know a Hamiltonian cycle in a graph and then uses graph isomorphism to prove his knowledge of the hamiltonian cycle without revealing it (for more details, see [2]). Typically the process involves several rounds of interaction. With each round, the verifier gets more confidence that the prover actually knows the secret. However, the proof is not guaranteed unless enough rounds are carried out.

Non-interactive proofs are a bit different from their interactive counterparts and consist only of a single message from the prover to the verifier. Non-interactive proofs are commonly used in modern cryptographic systems, since they are more efficient in terms of computational power. While interactive proofs are typically probabilistically sound and require multiple interactions, non-interactive proofs are deterministically sound. This means that in non-interactive proofs, the verifier can be absolutely certain of the prover's honesty from a single proof. Some well known examples of non-interactive proofs being used are in blockchain or smart contracts.

4.3 SNARK

SNARKs are a method to generate non-interactive proofs and are taken as an example since the ZKP-Puzzles application uses them. The properties of a SNARK lie in its name, as SNARK stands for:

- **Succinct**: The size of the proof is very compact, typically sub-linear or even constant, in relation to the size of the statement being proven. This means proofs can be verified very quickly, however generating takes more computational effort.
- **Non-interactive**: The prover sends the proof to the verifier, which can verify the proof without any further interactions.
- **Argument**: The proof is only sound based on computational difficulty.
- **Knowledge-sound**: It is not possible to create a proof without knowing the secret.

To understand how SNARKs work, it is necessary to understand some core components such as arithmetic circuits, DSLs, and the trusted setup.

4.3.1 Arithmetic Circuit

An arithmetic circuit is the statement to be proven. It encodes the logic and claims that there exists an input which satisfies the circuit's constraints. In order to create arithmetic circuits a Domain-Specific Language (DSL) is used. Examples for DSLs include Circom, Halo2, ZoKrates, Leo, Zinc and Snarky [3]. In those languages circuits are defined which represent the logic that is to be proven. Then the DSLs build and compile the circuits into SNARK friendly formats like R1CS, AIR and Plonk-CG. [4]

4.3.2 Circom

This section shows how a DSL works by the example of Circom, which is the DSL that is being used in the ZKP-Puzzles application. The syntax in Circom is specifically made to describe arithmetic operations. When writing a Circom circuit, signals are created as variables. Then arithmetic operations are applied on them to express the logic. These operations are turned into constraints, which are indicated by the "===" operator. They have to be satisfied for the proof to be valid. Circom only supports only **linear** and **quadratic** constraints. A *linear constraint* involves variables added or scaled by constants (e.g., $3a + b + 4 = 0$), while a *quadratic constraint* includes multiplication between variables (e.g., $a \cdot b - c = 0$). To better illustrate this there is the following example:

```
template ResultCheck() {
    signal input a;
    signal input b;
    signal input c;
    signal input expectedResult;
    signal output isValid;
    a + b * c === expectedResult;
    isValid <== expectedResult;
}

component main { public [ expectedResult ] } = ResultCheck();
```

Listing 4.1: Circom for result check

This circuit takes three private inputs `a`, `b`, and `c`, and a public input `expectedResult`. The expression

```
a + b * c === expectedResult
```

creates a constraint that must be satisfied. The output `isValid` is set to the expected sum. This allows a prover to convince a verifier that they know three secret values such that the first value plus the product of the second and third equals the public `expectedResult`, without revealing the actual values.

4.3.3 Rank-1 Constraint System

Rank-1 Constraint System (R1CS) is a mathematical format used to represent arithmetic circuits. It expresses constraints using the following form:

$$(A \cdot w) \circ (B \cdot w) = C \cdot w$$

Here, A, B, and C are matrices, which stand for the constraints from the circuit and therefore represent the input logic. w is the witness vector containing both public and private inputs. The $\cdot$ operator represents standard matrix-vector multiplication, and the $\circ$ operator represents the Hadamard product. This structure ensures that both sides of each constraint produce the same result, which is a requirement for the proof to be valid. [4]

4.3.4 Trusted Setup

In many SNARK systems a trusted setup is necessary. The trusted setup process, often called a ceremony, generates a proving key and a verification key. These keys are required by SNARKs. A secret is used to create these keys, which is later deleted after the initial setup. If someone else were to get a hold of this secret they could forge valid-looking proofs.

4.3.5 Powers of Tau

A common way to do the trusted setup is the Powers of Tau ceremony, which is also primarily used for zk-SNARKs, which are explained in a section below. In this ceremony many people work together to create a universal Structured Reference String (SRS), which is a collection of cryptographic data. The entire setup is based on the hardness Elliptic Curve Discrete Logarithm Problem, which states that for a given point P on an elliptic curve and a scaled point kP, it is computationally infeasible to calculate the scalar k. Powers of Tau uses *pairing-friendly elliptic curves* such as *BN254* or *BLS12-381*. [5]

The first participant starts the ceremony by choosing a secret scalar, which is a random number, and applying it to a generator point of such an elliptic curve. This results in a different point on the curve. Every participant contributes to the creation of this universal SRS by exponentiating the point from the previous participant with their own number. Since each participant keeps their scalar secret, no one knows the entire secret. The resulting universal SRS can then be specialized into a proving and verification key for a specific circuit. This is done in a second phase of the setup, called the circuit-specific setup, which utilizes the constraints of the circuit in the form of R1CS. [6] [7]

4.3.6 Universal SNARK

A universal SNARK is a SNARK system that supports multiple computations using a single trusted setup. The same proving and verification keys can be reused across many different computations. In traditional SNARK systems, a new setup is required for each specific circuit. SNARKs with this property are more practical for real-world applications.

4.4 zk-SNARK

A zk-SNARK adds the zero-knowledge component to a regular SNARK.

To create a zk-SNARK proof, an arithmetic circuit is described with a DSL like Circom and compiled in the form of R1CS. A witness is then generated. This is a file that contains the secret and all other values needed to show that the circuit's computation was done correctly. Together with the witness and the universal SRS created during the trusted setup, the proof can be generated. The resulting proof is a tuple of three elliptic points defined over the elliptic curve chosen during the trusted setup. With this proof, the public inputs and the verification key a verifier can check if the proof is sound.

4.4.1 Groth16

The ZKP-Puzzles application uses SnarkJS, which is a framework designed to work with zk-SNARKs. SnarkJS employs Groth16, which is a specific protocol for zk-SNARKs. Groth16 requires a trusted setup and can generate and verify proofs. It produces small proofs of constant size, and therefore has a very fast verification speed, making it well suited for a web application.

4.5 Reasoning & Alternatives

As the framework for generating and verifying ZKPs, SnarkJS was selected for its direct compatibility with JavaScript and TypeScript. Since these are the core technologies used in the application, SnarkJS is the best choice. It was decided to use Circom as a DSL because it is designed specifically for Groth16-based zk-SNARKs like SnarkJS and produces R1CS, which is a requirement for using SnarkJS. However there are multiple alternatives for the chosen technologies, that can also accomplish the same tasks:

- **Other zk-SNARK frameworks**: ZoKrates and Libsnark are popular zk-SNARK frameworks. ZoKrates focuses on the Plonk proof system and the integration with python. Libsnark is a C++ library and includes multiple zk-SNARK constructions like Groth16. These alternatives are not as suitable with our application as SnarkJS, because they are tailored to different programming languages.
- **zk-STARK**: zk-STARK stands for Zero-Knowledge Scalable Transparent Arguments of Knowledge. The key difference between them and zk-SNARKs is, that zk-STARKs are built on hashes and not elliptic curves. As a consequence no trusted setup is necessary. This technology was not chosen, because, in zk-STARKs, the resulting proof size is very large and therefore they are not suitable for web based proof verification. Another reason to use SNARKs instead of STARKs in a web environment is that SNARKs utilize elliptic curves which are well supported by web browsers and STARKs use hashes, which are harder to optimize in a browser. [8]
- **Bulletproofs**: Bulletproofs focus on range proofs, allowing a prover to prove that a certain number lies within a specified range. They do not require a trusted setup but are not suited for creating complex constraints and are therefore not usable for our application. This limitation can be addressed with Universal Bulletproofs, which support more complex and general computations beyond range proofs. While Universal Bulletproofs also do not require a trusted setup, they tend to be less efficient when it comes to verifying proofs.
- **PLONKish & Halo2**: PLONKish (UltraPLONK extension) is an upgraded version of PLONK which is an arithmetization like Groth16. PLONKish however not only allows for polynomial constraints but also in general allows for much deeper customization. Halo2 utilizes PLONKish to create it's circuits which are structured much differently when compared to Groth16. A Halo2 circuit utilizes it's constraint system where a gate is created from multiple constraints and added to the constraint system. The data used is stored in the chips. Each chip is a rectangular matrix (rows/-columns) with each column corresponding to a certain data-type, there are fixed, advice and instance columns. The fixed columns contain constants or selectors for

gates. A gate selector determines which gate is turned on or off and the constants are numeric values such as for instance a multiplication coefficient. In an advice column the prover stores the private values (witness) and finally the instance column contains the public input and output values. The chip is split into regions and these regions are determined by the gates that have been previously defined. For instance if the entire chip has one of each columns and there are two gates, one requiring a fixed value and 2 advice values and the other one requiring only a fixed value. Then the layout for the regions would be that for the first gate two rows are occupied by the region to provide the fixed cell as well as the two advice cells and one row for the second gate to provide the fixed cell. The chips are composed in a tree with the top level chip specifying and distributing its columns to the lower level chips. This allows for the functionalities of the chips to be combined. Lastly there are gadgets which function both as a facade and as an adapter allowing the circuit to have a simpler way of interacting with the chips functionalities. [9]

As much as this arithmetization is more customizable it is equally more complicated to construct which is why we decided against using it and utilizing Circom with Groth16 instead. The greatest difference between the two however is that Groth16 only allows for linear and quadratic constraints and PLONK allows for polynomial ones. This means had we chosen PLONK we could've most likely reduced the amount of constraints.

Chapter 5

Architecture

In this chapter the architectural structure of the project is described and explains the reasoning for key design decisions.

5.1 Technology Decisions

In general, the technologies not directly related to the ZKPs have been chosen based on experience and personal preferences.

5.1.1 Frontend

TypeScript: With JavaScript being the standard for web applications, we have decided to use TypeScript as we are already familiar with it and it offers additional benefits such as type safety, easier refactoring, and better IDE support.

React: Because from a users perspective the application will only be for solving puzzles it makes sense to create a single page application. Through experience react is our preferred library in assisting in creating a clean and easy to use user interface.

Anime.js: If necessary and if time permits, Anime.js will be able to add a final polish to further enhance the user experience.

5.1.2 Backend

In conjunction with the React-TypeScript frontend it makes sense to use Node.js and Express.js in the backend as these technologies harmonize well and work great to create a single page application.

Node.js: In order to use JavaScript persistently through the project Node.js is used. It enables asynchronous, non blocking operations to handle multiple requests efficiently.

Express.js: Express.js will be used as middleware for its lightweight and simple approach to connect our application to our database.

Gitlab OAuth: In order to ensure security when it comes to logins, the login process will be "outsourced" to gitlab using the OAuth functionality. This allows us to have unique users without managing sensitive login data ourselves.

5.1.3 Database

The database will store the user data, puzzles and connect the two in a separate table for each solved puzzle. Further, the database needs to be able to support multiple read and write connections at once since it is a web application. Considering the nature of this project being security oriented, it is wise to choose a database that ensures reliable handling of transactions with full ACID compliance.

Atomicity: This ensures that a transaction is only accepted if all parts of the transaction succeed. If only one part fails the entire transaction is aborted.

Consistency: After a transaction the database remains in a valid state. This ensures that no partial or corrupted state can occur.

Isolation: Whenever there are concurrent transactions they simulate sequential order to avoid any data corruption from an overlap.

Durability: Once a transaction is completed it is permanently saved, even if it crashes thereafter.

For these reasons the conclusion was drawn to work with **PostgreSQL**. The database is run inside a **Docker** container so that it can easily be terminated and restarted if necessary.

5.2 C4 Model Architecture

These architecture diagrams follow the C4 model. We've chosen to focus only on the first two levels—Context and Container—since Components and Code are likely to change frequently. We believe a high-level overview of our architecture is sufficient for this project.

5.2.1 C4 Context

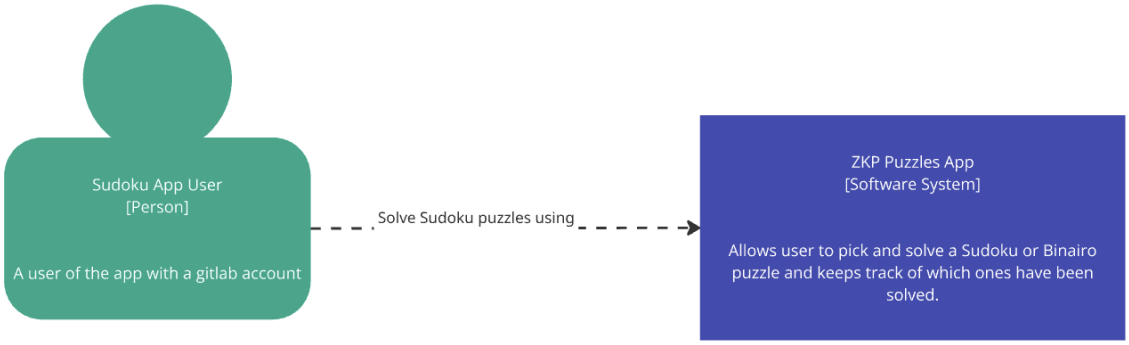

Figure 5.1: C4 Context

5.2.2 C4 Container

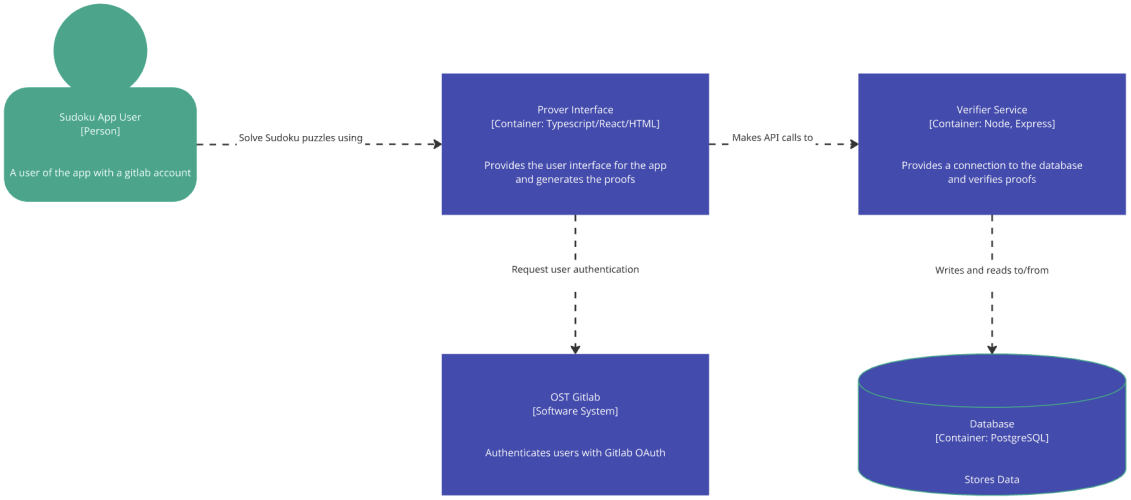

Figure 5.2: C4 Container

5.3 Database

The following ERD-diagram showcases the entities, types and relationships that are used in the PostgreSQL database for this project. The Sudoku puzzle itself is being stored as a concatenated string of each row, e.g. "849172653...". Only the unsolved puzzles are stored in the database. When a puzzle is solved a connection is made between the puzzle and the user (solved_puzzle).

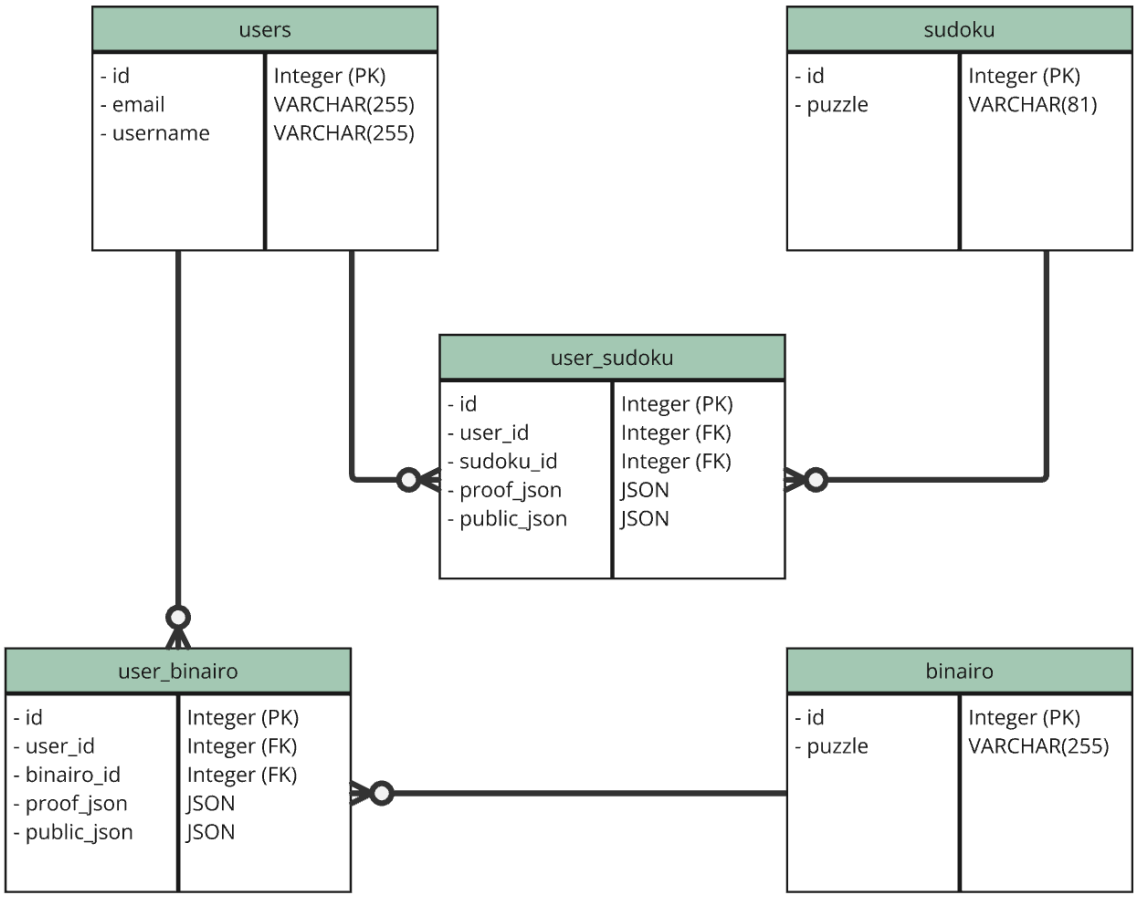

Figure 5.3: Entity Relationship Diagram

In total there is only the users table and a table for each puzzle type. The users and the puzzle tables are each connected with an intermediate table to ensure that only "one-to-many" connections and no "many-to-many" connections are given.

5.4 Application

Since React is used in the frontend the app is constructed using a component-based architecture that uses unidirectional data flow. The application is split up into components that each handle their own logic, state and rendering. This inherently allows for an easier extension and increases the manageability of the application. The unidirectional data flow means that a component can only pass information down to another component and never up. In order to trigger functions from overlying components the function must be given to the component to trigger.

5.5 Extension

Extension in this project would be adding more different types of puzzles. Thanks to Reacts component-based architecture as described beforehand the application is very extensible. However also the generating and verifying of the proofs along with the backend need to be extensible too.

The generating and verifying is equally extensible as it is designed using a modular architecture. For a new puzzle type another folder can be added in front- and backend and the necessary files generated from the circuit must be added. Lastly the database is also extensible by creating two new tables for each new puzzle type, one for the puzzle itself and one as a connection table to the user table to store the users proofs.

5.6 Scaling Performance

The main core of performance optimisation lies within the proof generation and its verification (GV). During the project the basic GV will be further optimised as much as the time allows. In the future further optimisations can be easily be applied by replacing the old files required for GV. If a different system and framework shall be used it would also be possible to replace those though it is more difficult. The biggest issue with replacing the circuit or frameworks entirely is that the existing proofs of the users will be invalid for the new system.

Chapter 6

Implementation

This chapter focuses on the most interesting aspects involved in the development of the application. The entire codebase can be viewed by people with access to the OST's GitLab.

6.1 Login - OAuth

To avoid dealing with login and account management in general, these responsibilities are outsourced to GitLab using its OAuth service. When a user clicks "login" on the landing page, they are redirected to the GitLab login where they must grant the application access to their user data. Next the user is redirected back to the application and in the background an exchange of authorization code and access token takes place to allow the application to get the username from GitLab. With this username the existing user from the applications database is retrieved and if no such user exists a new one is created. This allows for a smooth log-in experience as existing users only need to click the "login" button and are logged in in under 5 seconds. To further explain the exact requests and interactions between the front- and backend as well as GitLab and the applications database the following flow diagram illustrates the process:

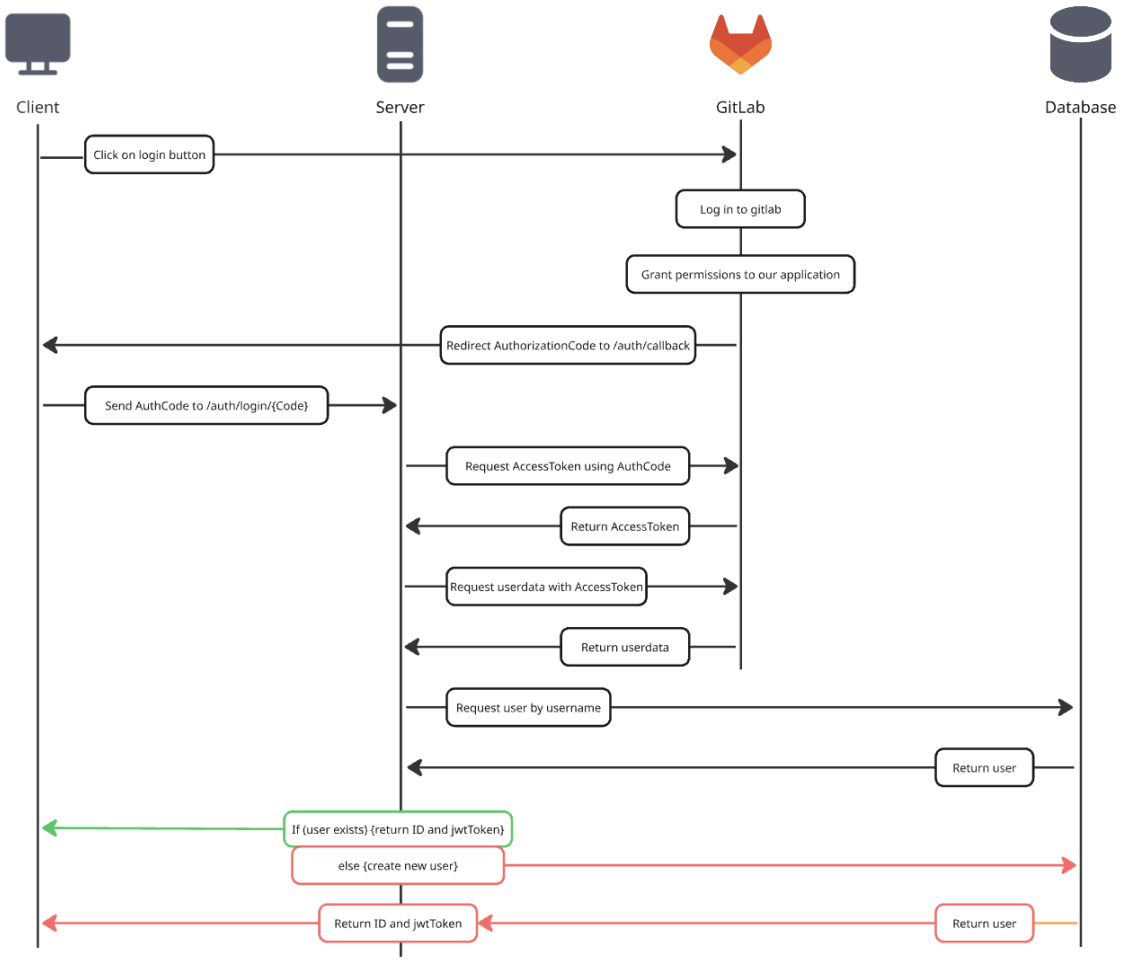

Figure 6.1: Login procedure flowchart

Since the application is a single page application, the callback from GitLab is handled by checking the url once for "/auth/callback" as soon as the page is mounted in the browser.

```
useEffect(() => {
    const path = window.location.pathname;
    if (path === '/auth/callback') {
      setAuthenticating(true);
    }
  }, []);
```

Listing 6.1: Searching the URL for /auth/callback

In the code authenticating is set to true at which point the Callback.tsx component is displayed and it's function is triggered. The callback page is a dark gray background with the words "logging in..." and the function sends the authorization code to the backend as described.

6.2 Maintainable & Extensible

Maintainability and extensibility were weighed more heavily to allow for further puzzles to be added easily. Following up on a suggestion from the advisor, first a ZKP for the Binairo puzzle is implemented in order to gain an understanding of all the components needed and how to use them. Therefore the application was initially developed in an open, extensible way, as the initial idea was to replace the Binairo circuit with the Sudoku circuit once completed. At a later point however it became clear that it makes more sense to keep both puzzles and turn it into a more general puzzle solving application that uses ZKPs for all the puzzles. With this in mind each component was developed as a frame to house a puzzle of any sort.

For instance in the frontend everything is kept dynamic using the PuzzleType enum. The React components then offer perfect extensibility by taking in data and generating the components completely dynamically. Some existing files need to be extended by a new puzzle but beside that only the PuzzleGrid needs to be newly created.

The proof generation itself is already fully dynamic as it only depends on the wasm and zkey files which are all accessed via dynamic paths that depend on the current PuzzleType. After this the proof- and public-JSON are sent to the backend along with the PuzzleType where the verification runs dynamically by the exact same principle.

6.3 Circuit Choice

A major decision for this project was which frameworks to use for the circuit as well as the generating and verifying of the proofs. In the beginning the advisor informed us of Circom and snarkjs which are the most common used frameworks. During our research, a big contender was Halo2, which uses PLONKish as described previously in 4.5. Halo2 offers full control and custom gates that allow for a low-level, performant circuit. Further, it offers the possibility to write constraints as polynomial equations rather than just linear and quadratic, as seen in Circom. This is a major difference as shown in the following example, where the number of constraints of a part of the Binairo circuit can be reduced from 36 to just 1 by chaining them all together with multiplication. In this example the constraint checks if there are any three neighbouring cells that contain the same value which would violate the Binairo rules. The tripletSumRow is the sum of 3 horizontally neighbouring cells and tripletSumCol is the same but vertically.

```
noTripletsRow[y*size + x] <== (tripletSumRow - 1) * (tripletSumRow - 2);
noTripletsColumn[y*size + x] <== (tripletSumCol - 1) * (tripletSumCol - 2);

//Reduced to:
singleConstraint <== (tripletSumRow0 - 1) * (tripletSumRow0 - 2) * ( ↵
tripletSumCol0 - 1) * (tripletSumCol0 - 2) * (tripletSumRow1 ...
```

Listing 6.2: Quadratic vs polynomial constraint

However, constructing a circuit in Halo2 requires much more code and is much more difficult to grasp for someone who has no experience in this topic at all. Given that the university does not offer any ZKP courses and given the very limited time to create this project it was decided to use Circom and SNARK over Halo2.

6.4 Constraint development

For each constraint it is made sure that the calculation results in 0 if the submitted solution is correct. This detail adheres to the underlying mathematical structure and increases the efficiency of the proof generation and verification. ZKP circuits, especially ones which result in an R1CS like Circom, create a system of polynomial equations to generate the proof. These polynomials are a series of quadratic equations in the form of A * B = C, where each such equation represents a computation. If a constraint is created and for instance it is known that for this constraint which is just a sum of two values, the values must always result in 20 otherwise the constraint is not met. In such cases, the constraint can be expressed as x + y - 20 = 0. In this manner it can be injected into the quadratic equation as (A = 1) * (B = x + y - 20) = (C = 0).

6.5 Proving to Verifying Process

While solving a puzzle the user can check the solution at any time. There is no detailed feedback on the provided solution as the solution is immediately run through the constraint system which fails if any constraint is not met or succeeds if every constraint is met. This means it is unclear which constraint failed and where the problem lies. To illustrate the process from proving to verifying the proof a simple flow chart has been created. The exact process is described below:

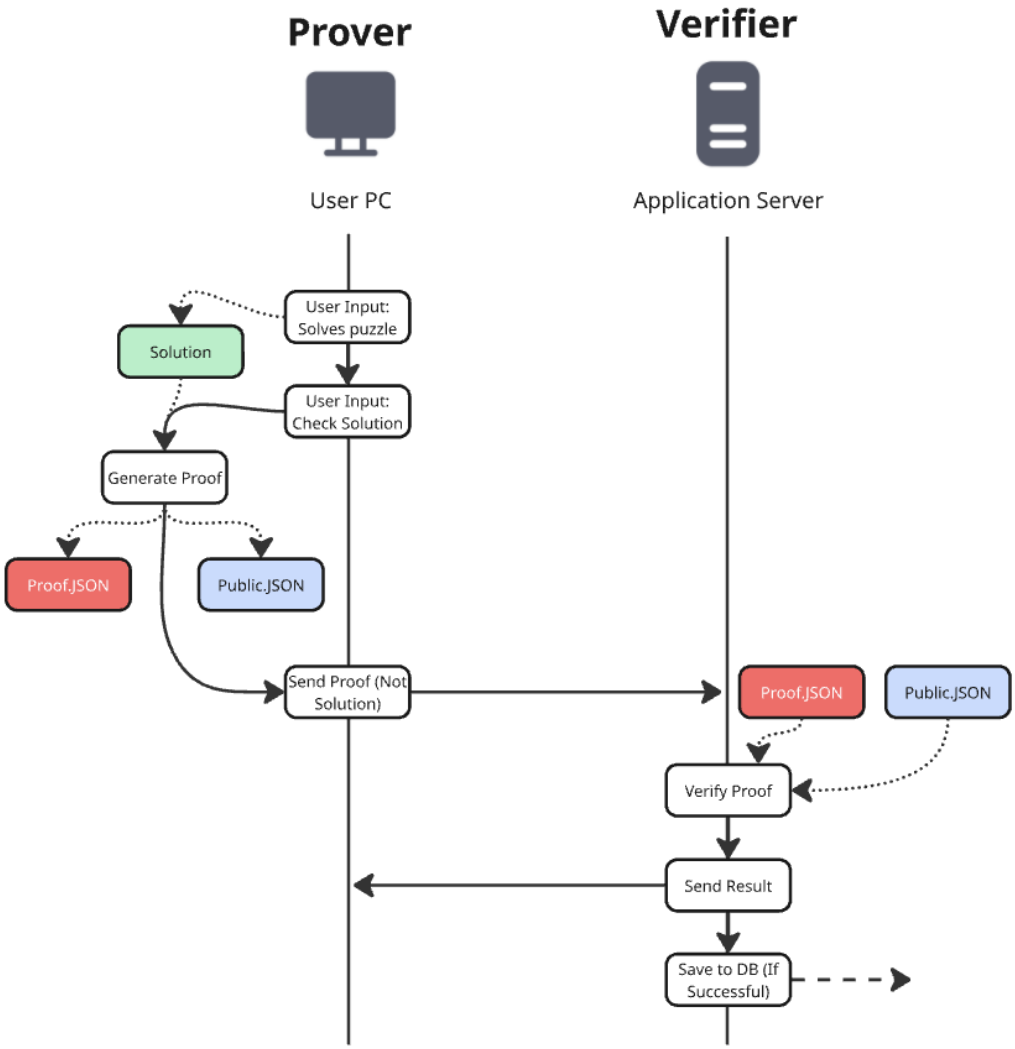

Figure 6.2: Simplified flowchart from proving to verifying

Using the snarkjs framework, the groth16 proof is automatically generated using the wasm and zkey files along with the solution of the puzzle. The results are the proof and publicSignals objects which are then sent together with the PuzzleType to the backend via a POST request to "/check-solution". If proof generation fails however, an error is thrown after which an appropriate message is displayed for the user.

On the server's side the verification is run using an npx command that is executed in a shell as follows:

```
1 const command = `npx snarkjs groth16 verify ${path.join(myPath, " ↵
  verification_key.json")} ${path.join(myPath, "public.json")} ${path.join( ↵
  myPath, "proof.json")}`;
2 const shell = process.platform === "win32" ? "powershell.exe" : "/bin/bash";
3
4 exec(command, { cwd: myPath, shell } ...
```

Listing 6.3: Executing the verification command

If an error occurs a response of status 500 is returned. If instead the proof is successfully verified then the userId and the original puzzle are extracted from the publicJSON. The puzzles id is requested from the database using the puzzle string which is all cell values appended into one string starting from top left to bottom right. Should no such puzzle exist in the database then the response status 404 is returned. On successful return of the puzzles id a new entry is created in the user_puzzleName table. The entry includes the userId, puzzleId, proofJSON and publicJSON. In order to prevent users from posting other users solutions the userId that is used to create the entry is the one found in the publicJSON. This means should someone manipulate the publicJSON to match the userId to their own id then the verification would fail. Otherwise, if they do not manipulate the file, the userId of the other user is read from the publicJSON and the solution is saved as an entry for the other user. To prevent duplicate issues with posting the solution multiple times, the entries in the user_puzzleName tables are unique where "on conflict" nothing happens. Lastly a message declaring the success is displayed for the user.

6.6 Sudoku Circuit

The Sudoku circuit is what connects the puzzle logic from the Sudoku to the ZKP. It is written in Circom and later compiled into R1CS. It is made of constraints that describe the logic of a Sudoku. Each row, column and box is individually checked to ensure that each number from 1 to 9 is present exactly once. The following code snippet demonstrates how each column is verified.

The function processes each column individually. For every column, it initializes an array of size 9 called *colHasEveryNumber*, with entires set to 0. Then it iterates over each cell in the column and performs nine equality checks over each cell using the *eqCol* component. This component takes the value of the current cell and compares it against each number from 1 to 9. If the cell's value matches the current number, *eqCol* returns 1, otherwise it returns 0. This result is added to the *i - 1*-th index of *colHasEveryNumber* where i is the number being checked. As a result, each entry in *colHasEveryNumber* reflects how many times a particular number (from 1 to 9) appears in the column. After iterating over all cells in the column, the program verifies that every entry in *colHasEveryNumber* is exactly 1 and therefore each number from 1 to 9 appears exactly once.

Similar logic applies to rows and boxes, with different ways to iterate over the grid.

```
for (var x = 0; x < 9; x++){
        var colHasEveryNumber[GRID_SIZE_MAX] = [0,0,0,0,0,0,0,0,0];
        for (var y = 0; y < 9; y++){
            for (var i = 1; i <= 9; i++) {
                eqCol[x][y][i-1].in[0] <== grid[x][y];
                eqCol[x][y][i-1].in[1] <== i;
                colHasEveryNumber[i - 1] = colHasEveryNumber[i - 1] + eqCol[ ↵
x][y][i-1].out;
            }
        }
        for (var i = 0; i < 9; i++) {
            colHasEveryNumber[i] === 1;
        }
    }
```

Listing 6.4: Example for column check

Last but not least there is a check to verify that the submitted Sudoku is not only a valid solution but also matches the unsolved Sudoku that was given. To do this the *isEqual* template from the publicly available circomlib is utilized. The final circuit consists of 4617 quadratic constraints, which is a medium sized circuit. The size directly impacts proving and verification speeds and has been optimized as much as possible.

6.7 Compiling The Circuit

Many of the files mentioned in this section are already described in chapter 4 and will not be further explained.

When the circuit is ready to be tested it is easiest to try it out on `https://zkrepl.dev/`. However once it needs to be implemented into an application the files should be generated manually to ensure full safety.

6.7.1 Generating Assisting Files

Step 1

First the *.r1cs*, *.wasm* and *.sym* are generated from the circuit. The symbol file maps constraint indices to lines in the circuit (Example: 2,2,2,main.original[0][0]) and is not necessary to generate or verify a proof but can be useful for debugging. Further, the generate_witness.js and witness_calculator.js are generated into a subfolder labeled by your circuits name _js and used later on in the proof generating process.

```
circom sudoku.circom --r1cs --wasm --sym
```

Listing 6.5: Generate r1cs, wasm and symbol file

Step 2

Next the r1cs file needs to be tied to the the powers of tau (SRS) to make it ready for usage with groth16.

```
npx snarkjs groth16 setup sudoku.r1cs pot16_final.ptau sudoku_0000.zkey
```

Listing 6.6: Trusted setup phase for groth16 protocol

Step 3

By adding a personal secret in this second phase of the trusted setup one is contributing to the ceremony and increasing the security.

```
npx snarkjs zkc sudoku_0000.zkey sudoku_final.zkey
```

Listing 6.7: Adding entropy

Step 4

Finally the verification key is generated from the final.zkey which is then used in the verification process.

```
npx snarkjs zkev sudoku_final.zkey verification_key.json
```

Listing 6.8: Adding own secret

6.7.2 Manual Proof Generation and Verification

Generate Witness

To generate the proof a witness file is generated using the wasm, input and a witness generating script.

```
node sudoku_js/generate_witness.js sudoku_js/sudoku.wasm sudoku-input.json witness.wtns
```

Listing 6.9: Generating witness

Generate Proof

Then using the final.zkey and the witness the proof.json and public.json are generated.

```
npx snarkjs groth16 prove sudoku_final.zkey witness.wtns proof.json public.json
```

Listing 6.10: Generating proof

Verify Proof

Lastly the proof.json and public.json are verified using the verification_key.

```
npx snarkjs groth16 verify verification_key.json public.json proof.json
```

Listing 6.11: Verifying proof

6.8 Animation

One problem with creating an application of which the core feature lies in security, is that (depending on the feature of course), if implemented correctly it is never noticed by the user. As the application is completed it is virtually no different from any other web application for solving puzzles. One can choose a puzzle, solve it and receives feedback based on the solution. To increase the "coolness factor" of the application it is decided that there

must be an option to get a deeper insight into what is going on behind the scenes. In order to depict to the user in a simplified manner that a proof is generated and sent to the server for verification an animation using animejs is created.

The entire animation for the proving and verifying process is contained in it's own component to separate it from the other code. The animations themselves were created using the animate function where a target is selected, css configurations are applied and the transition type set in "easing". To time and space out all the parts of the animation timeouts were used and adjusted to create a flow that is consistent but allows enough time for the user to read and understand what is happening. In the following example a button is blended in for 1 second after a 3 second delay.

```
  setTimeout(() => {
    animate('#continueAnimation', {
      opacity: [0, 1],
      duration: 1000,
      easing: 'easeInOutQuad'
    });
  }, 3000);
```

Listing 6.12: Example animejs code

In some parts of the animation the adding and removing of css classes from target elements was also used to further improve the animation in a simplified way. Lastly whenever the animate function animates an element it does so by editing and adding to the "style" attribute of an element. For this reason in order to reset the animations after a run the animated elements have their style attribute completely removed. This required all the animated elements not to use "style" but instead a css class and to add the css class "animated" in order to separate these elements so that the non-animated elements may still utilize the style attribute.

```
document.querySelectorAll('.animated').forEach(el => {
  el.removeAttribute('style');
});
```

Listing 6.13: Resetting the animated components

It is important to note that the entire animation is not actually performing any action. As soon as "check solution" is clicked the proof is generated, sent to the server and verified. After the response is received an animation is displayed based on whether the entire process was successful or not. This means by the time the animation starts the solution is already verified and saved or declined. [10]

6.9 Usability Testing Procedure

To perform usability tests, test scenarios are first created and then given to the test person. Their purpose is to guide the participant on what task they have to complete.

It is important that the scenarios are formulated in a way that encourages the participant to explore the interface and be able to solve the task on their own.

While the participant is doing the task, the test overseer observes closely and takes notes on any difficulties or usability issues encountered. After the test, there is an interview to review the test results and discuss the overall experience. Participants are also asked to provide additional feedback. Finally, if necessary, a series of general questions are reviewed to ensure certain topics are covered.

6.9.1 Test Scenarios

- You have an OST GitLab account. Please log into the site.
- Now you feel like solving a puzzle. Choose a puzzle and solve it.
- If you have solved your puzzle, can you find a way to check whether you solved your puzzle correctly?
- You've gotten feedback on your puzzle, but you don't understand what the application actually does. Is there a way for you to get more insight?
- You are done with your puzzle and want to see how many puzzles you and other people have solved.
- Before you are done, you feel like solving and verifying a different kind of puzzle.

6.9.2 Final Questions

These questions are only asked in the end if the usability tests and interview have not been able to answer them already.

- Is the design appealing to you and why?
- What are your thoughts on the color-scheme?
- Can you think of a feature that would improve the application?
- On a scale from 1 to 10 (not including 7) how would you rate the overall appearance?

On the last question the number 7 is excluded from the scale as it seems that people are highly inclined to pick the number 7 in order to be polite. However, by excluding the number 7 it is clear if people like (8) or dislike (6) whatever they are being questioned.

6.9.3 Screener Requirements

A screener is a description of a person that is suitable for the usability test. Any person meeting the requirements can be used as a test participant.

Since the tests focus on usability, there is no technical know-how required. A screener must:

- Be familiar with the rules of *Sudoku* and *Binairo*
- Speak English
- Know how to use a computer

6.9.4 Feedback

First test - 24.05.2025

Conclusion: The first usability test revealed that the general usability of the application was good. The participant were able to solve all the test scenarios without any help. However, there were some ambiguities and additional feedback. In the task where the visualize button was supposed to be used, the participants realized that they needed to use this button but didn't fully understand what it did. Another wish was to display the rules of the puzzles even if they were relatively familiar with the puzzle.

Solution: The decision was made to add a "?" icon next to the Visualize button to provide users with additional information. When clicked, two windows will appear. One appears below the Visualize button and includes a short description of the button with a link to the documentation for further information. The other window displays the rules of the current puzzle and is positioned to the right side of the screen if there is enough space, or centered if space is limited.

Second test - 07.06.2025

Resolution: The changes had a positive effect on the previously determined issues regarding the understanding of the application. The participants reacted positively to the new button and no further shortcomings were discovered.

Chapter 7

Results

This chapter includes the final result of the project and takes a look at the different releases. For each release the requirements are verified.

7.1 Releases

Although the development process was continuous, there were key moments when the application reached a state that was stable and contained a new feature for it to be marked as a release.

7.1.1 Alpha

The alpha release is a bare-bones application similar to a proof of concept prototype. The user interface is extremely crude and only the Binairo puzzle is implemented. The proof is generated in the frontend and verified in the backend. However, there are only two different Binairo puzzles.

7.1.2 Beta & Final Submission

The Beta is much like the final submission however the most significant difference being the addition of the user input from the usability testing. In the final weeks of the project the code was cleaned up and slightly optimized in some regions. Therefore the Beta and the final submission both are fully functional states that allow solving both Binairos and Sudoku puzzles as well as checking the high scores or visualizing the proving and verifying process.

7.2 Verify Functional Requirements

To verify the functional requirements the previously defined use cases in subsection 3.1.2 are tested on the application. If the use cases can successfully be performed the functional requirements are approved.

The results for the FR are the same for the Beta and final submission stages. Thus, they are only described once.

UC1: Access

Due to the nature of this project, no real server is rented to host the application. This is however possible to grant users access to the application through a web address. Hereby UC1 is approved.

UC2: Authentication

A user can authenticate themselves through the GitLab OAuth login but it is not possible to set a username in the application itself. However the username is retrieved from the GitLab account and can therefore be changed through GitLab. It is not possible to test this as editing the username has been disabled on our GitLab accounts. Given that the user management has been outsourced in this way the UC2 is considered approved.

UC3: Upload Sudoku

The Sudoku puzzles offered in the puzzle selection can all be solved and uploaded for verification. Therefore UC3 is approved.

UC4: Verify the Sudoku solution

As with UC3 when "uploading" a solution it is automatically verified which is why UC4 is also approved.

UC5: See the proving process

When visualize is enabled it is possible to see an animation showing what is happening in the background when a solution is uploaded. This animation shows in a simplified way how the proving and verifying process works. Hereby UC5 is approved.

Conclusion: All use cases are approved which means the application offers functionalities that cover all planned use cases.

7.3 Verify Non Functional Requirements

Considering the Alpha release is equivalent to a prototype the decision is made not to test the NFRs at this stage. Therefore the NFRs are only tested for the Beta and "Final Submission" stages. The tests are run according to the definitions in section 3.2.2.

7.3.1 Beta

NFR-1

On each of the listed browsers a puzzle can be solved and verified correctly. The animation plays as intended and the verification fails on an incomplete or false puzzle as expected. Therefore NFR-1 is approved.

NFR-2

After the jwtToken expires there is a JSON.parse error on screen as soon as any button is clicked that creates a request to the backend. It is not yet clear why this error is showing as the unauthorized responses are being caught after every request. Due to the presence of a user-unfriendly error message, NFR-2 is declined.

NFR-3

It takes exactly 4 clicks from the login page to verifying a solution: Login, Sudoku, 1 (puzzle selection), Check Solution. Therefore NFR-3 is approved.

NFR-4

Several valid Sudoku puzzles are successfully solved and verified as correct. This means the NFR-4 and thereby the completeness is approved.

NFR-5

After testing a false solution 3 times and a correct solution 10 times, the false solution is instant and difficult to time exactly should however be around 200ms. The correct solution however takes consistently under 1 second mostly around 0.85 seconds. This is clearly quite far from the limitation of 10 seconds and therefore NFR-5 is approved. (The specs used are - CPU/GPU: Apple M1, RAM: 16GB, Operating system, macOS sonoma 14.1.1)

NFR-6

NFR-6 is verified by testing five cases as stated in section 3.2.2. In order to test 1 and 2 the application is used normally. To test 3 and 4 a Sudoku is solved and the generated public and proof JSONs are copied to the side. Then the JS-console is opened in the browser and a request is sent manually including the JSON-files after having manipulated them to /check-solution. For the last test a new puzzle is inserted into the database where every field is empty.

Up until these tests are created it is not possible to send an invalid proof to the backend as it is only sent through the application if the generation is a success. During the tests it is noticed that such false proofs that are manually sent to the backend actually pass and return with a status 200 before crashing the backend server - even though the verification fails as shown in the backend server log. The problem is an asynchronous timing issue which allows the code to move on to saving an entry in the database even though the validation is still running. Then it tries to send a response with status 200 and 500 because it both passed and failed at the same time leading to the backend crashing.

1. Not filled out at all - The proof cannot be generated without a valid solution due to failing the constraints. This means it does not even reach verification and is therefore successfully declined.
2. Filled out with false solution - Here also the proof cannot be generated leading to this being declined successfully as well.

Considering the problems in the backend with manual requests NFR-6 is declined.

Conclusion: 2/6 NFRs are declined the others approved.

7.3.2 Final Submission

The NFRs 1, 3, 4 and 5 all remain unchanged and are still approved.

NFR-2

The JSON.parse error is successfully resolved. The functions weren't exited correctly after receiving a response with status 403 from a request. Now there are no known errors being displayed to the user. For this reason NFR-2 is approved

NFR-6

The previous timing error is solved by using util.promisify to create an async version of the exec command which is used to validate the proofs. This way the application waits for the response before continuing with the rest of the code.

```
1 const execPromise = util.promisify(exec);
2 const { stdout, stderr } = await execPromise(command, { cwd: myPath, shell ↵
  });
```

Listing 7.1: Example animejs code

In order to test 1 and 2 the application is used normally. To test 3 and 4 a Sudoku is solved and the generated public and proof JSONs are copied to the side. Then the JS-console is opened in the browser and a request is sent manually including the JSON-files after having manipulated them to /check-solution.

1. **Not filled out at all** - The proof cannot be generated without a valid solution due to failing the constraints. This means it does not even reach verification and is therefore successfully declined.
2. **Filled out with false solution** - Here also the proof cannot be generated leading to this being correctly rejected as well.
3. **Filled out but changing userId** - The validation successfully fails thanks to the integration of the userId in the proof generation process.
4. **Filled out but changing puzzle** - The validation successfully declines here for the same reason however in relation to the unsolved puzzle being integrated.

Considering all 4 tests of erroneous proofs are all declined successfully by the verification NFR-6 is approved.

Conclusion: All NFRs are successfully approved.

7.4 Final Product

7.4.1 Login

The login page has one button "login" through which the entire login process through GitLab is triggered.

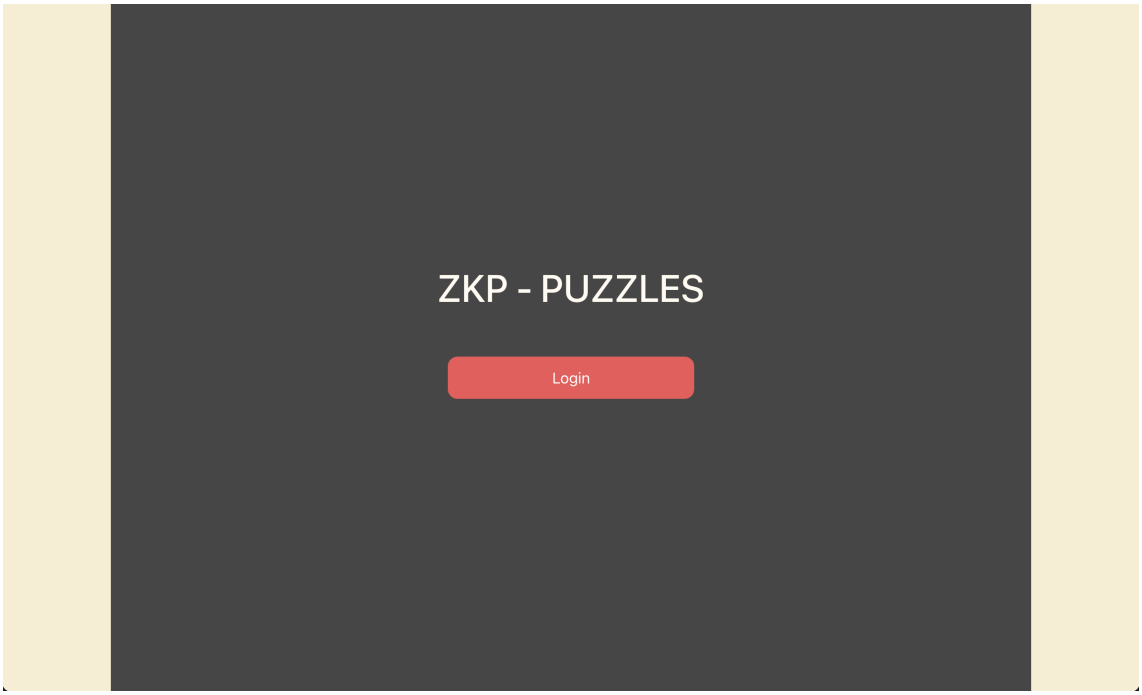

Figure 7.1: Login page.

7.4.2 Main Menu

The main menu is the only page that allows you to log out. It is also the only page that does not display the back button. On the main menu one can choose to check the high scores, play a Binairo or a Sudoku.

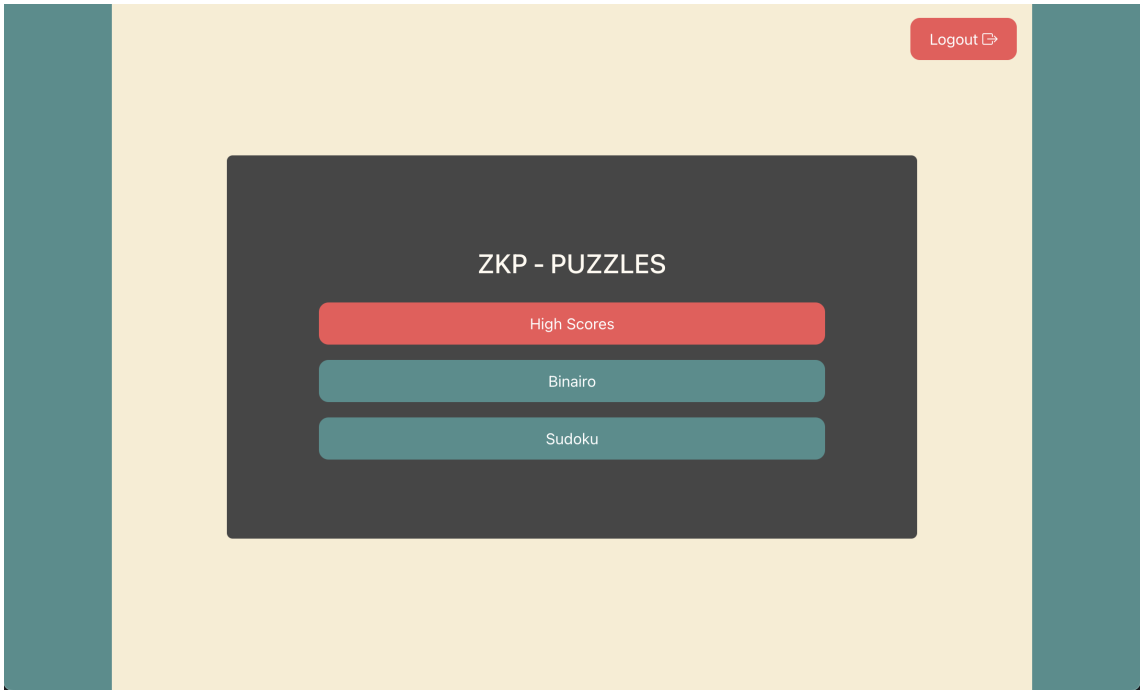

Figure 7.2: Main menu page.

7.4.3 High Scores

As mentioned in 6.2 the application is designed to be further extended. For this reason the high scores are displayed per puzzle in its own high score component. This way they can be extended by many more puzzle types. The first place is depicted with a trophy icon and a teal background, the second place gets a coral background and all the following users have no background color. In this example there are no further users.

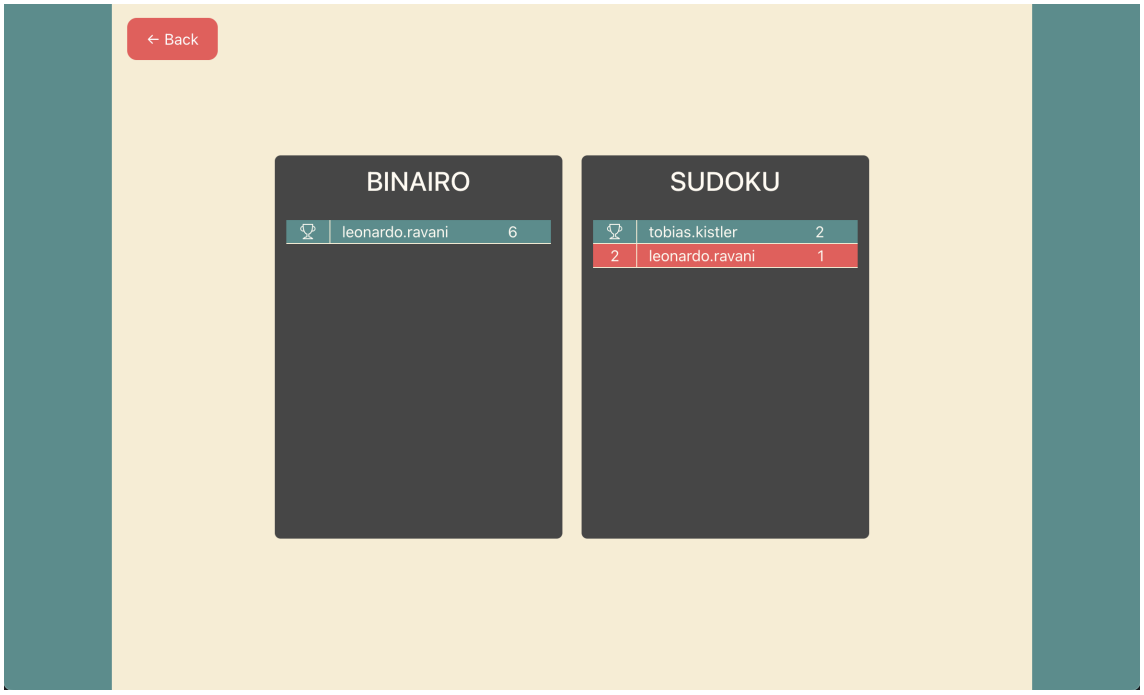

Figure 7.3: High scores page.

7.4.4 Puzzle Selection

Each puzzle available is listed in the box and displayed with a teal background to signify an unsolved puzzle or dark gray if it has already been solved. The puzzles can however be solved as many times as a user would enjoy.

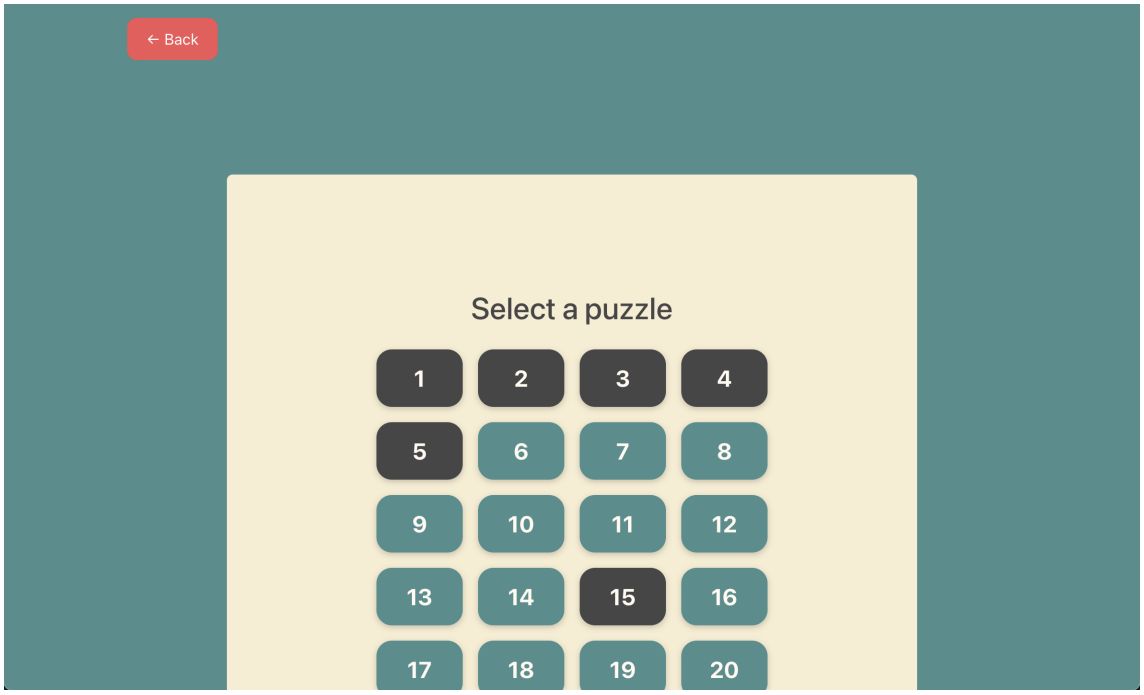

Figure 7.4: Puzzle selection page.

7.4.5 Binairo

The empty fields are marked with an X and a dark gray background. The fields that are part of the unsolved puzzle have an icon in the center to signify that these are unchangeable and correct to begin with. This ensures that if a user makes a mistake they can see which fields are definitely correct. Further below the "check solution" there is an "auto solve" button which allows for the puzzle to be filled out with the correct solution. The auto solve is however only shown and usable on puzzles for which the solution has been written down. The visualize button in the top right corner allows you to toggle the animation on or off. When it is toggled off there is simply a red or green flash with the text "WRONG" or "SOLUTION SAVED" to signal to the user if the solution is correct.

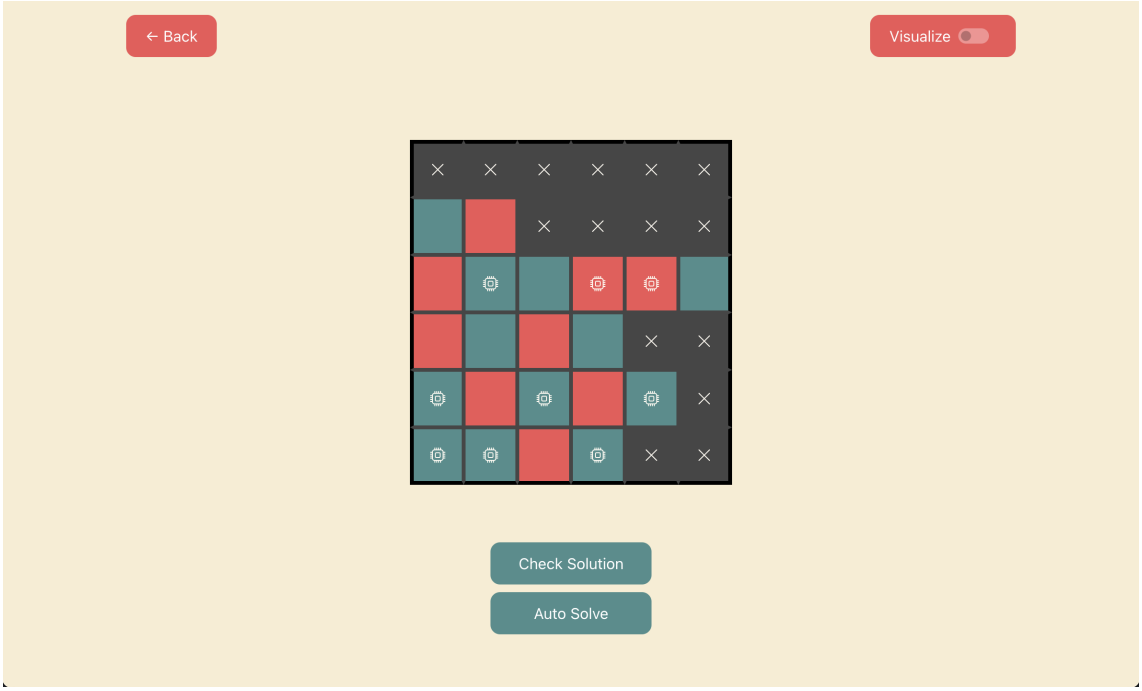

Figure 7.5: Binairo solving page.

7.4.6 Sudoku

The Sudoku page offers the exact same features as the Binairo page with the only difference being the puzzle itself. Here the fields which are part of the unsolved puzzle are marked with a red background while the fields that need to be filled in are marked with a teal background.

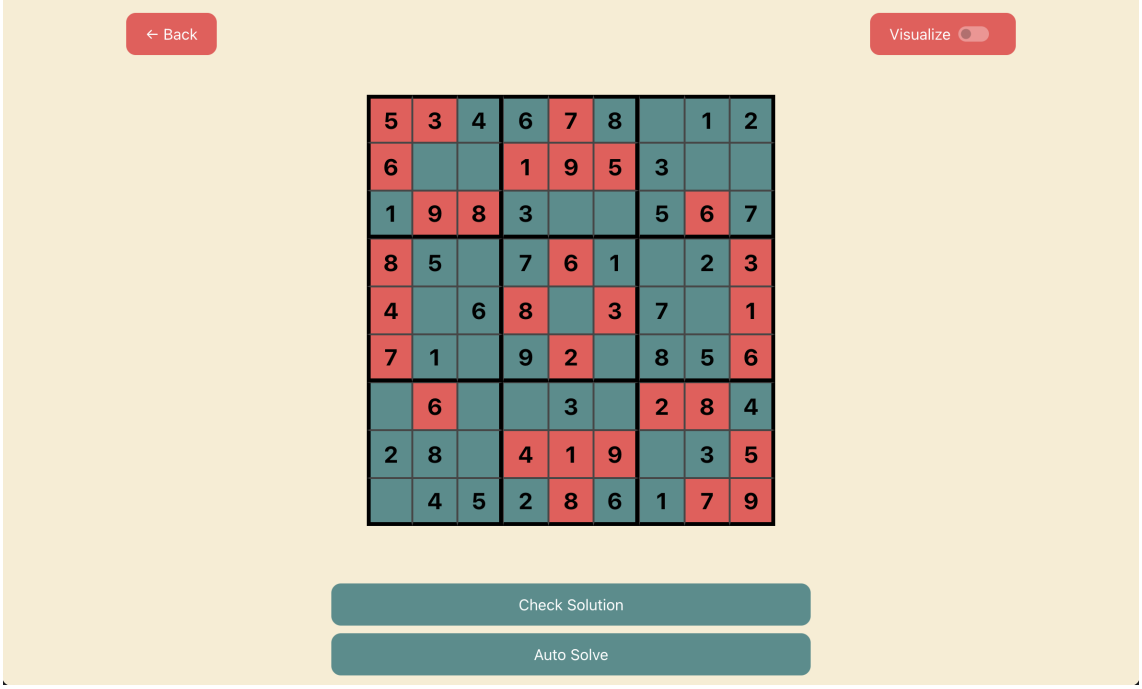

Figure 7.6: Sudoku solving page.

7.4.7 Animation

The animation can be toggled on or off using the "visualize" toggle in the top right corner of the application while on a puzzle solving page. For each step of the animation an image is provided and the process is described for better clarity.

Upon toggling the button and clicking "check solution" the rules for the puzzle according to which the constraints are calculated are displayed in a box that comes floating in from the top of the screen and are then checked or crossed off. As described previously while generating the proof it is not possible to get any details on which constraints failed. For this reason to validate the rules properly a client side validation has been implemented which validates the grid in JS and adjusts the rules accordingly.

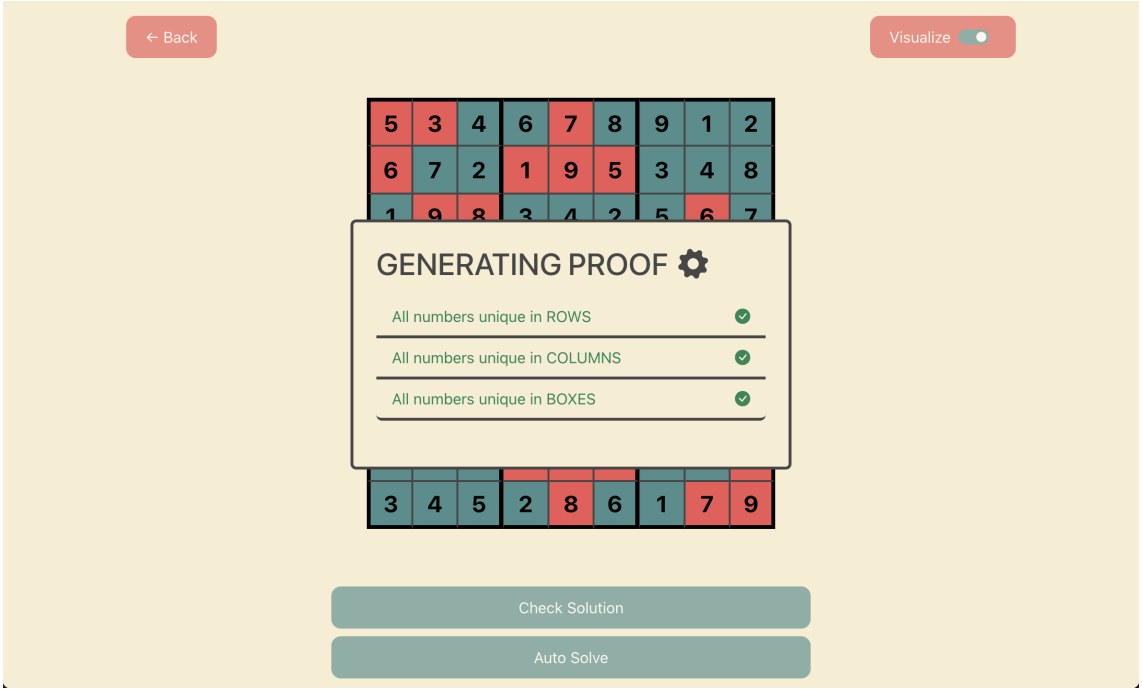

Figure 7.7: The proof is being generated.

Next three boxes expand from the center to fill out the screen. The box on the left displays the actual solution that has been provided by the user. The solution is however blurred out to signify that it is being kept secret. On the right hand side the proof.json and public.json are displayed for the user to understand what information travels to the server. After a few seconds a continue button is displayed.

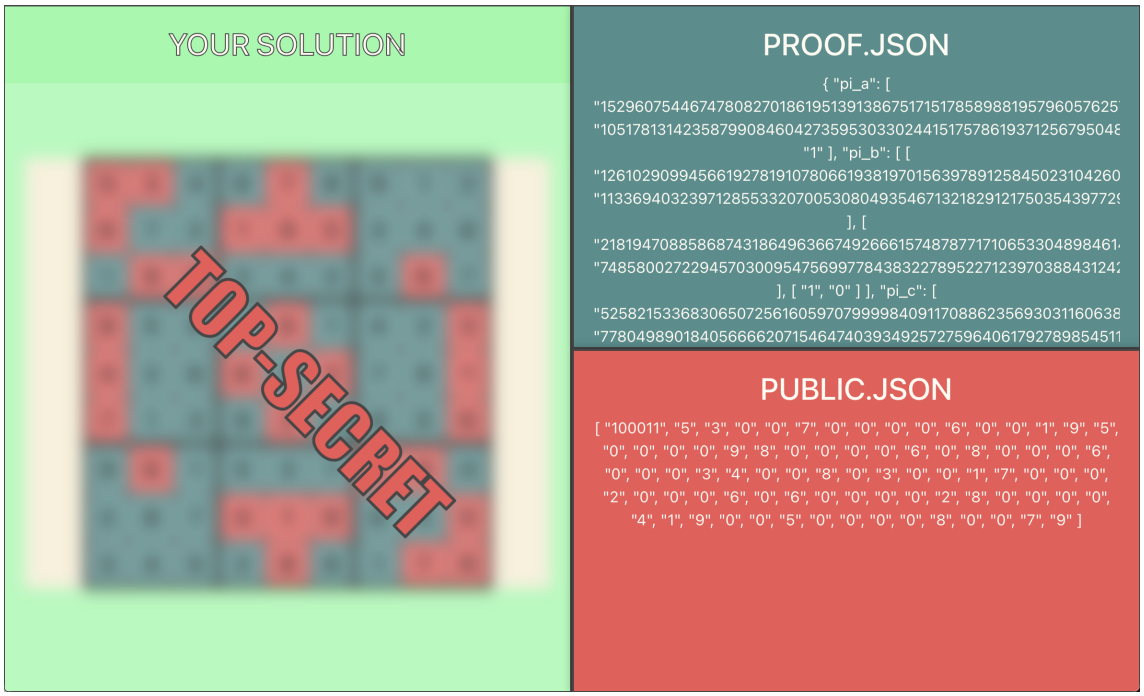

Figure 7.8: Hidden solution, proof.json and public.json are displayed.

For the last part of the animation the boxes all shrink down into small circles where the proof and public JSON move to the right showing that they are sent to the server while the solution one remains on the client. After that another box with rules, this time the rules of the verifier floats in from the top covering up the server. Here as with the rule box from before each rule is checked off and lastly a green flash covers the screen displaying "solution saved".

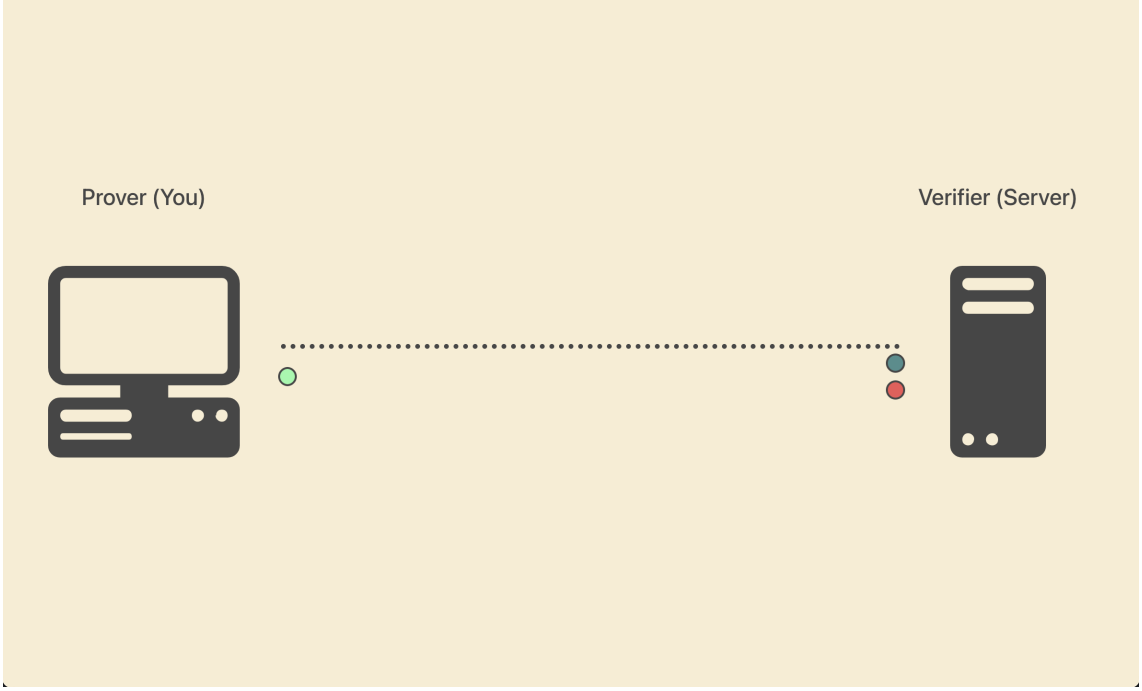

Figure 7.9: Prover holds solution, verifier holds proof.json and public.json.

In the case that the solution has errors and the proof cannot be generated the animation stops after crossing off the broken rules in the box and displaying a humorous GIF from Jurassic Park, featuring a cartoonish version of the character Dennis Nedry waving his finger. The gif has been downloaded from Tenor which allows for free non-commercial use as stated in the terms of service.

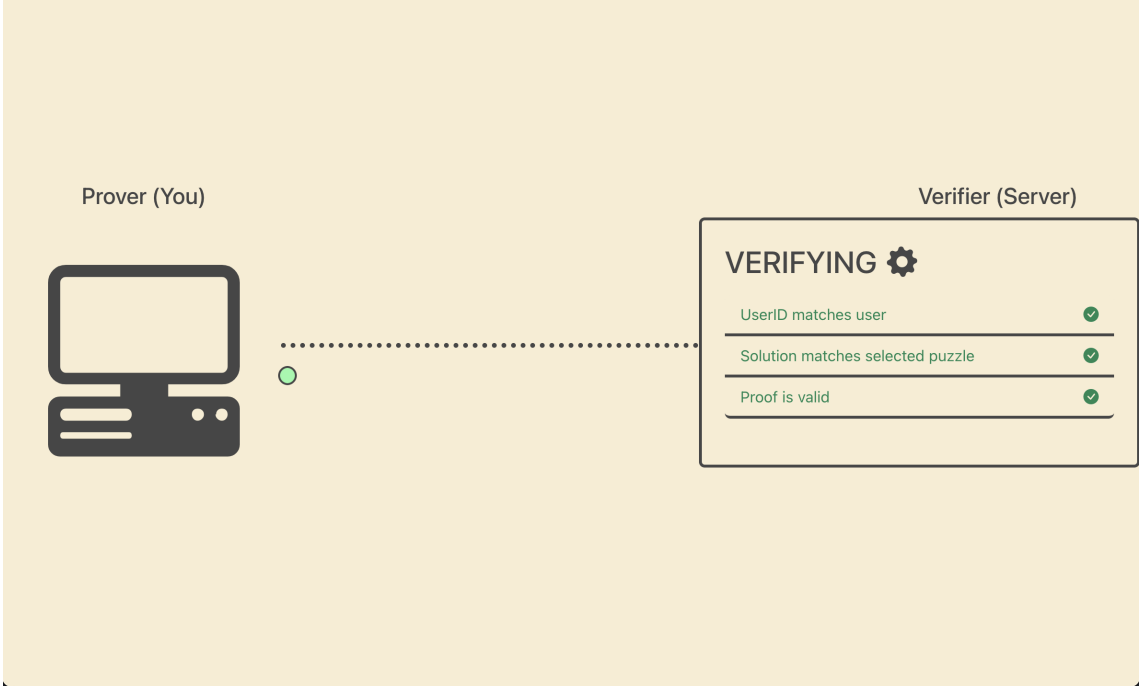

Figure 7.10: The solution is being verified.

Chapter 8

Conclusion

This section reflects on the result of the project and outlines potential enhancements and future use cases for the ZKP-Puzzles application.

8.1 Reflection

This project successfully fulfills the initial project assignment of becoming familiar with ZKPs and applying them to a Sudoku puzzle application. All the tasks outlined in the project assignment have been completed, but of course it is always possible to add more. The animation of the proving and verifying process can be considered an additional feature as the first idea was to just display the files that are generated in the process instead. Future possible additions to the application are described in the outlook below. The Chapter "Zero-Knowledge Proofs" presents the theoretical foundations about zero-knowledge and shows how ZKPs are applied in existing libraries. The focus there lies on the technology that is used in the ZKP-Puzzles application, however zero-knowledge is much bigger than that and a lot could be written about it. In addition, all the functional- and non-functional requirements are fulfilled and approved. One unresolved issue remains due to time constraints, which is that the browser history navigation (page back, page forward) is not fully functional. Instead users must rely on the "Back" button integrated in the application.

8.2 Outlook

There are currently no plans to keep developing the ZKP-Puzzles application in the future, as its primary purpose was to promote learning about the practical application of zero-knowledge systems. As the ZKP-Puzzles application does not have an active user base, there is no need to further develop it. However there are ideas to improve and expand the application. The following use cases show potential features that could be added:

UC6: Download proof
The user wants to download the proof so that they can show it to other people.

UC7: Upload proof
The user wants to upload the proof that they have downloaded before. They can see what their proof is for.

UC8: Verify high scores on device
The user does not trust the server and wants to receive proofs of other users' solutions in order to verify them themselves and build a leaderboard accordingly.

UC9: Adapt the application to mobile devices
The design must be adapted to mobile device screen sizes.

UC10: Add more puzzle types
One thing that was an important aspect during development, was making the application extensible. It was built in a way that allows for the easy addition of new puzzle types in the future, without the need for structural changes.

UC11: Mount application on a server
Deploying the application on a server would allow users to gain access through a browser without local setup, enabling external use.

Chapter 9

Additional Artifacts

This chapter presents supplementary artifacts that functioned as a base for the design and development of the application and elaborates on the design choices.

9.1 Low Fidelity Mockups

Given that sleek and simple designs are very modern and give a very clean impression, it is decided that only low fidelity mockups are necessary to map out a rough guideline for the development of the design. Many details have changed due to finding more appealing adjustments or even due to a change in how the underlying component is programmed. Since it is a small team of two people, no additional adjustments were made to the mockups throughout the development even if the design changed, since it would have been redundant. The colors are purposefully left open, although the hopes for a colorful turquoise were set, which were almost met, however, by implementing a muted teal. The changes from these mockups to the final product are all based on design preference. There are two very noticeable differences. For one, the title "Bimaru" since in the early stages of the project there was some confusion around which puzzle applies which rules and only by the beginning of the Beta stage was the title changed to "Binairo". As for the other difference, the way the JSON data is displayed in the mockups is unappealing and not as engaging as the animation. Important to note is that the mockup for the animation was created at a later stage than the rest of the mockups.

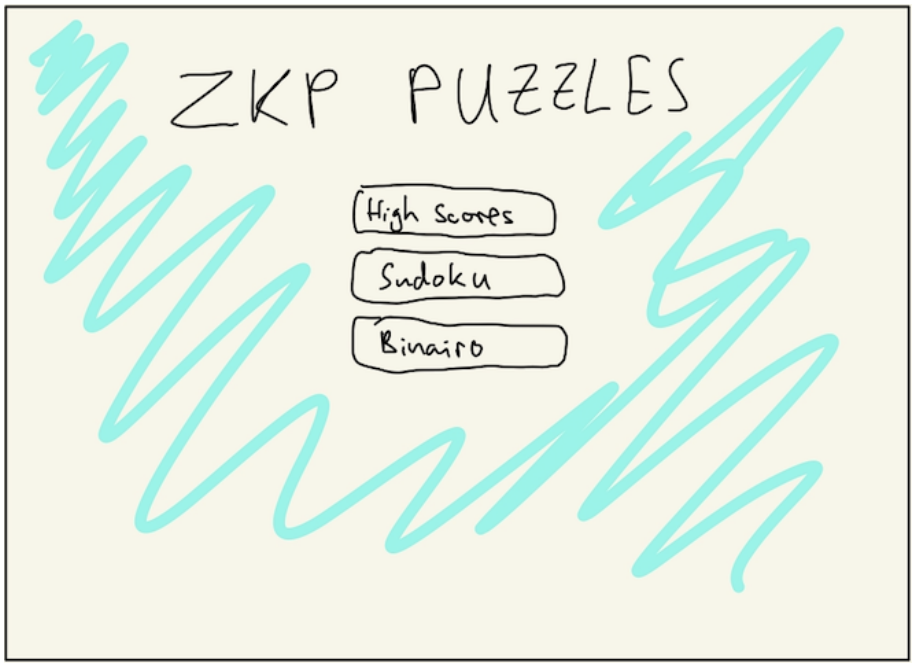

Figure 9.1: Main page

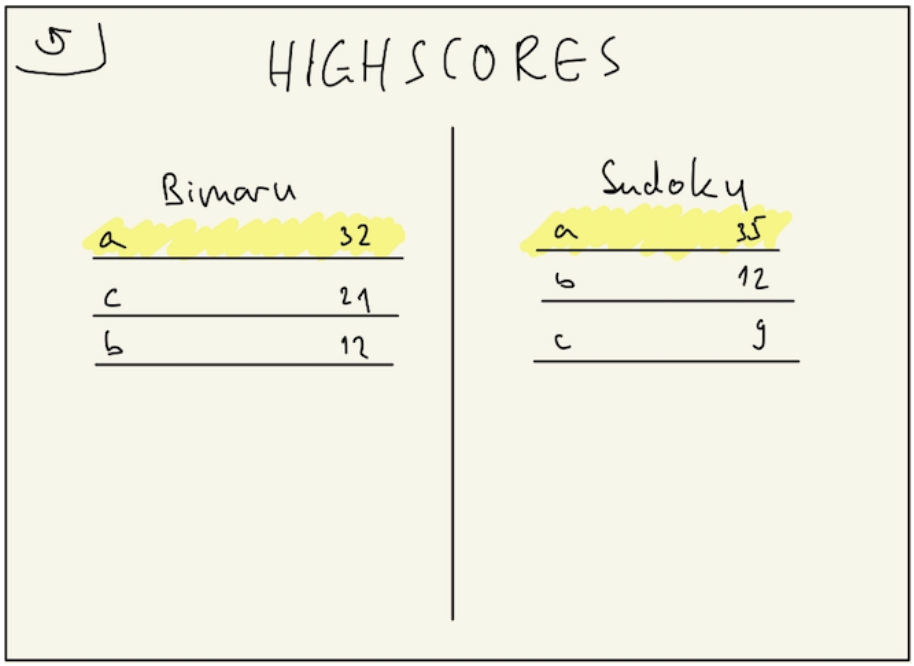

Figure 9.2: High Scores Page

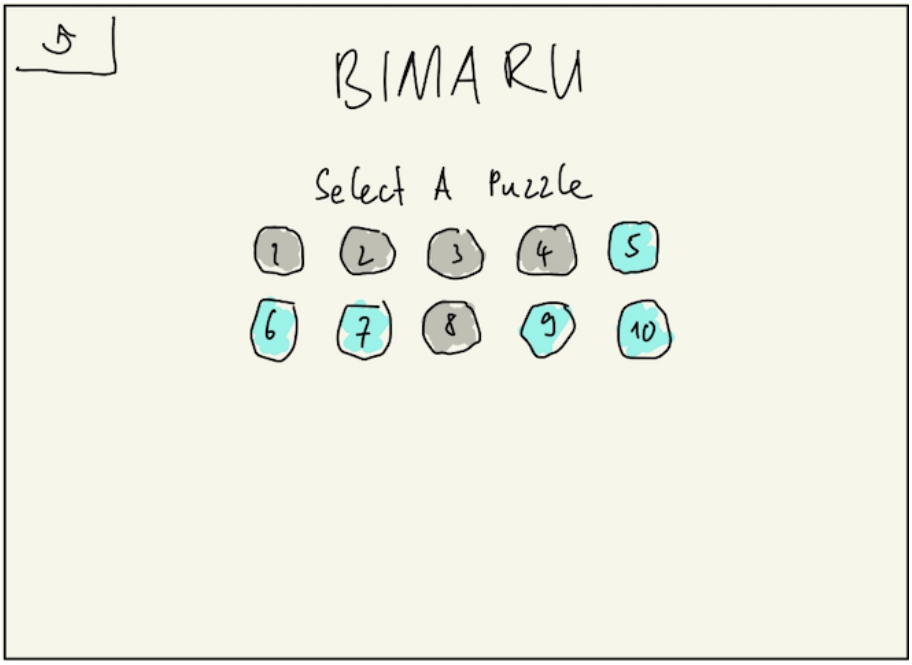

Figure 9.3: Puzzle selection for Bimaru page

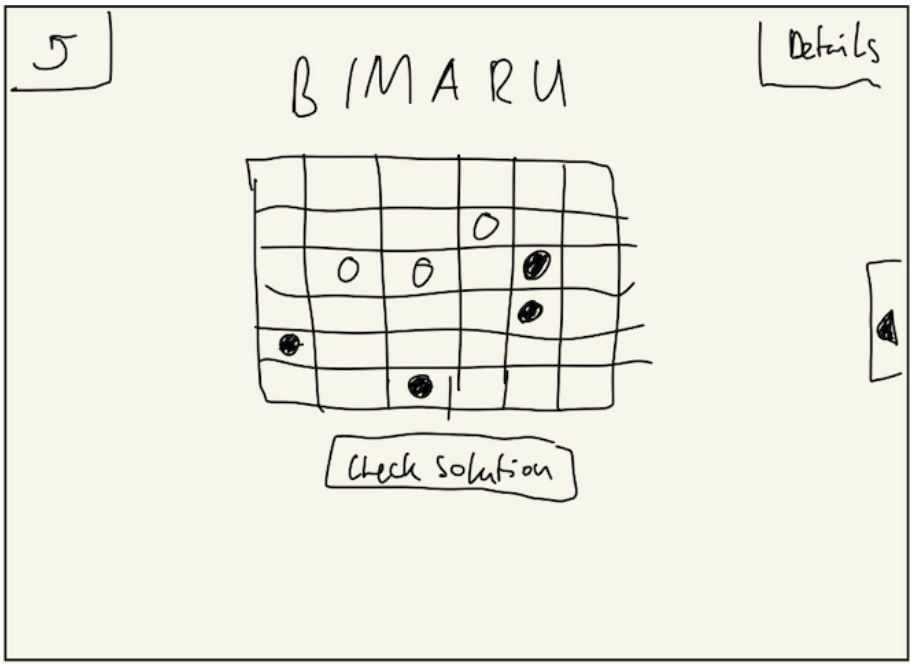

Figure 9.4: Puzzle solving page for Bimaru

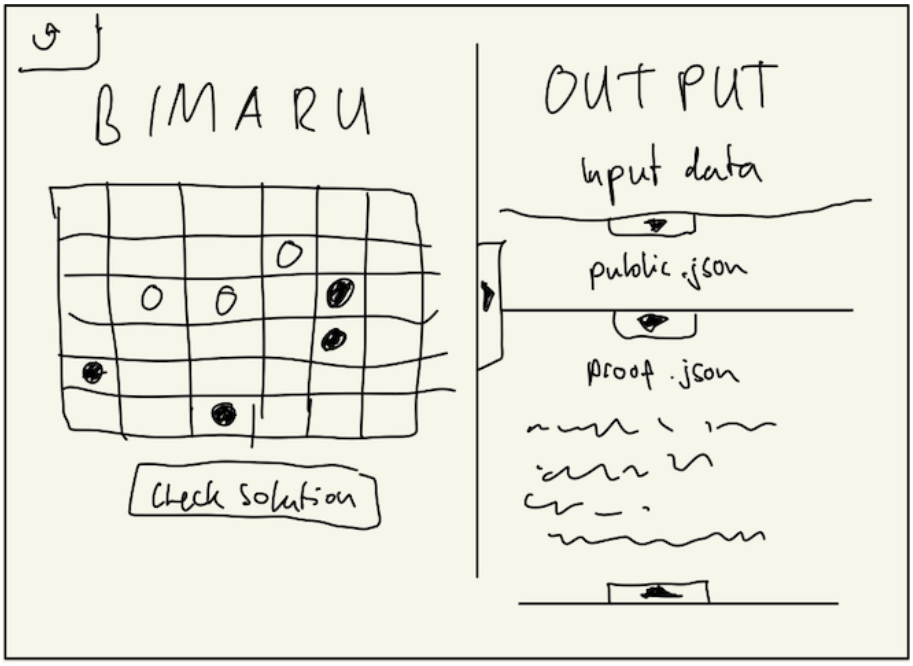

Figure 9.5: Details tab opened

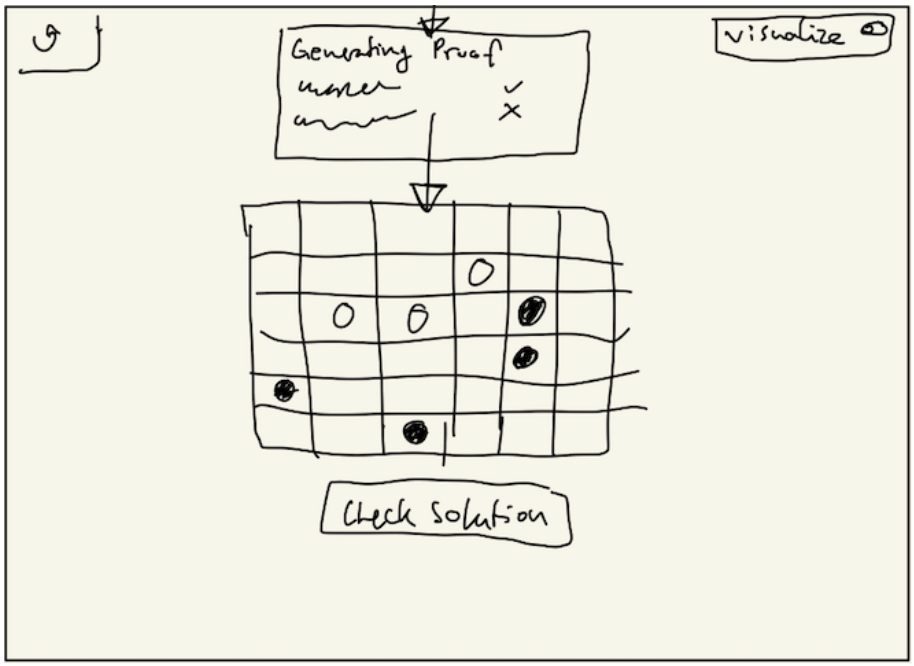

Figure 9.6: Animation step 1 - generating proof

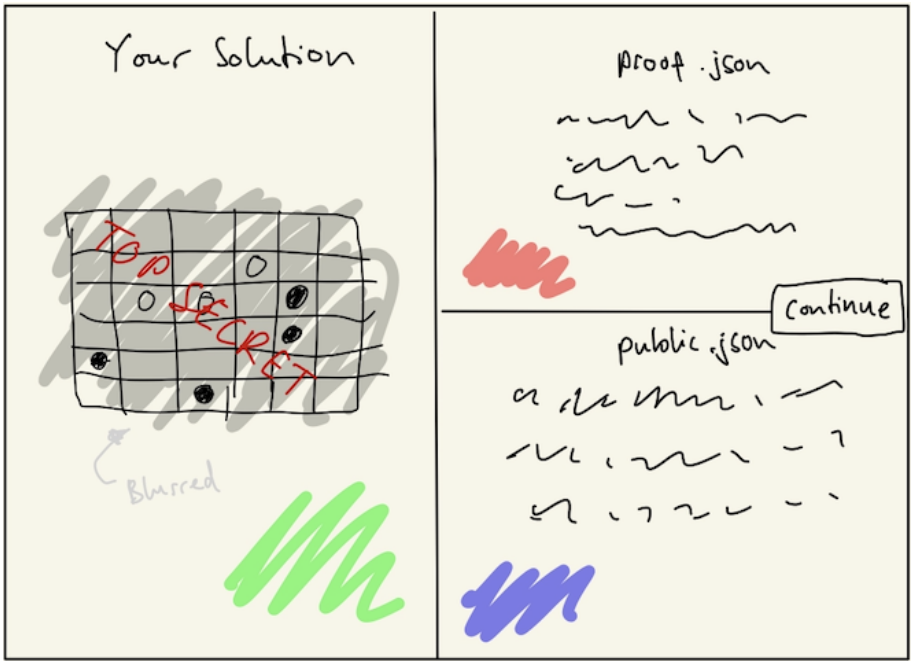

Figure 9.7: Animation step 2 - displaying details

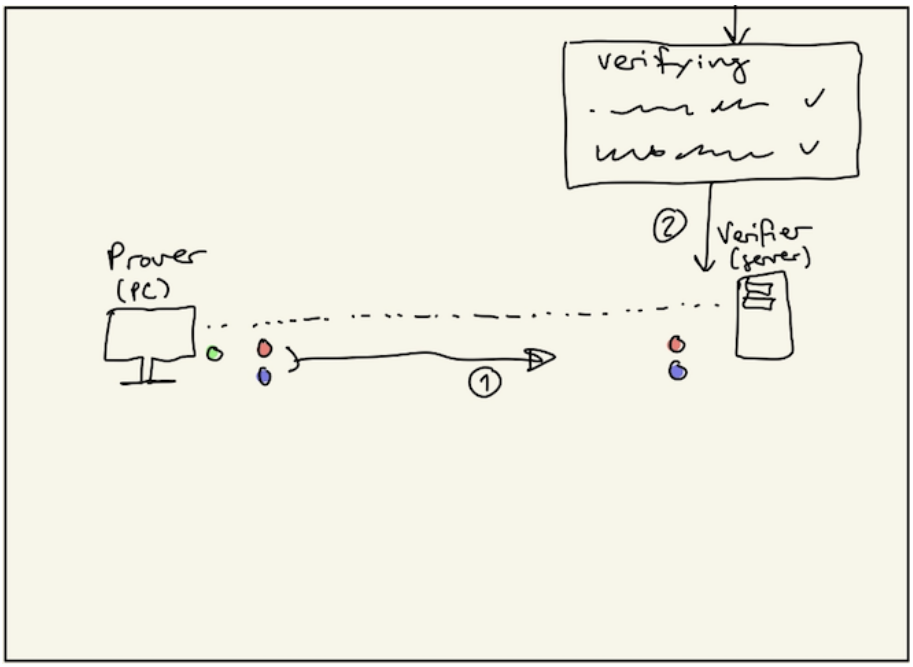

Figure 9.8: Animation step 3 - verify proof

9.2 Logo & Color Scheme

Fitting with the sleek and simplistic design the logo represents fields of a puzzle as they are most often squares of a grid. Each field is one of the primary colors used throughout the application.

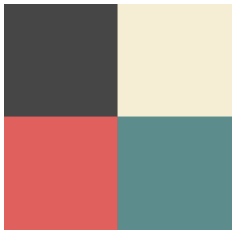

Figure 9.9: The logo

The primary colors are chosen as:

- #464646 – Deep charcoal gray: Very neutral while offering great contrast to white or ivory.
- #f8EDD2 – Ivory: Offers a bright color to contrast the gray. It mixes very well with the teal and coral, offering a more colorful alternative to white or a lighter gray.
- #4B8E8D – Muted teal: Muted means it was toned down using black, white or a mix of both. It remains a relatively vibrant cold color while avoiding being lurid.
- #F05656 – Coral red: It's a muted red which suits the other colors nicely for the same reason the teal was muted. Red and Blue in general offer a great contrast which makes them stand out much more next to each other.

The colors were hand-picked to compliment the sleek, clean design of the frontend while keeping playful elements as it is a puzzle solving application after all.

Chapter 10

Project Planning

This chapter describes the methodologies and tools used in the process of planning the project and compares them to the actual course of the project.

10.1 Planning

This section details project's planning, showcasing the structured meetings and cycles of development as well as a clear breakdown of the milestones.

10.1.1 Methodology

To manage the project the OST-made Scrum+ is utilized (a combination of Rational Unified Process (RUP) and Scrum). This allows us to keep the long-term plan in sight while remaining agile enough to make any necessary adjustments in case any of the risks were to occur or the time estimates are done poorly.

10.1.2 Roles and Responsibilities

Considering that only two people are working on the project it was decided to leave out Scrums traditional roles in favour of assigning responsibilities instead. As it is most important that both students work somewhat equal amounts on all parts of the project to grow an equal understanding of the topic, code and everything else that is connected, it doesn't make sense to assign fixed roles. What's left is only to assign responsibilities, i.e. tasks that can only be performed by one person. These tasks include:

"Leading scrum meetings" - Leonardo Ravani

"Writing meeting minutes" - Tobias Kistler

10.1.3 Meetings

Since the project is working with Scrum+ most of the meetings will be related to it.

- Meetings with the advisor: Every Friday 11:30 - 12:00

- Sprint Review/Retro: Every second Friday 12:00 - 12:30

- Sprint Planning: Every second Friday 12:30 - 13:30

- Weekly Sync: Every other Friday 12:00 - 13:00, Tuesday 11:00 - 12:00

10.1.4 Long-Term Plan

In order to plan ahead, a long-term plan has been established. This plan is a rough estimate and acts as a guide throughout the project. In the end of the project there is also a buffer which, if not needed, can be used to improve the app or add additional features.

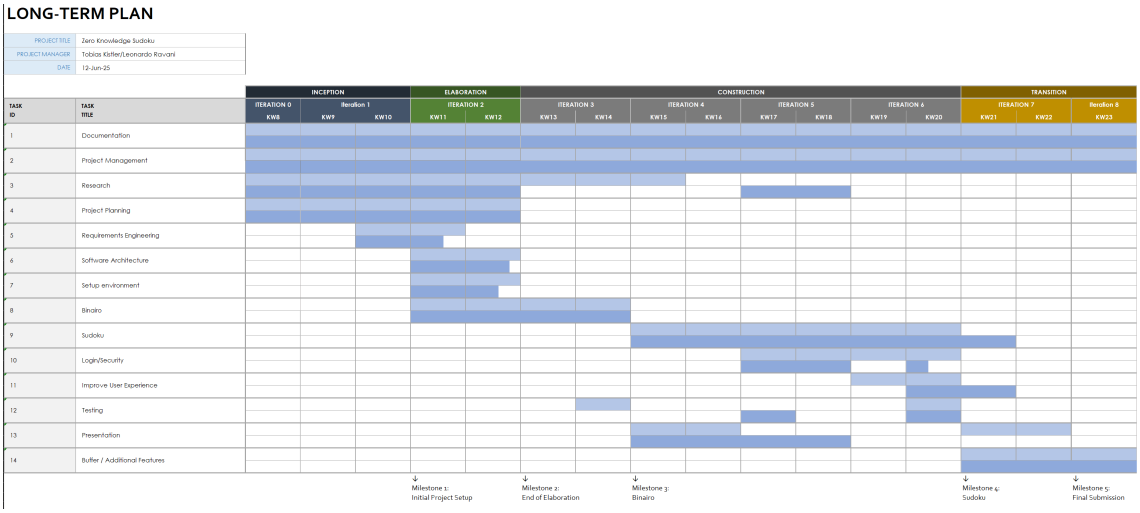

Figure 10.1: Long-term planning

10.1.5 Milestones

The project is divided into milestones to show how and when major activities and requirements are achieved.

Milestone 1: Initial Project Setup

The goal of the first milestone is to define the most important parts of the workflow, documentation and doing enough research for a general understanding.

Planned Deliverables

- Define and assign roles
- Plan necessary meetings
- Create long and short-term plan
- Define tooling
- Define workflow
- Identify and assess risk
- Research ZKP, circom and snarkjs

Milestone 2: End of Elaboration

This milestone focuses on the elaboration of appropriate functional and non-functional requirements of the project. A basic circuit for the binairo example should also be available at this point.

Planned Deliverables

- Define functional requirements
- Define non functional requirements
- Planning of these requirements
- Define quality measures
- Create basic circom circuit for Binairo

Milestone 3: Binairo

With this milestone, the aim is to complete the Binairo ZKP example so that the knowledge gained may be transferred to the Sudoku example.

Planned Deliverables

- Create frontend to solve Binairos
- Create slightly optimised ZKP circuit
- Create backend to verify circuit
- Functioning Demo for mid-term presentation

Milestone 4: Sudoku

The Sudoku milestone is the same as the Binairo milestone but for the Sudoku example. A higher grade of optimisation is focused on compared to the Binairo example.

Planned Deliverables

- Create frontend to solve Sudokus
- Create optimised ZKP circuit
- Create backend to verify circuit
- User experience optimisation
- Finalized research

Milestone 5: Final Submission

With this milestone, the final version of the Sudoku ZKP is delivered. This version should function correctly, bug free and be highly optimised. At this stage the application should represent a real life example.

Planned Deliverables

- Finish remaining features and add optional ones
- Finalized documentation

10.1.6 Short-Term Plan

Because this project is developed using an agile concept the short-term plan is essentially each sprint. The starting point of each iteration can be viewed in the long-term plan. Each sprint with the exception of 2 (first and last) has a length of two weeks. By the end of each sprint a meeting is held to plan the next one thereby creating the short term plan for the next to weeks. New tasks may be added during the sprint if more time is available, due to the planned tasks being completed in a shorter time than expected. Each task in the long term plan represents an epic which is further split up into tasks when planning a sprint.

10.2 Tooling

This section describes what kind of tools have been used to plan the project and keep an overview of the project progress.

10.2.1 Tracking

Jira is used to track issues and work hours. It is a project management tool that is especially useful when using Scrum. Jira provides a board that shows the workflow of the active sprint, a backlog containing all planned tasks, and time-tracking. Time is tracked on each ticket separately, and the total can be viewed in the "Worklogs Report." This helps to see which task has taken how much time and measure overall progress.

10.2.2 Time Tracking Results

As stated in the Bachelor's guide from the Department of Computer Science, the expected workload for a Bachelor's thesis should be 30 hours per credit resulting in 360 hours in a semester. The diagram below illustrates the actual workload of each team member. Leonardo exceeded the expected workload with 371.45 hours, while Tobias fell slightly below with 354.25 hours.

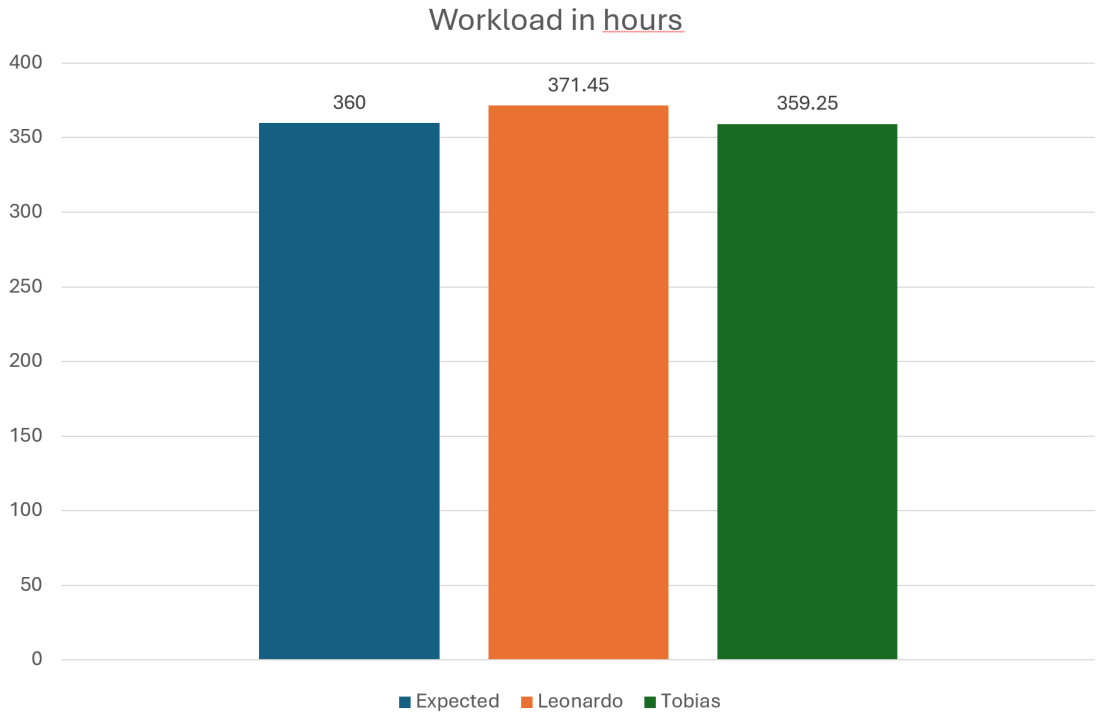

Figure 10.2: Workload by member

As time tracking was managed through Jira, with each ticket assigned an Epic, it is possible to analyze how much time was spent on each aspect of the project. The largest portion of time was spent on building the application with 329.25 hours making up 45% of the total workload. The second-largest category was documentation, with 221 hours, which accounts for 30.2%. Research followed with 96 hours (13.1%) and project management accounted for 60.5 hours (8.3%). The remaining 24.25 hours (3.3%) were spent on various task, such as preparing for the interim presentation.

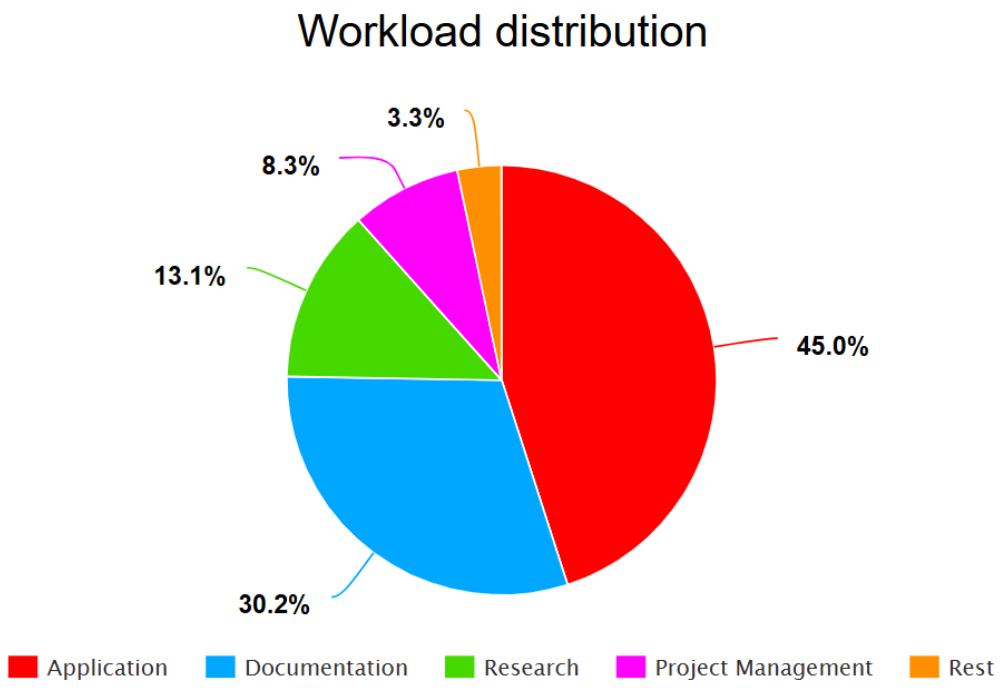

Figure 10.3: Workload distribution

10.2.3 Workflow

In order to keep track of our progress throughout the sprint, we defined a workflow. This workflow describes which states the ticket is going through until the task is done. When a new sprint is started, all tickets are initially set in the "TODO" section. They are either assigned to a person or remain unassigned, which means the ticket is to be done by both developers. If the developer wants to start working on a ticket, they move the ticket to "IN PROGRESS." This means that the task has been started but has not been completed. As soon as the task is done, the ticket can be moved to "IN REVIEW." There, the other developer can see that the task is completed and can review and test the work. If any issues are found, the ticket will be moved back to "IN PROGRESS." This process is repeated until the review is passed. Then the task is done, and the ticket can be moved to "DONE."

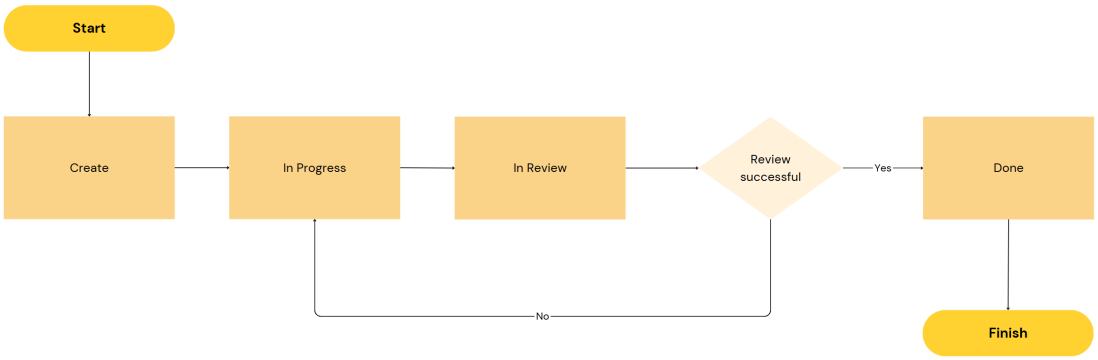

Figure 10.4: Jira ticket workflow

10.2.4 Directory Of Resources

Literature research and management	Google, swisscovery, Springer, ChatGPT
Data analysis and visualization	PowerPoint, Excel, draw.io, Sora, DBeaver, pgAdmin4, meta-chart.com
Idea generation	None
Translation	Google translate, DeepL, LEO
Prototyping	None
Coding	Visual Studio Code, Create React App, GitLab, ChatGPT
Text optimization, spelling and grammar check	Overleaf, ChatGPT
Collaboration and project management	Jira, MS Teams, WhatsApp
DevOps	Docker (Docker Desktop)

Table 10.1: Directory Of Resources

10.3 Quality Measures

The quality measures describe what actions are taken to improve a good quality of the product.

10.3.1 Git Workflow

To ensure high code standarts, a defined git workflow is used. After the initial setup, any additional features are implemented in a feature branch. These branches are named according to the convention of feature/ZKS-<Jira-ticket-number>-<feature-name>. After a feature is finished the developer resolves all the merge conflicts and starts a merge request. Merge requests are then reviewed by the other developer and are only merged into the main branch after approval. With this method, the risk of having major bugs is very low and the developers are up to date with all the code in the project. Another important upside of this method is that the main branch is always in a stable functioning state.

10.3.2 Test Strategy

Functional Testing: Before each merge request, the feature's functionality is tested manually. Automated functional testing is not planned at this stage, as its implementation would require quite a lot of time. This would be something to consider as an additional feature that is implemented in the end during the buffer if it is not needed to complete other more crucial features. The primary focus of these tests is to verify that the feature functions as described in the functional requirements.

Non-Functional Testing: Non-functional testing primarily focuses on usability testing. These tests are conducted alongside the development of the Sudoku application. However, since Binairo is implemented first, its frontend is already available for testing and improvements. Usability testing involves collecting feedback from potential users, implementing the necessary changes and conducting another round of tests with the same users to validate the improvements.

10.4 Risk Management

The Risk Management will display assumed risks in a graph and project their movement throughout the project according to changes that may positively or negatively influence each risk. The "risk matrix" will only be re-evaluated after a change has definitively affected the risks those changes will be described.

10.4.1 Iteration 1: Basis

This is the basis for the assumed risks that will be encountered throughout the project. It is created during the first iteration of the sprints. Risks are shown in the matrix judged by the damage potential and probability of occurrence. In the following version the risks are shown without any mitigations in mind, after the risks descriptions and mitigations another risk matrix for this first iteration is shown with the mitigations applied.

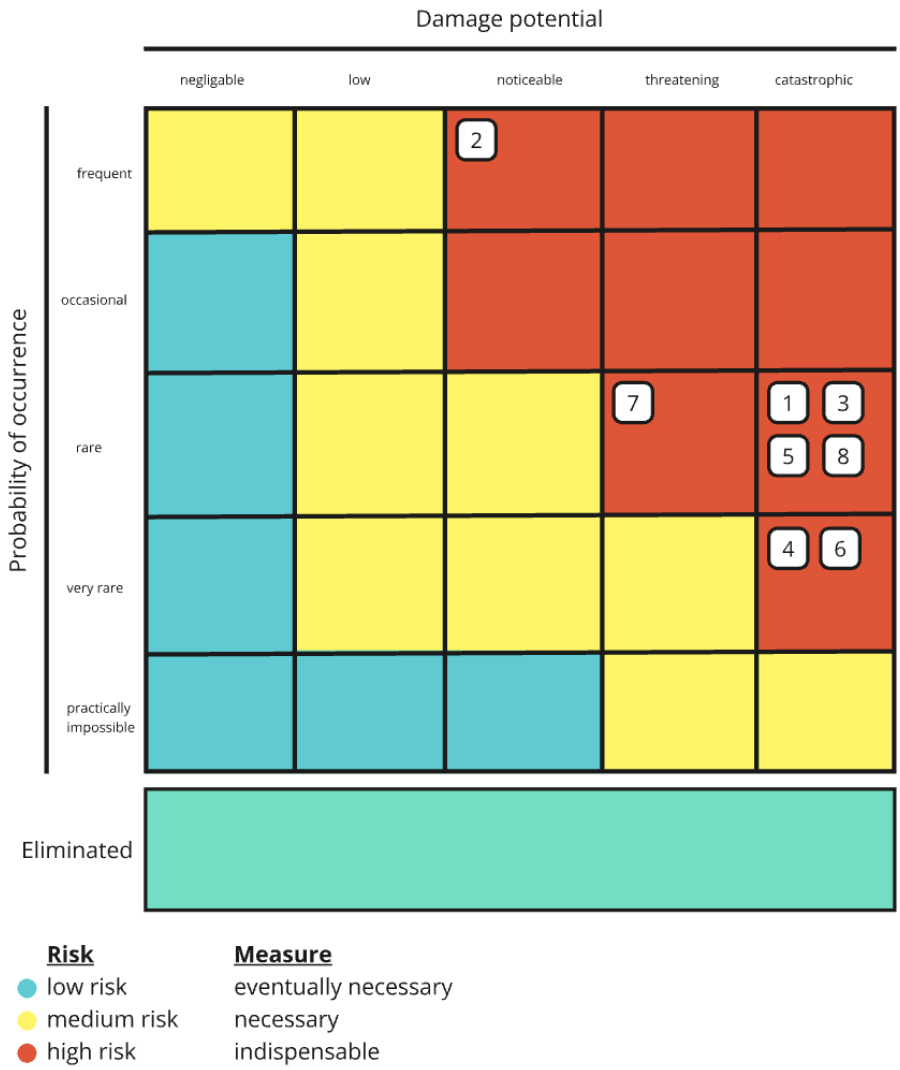

Figure 10.5: Risk Matrix 1 without mitigations

Risks

1. **Incorrect ZKP Implementation:** If the ZKP circuit is incorrectly implemented in a way that might not be noticeable at first, for example, providing true positives every time, but also very rarely false positives, it could lead to invalid proofs.

   Mitigation: Conduct extensive testing with different Sudoku solutions and perform peer code reviews. Performing a thorough research on the utilized libraries.

2. **Slow Proof Generation:** Creating an unoptimised proof generation process or verifier could lead to an impractical verification.

   Mitigation: Benchmark different ZKP proving systems (Groth16, Plonk, Stark), optimize circuit design, and cache precomputed elements.

3. **Backend Downtime:** If the backend server is down, users will be unable to submit their Sudoku solutions.

   Mitigation: Deploy the backend in a high-availability cloud setup, or use native verification with STARK/SNARK etc.

4. **Security Breach in User Authentication:** When working with user accounts there is always a risk of account hijacking.

   Mitigation: Outsource the authentication process by utilizing OAuth or other services.

5. **Fake ZKPs** Malicious users might submit precomputed or fake proofs to cheat the system.

   Mitigation: Utilize challenge-response verification and require the unsolved Sudoku or its ID to be given along with the proof.

6. **Settling with the wrong ZKP system:** Choosing technologies which only later on reveal difficulties or other problems could lead to large delays.

   Mitigation: The research needs to be thorough and if time permits one should even apply many different solutions to an example to see which combination is better suited for the final work.

7. **Team Member Absence:** If a team member is absent due to a more serious illness or personal reasons, progress may be delayed.

   Mitigation: Ensure that tasks can be reassigned easily given well-documented tasks and sharing key knowledge through meetings.

8. **Scope Creep:** Adding extra features down the line to, for instance, enhance the user experience of the final product, could slow down core development.

   Mitigation: Prioritize core functionality, and postpone non-essential features until later phases. All issues are tracked on atlassian jira.

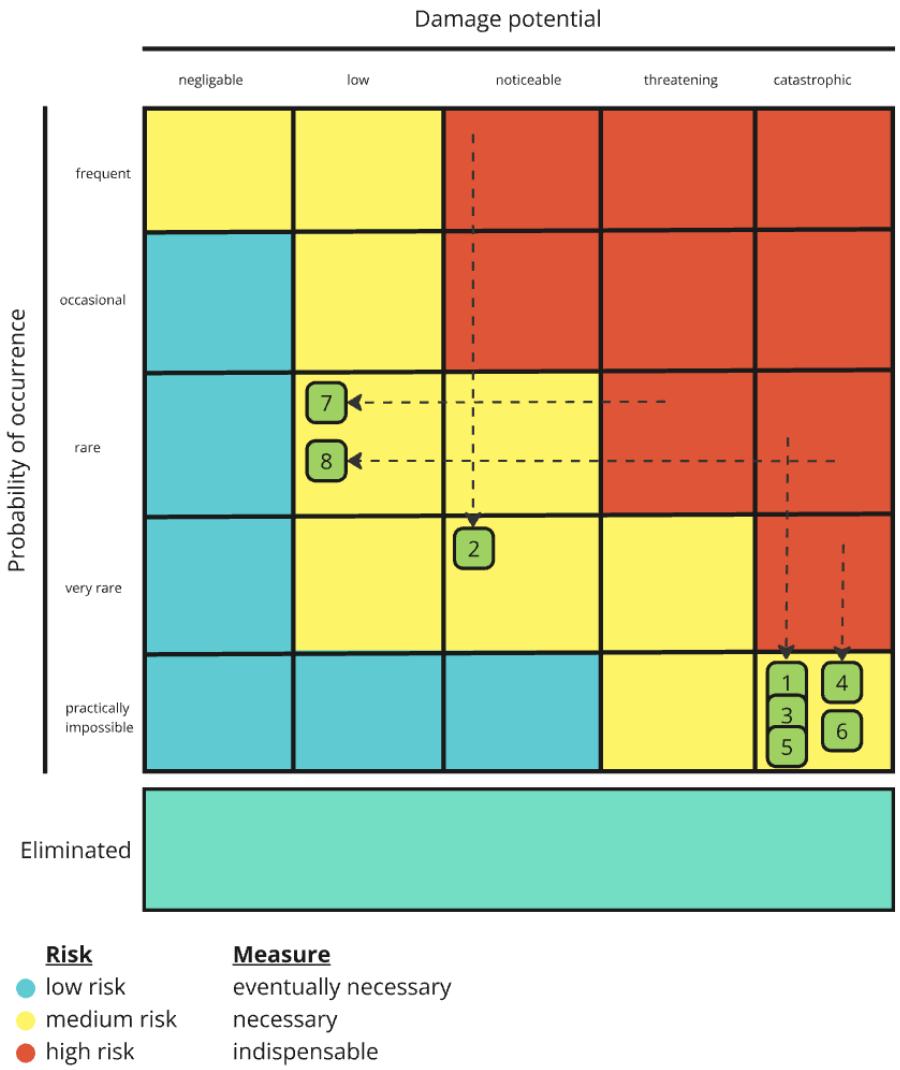

Figure 10.6: Risk Matrix 1 with mitigations

10.4.2 Iteration 2: Beta release

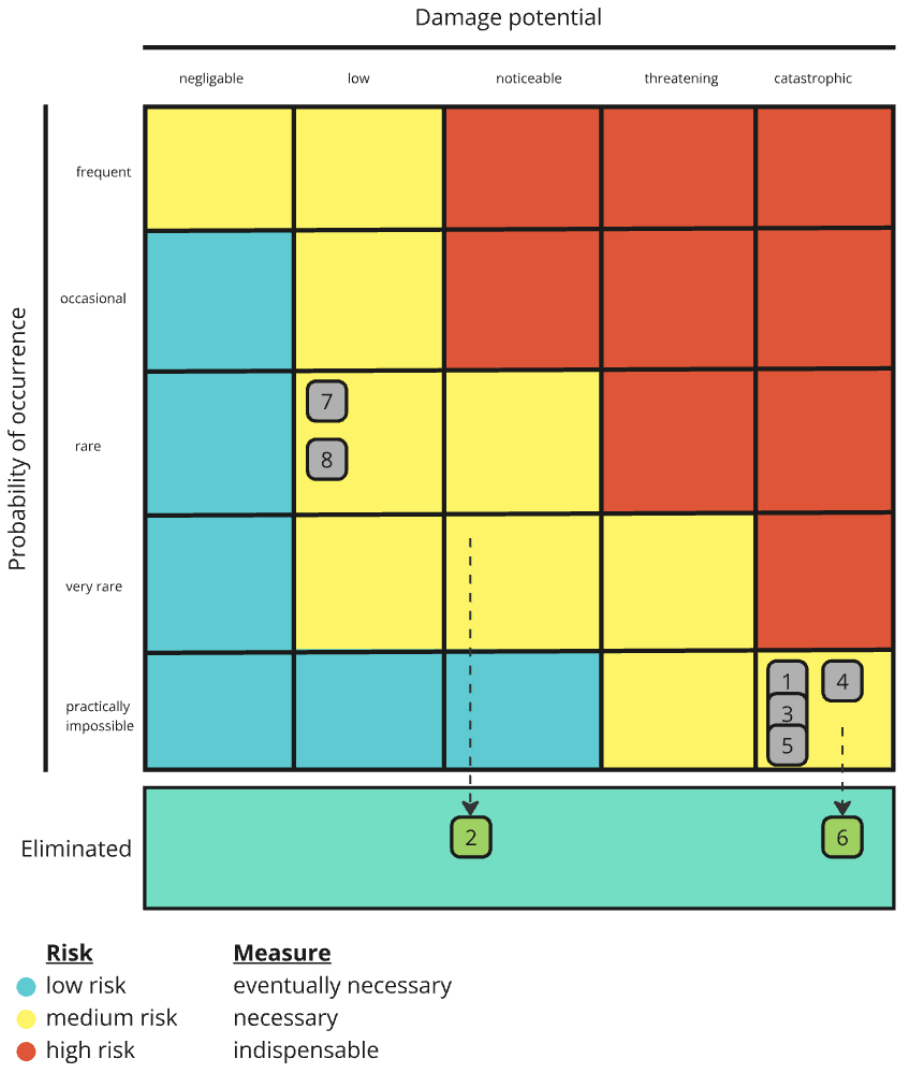

Figure 10.7: Risk Matrix 2

Changed Risks:

2. Slow Proof Generation:

Eliminated - After testing the generation and verification of the Sudoku ZKP it is safe to eliminate this risk as the entire procedure takes an average of 0.85 seconds.

6. Settling with the wrong ZKP system:

Eliminated - The functionality of the system has been proven, the proof can be generated in the frontend and successfully verified in the backend using snarkjs with groth16.

Unchanged Risks:

1. Incorrect ZKP Implementation:

It is impossible to guarantee a perfect ZKP circuit so a slight risk always remains.

3. Backend Downtime:

This problem can always occur as there is no time to create any distributed system.

4. Security Breach in User Authentication:

Although the authentication is outsourced to gitlab, the projects usermanagement could still remain a target to malicious actors.

5. Fake ZKPs

Although tests have been conducted it is also impossible to guarantee absolute safety.

7. Team Member Absence:

An absence can occur at any time.

8. Scope Creep:

The scope creep exists until the project is complete only in hindsight is it possible to judge if there was enough time.

List of Figures

List of Tables

Listings

Bibliography

[1] Wikipedia contributors, “Zero-knowledge proof,” 2024, accessed: 2025-05-08. [Online]. Available: https://en.wikipedia.org/wiki/Zero-knowledge_proof

[2] B. Maier, “Non-interactive zeroknowledge proofs based on hamiltonian cycles in large graphs,” Master’s thesis, Johannes Kepler University Linz, 2024, accessed: 2025-06-02. [Online]. Available: https://epub.jku.at/download/pdf/10031596.pdf

[3] B. R. C. on Decentralization AI, “Lecture 10.3: What is a zk-snark?” 2021, accessed: 2025-04-28. [Online]. Available: https://www.youtube.com/watch?v=gcKCW7CNu_M

[4] T. L. University, “Rank-1 constraint system with application to bulletproofs,” 2020, accessed: 2025-06-02. [Online]. Available: https://tlu.tarilabs.com/cryptography/rank-1

[5] E. Teske, “Pairing-friendly elliptic curves,” 2011, encyclopedia of Cryptography and Security, Springer US, accessed: 2025-06-12. [Online]. Available: https://doi.org/10.1007/978-1-4419-5906-5_256

[6] V. Buterin, “Exploring elliptic curve pairings,” 2017, accessed: 2025-04-28. [Online]. Available: https://vitalik.eth.limo/general/2017/01/14/exploring_ecp.html

[7] ——, “How do trusted setups work?” 2022, accessed: 2025-04-28. [Online]. Available: https://vitalik.eth.limo/general/2022/03/14/trustedsetup.html

[8] M. Asher, “Zero-knowledge proofs: Starks vs snarks,” 2021, accessed: 2025-06-03. [Online]. Available: https://consensys.io/blog/zero-knowledge-proofs-starks-vs-snarks

[9] Z. Developers, “The halo2 book,” 2023, accessed: 2025-06-03. [Online]. Available: https://zcash.github.io/halo2/index.html

[10] J. Garnier, “Documentation anime.js | javascript animation engine,” 2025, accessed: 2025-05-02. [Online]. Available: https://animejs.com/documentation