Step 1: Parsing the Source Code

The first step is to parse the source code into an Abstract Syntax Tree (AST). The `parseAST` function in the `Ast.Processing` module takes the source code as input and uses the Parsec parsing library to parse it according to the defined grammar rules.

Example source code:

```
var x = 10;
if (x > 5) {
    print("x is greater than 5");
} else {
    print("x is less than or equal to 5");
}
```

The `astParser` in the `Ast.Processing` module defines the grammar rules for the language. It uses parser combinators to specify the structure of the AST.

Visual representation of the parsing process:

```
             +------------------+
Source Code  |  var x = 10;     |
             |  if (x > 5) {    |
             |    print("...");|
             |  } else {        |
             |    print("...");|
             |  }               |
             +------------------+
                      |
                      | Parsing
                      v
             +------------------+
    AST      |  VariableDecl    |
             |    |- x          |
             |    |- 10         |
             |  If              |
             |    |- x > 5      |
             |    |- Print      |
             |    |   |- "..."   |
             |    |- Print      |
                |       |- "..." |
                +-----------------+
```

Step 2: AST Optimization

After parsing the source code into an AST, the `optimizeAST` function in the `Ast.ASTToRainbowToPLONK` module performs several optimizations on the AST. These optimizations aim to simplify and streamline the AST structure.

Visual representation of AST optimization:

```
             +-----------------+
Original AST |  VariableDecl   |
             |    |- x         |
             |    |- 10        |
             |  If             |
             |    |- x > 5     |
             |    |- Print     |
             |    |   |- "..." |
             |    |- Print     |
             |        |- "..." |
             +-----------------+
                      |
                      | Optimization
                      v
             +-----------------+
Optimized AST|  VariableDecl   |
             |    |- x         |
             |    |- 10        |
             |  If             |
             |    |- x > 5     |
             |    |- Print     |
             |        |- "..." |
             +-----------------+
```

In this example, the optimization process eliminates the unreachable `Print` statement in the `else` branch since the condition `x > 5` is always true based on the variable declaration `var x = 10;`.

Step 3: Converting AST to IR

The next step is to convert the optimized AST into an Intermediate Representation (IR). The `astToIRNodes` function in the `Ast.ASTToRainbowToPLONK` module performs this conversion. It recursively traverses the AST and maps each AST node to its equivalent IR node representation.

Visual representation of AST to IR conversion:

```
             +-----------------+
Optimized AST|  VariableDecl   |
             |    |- x         |
             |    |- 10        |
             |  If             |
             |    |- x > 5     |
             |    |- Print     |
             |        |- "..." |
             +-----------------+
                      |
                      | AST to IR
                      v
             +-----------------+
     IR      |  VariableDecl   |
             |    |- x         |
             |    |- IntLit(10)|
             |  If             |
             |    |- Greater   |
             |    |   |- Var(x)|
             |    |   |- IntLit(5)
             |    |- Print     |
             |        |- StringLit("...")
             +-----------------+
```

Step 4: Encoding IR to Rainbow Code

After converting the AST to IR, the next step is to encode the IR into Rainbow Code. The `encodeToRainbowCode` function in the `RainbowCode.Rainbow` module performs this encoding. It takes the IR as input and encodes each element of the IR into its corresponding hex color code defined in the `combinedEncoding` map.

```

             +-----------------+
     IR     |  VariableDecl   |
             |    |- x         |
             |    |- IntLit(10)|
             |  If             |
             |    |- Greater   |
             |    |   |- Var(x)|
             |    |   |- IntLit(5)
             |    |- Print     |
             |        |- StringLit("...")
             +-----------------+
                      |
                      | Encoding
                      v
             +-----------------+
Rainbow Code |  #FF5733 #33FF57 #FF33A5
             |  #0000FF #FF5733 #33FF57 #FF33A5
             |  #FF5733 #FF33A5
             +-----------------+
```

Rainbow Code:

+——+

| ● #FF5733 ● #33FF57 ● #FF33A5

| ● #0000FF ● #FF5733 ● #33FF57 ● #FF33A5

| ● #FF5733 ● #FF33A5

+——+

In this example, `VariableDecl` is encoded as `#FF5733`, `x` is encoded as `#33FF57`, `IntLit(10)` is encoded as `#FF33A5`, and so on. The resulting Rainbow Code is a sequence of hex color codes representing the encoded IR.

Step 5: Hashing the Rainbow Code

Instead of decoding the Rainbow Code back to IR, we can hash the Rainbow Code using a secure hash function like SHA-256. This step compresses the Rainbow Code into a fixed-size digest that can be efficiently used in the PLONK circuit.

Visual representation of hashing the Rainbow Code:

+——+

Rainbow Code | ● #FF5733 ● #33FF57 ● #FF33A5

| ● #0000FF ● #FF5733 ● #33FF57 ● #FF33A5

| ● #FF5733 ● #FF33A5

+——+

```
                        |
                        | Hashing (SHA-256)
                        v
               +------------------+
 Hashed Code   |  0x1a2b3c4d...   |
               +------------------+
```

Step 6: Integration with PLONK

The hashed Rainbow Code can be directly used as input to the PLONK circuit. The PLONK circuit takes the hashed code, processes it according to the PLONK protocol, generates a proof, and allows for verification of the computation.

Visual representation of the revised PLONK integration:

```
               +------------------+
 Hashed Code   |  0x1a2b3c4d...   |
               +------------------+
                        |
                        | PLONK Integration
                        v
               +------------------+
 PLONK Circuit|  Process          |
               |  Generate Proof |
               |  Verify          |
               +------------------+
```

By directly using the hashed Rainbow Code in the PLONK circuit, we can eliminate the need for decoding the Rainbow Code back to IR. This approach offers several benefits:

1. Security: Hashing provides an additional layer of security by irreversibly transforming the Rainbow Code into a fixed-size digest. Even if the hashed code is compromised, it would be computationally infeasible to recover the original Rainbow Code or the source code.

2. Compatibility: The hashed Rainbow Code is a compact and fixed-size representation that can be easily integrated into the PLONK circuit. It simplifies the input format and reduces the complexity of the circuit design.

3. Confidentiality: By using the hashed Rainbow Code directly, we maintain the confidentiality of the original source code. The hash function ensures that the original code cannot be reverse-engineered from the hashed code, providing an additional layer of privacy.

Visual representation of the entire process with the revised PLONK integration:

```
             +------------------+
Source Code  |  var x = 10;     |
             |  if (x > 5) {    |
             |    print("...");|
             |  }               |
             +------------------+
                      |
                      | Parsing
                      v
             +------------------+
    AST      |  VariableDecl    |
             |    |- x          |
             |    |- 10         |
             |  If              |
             |    |- x > 5      |
             |    |- Print      |
             |        |- "..." |
             +------------------+
                      |
                      | AST to IR
                      v
             +------------------+
     IR      |  VariableDecl    |
             |    |- x          |
             |    |- IntLit(10)|
             |  If              |
             |    |- Greater    |
             |    |   |- Var(x)|
             |    |   |- IntLit(5)
             |    |- Print      |
             |        |- StringLit("...")
             +------------------+
                      |
                      | Encoding
                      v
             +------------------+
Rainbow Code |  #FF5733 #33FF57 #FF33A5
             |  #0000FF #FF5733 #33FF57 #FF33A5
```

```
               |  #FF5733 #FF33A5
               +-----------------+
                        |
                        | Hashing (SHA-256)
                        v
               +-----------------+
 Hashed Code   |  0x1a2b3c4d...  |
               +-----------------+
                        |
                        | PLONK Integration
                        v
               +-----------------+
 PLONK Circuit|  Process        |
               |  Generate Proof |
               |  Verify         |
               +-----------------+
```

By directly using the hashed Rainbow Code in the PLONK circuit, we streamline the process, optimize for efficiency, and maintain the security and confidentiality of the original source code. This revised approach leverages the benefits of Rainbow Code encoding and PLONK integration while simplifying the overall workflow.

Here's a section comparing the proposed system to existing top cross-chain methods like Layer Zero, Cosmos IBC, and Wormhole:

Comparison with Existing Cross-Chain Methods

While the proposed system shares some high-level goals with existing cross-chain solutions, such as enabling communication and value transfer across different blockchain networks, it differs significantly in its approach and underlying mechanisms. Here's a comparison of the proposed system with some of the top cross-chain methods:

1. Layer Zero

Layer Zero is a decentralized network that aims to facilitate cross-chain communication and asset transfers. It uses a combination of smart contracts and a specialized consensus mechanism called "Zeroverse" to enable cross-chain functionality.

Key Differences from the Proposed System:

- Layer Zero relies on a separate network and consensus mechanism, while the proposed system leverages the PLONK circuit and zero-knowledge proofs for cross-chain computation and verification.
- Layer Zero requires the deployment of smart contracts on each participating blockchain, whereas the proposed system encodes the source code into Rainbow Code and uses the PLONK circuit for verification.
- Layer Zero does not inherently provide confidentiality or privacy for the cross-chain computations or data transfers, which is a key feature of the proposed system.

2. Cosmos IBC (Inter-Blockchain Communication)

Cosmos IBC is a protocol that enables communication and value transfer between different blockchains within the Cosmos ecosystem. It uses a hub-and-spoke model, where independent blockchains (zones) can connect to a central hub and communicate with each other through relay and packet-handling mechanisms.

Key Differences from the Proposed System:

- Cosmos IBC is primarily designed for communication and value transfer within the Cosmos ecosystem, while the proposed system aims to enable cross-chain computation and verification across different blockchain networks.
- Cosmos IBC does not provide inherent mechanisms for confidentiality or privacy of cross-chain data or computations, which is a core feature of the proposed system.
- The proposed system leverages the PLONK circuit and zero-knowledge proofs, which are not part of the Cosmos IBC protocol.

3. Wormhole

Wormhole is a cross-chain bridge that enables the transfer of assets and data between different blockchain networks, including Ethereum, Solana, and others. It uses a combination of smart contracts, trusted validators, and specialized messaging protocols to facilitate cross-chain communication.

Key Differences from the Proposed System:

- Wormhole primarily focuses on asset transfer and data communication, while the proposed system aims to enable cross-chain computation and verification using the PLONK circuit and zero-knowledge proofs.

- Wormhole relies on a set of trusted validators and specialized messaging protocols, whereas the proposed system uses the cryptographic properties of the PLONK circuit and hashed Rainbow Code for secure computation and verification.
- The proposed system provides inherent confidentiality and privacy for the cross-chain computations through the encoding and hashing of the source code, which is not a core feature of Wormhole.

One key advantage of the proposed system is its ability to execute arbitrary computational functions across different chains while maintaining confidentiality and privacy. By encoding the source code into Rainbow Code, hashing it, and integrating it with the PLONK circuit, the system can securely verify computations without revealing the original code or data.

This level of expressiveness and confidentiality is not easily achievable with existing cross-chain methods, which often rely on predefined functions, smart contracts, or messaging protocols that may require revealing sensitive information across different chains.

Additionally, the proposed system's use of the general-purpose PLONK circuit and its potential for better compatibility and interoperability across different blockchain ecosystems further highlight its potential advantages in terms of expressiveness and cross-chain computation.