Keelung - A Toolkit for Fast, Private and Secure Applications

BTQ Technologies Corp.

March 2023

Abstract

Zero-knowledge proof techniques are revolutionizing the world of privacy and computation. However, developing zero-knowledge applications is still a labor-intensive task and an error-prone process, partly because the development tools are still far from mature. We present Keelung, a domain-specific language and development toolkit for zero-knowledge applications. By raising the abstraction level, Keelung makes it easy for developers to focus on the business logic rather than the low-level circuit design of the zero-knowledge proof. Additionally, Keelung is embedded into Haskell, providing programmers with access to Haskell's rich ecosystem and extensive libraries, making the development of zero-knowledge applications less burdensome.

Contents

1 Introduction

In this paper we propose Keelung, a zero-knowledge (ZK) cryptography development toolkit for fast, private and secure applications. Keelung makes it simple to build zero-knowledge proofs (ZKPs) for applications regardless of the business logic or use case. Zero-knowledge cryptography has a bright future and we want to make it easier for teams to build and ship zero-knowledge applications.

1.1 Background

zk-SNARK In this paper, we focus on a subclass of ZKP techniques that enjoy particularly favorable properties. They are generally referred to as zk-SNARKs, or Zero-Knowledge Succinct ARguments of Knowledge. In a highly simplified form, the problem solved by a zk-SNARK can be roughly formulated as follows. A prover $\mathcal{P}$ wants to prove to a verifier $\mathcal{V}$, both computationally bounded, that $\mathcal{P}$ *knows* a w s.t. $y = f(x, w)$, where f is a public (ZK) "circuit" with public input x and public output y. Using a ZK proving system, $\mathcal{P}$ can generate a proof π that is relatively small, say $\mathcal{O}(\log |f|)$, where $|f|$ is the size of f measured in, e.g., number of arithmetic gates contained in f. With f, x, y and π, $\mathcal{V}$ can efficiently verify that $\mathcal{P}$ indeed knows a secret w s.t. $y = f(x, w)$, yet $\mathcal{V}$ learns nothing more about w beyond this fact.

Such a formulation makes zk-SNARKs useful for a wide range of internet applications. For instance, they can be used for selective revelation to verify a user's identity without disclosing their personal information, such as name, date of birth, or social security number. Similarly, hospitals can share relevant parts of patients' medical records with third parties, such as insurance agents, while protecting patients' privacy. Additionally, zk-SNARKs can verify the authenticity of transactions on a blockchain without recording or revealing certain details, such as the sender/receiver or the exact amount transacted. The succinctness of zk-SNARK proofs leads to a popular scaling solution for blockchains and similar systems, known as ZK rollups. We stress that in all these zkApps, $\mathcal{V}$ may be able to, and often indeed does, learn *some* information about w from the fact that w is a solution to the equation $y = f(x, w)$. This is not a problem in some zkApps such as ZK rollup, but when it is indeed a problem, developers should take extra caution to limit the information thus leaked. Keelung and any underlying ZK proving systems provide no guarantee whatsoever when it comes to such privacy breach.

Rank-1 constraint system (R1CS) R1CS is a popular intermediary representation of a ZK circuit, which we denote as f in the previous section. It is a type of constraint system, a mathematical structure that consists of a set of variables and constraints that define the relationships between those variables. To encode a ZK circuit f as mentioned previously in R1CS, we need to find matrices A, B, C of appropriate sizes s.t. $Az \odot Bz = Cz$, where $z = \begin{bmatrix} 1 & u & w & x & y \end{bmatrix}^t$, u

a set of auxiliary variables, and $\odot$ the Hadamard (coordinate-wise) product of vectors. The necessity of having u for encoding more complicated ZK circuits f should be apparent, as an R1CS expression can only natively encode (most but not all) quadratic relations in variables x, y, w. To encode a relation of a higher degree, for example, $y_i = x_i^3$, we can create an auxiliary variable u_i and set $u_i = x_i^2$. This way we can encode the original relation $y_i = x_i^3$ as $y_i = x_i u_i$, which can be encoded in R1CS fairly easily.

ZK DSL Domain-specific languages (DSLs) are programming languages created for a specific domain or application. They have become increasingly popular in recent years due to their greater efficiency and effectiveness in performing specific tasks compared to general-purpose programming languages. The benefits of using a DSL include providing a higher-level, more intuitive way to describe domain-specific computations, facilitating the involvement of non-experts in the domain, and abstracting away irrelevant details of the underlying computing systems. In the field of zero-knowledge application (zkApp) development, there has been a recent surge of new DSLs, such as Cairo, Cairo [1], Circom [2], Noir [3], ZoKrates [4], and others. While most of these DSLs are imperative, there have been attempts to adopt a more functional approach, such as Snårkl [5], which provides a basis and serves as a great inspiration for our work in this paper.

1.2 Use Cases of Zero-Knowledge Proofs

Zero-knowledge proofs are a potent tool with a broad range of applications in today's digital world. Here are six examples of how zero-knowledge proofs can enhance security and privacy in everyday transactions:

1. **Privacy-Preserving Analytics:** Zero-knowledge proofs can be used to enable secure and private data analysis without revealing sensitive information, making them useful for applications in business intelligence, data analytics, and machine learning. For instance, a company could use zero-knowledge proofs to demonstrate that its revenue is above a specific threshold without disclosing the actual revenue number.

2. **Identity Verification:** With the rise of online platforms, verifying identities has become increasingly important. Zero-knowledge proofs can enable secure identity verification without disclosing any unnecessary personal information. This can be particularly useful in cases where a user wants to prove their age or eligibility for a certain service without revealing sensitive personal details.

3. **Supply Chain Management**: In today's global marketplace, supply chains can be exceedingly complex. Zero-knowledge proofs can help ensure the authenticity and integrity of products throughout the supply chain, from manufacturing to delivery. This can help prevent counterfeit

products and guarantee that consumers receive the products they paid for.

4. **Anonymous Verifiable Voting**: Anonymous verifiable voting can prevent voter fraud and ensure the integrity of election processes. By providing a transparent and secure system that allows voters to verify that their vote was counted correctly without compromising their privacy, this voting method can help promote trust in the electoral process and encourage greater public participation.

5. **Machine Learning and Artificial Intelligence:** Using zero-knowledge proofs, machine learning inference can be performed on edge devices without sending the input to centralized servers. Proofs enable hiding both sensitive input data and the model parameters from public view and allow downstream entities verify the input was correctly processed to yield the reported output.

6. **Gaming:** Zero-knowledge proofs are an effective tool in gaming to allow players to prove that they have achieved certain in-game accomplishments or reached specific levels without revealing sensitive information about their game strategies or progress. This enhances the gaming experience by providing a secure and confidential way for players to earn rewards and recognition for their achievements without having to disclose any personal information or trade secrets.

2 Problems with privacy-preserving application development

Zero-knowledge proof techniques are gaining significant attention in the technology industry for their potential to bring revolutionary changes from the ground up, particularly in the field of privacy-centric applications. After nearly a decade of extensive research and development, however, developing ZK applications (zkApps) is still a labor-intensive task and an error-prone process. This is primarily because the field is still relatively young and the tooling ecosystem has not yet sufficiently matured. Here are some observations we noticed when we took a look at today's zkApp development landscape.

2.1 Lack of expressivity

Developers face a significant challenge when working with most existing zero-knowledge application (zkApp) development toolkits. These toolkits often lack a sufficient level of abstraction, requiring developers to possess a deep understanding of the underlying mathematical principles behind zero-knowledge proof (ZKP) techniques. This level of expertise is necessary to achieve the intended functionality of the zkApp and to avoid common errors. Additionally, developers using low-level programming languages must manually translate their high-level business logic, a complex and challenging process that can hinder development efforts. The requirement to adopt a new mode of thinking represents a significant barrier to entry for developers creating zkApps, leading to the potential for low-quality or insecure implementations.

2.2 Pre-quantum proving systems

Despite growing awareness and adoption of zero-knowledge proof (ZKP) technologies among researchers and developers, few are considering the long-term security implications of the ZK proving systems being utilized. Many of these systems are designed with a focus on performance and target pre-quantum computing environments. As a result, the ZK proofs generated by these systems are vulnerable to attacks by large-scale quantum computers, which undermines the fundamental concept of zero-knowledge. It is imperative that the development of zero-knowledge applications (zkApps) not be left behind in the transition to post-quantum cryptography, as it is essential to ensure the continued security of these systems.

2.3 Suboptimal/inefficient proofs

A significant limitation of programming in a low-level language is the need for developers to manually optimize their code for the downstream zero-knowledge (ZK) proving system while simultaneously encoding the intended zkApp's business logic. Hand optimization requires a deep understanding of the underlying ZK proving system, which can be highly challenging, particularly when dealing with complex data types such as high-dimensional arrays or vectors that are commonly found in a wide range of zkApps. Consequently, only a small number of highly skilled cryptographers/programmers possess the ability to implement reasonably optimized zkApps, creating a bottleneck that hinders the wider and more rapid adoption of ZKP technologies.

3 Hello, Keelung!

To address these challenges, we present *Keelung*, a domain-specific language (DSL) and toolkit designed to address the challenges of developing fast, private, and secure zero-knowledge applications (zkApps). Keelung aims to provide a simplified approach for developers to build and deploy zkApps irrespective of their use cases or business logic. Keelung offers a unique solution that defies the trade-off between speed and abstraction. By embedding itself in Haskell, a renowned programming language known for its safety and reliability, developers can write powerful and secure software with confidence. The incorporation of Haskell's advanced type system allows for high-level ZKPs with the ability to leverage Haskell's mature ecosystem and tooling. Haskell's functional programming language and advanced type system enable the development of correct programs with ease at any level of abstraction. Keelung's integration into Haskell allows for the use of its vibrant ecosystem and community, including features such as typeclasses, which provide a natural syntax for Keelung. Furthermore, developers can define custom datatypes on top of Keelung to meet their specific needs. Keelung's modular design allows developers to upgrade their ZKPs seamlessly from pre- to post-quantum. This is facilitated through the use of several modular ZK proving systems, both pre- and post-quantum, which the resulting circuits can plug-and-play with. This feature offers maximum flexibility to developers with minimal friction for upgrading zkApps when the need arises.

4 How Keelung solves these problems

4.1 Higher level of abstraction

Keelung offers a unique solution that overcomes the tradeoff between speed and abstraction by embedding itself in Haskell. This approach enables developers to leverage Haskell's advanced type system and mature ecosystem, providing the ability to write high-level zero-knowledge proofs (ZKPs) with confidence. Haskell is a functional programming language renowned for its safety and reliability, facilitating the construction of correct programs with any desired level of abstraction. By embedding Keelung in Haskell, developers can take advantage of the vibrant ecosystem and community that comes with the language. Haskell's Typeclass feature provides a natural syntax for Keelung while allowing developers to define custom data types to meet their specific needs.

4.2 Flexible ZKPs

Keelung has a modular design that allows for the straightforward upgrade of zero-knowledge proofs (ZKPs) from pre- to post-quantum, which is essential for ensuring the long-term security of zkApps. Keelung achieves this by providing several modular proving systems, both pre- and post-quantum, that a circuit can plug-and-play with. The modular architecture of Keelung enables developers to upgrade their zkApps with minimal friction when the time is right, providing the flexibility necessary to adapt to future security requirements. Keelung's modular design also allows for the integration of future advancements in zero-knowledge proof systems seamlessly.

4.3 Post-quantum proving systems

Keelung was designed with post-quantum security in mind. As such, developers have access to post-quantum proving systems, ensuring that proofs generated with Keelung are not vulnerable to quantum attacks. This approach enables developers to generate R1CS circuits with Keelung in combination with post-quantum proving systems, facilitating the development of zkApps with confidence in their long-term security. By incorporating post-quantum proving systems with Keelung, developers can confidently create zkApps that are secure, even in the face of quantum computing, laying the groundwork for a secure future for zero-knowledge proof technologies.

4.4 Cross-chain compatibility

One of the significant advantages of Keelung is the flexibility it provides developers in terms of the target environment for proof verification. Keelung enables developers to deploy zkApps in any virtual environment capable of zero-knowledge proof verification, providing a versatile solution that can adapt to different use cases and requirements. Having flexibility over the target verification environment is a valuable tool for developers seeking to deploy zkApps in different ecosystems. The ability to change the target environment also enables developers to remain flexible and agile in their approach, facilitating the development of innovative and groundbreaking zkApps across different virtual environments.

4.5 Fast compilation

Keelung is designed to provide extremely fast zero-knowledge proof generation, making it a valuable tool for developers seeking to optimize the performance of their zkApps. Keelung achieves this by leveraging Haskell's powerful abstraction capabilities to enable fast compilation of post-quantum cryptographic primitives. Keelung reduces the number of constraints associated with each circuit, which in turn drastically reduces the number of rich computations required at compilation time. This optimization approach enables Keelung to achieve faster zero-knowledge proof generation times, allowing developers to create efficient and high-performance zkApps that meet the demands of today's computing landscape. The reduction of constraints associated with each circuit also improves the scalability of Keelung, allowing it to handle larger and more complex circuits efficiently. This scalability feature is critical for developers looking to create zkApps for real-world applications that may involve a large number of participants and complex computations. In combination with Haskell's abstraction power, Keelung provides developers with a high-performance solution for zero-knowledge proof generation that is safe, efficient and scalable. This optimization approach enables developers to create fast and reliable zkApps that meet the performance demands of today's computing landscape while maintaining the high levels of security required in zero-knowledge proof technologies.

5 How does Keelung work?

A Keelung program will go through several stages: elaboration, type erasure, rewriting compilation, optimization, and circuit construction. At the end of a successful compilation, an R1CS circuit is generated. These stages are explained in more detail as follows.

5.1 Elaboration

Keelung provides developers with a variety of high-level constructs that do not necessarily exist in R1CS. In order to convert these constructs into R1CS, Keelung programs will need to go through an elaboration stage. For example, loops will be unrolled into repetitive instructions, while array manipulations will be resolved into simple variable assignments. Elaboration essentially strips away syntactic sugar, expressing a user program in a much smaller core language that is easier to process in the subsequent stages.

5.2 Type erasure

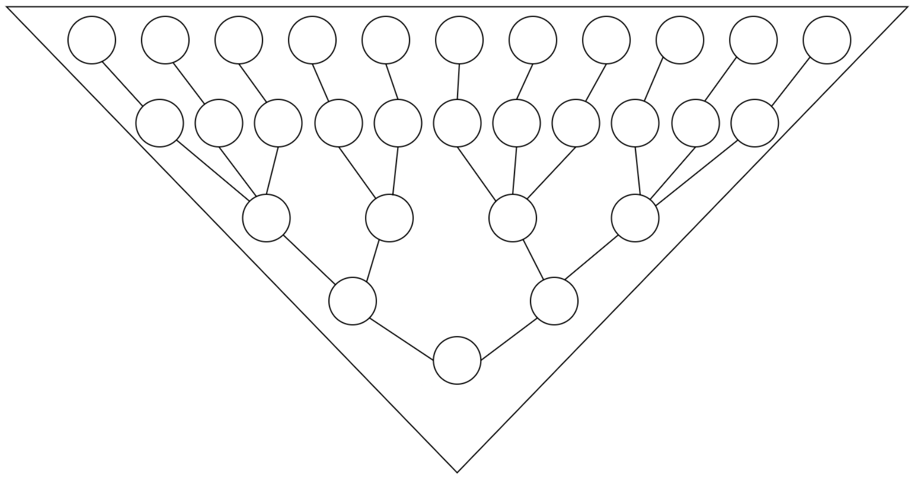
Like Haskell and other strongly-typed programming languages, Keelung helps developers to write safer programs by distinguishing between concepts like an array of Booleans or numbers. These type distinctions are erased at this stage to reduce overhead in the generated R1CS circuit. This is similar to a vanilla Haskell program: the type system is mostly visible at compile time to provide a strong guarantee of correctness; the information on types is then erased before the binary is generated so the runtime price we need to pay for having the extra safety guarantee is minimal.

5.3 Rewriting compilation

Complex expressions are then replaced with simpler equivalents to reduce the size of the resulting ZK circuit. Up until this stage, the whole program is still in the form of a syntax tree, with variables as leaves and operators as internal nodes. This syntax tree will be traversed and broken down into small constraints that describe the relationship between variables in the syntax tree. Take $\mathrm{A} = (\mathrm{B} * \mathrm{C}) + (\mathrm{D} * \mathrm{E})$ as an example. Because this expression is too complicated to directly encode into R1CS, it will be compiled into 3 smaller constraints: $\mathrm{A} = \mathrm{X} + \mathrm{Y}$, $\mathrm{X} = \mathrm{B} * \mathrm{C}$, and $\mathrm{Y} = \mathrm{D} * \mathrm{E}$. These constraints will then be used to construct an R1CS circuit (Figure 1).

5.4 Optimization

Many of these constraints may be redundant or can be replaced by simpler constraints that express equivalent relationships among the variables. Sometimes we can even eliminate certain variables altogether. Ideally, we would like to remove as many redundant constraints as possible as the number of constraints will significantly impact the size of the resulting ZK circuit. We have designed a set of optimization passes that will remove redundant constraints and simplify the remaining ones. These constraints will go through these optimization passes repeatedly until the number has been reduced to a bare minimum.



$$A = (B * C) + (D * E)$$

$$\Downarrow$$

$$\begin{aligned} A &= X + Y \\ X &= B * C \\ Y &= D * E \end{aligned}$$

Figure 1: An example of compiling a syntax tree into constraints

5.5 Circuit construction

The final stage is to construct an R1CS circuit from the remaining constraints. An R1CS circuit is made of a set of polynomials that describe the relationship between variables, and they are actually not that different from the constraints we have been obtained so far. All we need to do is to convert the constraints into polynomials and arrange them in an appropriate way to fit the selected ZK proving system's requirements.

6 Examples

By design, Keelung has a very small core language and leverages Haskell's rich ecosystem and tooling to achieve a wide variety of functionalities. The core language has native support for three primitive types: Field (the underlying field of the proving system), UInt n (unsigned n-bit integers), and Boolean.

Unsigned integer. It is straightforward to manipulate these primitive types in Keelung. However, the design of the unsigned integer type may require a bit more explaination. Unlike how it is handled in most functional programming languages, we decided to include some behavior of boolean and bitvector in it because it is very common in a lot zkApps to manipulate certain bits in an unsigned integer.

```
workMask :: Comp (UInt 32)
workMask = do
  word <- inputUInt @32
  return (word .&. 0xab)
```

Here we are taking a selection of bits from an unsigned integer by bitwise `and`'ing it with a bitmask `0xab`. It is safe to do because we know the bit length of the unsigned integer, so we can pad the bitmask with zeros to fit the length of `word`.

Dependent type. As we have seen, encoding the length of an unsigned integer in its type enables extra safety check at compile time.

```
example :: Comp (UInt 8)
example = do
  byte <- inputUInt @8
  word <- inputUInt @32
  return (byte .&. word) -- TYPE ERROR !!
```

For example, it is ambiguous to bitwise `and` two unsigned integers of different lengths. We can either extend the shorter one or truncate the longer one, and depending on the choice, we will have two different resulting types. It is unclear which one is the behavior that the programmer has in mind when he or she writes down this expression. Therefore, Keelung would reject this code snippet by throwing a type error, forcing the programmer to specify the exact behavior he or she intend to have.

Metaprogramming. With Keelung's metaprogramming capability, we can also use native Haskell functions to simplify the task of writing Keelung programs.

```
addN :: Field -> Comp Field
addN n = do
  x <- input
  return (x + n)

example :: Comp Field
example = do
  x <- addN 3
  y <- addN 5
  return (x * y)
```

Obviously this should translate to the following R1CS constraint: (x + 3)∗(y + 5) = out. This way the programmer can write down common programming idioms using Haskell functions, preferably with accurate and descriptive names, to make Keelung programs easier to understand and maintain.

Loop. Keelung is a functional DSL. This means that we usually use `map` and `fold` to implement control loops. In fact, we need to use the monadic versions of them in order to keep a record of the context, as well as to generate and accumulate the resulting constraints.

```
loopy :: Comp [Field]
loopy = mapM addN [0, 1, 2, 9876543]
```

For example, the effect of `loopy` is equivalent to loop over the four elements (0, 1, 2, and 9876543) in the array, applying the `addN` function to each of them.

Conditional. It is important to be able to express conditional branches in any programming languages.

```
is42 :: Comp Field
is42 = do
    x <- input
    cond (x `eq` 42)
        100 -- then
        1   -- else
```

While this should be self-evident, its R1CS circuit is a bit non-trivial and thus requires some elaboration. In R1CS, we usually need to “glue together” both branches of a conditional, effectively “executing” both of them and discard the irrelevant result:

```
  x - 42 = b_inverted
  1*b_inverted + 100*(1 - b_inverted) = out
```

Assertion. A zkApp naturally contains a number of assertions, expressing the relationships among the input and output variables. This is implemented in a monadic way in Keelung, and we also expose this interface to the programmer so that they can have finer control over assertions. For example, Keelung programmers can simply assert the equality between two input variables as follows.

```
example :: Comp ()
example = do
    x <- input
    y <- input
    assert (x `eq` y)
```

More complicated equality tests between compound variables can be constructed in a similar way.

7 Comparison

In this section, we will run a larger example of verifying Merkle tree membership, first in Circom and then in Keelung, to highlight the difference between the two DSLs. We choose Circom because it is among the most popular ZK DSLs, but the conclusion and insight should apply to other imperative ZK DSLs as well.

First, we show typically (but not necessarily optimally) how to prove that a leaf is in a given Merkle tree in Circom.

```
template MerkleTreeInclusionProof(nLevels) {
    signal input leaf;
    signal input pathIndices[nLevels];
    signal input siblings[nLevels];

    signal output root;

    component poseidons[nLevels];
    component mux[nLevels];

    signal hashes[nLevels + 1];
    hashes[0] <== leaf;

    for (var i = 0; i < nLevels; i++) {
        pathIndices[i] * (1 - pathIndices[i]) === 0;

        poseidons[i] = Poseidon(2);
        mux[i] = MultiMux1(2);

        mux[i].c[0][0] <== hashes[i];
        mux[i].c[0][1] <== siblings[i];

        mux[i].c[1][0] <== siblings[i];
        mux[i].c[1][1] <== hashes[i];

        mux[i].s <== pathIndices[i];

        poseidons[i].inputs[0] <== mux[i].out[0];
        poseidons[i].inputs[1] <== mux[i].out[1];

        hashes[i + 1] <== poseidons[i].out;
    }

    root <== hashes[nLevels];
}
```

We can see that, as part of the witness, we need to have an array of the hash values of the missing sibling subtree. In addition, we need to know whether the missing sibling subtree is to our left or to our right. This information is recorded in the array `pathIndices`. We then go through the authentication path from leaf all the way to root, verifying all the hash values along the path.

The same verification is implemented in Keelung as follows.

```
getMerkleProof :: Int -> Comp Field
getMerkleProof depth = do
  leaf <- inputField
  siblings <- inputs2 depth 5
  indices <- inputs depth
  (_, digest) <-
    foldlM
      ( \(i, digest) p -> do
          assert (digest `eq` choose p (access indices i))
          p' <- hash p >>= reuse
          return (i + 1, p')
      )
      (0, leaf)
      siblings
  return digest
```

As we can see, it seems a bit more concise because we use `foldM` (the monadic version of `fold`) to replace the loop in the Circom implementation. Using `fold`, we can express an extremely wide variety of language constructs and use patterns in imperative languages [6].

Intuitively speaking, a `fold` transforms an inductively defined data structure into another, possibly also inductively defined data structure in a systematic way. The specific transformation depends on the source data structure, which can be specified by the `Foldable` typeclass in Haskell. A canonical example is a linear array, but it can be easily generalized to other, more complicated data structures such as higher-dimensional arrays or trees. Given a source data structure, we then need to supply `fold` with a function of "stepwise specification" that takes (a) the current component of the source data structure to be folded, as well as (b) the element of the target type we have constructed so far, and then returns a new element of the target type resulted from folding the former (a) into the latter (b). This way, `fold` can start from an initial element of the target type and (figuratively) fold the source data structure into the target.

With this in mind, now we should be able to better understand this example; here the "stepwise specification" checks whether the hash value of the subtree we have computed so far matches the corresponding one in the given authentication path. If it does, we then compute the hash value of those of the two subtrees and pass it on as that of the current subtree. Starting from bottom and going up, this is precisely what we do to check whether a leaf is in a Merkle tree or not.

8 Roadmap

In the current design of Keelung, our own Keelung compiler is responsible for circuit compilation.

In our future roadmap, we plan to transform Keelung by integrating it with OpenZL, an open-source library that aims to bridge the gap between high-level languages and low-level libraries that require cryptographic primitives. This integration will allow Keelung to benefit from ECLAIR, the Embedded Circuit Language and Intermediate Representation, which will also handle the circuit compilation for the proving system in the choice of developers. Developers will be able to use Keelung to write high-level business logic and choose a suitable proving system plugin provided by OpenZL for their application. By integrating with OpenZL, Keelung will make it easier for developers to use the DSL and choose between different proving systems, resulting in a more powerful and efficient tool for secure application development (Figure 2).

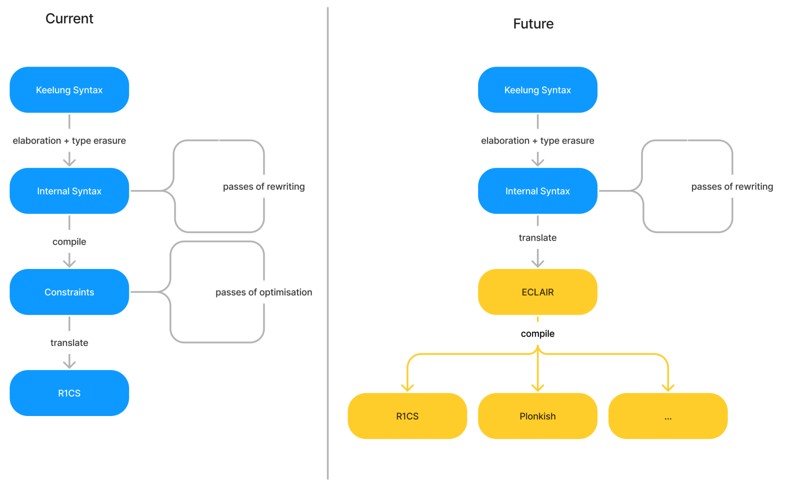


Figure 2: The comparison between the current design and future integration

9 Conclusion

In this paper, we have introduced Keelung, a DSL and development toolkit for building fast, private, and secure zkApps. We have argued for the importance of raising the level of abstraction in zero-knowledge application development which Keelung achieves by abstracting away the details of the underlying zero-knowledge proving system. While still in its early stages of development, Keelung shows promise in its potential to make ZK programming more accessible and user-friendly. We plan to extend Keelung's capabilities by adding native support for important operations like integer division and extending its backend to support more families of zero-knowledge prooving systems, such as PLONK. We ultimately hope that Keelung will play a crucial role in developing zkApps, bringing the focus closer to the application's business logic and making ZK programming easier for developers. Interested developers can access the alpha version of Keelung on Github and visit the Keelung website for more information.

References

[1] `https://starkware.co/cairo/`

[2] `https://docs.circom.io/`

[3] `https://aztec.network/noir/`

[4] `https://zokrates.github.io/`

[5] G. Stewart, S. Merten, and L. Leland. "Snårkl: Somewhat practical, pretty much declarative verifiable computing in Haskell." *International Symposium on Practical Aspects of Declarative Languages.* pp. 36–52, Springer 2018.

[6] G. Hutton. "A tutorial on the universality and expressiveness of fold." *Journal of Functional Programming*, 9(4), pp. 355–372, 1999.