Leo : A programming language for Formally Verified, Zero-Knowledge Applications.

- Dylan Kawalec

White paper Synthesis

Abstract

What do decentralized applications (dApp's) on suffer from?

1. Limited **run time**: "It is the time that a program is running alongside all the external instructions needed for proper execution".
2. Minimal **Stack Size:** the minimal size of the dynamic array often found in distributed ledger technologies. Found to be slightly problematic in the Solidity Programming language used in Ethereum virtual machine Layer 1/2 Distributed ledger technologies.
3. Restrictive **instruction sets:** Often referred to as a RISC (reduced instruction set computer); this is referring to when a computer is designed to simplify the individual instructions given to the computer to accomplish tasks.
4. Minor (a) : ***Front running attacks*** and ***consensus instability*** :

a) "Front-running is an attack where a malicious node observes a transaction after it is broadcast but before it is finalized, and attempts to have its own transaction confirmed before or instead of the observed transaction." — encs.concordia

Explain : Decentralized Exchanges sign orders from off-chain orders, and smart contracts execute customer orders automatically on the blockchain. These orders are seen on chain by the public. Let's picture an example of **front-running**. — *If Alice makes an order without the proper amount of gas fee's to pay for the order, and a random bot on the network finds Alice's order and pays a higher gas fee, then that bot will win the order due to this public facing order to commit the transaction on the network, winning over everyday users like Alice.*

Note : Bot's often exploit and take advantage of users on-chain very often.

The **Leo DSL** (domain specific language) provides builders to create (c) : *formally verified, Zero-knowledge proof application* language. **Leo** is NOT restricted by runtime, stack size or instruction sets; Leo mitigates **Miner Extractable Value** (MEV), which is a catch all term for all the ways miners can leverage their control over inclusion/ordering of transactions to capture revenue for themselves. **Leo** is *succinctly verified*, which technically means that programs are briefly and clearly expressed by anyone able to verify transactions & signatures.

c) “**Formal Verification:** The act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics.[1]" — Wikipedia

~ Leo introduces rich new technical features ~

1. **Testing frameworks** : Referred to as *Testnet3,* Allows Developers to write private applications on the Aleo Virtual Machine (AVM); which is known as *the first* zkVM (Zero-knowledge virtual machine) allowing for fully shielded transactions, similar to zCash.

2. **Package Registry** (*page 30*) : A collection of open sourced packages, also known as *rust crates,* that developers can download and implement into their programs similar to other package libraries other programming languages offer. “Using registered packages also prevents multiple instantiations of the same circuits on a given ledger – decreasing transaction circuit bloat.”—WP

3. **Remote Compiler** : We believe this is referring to the fact that the AleoHQ library can be executed in its own IDE and compiled on/offline named Aleo Studio**.**
— “It takes your code, constructs the proof circuit that represents the computation you want to run, populates the input wires with the values you want, runs the circuit, generates proving/verifying keys, and then combines all the relevant data into a ZKP. For example, you could run it on the input data in the image below.” — golden.com/wiki/Leo

4. **Import Resolver** (*page 29*) : “The import resolver runs during the ASG conversion compiler phase and stores imported identifiers, circuits and functions for lookup in during program execution”.

Introduction

Distributed ledgers offer an immutable history of state transitions, which ensures that everyone has the same exact view of the data at the same time. These systems are referred to as a trust-less peer-to-peer financial mechanisms, which gave rise to what’s known as a “cryptocurrency”. Blockchains have opened up new P2P applications, revolutionizing gaming, finances, governance, social networking and data sharing.

~ Three core issues distributed systems face ~

1. **Scalability Problem** : The settlement time / amount of transactions that are able to settle on the ledger at any given moment. Miners tend to carry a lot of the overhead for distributed ledger technology due to the vast amount of settlements occurring on any given public ledger.

Miners re-execute every single transaction, which can be cumbersome. This stalls entire systems and is highly wasteful of computational resources, impacting the environment. To avoid this this issue, blockchains use *gas* to disincentivize DOS stacks (denial-of-service), to avoid any one entity to flood the ledger in order to make the services provided by the blockchain network unusable. Even though gas is used to regulate bad behavior on chain, an problem known as **Verifier's Dilemma** can still occur, which causes low node participation in both verification and blockchain storage resulting from the Verifier's Dilemma undermines the security and integrity of the blockchain. This has resulted in blockchains either forking or shut down, making any and all decentralized services unusable worldwide.

Lastly, since users are competing for timeshares to make executions, we circle back to the fundamental concerns Leo addresses directly (*limited running time*, *minimal stack size, and restrictive instruction sets*).

2. **Privacy Problem**: This should not be confused with "Security"; privacy simply refers to the protection users have when interacting with publicly distributed technology. Most networking applications ensure users a level of confidentiality or psudoanonimity to protect them from malicious actors.

The strength and weakness of all chains is, "*The history of all state transitions must be executable by all parties.*" Ledgers like bitcoin reveal all information the sender, receiver and amount spent by all parties interacting with currencies on blockchains. "Not only does this reveal the private financial details of the individuals and businesses who use such a system, but it violates the principle of fungibility, a fundamental economic property of money". — WP

Traditional applications such as (a) *Dark Pools* simply cannot exist on public blockchains due to this very issue. Systemic risks are posed to decentralized ledgers; as mentioned previously, front-running & arbitrage attacks creates "Miner-Extractable Value" (MEV),

a) **Dark Pools** : "A dark pool is a privately organized financial forum or exchange for trading securities. Dark pools allow institutional investors to trade without exposure until after the trade has been executed and reported. Dark pools are a type of alternative trading system (ATS) that gives certain investors the opportunity to place large orders and make trades without publicly revealing their intentions during the search for a buyer or seller." — Investopedia

3. **Audibility Problem**: In order for users to comply with regulatory compliance, they must provide valid proof that they are interacting with financial systems to prevent fraud, embezzlement or other crimes against humanity. We do this globally; there is no exception for blockchain technology.

"Informally, zero-knowledge proofs enable one party (the prover) to attest to another party (the verifier) that a known computation was executed honestly for some *private inputs". — WP*

Zero-knowledge proof technology allows users to encrypt and make undeniable assumptions about the validity of said transactions, ensuring the same level of integrity seen in today's records committed on public ledgers. However, in order for users to use such powerful methods of encryption, the same cryptographic primitives must be built into the software stack the chain is built upon, and later expressed by programmers in the same manner, which requires *specialized domain expertise.* Most existing solutions for ZKP (Zero-Knowledge proofs) "*have weak guarantees of correctness and safety*". The lack of assurance even the most talented developers face is launching faulty systems to the open source, which could potentially harm those interacting with these ZKP's.

1.1 Our contributions

LEO

I. **Statically-typed** : "Statically typed is a (a) programming language characteristic in which variable types are explicitly declared and thus are determined at compile time. This lets the compiler decide whether a given variable can perform the actions requested from it or not." — Techopedia

a) Haskell, FORTRAN, JAVA, C, C#, C++, Ada, JADE, Pascal, ML, Perl, Scala

II. **Functional Programming Language** : "Functional programming languages are specially designed to handle symbolic computation and list processing applications. Functional programming is based on mathematical functions." — tutorialspoint — LEO is also *intuitive*, which means that the semantics of the language is compact and easier to understand than most languages. Programs in Leo are compiled and executed offline.

III. **Correct-by-construction**: "A compiled program can be formally verified with respect to its high-level specification, producing a proof of semantic equivalence between its input and output. Since LEO programs can be mathematically represented, strong guarantees of correctness and safety are achieved by way of formal methods." — WP ;

Summary of understanding: LEO's *type inference* allows programs to achieve formal verification due to the data equivalence of its original input in accordance to the output of that original input. This check-sum is done at compile time to assure that a program is working as intended, using strong cryptography to prove the correctness of the formal methods used during this intermediary "*correct-by-construction*" session.

IV. **Verifiable Computation**: "*an execution of a compiled program can be succinctly verified by anyone with respect to its inputs in zero-knowledge.* LEO program executions produce a non-interactive, zero-knowledge proof attesting to the validity of the computed output that is cheap to verify." — WP

Summary of understanding: Verification happens in ten of milliseconds depending on the complexity of the program, and can be validated by anyone (validators/POS) on the network. The nature of the ZKP is non-interactive — a security guarantee. The sequential formal methods formulating the Zero-knowledge proof in the Leo program happens before the transaction is deposited, which means the program cannot suffer from a "man-in-the-middle" attack. The validators on the Aleo network cannot influence a change between ***prover*** and the ***verifier*** of a

LEO program. In other words, the proof is "succinctly verified" off-chain, and then validated on chain and is immediately verified by the receiving party, *the verifier.* This is a mouthful, so I suggest you do you due diligence to articulate **Verifiable Computation** in your own words.

Note: *miner re-execution* is not needed since the computation of the ZKP happens off chain and is validated by the LEO program itself. We are fortunate of how **intuitive** LEO is; this feature saves programmers a lot of time when building Zero-knowledge Proof applications.

Get Excited! : There is no "**Verifier's Dilemma**" nor a need for mechanisms such as ***gas***, users do not compete with one another for time-shared resources. Leo programs are bound only by the computation resources of the machine executing the program, eliminating ***run time, stack size and instruction set*** problems!

A high level syntax, giving us the highest level of expression.

Leo Properties

I. **Composability**: "*A user may combine arbitrary functions from compiled programs of their choice, constructing novel programs for execution.* LEO supports inter-process communication for functions to exchange data with one another while preserving program isolation to prevent malicious users from interfering with the execution of the program" —WP

Summary of understanding :

— ***Arbitrary functions*** are "symbols that may be considered to represent any one function of a set of functions."— Merriam Webster.

In Leo, programmers can choose from a variety of remade packages, or create their own and contribute to the DSL itself.

— ***inter-process communication of functions*** (IPC) : "A set of programming interfaces that allow a programmer to coordinate activities among different program processes that can run concurrently in an operating system. This allows a program to handle many user requests at the same time." — techtarget

Leo programs execute in a somewhat concurrent fashion; the program imports the necessary dependencies from the package manager into a sort of black box environment, compiled the program and checks for errors while doing so in congruence with the information it has about the package it originally sourced (online=>offline storage compilation).

II. **Decidability**: "A user may know with certainty from their high-level specification what the compiled program will execute. LEO is intentionally Turing-incomplete, avoiding "Turing complexity" to enable strong guarantees of both time-safety and memory-safety. This means a user can statically analyze the entire call graph of a LEO program, and detect type definition bugs, improper type casts, and unintentional mutability errors at the time of compilation." — WP

Summary of understanding :

— ***Leo is Turing-incomplete :*** Though the Leo language is not complete, this is useful for security and memory management analysis when scouting for errors. Leo introduces safer constructs such as bounded recursion to "***unroll call stacks at compile time to optimize for reduced instructions steps and minimized memory usage, along with guarantees of termination.***" — WP

— ***call graph :*** "A Call Graph allows the user to view the relationship between Parents and Child subroutines in their program. In essence, Each node in the Call Graph ... can be thought of as the "Parent function" and each node as the "Child function." — Microchipdeveloper . Think of a mind map / flowchart if you need a mentally imagine what this looks like.

— ***type definition : "***Typedef is used to create new names for types and structures. It can save the user some typing when using a structure or variable data type.***"*** — syntaxdb . Leo safely detects changes in defined types throughout the program before compilation, similar to the rust programming language. A program will not complete itself if it is not correct.
note : The programming language **Rust** is the backbone of Leo

— ***type casts : "*** Type casting refers to changing a variable of one data type into another. The compiler will automatically change one type of data into another if it makes sense. For instance, if you assign an integer value to a floating-point variable, the compiler will convert the int to a float**."** — Developer Insider . The Leo compiler is intuitive enough to find improper type casts, saving you nasty bugs in the long run.

— ***mutability errors :*** A mutable value is one that can be changed without creating an entirely new value. At the time of compilation, the Leo compiler will throw you an error if you had mistakenly changed a value that changed unexpectedly.

III. **Universality:** " A user does not need to facilitate a cryptographic ceremony to achieve verifiable computation for their program. LEO is a universal compiler, meaning it is able to efficiently derive the program-specific public parameters for use in a compiled program. " — WP

Summary of understanding : There is not a need to have a trusted set up between users of the network and the validators to exchange encrypted notes to verify if the deposit's are accurate. The program is uses a set of trusted parameters to ensure the integrity of the program's functionality. The parameters of this compiler cannot be skewed in any way, ensuring the security of the program executes as intended for every user / programmer at the time of the programs execution.

Primitive Leo Features

I. **Private by Default :** Leo programs resist miner front-running and arbitrage attacks, since all state transitions are not publicly disclosed unless by choice. We call this unique feature, *selective disclosure.* Users are ensured that their transactions are private to them unless they choose to have a 3rd party view their account's state transitions for various purposes.

Reasons to selectively disclose private state transitions
- State regulatory compliance.
- Internal or 3rd party tax audits
- CPA consultation
- Financial service's such as asset protections

II. **Formal Verification** : Programs are guaranteed to be strong and correct during compilation. *Implicit casting* is strictly forbidden, but *typecasting* is not forbidden, giving the programmer more control over their program. This ensures that the input and output semantic representation's the compiler is completing is exactly what the programmer expects.

— ***Implicit casting :*** "The process of converting one type of object and variable into another type is referred to as **Typecasting**. When the conversion automatically performs by the compiler without the programmer's interference, it is called **implicit type casting** or widening casting." — Javapoint

III. **Remote Compilation** : Since users are expected to generate expensive and time insensitive proofs with their own computational resources, users can dedicate the execution of a program to even the most untrustworthy of users. Since the verification of the ZKP is relatively cheap, users can verify the correctness of the proof extremely quickly. Since Leo programs are verified, this makes the mobile / smart device industry a ripe environment for Leo applications to deploy too since computational cloud services can run the program for you.

IV. **A Perspective on production readiness** : Constructing a full fledged Aleo/Leo program is no small feat. The language is rather robust and has a long development roadmap ahead of itself in order for it to be made for real-world applications. However, the methodology of Leo was built to last the test of time. As contributors continue to work with Leo, the community and its users will grow stronger everyday.

Approaches to Circuit Synthesis	Built Systems	R1CS Compatible	Testing Framework	Package Registry	Import Resolver	Remote Compiler	Formally Verified
Gadgets	[SCI, bela, jsn, ark, BCG+20]	✓	✓		⋆		
CPUs	[BCTV14a, BCTV14b, WSR+15]	✓					
DSLs	[PGHR13, BCG+13, BFR+13, KPP+14, CFH+15, ET18, GN20, OBW20]	✓	⋆				
Leo	This work	✓	✓	✓	✓	✓	✓

1.2 Related work

Gadgets | **Handcrafted Circuits**
- A circuit in computer science is how the outputs of a mathematical function is computed given an input. A circuit will pass input values through an order of **gates**. Each gate computes a function, which is a subroutine that runs a sequence of programatic instructions that perform a specific task. [SCI, bela, jsn, ark and BCG+20-(PDF)] Users would have to have a solid understanding of native/constraint/non-native constraint type operations to write the logic to craft circuits.

Processing | **CPU-style circuits**
- [BCTV14a, BCTV14b] these types of handcrafted circuits attempt to simulate a simple CPU, processing bits of machine code in multiple steps using the CPU's own memory to access and intermediate state representation. If architected improperly, this approach can incur large performance costs, and requires tailored **elliptic curves** in order for the logic to work. However costly this approach may be, Leo does take this approach since CPU circuits can be customized to assure strong safety and correctness.

Writing ZKP | **DSLs**
- [OBW20, ET18] high-level languages that compile to circuits is a highly effective way to perform ZKP operations. Leo makes no comparison to other methods, other than the fact that Leo introduced *formal methods for circuits,* which attests to the correctness of a program which is an important feature. As you already know, the Leo DSL introduces the first known *testing framework for circuits*, to build more secure applications — as well as *import resolution of packaged circuits* to verify the integrity of **remote compiled** programs.

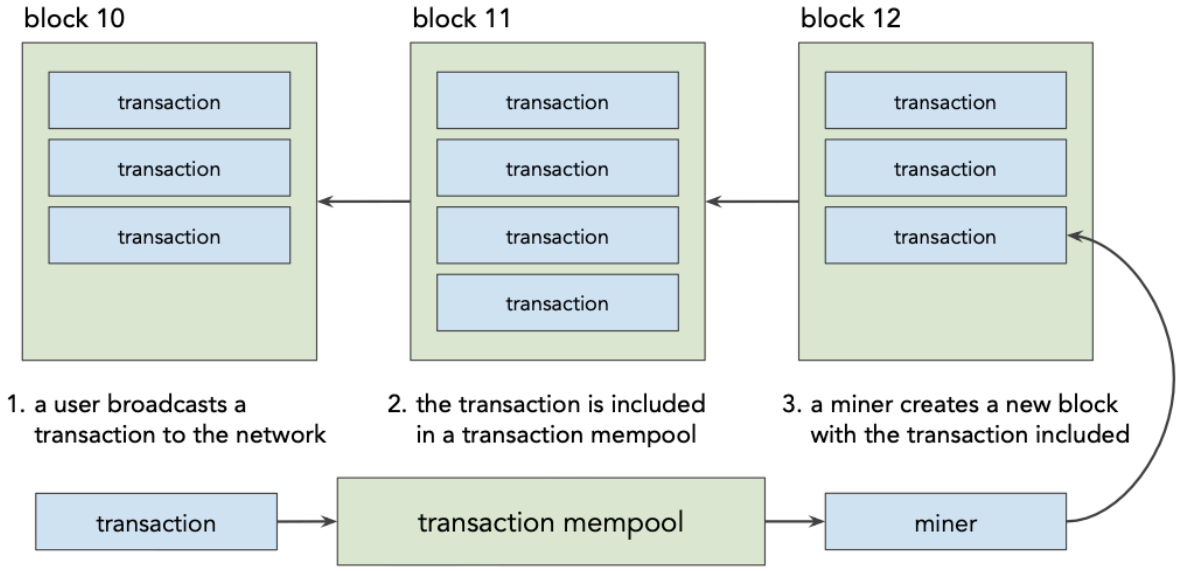
Synthesis of Circuits
1. What are the trade off's for using other DSLs in comparison with LEO?
2. What sort of contributor following does Leo have in comparison to other DSLs?
3. What are some of the major criticisms of Leo, and how do programmers combat them?
4. Looking back through Leo's history, why did the project initially start; which problems was the team initially trying to solve?
5. When the team began architecting Leo, what was missing in production for proving ZKP's globally?; What were the issues being faced pre-existing production?; And which aspects for proving ZKP's needed to improve on a circuit level to move the Pri-Fi industry forward ?

2 Background

Here is some context before you begin reviewing this section

1. *Ledger based systems* - ie. Blockchains.
2. *zkSNARKs* - zero-knowledge Succinct Non-interactive Argument of Knowledge.
3. *R1CS* - rank-1 constraint systems

2.1 Ledger-based systems



Ledger-based systems, like distributed ledgers seen on blockchains, maintain a high level of data integrity for all parties to view, allowing them to aggregate data in a variety of ways. Ledgers cannot change the history (*immutable history*) of statements that are appending (*append-only*) to each and every block. When users interact with these systems, it's done so in a *trustless* manner, meaning there is no 3rd party middleman approving transactions being submitted within these *peer-to-peer* networks like bitcoin. Ledgers like Ethereum embed trustless function calls to programs known as Solidity smart contracts, which allow users to interact with each other using dApp's (*Decentralized Applications*). No single entity has authority or control over these systems, "and they allow any party to construct, append, and validate a transaction in the system."—WP

Ledger-based systems mentioned:

- [Nak09 [Bitcoin], Goo14[Tezos], Woo17[Ethereum], PB17[Plasma], Fil17[Filecoin], ZCa17[ZCash], EOS18[EOS], Mat18[Matter Labs / zkSync], KB18[Cosmos], Ner19[Nervos Network], Spa19[Spacemesh Protocol]].

2.2 Zero-knowledge proofs

zkSnarks

ZKP's (Zero-knowledge proofs) ensure strong privacy and efficiency guarantees using cryptographic primitives that allow a *prover* to convince a *verifier* that a particular statement is true without revealing any of the information to the entity verifying said contents.

- Example: I know `x` such that `y = function(x)`
- In English : "*I know the secret, I can't tell you the secret. But I can prove to you that I know the secret.*" — Howard Wu

Leo defines it's use of ZK as ***publicly-verifiable preprocessing zkSNARK*** [BCI+13, GGPR13], which consists of three algorithms — a *setup, prover and verifier*.

Components of zkSNARKS	
F	Predicate of knowledge, checks if x is and encryption of w and that w is a valid payment.
pk	Proving key
vk	Verification key
X	Public input, encryption of payment details
W	Secret input, private payment details
π	Output / valid proof

I. **Setup :** I have a predicate of knowledge (f) which outputs both a proving key and verification key. This is considered to be a trusted setup, since the steps proceeding the setup involve values that must remain a secret. However, ZK implementations like Monero mitigate this requirement through the use of bulletproofs, which is a trustless setup.
Bulletproofs : *are short non-interactive zero-knowledge proofs that require no trusted setup. A bulletproof can be used to convince a verifier that an encrypted plaintext is well formed .*
The proving key and verification key are only ran once as public parameters and are no longer needed after the Setup sequence.

- Setup $(F) \rightarrow (pk_F, vk_F)$

II. **Prover :** First let's imagine that you are the person acting as the prover. The prover takes the proving key as a public input (x) for the predicate of knowledge (F), and replaces (F) with a private input (w) and then outputs a valid proof (π). This prover (π) essentially "proves" that the predicated input of (F) and the public input of (x), attests too the fact that I as the **Prover** know a private input ('w', the secret), and I can prove the following statement ::

—'**My Predicated-input function consists of my public input, and private input ... this is true'**.
—The same is said in more formal approach in the Leo White-paper; "**given F and x, I know a secret w such that F (x, w) = true**" .

- Prover (pk_F, x, w)

III. **Verifier :** Now, let us imagine that our Verifier is another person that you the prover are interacting with. You give the verifier your proving statement, and our verifier takes the verification key, the public input (x) for the predicated knowledge that was initially Set (F), and the valid output

proof (π). The verifier will know if the outputs are True, if and only if, the proof the prover handed the verifier is valid and correct, otherwise, the output the verifier will attempt to make is going to be false. "The verifier can be run by anyone who wishes to verify a claim to the statement." —WP

- Verifier (vk_F , x, π) → true/false

ϕF	R1CS instance, related to circuits of logical gates
π	Valid input prover / a proof
p	Large prime order of inputs
k	Number of inputs
N	Number of variables
M	Number of constraints
ϕ	Phi (1.61803…)
A	Left coefficient
B	Right coefficient
C	Output coefficient

Note : Performant (and universal) state-of-the-art circuit-specific **SRS** (*preprocessed universal* / updatable **structured reference strings**) like Gro16 can be used for proof systems that require cryptographic "ceremonies", which many applications require for real-world deployment's.

2.3 Rank-1 constraint systems (R1CS)

Define R1CS: "R1CS (rank-1 constraint systems) define **a set of bi-linear equations which serve as constraints suitable for ZK proofs**. They describe the execution of statements written in higher-level programming languages and are used by many ZK proof applications, but there is as yet no standard way of representing them." — ZKproof.org

Define a Logic gate: "A logic gate is **a device that acts as a building block for digital circuits**. They perform basic logical functions that are fundamental to digital circuits. Most electronic devices we use today will have some form of logic gates in them." — techtarget.com

Operations of a zkSnark using R1CS

1. zkSNARK proof systems require the function of (F, predicate of knowledge) as an instance of R1CS, which is closely related to circuits of logical gates.

What a zkSNARK proof is doing is attesting that the set of constraints is satisfiable, meaning, the **size** of the (R1CS) ϕF is related to the **execution time** of (F).

2. Next, we describe the type of computation used to invoke the zkSNARK. Since R1CS is composed of constraints, the **size** of the R1CS (ϕF) expresses the relation is has for the function (F) in a mathematical form. Thus, instance ϕF is now invoked for the **Prover** to providing a set of values that represent the public(X) and private inputs (w replaces F).

3. These values are then instantiated and are represented in a field of a large *prime order* (p).

Note: Integer types are much less than (p)

4. In order to get a valid proof (π), the R1CS instance (ϕF) *"***size"** is placed over the field of prime order (p) and parameterized by:

> the number of inputs (k)

> number of variables (N), with (k) being less than or equal to (N)

> and the number of constraints (M)

5. The instance of Phi (ϕ) is composed of a **tuple** : an ordered set of values.

> tuple == (k, N, M, **a, b, c**, p)

> **a, b and c are** (1 + 'number of variables' (N)) × 'number of constraints' (M)

matrices over the field.

6. We now define the input for ϕF as a vector (*a type of array that is one dimensional)* $\boldsymbol{x}$ in our Field raised to the power of out inputs (k) — F_k .

7. The witness for ϕF as a vector $\boldsymbol{w}$ in our field raised to the power of our number of variables minus our number of inputs — F_{N-k}

8. A value input-witness pair (x, w) satisfies *ϕF,* if and only if, we let $\mathbb{z}$ equal (1, $\mathbb{x}$, $\mathbb{w}$) be a vector of our field to the power of one plus our number of variables — F_{1+N}

$$\left(\sum_{i=0}^{N} \mathbf{a}_{i,j} z_i\right) \cdot \left(\sum_{i=0}^{N} \mathbf{b}_{i,j} z_i\right) = \left(\sum_{i=0}^{N} \mathbf{c}_{i,j} z_i\right)$$

This constraint form above is when *predicated knowledge*, the function (F), is reduced to an R1CS instance *ϕF . For example, If you were attempting to describe the instantiation of a boolean variable, you can introduce a constraint that restricts the variable to take of a value of sticky 0 or 1 by checking:* (variable) * (variable - 1) = 0

Don't let the math scare you! You do not need to go digging back into you old notebooks to try to solve this alone. I know math can be intimidating for programmers, especially for those learning an entirely new programming language. I assure you, once you see how a zkSNARK is broken down into bite size pieces, it's no different than tinkering around with an electrical circuit board.

Here's a Challenge !

I highly recommend you create a physical color coded mind map of the steps listed above to try to visualize the sequence of a zkSNARK. The better you understand the logic of a ZKP, the better you will be creating your own private applications in Leo.

Each ***constraint*** in the ***relation*** can be thought of as representing a *logical gate*, which is a device that implements a *Boolean* function; "true or false". Leo was designed to handle these sorts of complexities for you since circuits are easily reducible to these sorts of forms.

Define Constraint: "In mathematics, a **constraint** is a condition of an optimization problem that the solution must satisfy. There are several types of constraints—primarily equality constraints, inequality constraints, and integer constraints. The set of candidate solutions that satisfy all constraints is called the feasible set." — Wikipedia

Example of constraints: "The variable in a word problem would be what we don't know, and the constraint is the limit to the value of what we don't know. So, for example, if Vic works a certain number of days a month. So, if you see this phrase in a word problem, and it says a certain number of days, but we don't know the number of days, we'll call that X days, and that's what tells us it's a variable. We don't know the value, but it's a specific number. So, that's our variable. And, then, our constraint is zero and 31 are our constraints because you can't have more than 31 days in a month, and you can't have fewer than zero. So, our constraints are anywhere between zero and 31." — Charlie Kasov, synonym.com

Define Relation: "A relation between two sets is a collection of ordered pairs containing one object from each set. If the object x is from the first set and the object y is from the second set, then the objects are said to be related if the ordered pair (x, y) is in the relation." — mathinsight

If you remember back from algebra class, this is like the **domain** and **range** of a parabola.

2.4 Programming languages, compilers, and formal methods

If you've played around with multiple programming languages before, then you know that each language is its own sugary syntactical and semantic features. Even in the last decade or so, brand new paradigms of programming emerged, like imperative, functional, object-oriented, and logic programs had been born like the Rust programming language (July 7th, 2010). Languages are often created to better represent distinct models of computing for specific input/output operations. Languages are like tools in a car shop, we don't use a plunger to fix a dent in the car, instead we use a dent puller. See, humans can do something right for once.

New languages often have new bounded computations, and there sometimes exists a *reduction to NP* relation by which their logic can be mathematically represented *:*

- *NP,* this essentially refers to the set of problems for which it is easy to verify a proposed solution.
- P, is the set of problems that are **easy to solve** and for which it is easy to verify a proposed solution.
- "If I know how to do everything very well, does it mean that I can always do things well?"

The Benefit of using *Formal Methods* for real-world applications
Given these conditions, programming languages can compute outputs that are flexible enough to compute a Zero-knowledge proof. Before circuits had to be created using only specialized hardware and machinery which would then allow existing software to compute various outputs. However, since the outputs of ZKPs are amenable with the outputs that existing programing language's produce, certain programming languages like **Leo** are able to write these circuits using only software without the need for specialize hardware.

This is a critical point since Leo uses *formal methods* to produce outputs that reduce the impact of bugs and vulnerabilities most programs face, especially those that currently support the use of smart contracts. In 2022 alone, several Multi-Million dollar Smart contract bridge hacks occurred, resulting in an entire shutdown of major blockchain networks like the Horizon, Ronin and BNB bridge. Formal methods of computation can bring the assurance for

systems that rely on the precision and correctness of specific applications, like bridges. Before *formal methods* were introduced to programming languages, they had to be written using natural language and source code was written to implement their specifications. Languages are like tools, and some tools programmers use either deliver half baked or complete solutions for the specific problems they are required to solve. OOP's and Functional languages help make programs understandable, while functional languages make it easier to prove correctness with respect to a specification. The trade off is that it is often difficult to use new tools to get a job done; but as the saying goes, "Nothing great ever came easy".

Compilers
Formal methods gives us access to *formal grammars* which have helped improve parsing correctness, but hand-written *optimizing compilers* have remained the norm.

Formal Grammars : "A *formal grammar* is a set of rules for rewriting strings, along with a "start symbol" from which rewriting starts." — Wiki

Optimizing compiler : "In computing, an *optimizing compiler* is a compiler that tries to minimize or maximize some attributes of an executable computer program. Common requirements are to minimize a program's execution time, memory footprint, storage size, and power consumption (the last three being popular for portable computers)." — Wiki

These machines had to be formally modeled that depended on post-hoc verification to provide correctness; these verified compilers had some success like CompCert since it reduced the compilation time during preprocessing.

Preprocessing : "In computer science, a preprocessor (or pre-compiler) is a program that processes its input data to produce output that is used as input to another program. The output is said to be a preprocessed form of the input data, which is often used by some subsequent programs like compilers." — Wiki

A *verified compiler* is meant to build a ***verifying compiler***, where instead of making the compiler always generate an output that is logically equivalent to it's input, a verifying compiler generates a proof of the semantic equivalence between the input and output every time it's ran.

Verifying compiler : "A verifying compiler compiles specified programs, which contain normal program text and additional text specifying desired properties of the program. A verifying compiler attempts to prove that the specified properties hold for all possible inputs to the program." — theory.standford.edu

The compilers make use of formal method tooling to formalize and prove properties of programming languages and their compilation; like the theorem prover ACL2 [KM]. ACL2 has been used for software modeling and verification in academia and industry production due to its proven efficiency comparable to mainstream programming languages.

3 Design of the Leo Programming Language

“We begin by describing the design of LEO. After introducing the **environment** that LEO programs execute in Section 3.1, we describe in overview the syntax of our language in Section 3.2, and its static and dynamic semantics in Section 3.3 and Section 3.4 respectively. We discuss its **R1CS** semantics in Section 3.5. For a comprehensive description of the language, refer to the LEO Language Formal Specification [Cog]. “ — WP

3.1 Environment

All Leo programs execute in an offline ledger where their transactions produce state transitions about zero knowledge applications. The state transition is encrypted into a privacy-preserving transaction that is broadcasting to an open network. Programmers can choose to not broadcast their transactions to the open network using the built-in *Testnet3* Leo framework.

Security Guarantee’s

1. **Execution correctness** : Malicious parties cannot create valid state transitions without access to your private key and the application state of a user.

2. **Execution privacy** : Selective disclosure guarantee, which ensures that users reveal only the information that is intended to be made public from an application issuing a transaction.

3. **Transaction non-malleability** : When a transaction is transiting to the ledger, it cannot be modified by malicious parties protecting the integrity of data committed to the open network.

3.2 Syntax

Question : Where is [Cog] Developer documentation to describe the AST’s in depth?
Leo’s Context-free grammar defines *abstract syntax* which consists of *abstract syntax tree’s* (**AST’s**). Common data types and operations are supported by Leo, and the distinctions between abstract and concrete syntax is minimal.

3.2.1 Types

Leo uses both **scalar** and **aggregate types**. All values of scalar’s are *atomic*, while all values of aggregates *contain other values*. Data types are usually either ~~Void~~, Scalar or Aggregate. Scalar types make use of Pointers, Arithmetic types (including *integral* (signed and unsigned integers) and *floating types*) and Enum’s. Other Scalar’s types you may be familiar with are Bool’s and Strings. Aggregate types give us access to things like Unions, Structures and array’s. A good way to remember them is that Scalar’s contain a single value while aggregates are collections of scalar values.

Scalar Integer types

- Leo Signed and Unsigned Int size - [8, 16, 32, 128 bits]
    - Int consists of type *addresses* (for Aleo accounts on the distributed ledger)
    - Int consists of type *field elements* (values of a prime field (p))

- Int consists of type *group elements* (elliptic curve points)

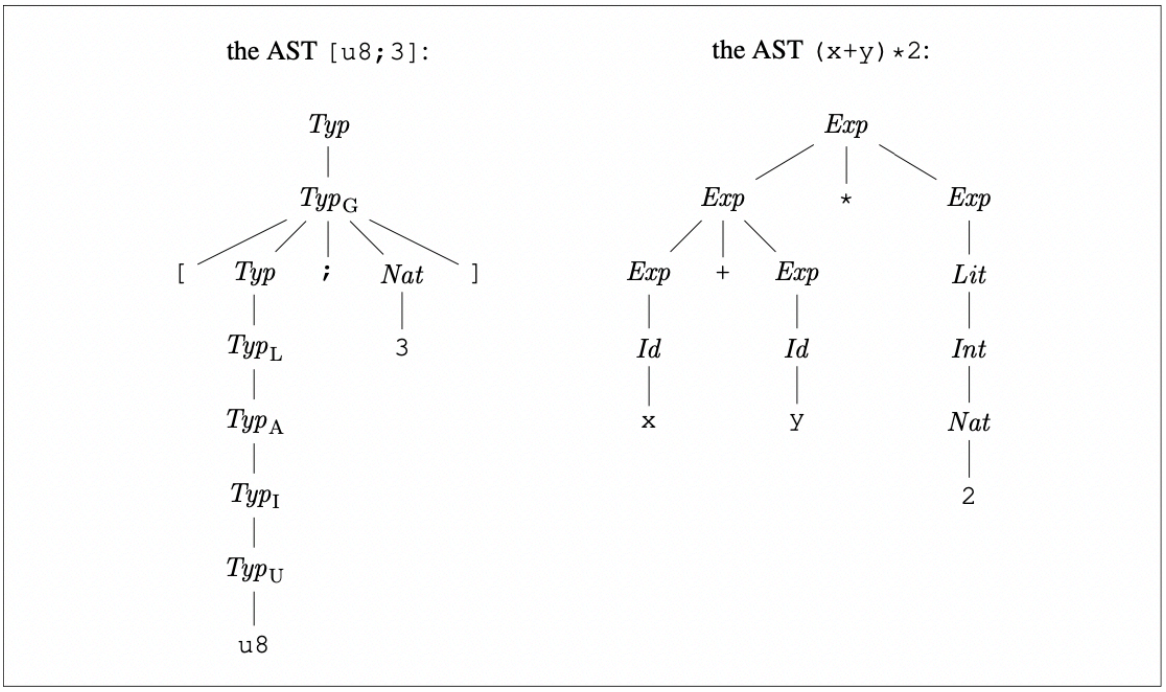
Aggregates

- *Tuples*, which consist of sequences of zero, two or more values of possibly different types.
- *Arrays*, which consist of sequences of one or more values of the same type.
    - *Nested Arrays* (arrays of arrays) for multidimensional array construction.
- *Circuit types* are referenced by names (or by `Self` inside of their own definitions) and whose values consist of unordered collections of values off possibly different types, which are accessed by name. A circuit type is similar to a **class type** in other programming languages.

Define class type : "A class is **a group of objects that share common properties and behavior**. For example, we can consider a car as a class that has characteristics like steering wheels, seats, brakes, etc. And its behavior is mobility." — techvidvan

Reminder : "In theoretical computer science, a circuit is **a model of computation in which input values proceed through a sequence of gates, each of which computes a function**. Circuits of this kind provide a generalization of Boolean circuits and a mathematical model for digital logic circuits." — Wiki

AST



Syntax

identifiers	Id	::=	`...`
package names	Pkg	::=	`...`
addresses	$Addr$	::=	`...`
strings	Str	::=	`...`
annotations	Ann	::=	`...`
natural numbers	Nat	::=	`0` | `1` | `2` | `...`
integer numbers	Int	::=	Nat | `-1` | `-2` | `...`
unsigned integer types	Typ_{U}	::=	`u8` | `u16` | `u32` | `u64` | `u128`
signed integer types	Typ_{S}	::=	`i8` | `i16` | `i32` | `i64` | `i128`
integer types	Typ_{I}	::=	Typ_{U} | Typ_{S}
arithmetic types	Typ_{A}	::=	Typ_{I} | `field` | `group`
scalar types	Typ_{L}	::=	`bool` | Typ_{A} | `address`
circuit types	Typ_{C}	::=	Id | `Self`
aggregate types	Typ_{G}	::=	`(`Typ^*`)` | `[`Typ`;`Nat`]` | `[`Typ`; (`Nat^*`)]` | Typ_{C}
types	Typ	::=	Typ_{L} | Typ_{G}
coordinates	$Coor$	::=	Int | `+` | `-` | `_`
literals	Lit	::=	`true` | `false` | Nat Typ_{U} | Int Typ_{S} | Int | Int `field` | Int `group` | `(`$Coor$`,`$Coor$`)group` | `address(`$Addr$`)`
expressions	Exp	::=	Id | `self` | Lit | Id`(`Exp^*`)` | `!` Exp | Exp `&&` Exp | Exp `||` Exp | `-` Exp | Exp `+` Exp | Exp `-` Exp | Exp `*` Exp | Exp `/` Exp | Exp `**` Exp | Exp `<` Exp | Exp `<=` Exp | Exp `>` Exp | Exp `>=` Exp | Exp `==` Exp | Exp `!=` Exp | `if` Exp `?` Exp `:` Exp | `(`Exp^*`)` | Exp`.`Nat | `[(...`$^?$$Exp$`)*]` \| `[`$Exp$`;`$Nat$`]` \| `[`$Exp$`; (`$Nat^*$`)]` | Exp`[`Exp`]` | Exp`[`$Exp^?$`..`$Exp^?$`]` | Typ_{C} `{(`Id`:`Exp`)*}` | Exp`.`Id | Exp`.`Id`(`Exp^*`)` | Typ_{C}`::`Id`(`Exp^*`)`
print console functions	$Print$	::=	`log` | `debug` | `error`
statements	Stm	::=	Exp`;` | `return` Exp | `let ((mut`$^?$ Id`)*)(:`Typ`)`$^?$ `=` Exp`;` | `(if` Exp `{`Stm^*`} else)*` `({`Stm^*`})`$^?$ | `for` Id `in` Exp`..`Exp `{`Stm^*`}` | Exp `=` Exp`;` | Exp `+=` Exp`;` | Exp `-=` Exp`;` | Exp `*=` Exp`;` | Exp `/=` Exp`;` | Exp `**=` Exp`;` | `console.assert(`Exp`);` | `console.`$Print$`((`Str`,`Exp^*`)`$^?$`);`
function definitions	Fun	::=	Ann^* `function` Id `(((mut` | `const)`$^?$$Id$`:`$Typ$`)*)` `(->` Typ`)`$^?$ `{`Stm^*`}`
member definitions	Mem	::=	Id`:`Typ | `(mut`$^?$ `self)`$^?$ Fun
circuit definitions	$Circ$	::=	Ann^* `circuit` Id `{`Mem^*`}`
paths in packages	$Path$	::=	`*` | Id `(as` Id`)`$^?$ | Pkg`.`$Path$ | `(`$Path^*$`)`
import declarations	Imp	::=	Ann^* `import` Pkg`.`$Path$`;`
files	$File$	::=	$(Imp$ | $Circ$ | $Fun)^*$

3.2.2 Definitions

Example

Fun **::=** *Ann** `function` *Id* (`mut`$^?$ `self`)$^?$, (`mut` | `const`)$^?$ *Id:Typ*)* (-> *Typ*)$^?$ {*Stm**}

A **Function Definition** may be a circuit type or at the top-level, which means that it doesn't need to be apart of any circuit type. Members and top-level functions have the same form, but if they aren't apart of a circuit type, then they cannot have `self` inputs.

- Functions with the *mut* modifier can be modified (e.g. assigned) inside the function; otherwise they cannot change (e.g. immutable)
- Function inputs with the const modifier are declared (and checked) to take only compile-time constant values (read section 3.3).
- When the function output type is the empty tuple type, it may be omitted.

Circuit type definitions consist of the name of the type (`function`) with one of more **members.**

- **Circuit Type :** Resembles a class in an object-oriented language, where member variables resemble 'fields' (a term avoided in the Leo Language to avoid confusion with 'prime fields') and member functions resemble methods. Circuit types do NOT support inheritance

  Members
  - *Member variable* : **name + type** : it defines a name and typed component of the values of the circuit type.
  - *Member function* : This will consist of a function definition with an optional `self` parameter that receives a value of the circuit type.
    - Member function's with `self` inputs can be modified inside of a function definition; otherwise they are immutable.

Note : *ALL MEMBER VARIABLES ARE ASSOCIATED TO INSTANCES (VALUES) OF THE* ***CIRCUIT TYPE.***
In object-oriented terminology, there are no static member variables.
Essentially, objects do not share heap memory.

function definitions	*Fun* ::= *Ann** `function` *Id* (((`mut` | `const`)$^?$ *Id* : *Typ*)*) (-> *Typ*)$^?$ {*Stm**}
member definitions	*Mem* ::= *Id* : *Typ* | (`mut`$^?$ `self`)$^?$ *Fun*
circuit definitions	*Circ* ::= *Ann** `circuit` *Id* {*Mem**}

The grammar rule for member definitions suggests that the `self` input starts the definition of a member function. If the self input is present, it is stated prior to the other inputs.

3.2.3 Expressions

To be continued…