LEO: A Programming Language for Formally Verified, Zero-Knowledge Applications

Collin Chin
collin@aleo.org
Aleo

Howard Wu
howard@aleo.org
Aleo

Raymond Chu
ray@aleo.org
Aleo

Alessandro Coglio
acoglio@aleo.org
Aleo, Kestrel

Eric McCarthy
ericm@aleo.org
Aleo, Kestrel

Eric Smith
esmith@aleo.org
Aleo, Kestrel

May 18, 2021

Abstract

Decentralized ledgers that support rich applications suffer from three limitations. First, applications are provisioned tiny execution environments with *limited running time*, *minimal stack size*, and *restrictive instruction sets*. Second, applications must reveal their state transition, enabling *miner frontrunning attacks* and *consensus instability*. Third, applications offer *weak guarantees of correctness and safety*.

We design, implement, and evaluate LEO, a new programming language designed for *formally verified, zero-knowledge applications*. LEO provisions a powerful execution environment that is not restricted in running time, stack size, or instruction sets. Besides offering *application privacy* and mitigating *miner-extractable value* (MEV), LEO achieves two fundamental properties. First, applications are *formally verified* with respect to their high-level specification. Second, applications can be *succinctly verified* by anyone, regardless of the size of application.

LEO is the first known programming language to introduce a *testing framework*, *package registry*, *import resolver*, *remote compiler*, *formally defined language*, and *theorem prover* for general-purpose, zero-knowledge applications.

Keywords: decentralized applications; zero knowledge proofs; programming languages; formal methods

Contents

1 Introduction

Decentralized ledgers are append-only systems that offer an indisputable history of state transitions and ensure that all parties have access to the same view of data on the ledger. This mechanism enables multiple parties to collaborate with minimal trust assumptions, and has been a catalyst for peer-to-peer payment systems (cryptocurrencies). Notably with the advent of systems that enable peer-to-peer applications, decentralized ledgers have become the foundation for new paradigms of financial systems, governance, social networking, digital gaming, and data sharing. In this work we address three limitations of decentralized applications, the first about *scalability*, the second about *privacy*, and the third about *auditability*.

The scalability problem. The remarkable power of decentralized ledgers comes at a cost: *the computational integrity of these systems is achieved via direct execution of state transitions*. In these systems, miners carry the overhead of this transaction execution. Unfortunately, this creates three fundamental problems.

First, each miner must re-execute every transaction (and its associated computation) in order to verify its validity. This technique is not only *highly wasteful* of computational resources, but it creates a transaction throughput problem that users who send transactions containing large (or even unbounded) computations *run the risk of stalling the system*. Thus, in order to discourage denial-of-service attacks, current decentralized ledgers utilize mechanisms such as *gas* to try to economically bound this risk.

Second, while these mechanisms reduce the feasibility for a denial-of-service attack, *it does not mitigate it*. When a computationally-expensive transaction is sent, it simply may not be economically profitable for miners to verify the transaction, creating a problem referred to as the "Verifier's Dilemma" [LTKS15]. In practice, these problems have resulted in successful forks and attacks on decentralized ledgers [Bit15, Eth16].

Third, these mechanisms fundamentally limit the expressivity and functionality of decentralized applications. By re-executing every transaction, users are *competing against one another* for compute cycles from miners, time-sharing the limited resources across all parties. Consequentially, applications are provisioned tiny execution environments with *limited running time*, *minimal stack size*, and *restrictive instruction sets*.

The privacy problem. Current decentralized ledgers severely limit applications as a consequence of their design. Recall the core strength of decentralized ledgers is also their core weakness: *the history of all state transitions must be executed by all parties*.

In ledger-based systems such as Bitcoin [Nak09], this means means every payment transaction reveals the payment's sender, receiver, and amount. 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. This problem has motivated prior work to achieve meaningful privacy guarantees for payments [ZCa17, BCG$^+$14].

Applications in current decentralized ledgers suffer from two problems. First, the lack of privacy severely limits the direct application of decentralized ledgers. When the state transition of every application is public, so is the history of function calls and its associated parties. In practice, this *limits the types of applications* that current ledgers can support, such as dark pools. Second, recent work has shown the lack of privacy enables miner frontrunning and arbitrage attacks, creating a problem referred to as "Miner-Extractable Value" (MEV) [DGK$^+$19, EMC19]. Beyond exploiting the inherent weaknesses of current ledgers for financial motivations, miners can prioritize transaction ordering at their discretion to create *consensus instability*. These consensus-layer security risks pose *systemic risks to decentralized ledgers*.

The auditability problem. To address some of the shortcomings of decentralized ledgers, zero-knowledge proofs have received much attention for their ability to achieve strong privacy, integrity, and efficiency guarantees [BCG$^+$20]. 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*. For example, while Bitcoin requires users to broadcast their private payment details in the clear, zero-knowledge proofs enable users to broadcast *encrypted* transaction details by *proving* the validity of the payment without publicly disclosing the contents of the payment.

However this approach faces two fundamental challenges. First, while zero-knowledge proofs are a promising technology for decentralized ledgers, writing efficient zero-knowledge applications requires *specialized domain expertise*. Not only does this cryptographic primitive need to be integrated into the decentralized ledger it operates on, but applications in this construct must be expressed in mathematical representations that are simply unapproachable to most users. Second, existing solutions to architect zero-knowledge applications have *weak guarantees of correctness and safety*. Beyond correctly constructing the application itself, there is a lack of guarantees that the application itself is executed correctly with respect to the prover and verifier. Unfortunately, this has resulted in serious deployment risks [SWB19] and critical bugs [Tor19, Sem19, Bas20].

In summary, there is a dire need for auditable techniques to architect zero-knowledge applications for use in decentralized ledgers. Prior works only partially address this need, as discussed in Section 1.2.

1.1 Our contributions

We present LEO, a new programming language designed for *formally verified, zero-knowledge applications*. LEO is a statically-typed, functional programming language, built with intuitive semantics that enable users to write decentralized applications that attest to the correctness of their offline compilation and execution. The LEO compiler simultaneously achieves two fundamental properties:

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

- **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.

Since LEO programs are verifiably computed and can be verified in tens of milliseconds, *regardless of the offline computation*, they can be used to realize highly-efficient state transitions on a decentralized ledger *without miner re-execution*. As 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. Instead, LEO provisions a powerful execution environment that is not restricted in running time, stack size, or instruction sets for the application, and is bounded merely by the computational resources of the machine that it executes on. LEO offers an expressive syntax to achieve a high-level programming language with the following properties:

- **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.

- **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.[1]

- **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.

Private by default. In LEO, applications are *private by default*, expanding the types of rich applications that can be supported by decentralized ledgers. In addition, the private nature of decentralized applications written in LEO affords them resistance to miner frontrunning and arbitrage attacks, as their state transitions are not required to be publicly-disclosed on a decentralized ledger. [2]

Formal Definition. The LEO language has a formal definition of its syntax and semantics. This formal definition not only provides a precise definition of the language, but also forms the basis for formal proofs of correctness of the compilation of LEO to R1CS. Very few languages are formally defined.

Formal verification. The LEO compiler is architected for strong guarantees of correctness and safety. The language is designed with conventional syntax and obvious semantics—for example, implicit casting is forbidden. The LEO compiler automatically translates LEO programs to a mathematical representation, whereby it is designed to perform formal verification of the semantic equivalence between its input LEO programs and its output representation. This is a foundational improvement in the trustworthiness of the resulting zero-knowledge proof. See Section 4.2 for details.

Remote compilation. Verifiable computations come with the added cost of producing a zero-knowledge proof to attest to the correctness of the offline execution. As the size of a program grows, both the time and memory resources required to produce a valid proof also grows, which could become infeasible for a user.

To address this challenge, LEO supports *remote compilation*. By its inherent nature of being verifiable, a user may delegate the computation of a LEO program to an untrusted worker who performs the offline execution, producing a computed output along with its zero-knowledge proof that a user may subsequently check. And as verification of succinct zero-knowledge proofs is cheap, the user is able to efficiently verify the correctness of the offline execution in tens of milliseconds.

This feature is of particular relevance for real-world deployments, because it enables users on mobile phones or smart devices to run real-world applications that they otherwise could not run trustlessly today.

A perspective on production readiness. LEO affords reasonable efficiency however is by no means a lightweight construction. We have, after all, set ambitious goals: LEO is the first known programming language to introduce a *testing framework*, *package registry*, *import resolver*, *remote compiler*, *formally defined language*, and *theorem prover* for general-purpose, zero-knowledge applications.

In light of the foregoing ambitious goals, we note that these language features are an evolution of ongoing development to implement and optimize each feature as described in this work to achieve a production-ready framework for zero-knowledge applications. We have, in our opinion, laid the groundwork for a methodology in developing zero-knowledge applications that will prove foundational, not only for its correctness and safety guarantees, but also for its real-world applicability.

[1]Note that Turing-incomplete does not mean LEO cannot support language features like recursion. Rather, LEO introduces a safer construct of *bounded recursion* to unroll call stacks at compile time to optimize for reduced instruction steps and minimized memory usage, along with guarantees of termination.

[2]Note that this is only the case for deployments of decentralized ledgers that themselves are *private by default*, such as Zexe [BCG$^+$20]. In light of such cases, the risks associated with MEV would be mitigated.

1.2 Related work

Approaches to Circuit Synthesis	Built Systems	R1CS Compatible	Testing Framework	Package Registry	Import Resolver	Remote Compiler	Formally Defined	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	✓	✓	✓	✓	✓	✓	✓

Figure 1: Comparison of various approaches for circuit synthesis. ⋆ means the feature is supported in limited scope.

Handcrafted circuits. There has been a long lineage of libraries that offer interfaces for handcrafting circuits [SCI, bela, jsn, ark, BCG+20]. While this approach typically results in circuits composed of far fewer constraints than other approaches, it requires the user to understand the semantics between *native*, *constraint*, and *non-native constraint* type operations. In contrast, our approach seeks to abstract these notions, requiring minimal knowledge of circuits for the user.

CPU-style circuits. Another line of work built upon the notion of handcrafted circuits to simulate a simple CPU, by processing machine code one or more steps at a time, and attesting to the validity of memory accesses and intermediate state representations [BCTV14a, BCTV14b, WSR+15]. While conceptually simple, this approach has shown to incur enormous performance costs, and requires tailored sets of elliptic curves in order to work, limiting the domain of its applicability. We view our work as complementary to this approach, since CPU-style circuits may be architected with more resilient guarantees of correctness and safety by using LEO.

Domain-specific languages. Several works [PGHR13, BCG+13, BFR+13, KPP+14, CFH+15, ET18, GN20, OBW20] have explored developing techniques for compiling to a circuit from a high-level language. Recent work has shown that this is fairly performant to hand. In our work, we make no claims to the efficiency of our construction in comparison to existing languages. Rather, we expand on this lineage to introduce *formal methods for circuits*, which attest to the correctness of program compilation. We also introduce the first known approach for a *testing framework for circuits*, *import resolution of packaged circuits*, and so on.

2 Background

We cover necessary background on the building blocks of LEO: we cover the current landscape of *ledger-based systems* (Section 2.1), introduce a form of zero-knowledge proofs called *zkSNARKs* (Section 2.2), describe the notion of *rank-1 constraint systems* (Section 2.3), and provide a high-level description of *programming languages, compilers, and formal methods* (Section 2.4).

2.1 Ledger-based systems

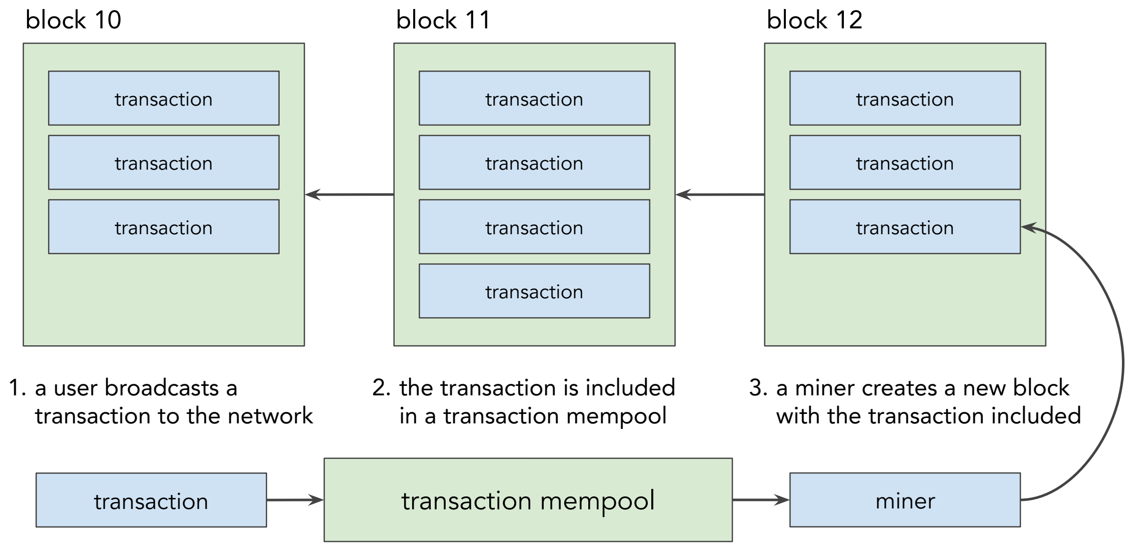

block 10
block 11
block 12
transaction
1. a user broadcasts a transaction to the network
2. the transaction is included in a transaction mempool
3. a miner creates a new block with the transaction included
transaction mempool
miner

Figure 2: Components of a decentralized ledger.

Ledger-based systems provide a mechanism to maintain data in a *consistent manner* across a wide range of parties, even in the presence of corrupted parties. Ledgers are defined as an *append-only* data structure that offers an *immutable history* of all transactions logged in the system. Recently, ledger-based systems have received much attention from academia and seen several industry deployments [Nak09, Goo14, GM16, Woo17, PB17, Fil17, JPM17, QED17, ZCa17, MS18, CZK+18, EOS18, Mat18, KB18, Ner19, Spa19, LP19, TBT19, Lib19, BCG+20].

In peer-to-peer payment systems such as Bitcoin [Nak09], each transaction records *payment details*—including a sender, receiver, and amount. Thus, users of the system transact payments with one another, *without the need for a trusted third party*. The advent of peer-to-peer applications on systems such as Ethereum [Woo17] extend the functionality of ledgers, whereby transactions not only record payment details, but also *embed function calls* to programs. These peer-to-peer systems are considered *decentralized* as no single authority controls the system, and they allow *any party to construct, append, and validate* a transaction in the system.

2.2 Zero-knowledge proofs

Cryptographic proofs offering strong privacy and efficiency guarantees, known as *zero-knowledge Succinct Non-Interactive ARguments of Knowledge (zkSNARKs)*, have received significant interest from academia and industry [Gro10, Lip12, BCI+13, GGPR13, PGHR13, BCG+13, BFR+13, DFKP13, BCTV14b, KPP+14, ZPK14, BCG+14, CFH+15, WSR+15, CFH+15, Gro16, JKS16, KMS+16, NT16, DFKP16, BCS16, GM17, BBC+17, MBKM19, CHM+19, GWC19].

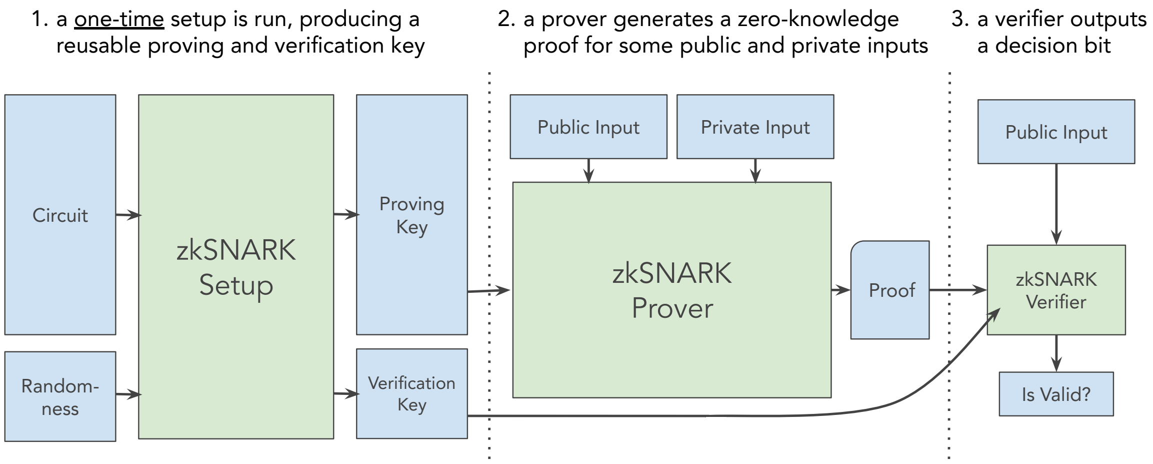

1. a one-time setup is run, producing a reusable proving and verification key
2. a prover generates a zero-knowledge proof for some public and private inputs
3. a verifier outputs a decision bit
Circuit
Random-ness
zkSNARK Setup
Proving Key
Verification Key
Public Input
Private Input
zkSNARK Prover
Proof
Public Input
zkSNARK Verifier
Is Valid?

Figure 3: Components of a zkSNARK. Blue boxes represent data structures while green boxes denote the parties or phases of the proof system.

A zkSNARK allows a *prover* to convince a *verifier* of a statement of the form "*given a function F and input x, I know a secret w such that $F(x, w) = \mathsf{true}$*". The notion of a zkSNARK, formulated in [Mic00, GW11, BCCT12], has several definitions, and we consider one known as a *publicly-verifiable preprocessing zkSNARK* (see [BCI+13, GGPR13]). This definition of a zkSNARK consists of three algorithms—a *setup*, *prover*, and *verifier*:

- $\mathsf{Setup}(F) \rightarrow (\mathsf{pk}_F, \mathsf{vk}_F)$ – The setup takes as input a predicate F and outputs a proving key pk_F and a verification key vk_F. The setup is often considered *trusted* as its intermediate computation steps involve values that must remain secret, however ongoing work, such as [BCG+15, ZCa16, BGM17, BGG18], offers mitigations for this requirement. The setup outputs keys $(\mathsf{pk}_F, \mathsf{vk}_F)$ which are used as public parameters, and in particular, only needs to be run once.

- $\mathsf{Prover}(\mathsf{pk}_F, x, w) \rightarrow \pi$ – The prover takes as input the proving key pk_F, public input x for F, and a private input w for F, and outputs a proof π. The proof π attests to the statement "*given F and x, I know a secret w such that $F(x, w) = \mathsf{true}$*", while revealing no information about w beyond what is implied by the statement. The prover can be run by anyone who wishes to prove their claim to the statement.

- $\mathsf{Verifier}(\mathsf{vk}_F, x, \pi) \rightarrow \mathsf{true/false}$ – The verifier takes as input the verification key vk_F, public input x for F, and proof π, and outputs true if the proof is valid and correct, otherwise outputs false. The verifier can be run by anyone who wishes to verify a claim to the statement.

zkSNARKs enables applications to offer strong guarantees for users. For example, zkSNARKs are the core technology of Zcash [ZCa17, BCG+14], a popular cryptocurrency designed to preserve a user's payment privacy. In this cryptocurrency example, w is the private payment details, x is the encryption of the payment details, and F is a predicate that checks that x is an encryption of w and that w is a valid payment.

In recent years, there has been work to develop *preprocessing zkSNARKs* with a *universal* and *updatable* structured reference string (SRS) [MBKM19, CHM$^+$19, GWC19]. Many applications for real-world deployment requires sampling the SRS using cryptographic "ceremonies" []. This is to ensure there exists no single party who is entrusted with sampling the SRS.

Recent work [CHM$^+$19] has shown that proof generation techniques for an R1CS instance ϕ using a universal SRS achieves practical performance capabilities for both the zkSNARK prover and verifier. We note that the efficiency gap between universal SRS and state-of-the-art circuit-specific SRS [Gro16] is competitive. For the purpose of this paper, it suffices to state that there exists performant (*and universal*) proof systems for the circuit-specific SRS to be used in.

2.3 Rank-1 constraint systems

A zkSNARK proof system requires expressing a function F via a high-level programming language, as an instance of a *rank-1 constraint system (R1CS)* ϕ_F relation, which is closely related to circuits of logical gates. A zkSNARK proof then attests that such a set of constraints is *satisfiable*. The size of ϕ_F is related to the execution time of F.

We now describe the type of computation used to invoke a zkSNARK proof system. Informally, an R1CS instance ϕ_F is a relation composed of constraints expressing the function F in mathematical form. The instance ϕ_F is invoked by providing a set of values representing the public and private inputs. In our case, values are instantiated in a field $\mathbb{F}$ of a large prime order p, where we may implicitly assume that all integer types are (much) less than p.

Formally, an R1CS instance ϕ_F over $\mathbb{F}$ is parameterized by the number of inputs k, number of variables N (with $k \leq N$), and number of constraints M. The instance ϕ is composed of a tuple $(k, N, M, \mathbf{a}, \mathbf{b}, \mathbf{c}, p)$, where $\mathbf{a}, \mathbf{b}, \mathbf{c}$ are $(1 + N) \times M$ matrices over $\mathbb{F}$. We define the *input* for ϕ_F as a vector x in $\mathbb{F}^k$, and the *witness* for ϕ_F as a vector w in $\mathbb{F}^{N-k}$. A valid input-witness pair (x, w) *satisfies* ϕ_F if, letting $z = (1, x, w)$ be a vector $\mathbb{F}^{1+N}$, the following holds for all $j \in [M]$:

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

A function F that is reduced to an R1CS instance ϕ_F is represented in the constraint form above. Each constraint in the relation can be thought of as representing a logical gate, and circuits are easily reducible to this form. We view $\mathbf{a}, \mathbf{b}, \mathbf{c}$ as containing the 'left', 'right', and 'output' coefficients respectively. For example, to describe the instantiation of a boolean variable, one can introduce a constraint that restricts the variable to take on a value of strictly 0 or 1 by checking: $(variable) \cdot (variable - 1) = 0$.

2.4 Programming languages, compilers, and formal methods

Programming languages offer a variety of beneficial syntactic and semantic features. Over the years, specific paradigms for programming have emerged—imperative, functional, object-oriented, logic, to name a few. Broadly, these approaches represent distinct models for computing an output for a given input. And for these bounded computations, there exists *reductions to NP relations* by which their logic can be *mathematically represented*. Under these conditions, we see that programming languages describe computations that are *amenable to zero-knowledge proofs*.

In recent years, growing concerns over the impact of bugs and vulnerabilities in programs has fostered the wider development and application of *formal methods*, which are techniques and tools based on formal logic that aim to establish desired properties of programs by way of mathematical proofs.[3]

Formal methods can be applied to specifications, to source code, to compilers, and to machine models and machine code. The assurance case for a system depends on the precision and correctness of the specification and on the formal proof that the machine code correctly implements the specification.

Formal specification languages can be challenging for most developers, so usually specifications are given in natural language and source code is written to implement the specifications. With correctness in mind, developers realize that their choice of a programming language makes a big difference as to whether they can validate the specification and be sure the program meets the requirements. Control structures replacing gotos, object-oriented languages, and functional languages all help make programs understandable. Functional languages in particular make it easier to prove correctness with respect to a specification.

For compilers, formal grammars have helped improve parsing correctness, but hand-written optimizing compilers have remained the norm. Most verified applications depend on post-hoc verification where the machine is formally modeled and the program is proved correct according to the specification. Some *verified compilers* have had some success, such as CompCert [Ler09], for a subset of C without preprocessing.

An alternative approach to building a verified compiler is to build a *verifying compiler* [CGP$^+$97, NL98], i.e. one that generates, every time it is run, a proof of the semantic equivalence between its input and output. This is often an easier task than to prove that the compiler itself always generates an output that is semantically equivalent to its input.

While in principle the mathematical proofs in formal methods may be carried out with "pencil and paper", they can be more reliably and efficiently realized with tools, such as theorem provers, model checkers, SMT and SAT solvers, and static analyzers [KM, INR, Unia, Unib, FBK, SRIa, Belb, Vor, SRIb, Mic, CCF$^+$].[4] The formal methods community has used these tools not only to prove properties of programs, but also to formalize and prove properties of programming languages and their compilation.

In particular, for the purpose of this paper, we mention ACL2 [KM], an industrial-strength theorem prover that is used not only in academia but also in industry. ACL2 features a formal logic based on purely functional Common Lisp, powerful and automated proof procedures, and the ability to run formal specifications (when executable) with similar efficiency to mainstream programming languages. ACL2 has been used in a variety of application domains, ranging from hardware to software modeling and verification.

[3]These are proofs of mathematical theorems, different in nature from cryptographic zero-knowledge proofs. A major theme of this paper is the application of mathematical proofs to zero-knowledge proofs. Another interesting relation (though not explored in this paper), is the application of zero-knowledge proofs to mathematical proofs, i.e. "there exists a secret proof P for a non-secret theorem T".

[4]This is a very partial list of references.

3 Design of the LEO Programming Language

We begin by describing the design of LEO. After introducing the environment in which 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].

3.1 Environment

LEO programs execute in a ledger-based system where users can execute offline computations and produce transactions, attesting to the correctness of these computations. At its core, LEO produces *state transitions about zero-knowledge applications*. These state transitions are then encrypted into privacy-preserving transactions that are broadcast to an open network.

Our design achieves several security goals. Informally, LEO ensures *execution correctness*, meaning malicious parties cannot create valid state transitions without access to the private key and application state of a user. In addition, LEO offers *execution privacy*, which ensures transactions reveal only the information that is intended to be made public from the application. Lastly, our approach achieves *transaction non-malleability*, which protects a transaction from modification by malicious parties in transit to the ledger.

3.2 Syntax

We define the syntax of LEO with the context-free grammar in Figure 4. More precisely, this grammar defines the *abstract syntax*, which consists of *abstract syntax trees* (*ASTs*). Figure 5 presents two examples of ASTs. The *concrete syntax* used for writing LEO programs closely resembles this construct and is described with greater detail in the LEO developer documentation [Aleb] and by the ABNF grammar of LEO [Alea].

Our syntax is designed with the intent to support data types and operations that a typical developer would come to expect. To maintain an intuitive description of our design, we keep the distinction between abstract and concrete syntax to a minimum. For brevity, the lexical details of identifiers and other entities are omitted in Figure 4.

3.2.1 Types

LEO has *scalar* types and *aggregate* types: values of scalar types are atomic, while values of aggregate types contain other values.

The scalar types consist of a type of *booleans* and types of *signed and unsigned integers* of five sizes (8, 16, 32, 64, and 128 bits). In addition, they consist of a type of *addresses* (for accounts in the ledger), a type of *field elements* (i.e. values of the prime field), and a type of *group elements* (i.e. elliptic curve points).

The aggregate types consist of types of *tuples* (i.e. sequences of zero, two, or more values of possibly different types) and types of *arrays* (i.e. sequences of one or more values of the same type). Types of arrays may be nested (i.e. arrays of arrays) to construct multidimensional arrays. The aggregate types also include *circuit* types, which are referenced by names (or optionally by `Self` inside their own declarations) and whose values consist of unordered collections of values of possibly different types, which are accessed by name. A circuit type is similar to a class type in other programming languages.

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

Figure 4: The abstract syntax of LEO.

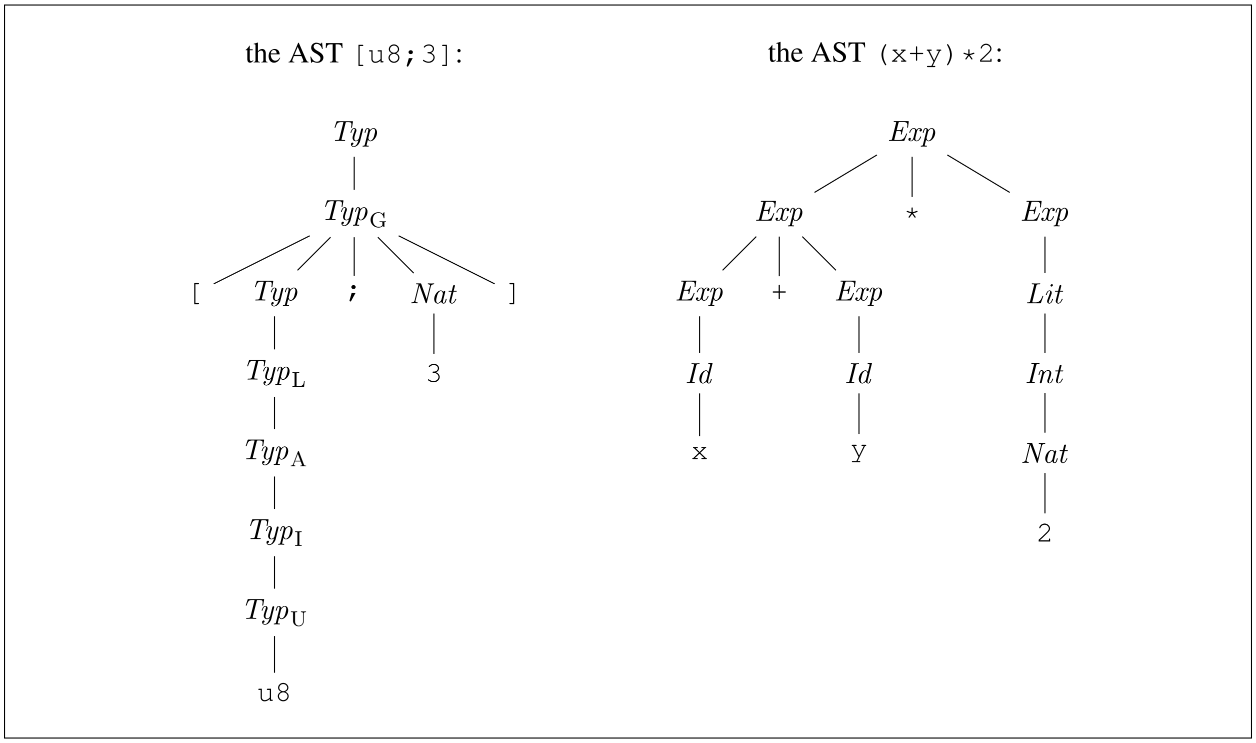


Figure 5: Examples of abstract syntax trees (ASTs).

3.2.2 Function and Circuit Declarations

A *circuit type declaration* consists of the name of the type along with one or more *member declarations*, which are of two kinds. A *member variable declaration* consists of a name and a type: it defines a named and typed component of the values of the circuit type. A *member function declaration* consists of a function declaration with an optional `self` parameter that receives a value of the circuit type. A circuit type resembles a class in an object-oriented language, where member variables resemble fields[5] and member functions resemble methods. However, LEO circuit types currently do not support inheritance. Furthermore, all member variables are associated to instances (i.e. values) of the circuit type; in object-oriented terminology, there are no static member variables.

A *function declaration* may be a member of a circuit type (see above) or at the top level, i.e. not part of any circuit type. Member and top-level functions have the same form, except that the latter cannot have `self` parameters. Function parameters with the *const* modifier are declared (and checked) to take only compile-time constant values (as described in Section 3.3); this also applies to the `self` parameters of member functions. The `mut` modifier for the `self` parameter indicates that the target circuit value can be modified by the function. When the function output type is the empty tuple type, it may be omitted.

While the grammar rule for member declarations in Figure 4 suggests that the `self` parameter starts the declaration of a member function, an equivalent rule for the concrete syntax may be stated as in Figure 6. That is, the `self` parameter, if present, occurs just before the other parameters. In Figure 4, we define this rule in the abstract syntax in a manner that reduces repetition and highlights shared structure.

[5]However we avoid this term in LEO so as to avoid confusion with prime fields.

$$\mathit{Fun} ::= \mathit{Ann}^{*}\ \texttt{function}\ \mathit{Id}\ \texttt{(}((\texttt{mut} \mid \texttt{const})^{?}\ \texttt{self})^{?}\texttt{,}(\texttt{const}^{?}\ \mathit{Id}\texttt{:}\,\mathit{Typ})^{*}\texttt{)}\ (\texttt{->}\ \mathit{Typ})^{?}\ \mathit{Blk}$$

Figure 6: An alternative abstract syntax rule for circuit member functions that is close to concrete syntax.

3.2.3 Expressions

Variables. *Expressions* include *variables*, which may be function parameters or defined locally in the function. For circuit member functions that include the `self` parameter, the special variable `self` refers to the instance of the circuit type (i.e. value of the circuit type) that the function is called on.

Literals. Expressions also include *literals*, which denote constant values of scalar types. An untyped literal is permitted when its type may be inferred (as described in Section 3.3). Group literals may be defined in one of two ways: (1) a group literal may be defined as a single integer, which represents the result of a scalar multiplication using the generator point of an elliptic curve by the integer, or (2) a group literal may be defined as two coordinates, which represent the elliptic curve point directly in affine form. In the latter case, one of the two coordinates may be omitted, in which case it is calculated as part of the static semantics (as described in Section 3.3), based on the specified sign criteria of the candidate point and the requirement that the resulting point must be on the curve.

Unary and binary operations. There are *unary and binary operations* for boolean connectives (negation, conjunction, disjunction), arithmetic operations (addition, subtraction, multiplication, division, power) on not only integers but also field and group values (subject to definitional restrictions), orderings on integers, and (non-)equality for all values.

Tuples. There is a *tuple constructor* that forms tuples from components, and a *tuple accessor* that accesses components in tuples via 0-based indices.

Arrays. There are two *array constructors*: one which directly lists the contents in order, either as elements or as spreads (subarrays) that are spliced in, and one which initializes all indices of the array with a specified value. There are two *array accessors*: one which extracts an element at an index, and one which extracts a subarray from a range of indices (which, if omitted, defaults to the start or end of the array).

Circuits. There is a *circuit constructor* that specifies an expression for each member variable, and a *circuit accessor* that extracts a member variable by its name.

Function calls. There are three kinds of *function calls*. A top-level function call consists of a name and its arguments. A member function without the `self` parameter is called by prepending the circuit type to the name, separated by two colons (`::`); this resembles a static method call in an object-oriented language. A member function with the `self` parameter is called by prepending an expression of the circuit type to the name, separated by a dot (`.`); this resembles an instance method call in an object-oriented language.

Conditionals. There is a *ternary conditional* expression, which consists of a test expression followed by a question mark (`?`) and two subexpressions that are separated by a colon (`:`).

3.2.4 Statements

Expression statements. *Statements* include *expression statements*, i.e. expressions followed by a semicolon (`;`). These expressions must return an empty tuple (i.e. “no value”) as LEO disallows discarding values in expression statements. These expressions are typically function calls that are used for side effects, such as incrementing a member variable in a circuit value.

Return statements. Values are returned from functions to their caller via *return statements*. When a function terminates execution without an explicit return statement, it is equivalent to an implicit return statement with an empty tuple.

Variable definition statements. Local variables are introduced via *variable definition statements*, each of which defines one or more variables. The `let` variant indicates that the variable can depend on program inputs and can be modified; the `const` variant indicates that the variable only depends on literals and other constant variables. The type(s) may be omitted, if they can be inferred (as described in Section 3.3).

Assignment statements. *Assignment statements* modify the values of variables, or components thereof. The left expression starts with a variable and may continue with a sequence of tuple accessors, array element and range accessors, or circuit accessors. Besides *simple assignments*, which merely assign the value of the right expression, there are *compound assignments*, which assigns the result of applying an operator to the values of the left and right expressions.

Conditional statements. *Conditional statements* consist of one or more branches, each composed of a test expression along with a block, terminating with an optional block that executes if all test expressions fail.

Loop statements. *Loop statements* consist of an integer variable that is automatically incremented from a specified start to a specified limit as the block is repeatedly executed.

Console statements. *Console statements* are defined for testing and debugging purposes. Namely, console statements are compiled to R1CS for constraint evaluation, however their execution is not supported in a zkSNARK proof system. These statements are used to test assertions and to print out messages during compiler execution. The messages are specified via Rust-like formatted strings.

3.2.5 Annotations

Circuit and function declarations may contain *annotations*. We omit their details in Figure 4 for brevity. LEO currently supports a limited set of annotations designated for test functions.

3.2.6 Files, Imports, and Packages

A *file* contains declarations of functions and circuit types, in any order. A file may contain *import declarations*, used to reference functions and circuit types from other files or libraries. A *package* is a bundled collection of LEO functions and circuit types.

3.3 Static semantics

The *static semantics* of LEO consists of compile-time requirements that must be satisfied in order for a LEO program to be compiled to R1CS. These requirements are best described (below) in terms of a sequence of phases that check and transform the ASTs obtained from parsing LEO code.

3.3.1 Canonicalization

The grammar offers a number of syntactic sugars to represent common language constructs in simpler terms:

- Inside a circuit type declaration, `Self` is equivalent to the name of the enclosing circuit.
- A multidimensional array type is equivalent to a suitable nest of monodimensional array types.

- A multidimensional array construction expression (the kind that fills all the elements with one value) is equivalent to a suitable nest of monodimensional array construction expressions.

- A compound assignment is equivalent to a simple assignment with the left expression copied on the right along with the operator. The equivalence holds because the left expression is checked to be free of side effects (see below).

- A function declaration without an explicit output type is equivalent to one with the empty tuple type as output type.

These equivalent forms provide convenience to the developer but complicate the processing of ASTs. For instance, to check whether two types are the same, it would be necessary to go beyond syntactic equality, e.g. to recognize that `Self` is the same as the enclosing circuit type.

Thus, the first phase of the LEO static semantics is *canonicalization*, which turns the aforementioned equivalent forms into canonical ones:

- Every occurrence of `Self` is replaced with the name of the enclosing circuit type. It is an error if `Self` occurs outside a circuit type.

- Multidimensional array types are turned into the corresponding nests of monodimensional array types.

- Multidimensional array construction expressions are turned into the corresponding nests of monodimensional array construction expressions.

- Compound assignments are turned into the corresponding simple assignments with the left expression and the operator copied in the right expression.

- Function declarations without output types are given the explicit empty tuple type.

The resulting ASTs are simpler to process. For instance, two canonical types are the same if and only if they are syntactically equal.

3.3.2 Type checking and type inference

Every LEO expression must have a compile-time type. The type of an immediate subexpression must have an appropriate type, and contribute to the determination of the type for its containing expression. For example, the binary operation + for addition can be applied only to two subexpressions with the same arithmetic type (such as an integer, field, or group) and the binary expression has the same type.[6] As another example, a member variable access operation may be applied only to an expression of a circuit type which contains that member variable, and the type of the member access expression is the type of that member variable.

As part of the determination of the types of expressions, it is natural to also check related requirements, e.g. that only variables in scope are referenced in expressions. Indeed, if the variable is not in scope, it is impossible to determine its type.

Since return statements yield values, the LEO static semantics also assigns types to statements, including an indicator that a statement returns nothing and simply lets execution continue past it, as is the case of an assignment statement, for instance. This indicator is similar to `void` in some languages, however LEO does not support such a construct in its syntax. Nonetheless, this discussion uses ‘type’ to include this indicator.

[6]The current type requirements for operators in LEO may be relaxed in the future, possibly with implicit widening conversions.

Since conditional statements may return different types in different branches, the static semantics assigns finite sets of types to statements, representing all possible outcomes.[7] The body of a function must return a singleton set containing the output type of the function.

As discussed in Section 3.2, the LEO syntax allows untyped literals and untyped variable definition statements. If none of those types are actually omitted, then *type checking* applies, which consists of a number of rules such as the ones exemplified above. Type checking ensures that the rules are satisfied, at the same time calculating types of expressions and statements.

If any literal or variable definition omits a type, then *type inference* applies: the omitted types must be inferred in such a way that all the type checking rules are satisfied. There are three possible situations:

1. There is no way to infer missing types that satisfies the rules: the program is type-incorrect, and is rejected.

2. There are two or more ways to infer missing types that satisfies the rules: the program is type-ambiguous, and is rejected.

3. There is exactly one way to infer missing types that satisfies the rules: the program is type-correct, and is accepted.

As mentioned in Section 3.2, variables and function parameters may be constant (i.e. `const`) or not. The type checking rules forbid assigning non-constant values to constant variables or to components of constant variables; they also forbid passing non-constant arguments to constant function parameters. Besides calculating types of expressions, the type checking rules also calculate whether an expression is constant or not, which is the case when it only depends on literals and on `const` variables.

As part of type checking and inference, integer literals are checked to be in the ranges of their types. Missing coordinates of group literals are also inferred during type checking and inference as alluded to in Section 3.2; group literals with explicit coordinates are also checked to be points of the elliptic curve.

The type checking and inference phase takes place just after the canonicalization phase. It transforms ASTs by inferring missing types, ensuring that the type checking rules are satisfied. The resulting ASTs have no missing types, and no missing group literal coordinates.

3.3.3 Optimizations

Since R1CS are flat structures, in order to translate a LEO program to R1CS, it is necessary to "flatten" programs through optimizing checks and transformations. These optimizations are part of the LEO static semantics because a LEO program is deemed illegal if they fail. These optimizations are orthogonal to executing LEO as a traditional programming language according to the dynamic semantics described in Section 3.4. Rather they are dictated by the intended purpose of LEO for representing zero-knowledge circuits in high-level constructs that resemble typical programming languages.

Constant folding and propagation. The LEO static semantics and compiler attempt to detect or rule out as many errors as possible statically. During constant folding, basic integer operations including addition, subtraction, multiplication, division, exponentiation, and negation are computed between constant values. The same principle is applied to operations for other datatypes that are determined to be constant including: booleans, fields, groups, arrays, and tuples. During constant propagation, constant values are substituted in later expressions which may be further optimized by an additional constant folding phase. The following

[7]This is not the case for conditional *expressions*, whose branches are required to have the same type.

invalid operations are detected during constant folding and propagation: integer division by zero, integer overflow, integer underflow, out of bounds array access, non-constant array sizes.

Loop unrolling. All iteration statement loops must be unrolled. This requires the loop start and limit expressions to be compile-time constants, which is established via constant folding and propagation.

Function inlining. All function calls must be inlined. This requires recursion to be statically bounded.

Constant folding and propagation, loop unrolling, and function call inlining constitute a static semantic phase that takes place after type checking and inference. If successful, the result is a significantly transformed AST that is amenable to translation to R1CS—through successive transformations.

3.4 Dynamic semantics

The *dynamic semantics* of LEO describes the execution of LEO in terms of an abstract machine with a computation state that is manipulated by expressions and statements. This is similar to other programming languages, independent of the zero-knowledge orientation of LEO: this is intentional so that developers may write "normal" programs in LEO, conceptualizing execution of their programs in familiar terms, while compiling the programs transparently to R1CS.

3.4.1 Computation states

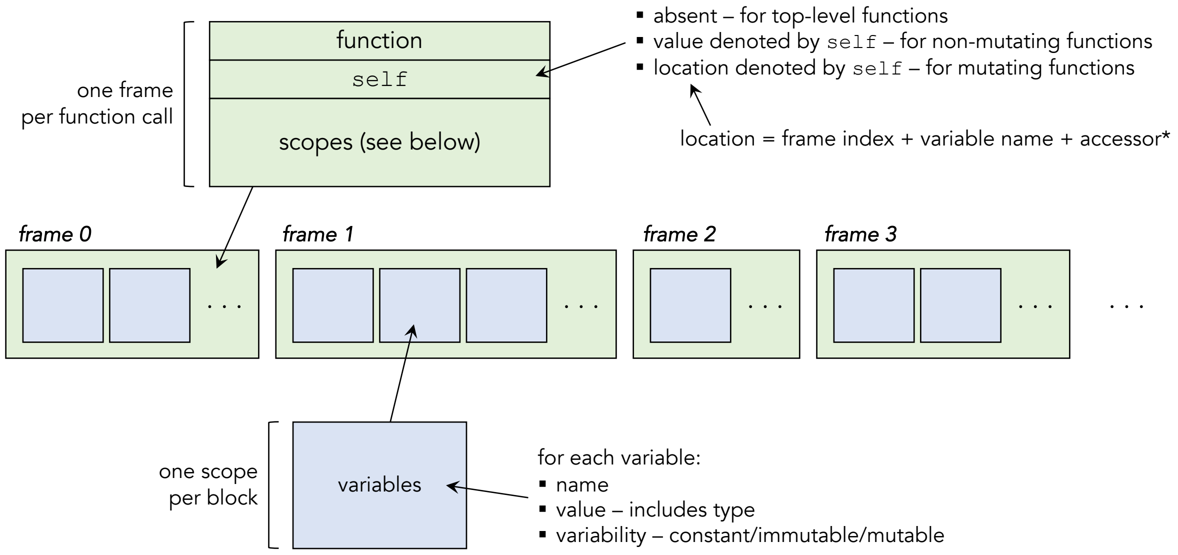

function
self
scopes (see below)
one frame per function call
▪ absent – for top-level functions
▪ value denoted by self – for non-mutating functions
▪ location denoted by self – for mutating functions
location = frame index + variable name + accessor*
frame 0
frame 1
frame 2
frame 3
one scope per block
variables
for each variable:
▪ name
▪ value – includes type
▪ variability – constant/immutable/mutable

Figure 7: Depiction of a generic LEO computation state.

A *computation state* is depicted in Figure 7, which shows the variables and their organization.

A computation state contains a sequence of *frames*, each corresponding to the invocation of a LEO function. The sequence of frames is a stack (last-in, first-out): a function call pushes a new frame onto the stack, and a function return pops the top frame from the stack.

Each frame contains a sequence of *scopes*, corresponding to the block structure of the statements in the function associated to the frame. The sequence of scopes is also a stack: entering a block pushes a new scope onto the stack, and exiting a block pops the top scope from the stack.

Each scope contains information about one or more variables, particularly the value stored in each variable—there are no uninitialized variables in LEO, by design. The bottom (i.e. outermost) scope in a frame is initialized with the function's arguments when the function is called. The execution of a variable definition statement extends the top (i.e. current) scope. Entering a block pushes an empty scope. Exiting a block pops the top scope, destroying the variables in that scope; since LEO has no pointers, there may never be stale pointers.

Each frame also includes optional information about `self`. There is no `self` information for top-level functions or for circuit member functions without `self` parameter. For circuit member functions with `self` parameter, the information is just the value when the `self` parameter has no `mut` modifier, otherwise it is a location, i.e. a form of pointer (see below for more detail).

3.4.2 Execution

Expressions and statements are *executed* stepwise, as in other programming languages.

Expressions are mostly side-effect-free. The only expressions that may have side effects are calls of circuit member functions with a `mut self` parameter. Any modification of `self` inside such functions, via local assignments or via further calls of functions with a `mut self` parameter on `self`, persists on the call's target after the function returns. The target is not copied when the function is called; rather it is passed to the function by reference (i.e. like a special pointer). However, this reference can only stay within `self` in the called function; it cannot be stored elsewhere, except for being passed to another `mut self` function. As shown in Figure 7, this reference consists of the index of the frame where the value lives (which may be passed to later frames via calls with `mut self`), the name of the variable in that frame (variables have unique names across the scopes of a frame, because LEO prohibits shadowing), and a sequence of zero or more accessors (e.g. array indices). In all other cases, function arguments are copied from caller to callee, i.e. they are passed by value.

Since expressions may have side effects, their order of evaluation matters. In LEO, expressions are evaluated from left to right. Conditional expressions are non-strict (as far as the dynamic semantics is concerned): if the test is true, the 'then' branch is evaluated and the 'else' branch is ignored; if the test is false, the 'else' branch is evaluated and the 'then' branch is ignored.

Statements are executed one after the other. Besides the side effects in the contained expressions, described above, statements have side effects of their own. Variable definitions create new variables, and assignments modify existing variables. Loop variables are automatically created when the loop starts and are automatically incremented at each iteration. Return statements immediately end the execution of a function, ending any loops in progress. Conditional statements are non-strict: only the block whose test holds is executed (or the `else` block if present and if no test holds), ignoring the other blocks. Console print statements have obvious input/output side effects; console assert statements stop execution if the assertions are not satisfied.

3.4.3 Entry points

A LEO program has a `main` function, which is the entry point of the application implemented by the program. The inputs to the `main` function are provided by the environment, and the outputs are returned to the environment. LEO is designed to support dynamic entry points, built to provide richer interactions with LEO applications.

Language decidability. We emphasize that LEO code is guaranteed to terminate, meaning the `main` function, and every other LEO entry point, is able to be semantically described as a mathematical function that maps its inputs to its corresponding outputs.

3.5 R1CS semantics

Even though LEO appears as an ordinary programming language, it is specifically designed to efficiently compile zero-knowledge applications written in LEO to R1CS. The fact that a LEO program is compiled to R1CS means that there is an R1CS instance associated with each LEO program. We regard this as the *R1CS semantics* of LEO. We emphasize that ordinary programming languages do not have R1CS semantics, limiting their ability to enforce *correct-by-construction* programs or achieve *verifiable computations*.

We abstract the details of our translation mechanisms from LEO code to R1CS. While there are many correct ways to translate a LEO program to R1CS, our design prioritizes circuit size and efficiency. As mentioned above, the LEO `main` function, according to the dynamic semantics, denotes a mathematical function

$$y = F(x)$$

where x is the input and y is the output. For simplicity, we consider a LEO function with a single input and a single output here, but this easily generalizes to multiple inputs and outputs (the latter returned as a tuple).

An R1CS relation is a set of constraints over a set of variables. In programming language terms, an R1CS relation is synthesized from a LEO function, and composed of *function input variables*, *function output variables*, and *auxiliary program variables*. For simplicity, we consider a relation with one input variable x, one output variable y, and one auxiliary variable a, which may easily be generalized to multiple variables of each kind. Logically, the R1CS constraints denote a mathematical relation

$$R(x, y, a)$$

over input, output, and auxiliary variables: the relation holds exactly on those values that satisfy all the constraints. By existentially quantifying over the auxiliary variable, we can define an input/output relation for the R1CS relation as follows:

$$P(x, y) \triangleq [\exists a.\ R(x, y, a)]$$

The correctness of the R1CS with respect to the LEO code is mathematically expressed as

$$\forall x, y.\ [y = F(x)] \Longleftrightarrow P(x, y)$$

This says that: (1) if LEO computes output y from input x, then the R1CS is satisfied by input x and output y (for some a); (2) if the R1CS is satisfied by input x and output y (for some a), then LEO computes output y from input x. That is, the R1CS represents all and only the LEO computations.

The above mathematical assertion relates LEO's dynamic semantics and R1CS semantics. The function F is defined by the dynamic semantics; the relation R is defined by the R1CS semantics.

4 Design of the LEO Compiler

We describe the design of the LEO compiler, which is based on the design of the LEO language in Section 3. Unlike ordinary compilers, the LEO compiler generates machine-checked formal proofs of the correctness of its operation. The proofs are checked by the ACL2 theorem prover (briefly discussed in Section 2.4), based on an ACL2 formalization of LEO. We note that some design features described in this section may not be fully implemented or optimized yet; see Section 5. After presenting the architecture of the compiler in Section 4.1, we discuss the generation of proofs in Section 4.2. We discuss the compiler phases in more detail in Section 4.3 to Section 4.9.

4.1 Architecture

We begin by describing the phases of the compiler. At a high level, the LEO compiler consists of a sequence of phases that translates LEO code text into an R1CS circuit. This sequence is depicted in Figure 8.

Each phase performs a transformation between artifacts of the same kind or of different kinds, and the transformation generally involves checks on the input that result in an error when not satisfied. Unsurprisingly, some of the phases of the compiler correspond to the phases that define the LEO static semantics (see Section 3.3). All of the phases are described in detail following our description of proof generation in Section 4.2.

4.2 Proof generation

Each phase of the LEO compiler not only carries out its transformations and checks, but also generates a machine-checkable proof of correct operation each time the phase runs. This is depicted in Figure 9 for a generic phase that transforms ASTs, but the same applies to phases that transform different kinds of artifacts.

Conceptually, the illustrated compiler phase is a mathematical function ϕ that transforms an AST a into an AST $\phi(a)$—or returns an error indication if some check on a fails. These are native ASTs, i.e. values of the native data types that represent ASTs.

As stated earlier, the proofs are based on an ACL2 formalization of LEO. This formalization includes, in particular, ACL2 data types that represent ASTs; these ACL2 ASTs are similar to the native ASTs mentioned above, but are different values in a different language. Thus, we use a converter from native ASTs to ACL2 ASTs. More precisely, the LEO compiler optionally generates JSON representations of native ASTs (and other artifacts between phases). These JSON representations are converted to ACL2 ASTs. Conceptually, the converter is a mathematical function α that converts a native AST a into an ACL2 AST $\alpha(a)$.

For each compiler phase, the ACL2 formalization of LEO includes a notion of correctness of the compiler phase. Mathematically, this is a binary relation $\mathcal{C}$ between ASTs (or other artifacts): $\mathcal{C}(a, a')$ holds exactly when a' is a correct output of the input a for that compiler phase. For instance, for the type checking and type inference compiler phase, the notion is the one stated in Section 3.3: a' is the same as a except that it may have additional types that are missing from a, and a' satisfies all the type checking rules. As this case exemplifies, there may be more than one correct a' for each a: this provides more flexibility in the implementation of the compiler phase.

When compiler phase ϕ runs on a yielding $\phi(a)$, the correctness of this run is expressed by the formula $\mathcal{C}(\alpha(a), \alpha(\phi(a)))$: the output of the phase is correct with respect to the input. The conversion α, described above, is needed to turn the JSON representation of the native ASTs into ACL2 ASTs. The compiler phase also generates an ACL2 proof of that formula, which is checked by the ACL2 theorem prover.

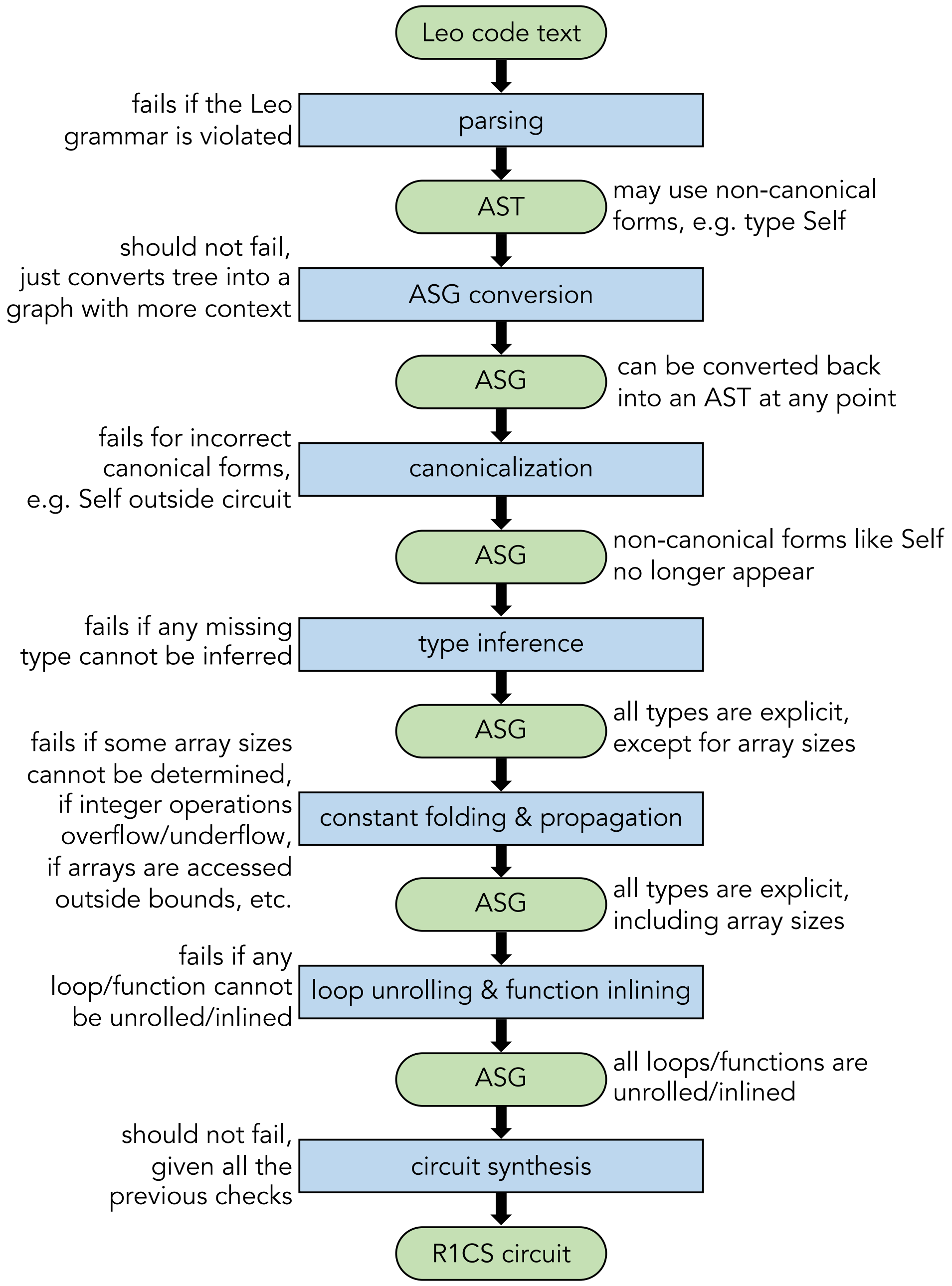
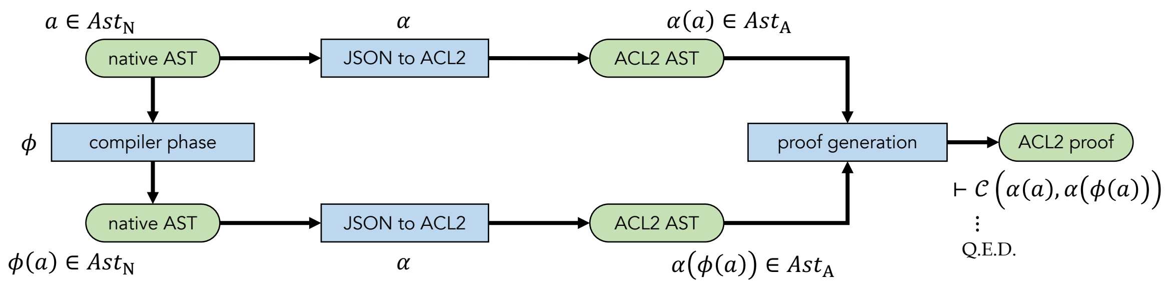

Leo code text
fails if the Leo grammar is violated
parsing
AST
may use non-canonical forms, e.g. type Self
should not fail, just converts tree into a graph with more context
ASG conversion
ASG
can be converted back into an AST at any point
fails for incorrect canonical forms, e.g. Self outside circuit
canonicalization
ASG
non-canonical forms like Self no longer appear
fails if any missing type cannot be inferred
type inference
ASG
all types are explicit, except for array sizes
fails if some array sizes cannot be determined, if integer operations overflow/underflow, if arrays are accessed outside bounds, etc.
constant folding & propagation
ASG
all types are explicit, including array sizes
fails if any loop/function cannot be unrolled/inlined
loop unrolling & function inlining
ASG
all loops/functions are unrolled/inlined
should not fail, given all the previous checks
circuit synthesis
R1CS circuit

Figure 8: LEO compiler phases. Blue boxes are phases. Green boxes are artifacts.



Ast_N	set of native ASTs of Leo
Ast_A	set of ACL2 ASTs of Leo
$\phi : Ast_N \to Ast_N$	compiler phase (in Rust) – a function from ASTs to ASTs
$\alpha : Ast_N \to Ast_A$	converter from JSON (native) ASTs to ACL2 ASTs
$\mathcal{C} \subseteq Ast_A \times Ast_A$	formalized correctness of compiler phase – an I/O relation between ASTs

Figure 9: Proof generation for a compiler phase.

By generating a proof for each phase, we can compose the proofs and correctness formulas to obtain an end-to-end proof of the correctness of the whole compilation process. The end-to-end correctness theorem says that the LEO program text is correctly compiled into R1CS. This encompasses all the compiler phases, from parsing all the way to circuit synthesis. The notion of correctness (end-to-end and by phase) is defined in terms of the ACL2 formalization of the LEO syntax and semantics, including static, dynamic, and R1CS semantics.

This is a *verifying compiler* approach, as opposed to a *verified compiler* approach. Every time the compiler runs, we formally verify that it produces the correct output for the input. This formal verification is designed to be simple, at the level of proof checking (and not of proof search), so the compiler generates sufficiently detailed information for the proof to be efficiently checked by the ACL2 theorem prover.

4.3 Parsing

The LEO parser is based on the ABNF grammar of LEO [Alea]. This grammar is also part of the ACL2 formalization of LEO: it is used to formally define the concrete syntax of LEO. ABNF (Augmented Backus-Naur Form) [CO08, Kyz14] is an Internet standard, which we have previously formalized in ACL2; we have also previously developed a formally verified ABNF grammar parser [Cog18], which we use to parse the ABNF grammar of LEO into a formal representation in ACL2 of the concrete syntax of LEO. This forms the basis for supporting the generation and checking of proofs of correct parsing for this phase.

The LEO parser consists of a lexer which converts LEO code text into tokens, and a parser which converts tokens into data structures that represent expressions, statements, and functions which combine to form an AST. Code text cannot be ambiguous; if a file parses, it has exactly one valid AST. The benefit of this approach is that lexical and syntactical errors are caught as early as possible when parsing. The parser also identifies deprecated syntax and recommends the new approach to developers.

4.4 AST conversion

The parsing phase outputs an abstract syntax tree (AST) which strips unnecessary nodes from the tree such as semicolons, comments, and other non-executed parts of code. A framework for error handling is established at this phase, introducing spans and formatted logging. The framework is consistent across future compiler phases which ensures that semantical and logical errors are caught and propagated cleanly. The AST conversion phase does not fail as the goal is merely to reorganize the tree—not rewrite it. The native AST may be serialized directly into a JSON format that can be converted to an ACL2 AST as explained in Section 4.2.

Many of the formal proofs for the LEO compiler phases are based on ASTs. To calculate the envisioned end-to-end proof described in Section 4.2 the compiler must perform a conversion from AST to JSON AST to ACL2 AST at each intermediate phase. We remark that the final proof only involves the LEO code text and R1CS circuit, while the intermediate artifacts are auxiliary. Errors encountered during parsing between AST formats will result in an invalid proof and a failed compilation.

4.5 ASG conversion

To efficiently perform optimization passes on programs in LEO, we capture the semantics of program nodes in an abstract semantic graph (ASG). The ASG is at a higher level of abstraction than the AST and serves as an intermediate representation that is conducive for further optimizations and transformations. Converting to an ASG from an AST identifies scoping failures such as unknown identifiers, unknown types, and immutable variable mutation. Each graph node gains additional context such as parent functions, parent circuits, and expected types. Every ASG may then be converted back into an AST for proof generation between each phase.

ASG structures can express more complexity than ASTs. This additional complexity is not necessary for proof generation. In addition, the simple, non-cyclic tree structure of ASTs can be more easily expressed in ACL2. Since ASGs may be converted back to AST, the ACL2 proofs are expressed on ASTs. We remark that, for the purpose of the end-to-end proof from LEO program text to R1CS, it is not necessary to explicitly verify the correctness of the conversions between ASGs and ASTs. The reason is that the end-to-end proof involves only the LEO program text and the R1CS, while the intermediate artifacts are auxiliary. Errors encountered during AST to ASG conversion or ASG to AST conversion will result in an invalid proof and a failed compilation.

4.6 Canonicalization

This compiler phase performs the canonicalization transformations described in Section 3.3.1. This phase is fairly simple, and helps simplify subsequent phases. For instance, as explained in Section 3.3.1, the fact that all the types are canonical after this phase lets the type checking and inference phase use syntactic equality to check whether two types are the same.

The ACL2 formalization of LEO includes a definition of the canonicalization relation. The proof generated for this compiler phase says that the canonicalization performed by the compiler is the same one prescribed by the ACL2 formalization.

4.7 Type inference

This compiler phase performs the type checking and inference activities described in Section 3.3.2.

Each variable and literal in a program must have an explicit type before translating to R1CS. The correct types for implicitly-typed variables and literals are inferred from contextual type information associated with function arguments, function return types, circuit members, and other variable names. The compiler identifies type checking failures as described in Section 3.3.2 and recommends adding explicit type definitions at the exact locations that would satisfy the type checking rules.

The ACL2 formalization of LEO includes a definition of the type checking rules, of the notion of adding types to ASTs, and of the notion of type inference solutions. The proof generated for this compiler phase says that the output AST is a type inference solution of the input AST.

4.8 Optimizations

These compiler phases perform the constant folding and propagation, loop unrolling, and function inlining optimizations described in Section 3.3.3.

Any constant evaluation that can be performed before translating to R1CS will create lighter programs with fewer constraints. In addition, evaluating constant expressions may reveal illegal code which would otherwise cause some R1CS circuits to always fail. Each optimization phase takes as input an ASG and outputs an ASG - both of which can be converted to an AST. It is common for optimizing passes to be run several times during compilation. For example, a variable definition statement may have its expression evaluated during constant folding and then its variable substituted during constant propagation. An additional constant folding pass can further optimize the substituted statement which may open up more potential for another constant propagation pass. Therefore, it is possible to execute a run of the compiler that performs optimizing passes repeatedly. Since each pass of the compiler is verified individually by generating a proof between the input and output ASTs, a chain of proofs can be generated for the entire compilation.

The proof generated for this compiler phase says that the output AST is functionally equivalent to the input AST (according to the LEO dynamic semantics formalized in ACL2), and that the output AST has no loops (because they have been unrolled), no function calls (because they have been inlined), and all known array sizes (because of constant propagation and folding).

4.9 Circuit synthesis

The final phase of the compiler translates an ASG into an R1CS relation. Our approach takes advantage of the notion of handcrafted circuits (as described in Section 1.2) to compose efficient R1CS relations. LEO makes use of a library of R1CS gadgets that define a fixed number of R1CS constraints for every syntax primitive in the language. An instance of a LEO program in ASG form expresses syntax primitives in a format that may be directly translated into R1CS relations by substituting one or more R1CS gadgets.

One may easily evaluate the execution cost of different programs by outputting the number of constraints generated by circuit synthesis. LEO provides a testing framework that enables developers to synthesize individual program components and compare the cost of different approaches without having to handwrite separate circuits for each program. This modular design abstracts the concept of low level constraint writing in a composable way.

Those who wish to write their own circuits may do so by using a circuit declaration in a LEO program. For example, LEO supports a `u8` type that maps to an unsigned integer gadget which stores eight boolean gadgets. If one wishes to construct a `u4` unsigned integer type, they may define a new `u4` circuit with four boolean type circuit members. The circuit synthesis phase will generate R1CS constraints for the new type that are as efficient as the native implementation.

By using a synthesizer for *preprocessing arguments with a universal and updatable SRS*, we can forgo the sampling of an SRS that depends on individual circuits. Instead the SRS is *universal*, meaning the sampled SRS is able to support any circuit up to a given size bound by enabling any party, in an offline phase *after* the SRS is sampled, to publicly derive a *circuit-specific SRS*.

The proof generated for this compiler phase says that the generated R1CS constraints are a valid representation of the LEO program according to the formal notion described in Section 3.5. Besides the ACL2 formalization of LEO already discussed, we have an ACL2 formalization of R1CS, that this proof relies on.

4.10 Verifying R1CS gadgets in LEO CORE

To preserve the zero-knowledge feature of an application, it is important to apply sufficient computation to private inputs so that they cannot be discovered from the public inputs and outputs. For the LEO programmer's convenience, we provide a native implementation of the Blake2s cryptographic hash [SA15] that is callable from LEO programs. Additional native methods are planned.

To preserve the assurance case for LEO programs that include Blake2s and other native methods, we have adapted formal methods to R1CS to be able to formally verify R1CS relations that are not created by compiling LEO programs. Our approach for doing so is based on the ACL2 theorem prover [KM] and Kestrel Institute's Axe [Smi] toolkit, which is implemented as an ACL2 program. We use our formalization of the semantics of R1CS relations in ACL2. This formalization defines the syntax of an R1CS as an ACL2 object and defines the semantics of such objects by characterizing what it means for an R1CS to hold over a valuation (an assignment of values to its variables). The formalization is based on our ACL2 formalization of prime fields, and both are available in the open-source community libraries of ACL2.

The verification process takes as input an R1CS in our format (perhaps translated from other format, such as a JSON representation). We then lift the R1CS into an equivalent logical term. For this, we use Axe's R1CS Lifter. The Lifter first forms a mathematical term asserting that the R1CS holds over arbitrary symbolic variables. It then applies the Axe Rewriter to transform that term into an equivalent expression over the prime field operators. To do this, the Axe Rewriter opens up the functions that define the R1CS semantics, producing a logical term equivalent to the R1CS but no longer depending on the R1CS semantics functions. The resulting term is essentially the conjunction of the constraints in the R1CS, expressed in terms of the underlying prime field operations, but some simplifications may also have been applied.

To verify an R1CS, we need a specification to compare it to. For this, we form an ACL2 term expressing the relationship that should hold over the inputs and outputs of the R1CS (but not its intermediate variables). When possible, we use our library of existing bit-vector and cryptographic functions. Typically, the specification says that the output variables of the R1CS are equal to some expression over the input variables. We then call the Axe R1CS Prover to attempt to prove that, whenever the R1CS holds (over all of its input, output, and intermediate variables), the specification relation also holds (over the inputs and outputs). We call this the "forward direction" of the proof—this corresponds to the 'if' implication in the equivalence formula in Section 3.5, if F denotes the specification (e.g. of Blake2s) instead of the semantics of a Leo program. In the future, we plan to also consider the backward direction (the claim that, if the specification holds over the inputs and outputs, then there exist values for the intermediate variables such that the entire R1CS holds)—this corresponds to the 'only if' implication in the equivalence formula in Section 3.5, if F denotes the specification (e.g. of Blake2s) instead of the semantics of a Leo program. We always assume that all values are field elements, and, if necessary, additional assumptions on the input variables can be passed to the Prover (such as claims that some of them are single bits).

The Axe Prover supports two main proof tactics, rewriting and variable substitution. Rewriting is used to solve the constraints, attempting to turn each one into an equality of a variable and some other term not involving that variable. This is reasonable, as the variables in an R1CS can often be ordered, with each constraint defining one or more new variables in terms of previously defined variables. After solving for a variable, substitution can be used to replace it, everywhere in the proof, with the term that it is known to be equal to. The rewriting process applies rewrite rules from our large and growing library of verified rules. Crucially, all of these rules are proved as theorems in the ACL2 system, so additional rules can be added without compromising the soundness of the proof system. Our library contains many rules that recognize R1CS gadgets, covering a variety of common operations such as XORing, bit packing, bit rotations, etc. Often several rounds of rewriting and substitution are necessary, as the Prover turns the information in the R1CS into more usable forms. For example, an XOR constraint cannot be solved until its input values are known to be single bits, but this may not be known until the constraints that define those values are themselves solved. The Axe Prover typically applies a sequence sets of rules, exhaustively applying rewriting and substitution with one rule set before moving on to the next one. This allows us to do the proof in stages, each of which handles one kind of operation (e.g., recognizing and transforming bit packing gadgets, or moving negated addends to the other sides of equalities). Once every intermediate variable has been eliminated by substitution, we are usually left with a single term for each output of the R1CS, representing it as a function of (a subset of) the input variables. Often these terms no longer contain any prime field operations, because rewrite rules have recognized all of the R1CS gadgets and replaced them with more traditional bit-vector operations (concatenation, addition, rotation, etc.). The final step of the proof is usually to allow the Prover to expand the functions in the specification, unrolling all of the (bounded) recursion to expose simple bit-vector operations. If all goes well, each of the output bits of the specification will have an expression that exactly matches the expression derived from the R1CS (perhaps after some normalization, also done by rewriting). This completes the proof.

5 Implementation

In Section 5.1, we describe our current implementation of the LEO compiler. In Section 5.2, we define the cryptographic building blocks that are used to evaluate our construction. In Section 5.3, we describe our formal definition of LEO and the proofs about the formal definition itself. In Section 5.4, we describe the developer tools that complement the LEO compiler.

5.1 Compiler implementation

We have implemented the LEO compiler in a Rust library [8]. The LEO compiler parses LEO programs, checks the static semantic requirements, and compiles them to R1CS. We have implemented the building blocks of formal verification for a subset of the LEO compiler phases in ACL2[9], and we are working on expanding it to the remaining compiler phases. All of the LEO compiler is under active development.

5.2 Cryptographic instantiations

Prime	Value	Size in bits	2-adicity
r	`0x12ab655e9a2ca55660b44d1e5c37b00159aa76fed0000001 0a11800000000001`	253	47
p	`0x1ae3a4617c510eac63b05c06ca1493b1a22d9f300f5138f1 ef3622fba094800170b5d44300000008508c00000000001`	377	46

Figure 10: The E_{BLS} elliptic curve (commonly known as BLS_{12}-377).

Elliptic curves. Our construction is able to support a variety of elliptic curves. For our implementation strategy, we selected a curve instantiation E_{BLS} from the Barreto-Lynn-Scott (BLS) family with embedding degree 12. Our choice conservatively achieves 128 bits of security. Both $\mathbb{F}_r$ and $\mathbb{F}_p$ have multiplicative subgroups of order 2^α for $\alpha \geq 40$ (both the scalar field and base field have high 2-adicity). This allows our choice of zkSNARK proof system to support R1CS instance sizes that are in the billions.

zkSNARKs. Our construction is able to support a variety of zkSNARK proof systems. Namely, so long as the zkSNARK language supports *R1CS relations*, we are able to execute proof generation for a compiled LEO program. Concretely, we rely on the MARLIN proof system, which is a preprocessing zkSNARK with a universal and updatable SRS [CHM+19], used over the elliptic curve E_{BLS}.

5.3 Formal definition

The formal proof generation described in Section 4.2 relies on a formalization of the LEO syntax and semantics, which we are developing in the ACL2 theorem prover. This formalization mirrors the LEO Language Formal Specification [Cog]; the latter is written in a generic mathematical notation that should be accessible to readers unfamiliar with ACL2.

The ACL2 formalization of LEO currently covers:

- The ABNF grammar [Alea] (see Section 4.3).

[8]https://github.com/AleoHQ/leo
[9]https://github.com/AleoHQ/leo-semantics

- The abstract syntax (see Section 3.2).
- The canonicalization phases and type checking and inference phase of the static semantics (see Section 3.3).
- The dynamic semantics (see Section 3.4).

The ACL2 formalization is also being extended to cover:

- The constant propagation and folding, loop unrolling, and function inlining phase of the static semantics (see Section 3.3).
- The R1CS semantics (see Section 3.5).
- The extra-grammatical requirements that, together with the ABNF grammar, define the concrete syntax (see Section 3.2).

When completed, this provides a complete formal specification of LEO. Adding the remaining parts listed above poses no technical problems.

The ACL2 formalization of LEO includes not only the definition of the syntax and semantics of LEO. It also includes theorems, verified via the ACL2 theorem prover, about these definitions, which serve to validate the definitions. These theorems express expected properties of the definitions; failure to prove one of these theorems would uncover an inadequacy in the definition. These theorems are distinct from, and in a way complementary to, the ones generated by the LEO compiler (see Section 4.2).

A simple example of these theorems is the idempotence of the canonicalization transformation: applying the transformation twice is equivalent to applying it once. We are adding more of these theorems to the formalization. A more complex example of these theorems, which we plan to add, is type soundness, namely the fact that, if a program satisfies all the type checking rules (after type inference), then its execution will never result in certain kinds of run-time errors. This is a fundamental property, that is often proved in formalizations of programming language: it provides a major validation of the design of the language, and of the correct relation between static and dynamic semantics.

5.4 Developer tooling

LEO is designed with developer usability in mind. We give an overview of the testing framework, import resolver, and package registry unique to LEO that enable rapid development of zero-knowledge applications.

Testing framework. The LEO testing framework consists of console statements, local unit test functions, and custom input integration test functions. All of the tools in the LEO testing framework can be used without *specialized domain expertise*. We designed a formatted string syntax that can be used anywhere in a LEO program to print the value of an expression. The console logging and console debug statements print formatted strings and do not generate constraints; they are specifically provided to the developer for constant value debugging. LEO supports a `@test()` annotation that allows for fine-grained unit and integration testing of programs at any entry point. Any custom identifier, function, or circuit can be instantiated in a testing context and checked for correctness using constant values. Correctness checks can be enforced via console assert statements which will enforce an equality constraint in a testing context. The `@test()` annotation can be invoked with a file input argument to test functionality across multiple environments. Each environment synthesizes the circuit on a clean constraint system and relays any compilation or constraint satisfaction errors back to the developer.

Import resolver. Import declarations in LEO (as described in Fig. 4) support local file parsing, module access, and definition aliasing. The import resolver runs during the ASG conversion compiler phase and stores imported identifiers, circuits and functions for lookup in during program execution. Resolving imports in the ASG prevents code duplication, since files that import the same dependency can reference the same node in the graph—saving R1CS constraints.

Package registry. LEO integrates directly with the first known registry for zero-knowledge circuits. Successfully compiled projects can be packaged and published directly to the registry. Similarly, any public package may be cloned or added to a current project and imported into a file. The package registry supports publishing as an organization, a user, or a user within an organization. This promotes circuit ownership and gives developers more confidence for packages published by reputable organizations and users. Using registered packages also prevents multiple instantiations of the same circuits on a given ledger—decreasing transaction circuit bloat.

6 Evaluation

In Section 6.1, we begin by defining our methodology for evaluating the LEO compiler. In Section 6.2, we define the set of LEO programs that we use to benchmark the performance of our implementation. In Section 6.3, we evaluate the running time for each phase of our compiler separately. In Section 6.4, we evaluate the performance for formally verifying a complex R1CS gadget in LEO CORE.

6.1 Methodology

We evaluate the LEO compiler on a set of programs, measuring the running time of the major phases of our compiler. The complete set of phases is outlined in Section 4.1, and for simplicity, we choose to prioritize our measurements on the time from (1) LEO code to AST, (2) AST to ASG, (3) ASG to R1CS, and (4) LEO code to R1CS. Note that (4) is the sum of (1)–(3). All reported measurements are taken on a machine with an AMD Ryzen Threadripper 3960x at 3.8 GHz with 64 GB of RAM.

6.2 Programs for evaluation

In our benchmarking, we implemented and evaluated three distinct LEO programs of varying constraint sizes: a Pedersen hash (Section 6.2.1), bubble sort (Section 6.2.2), and least-squares linear regression (Section 6.2.3).

6.2.1 Pedersen hash

We implement a simple 256-bit Pedersen hash circuit by sampling a random vector g of 256 group elements as public parameters, and for a given message x of 256 bits, our circuit evaluates $g_0^{x_0} g_1^{x_1} \dots g_{255}^{x_{255}}$. Figure 11 shows the code used to instantiate the Pedersen hash circuit.

```
circuit PedersenHash {
    parameters: [group; 256],

    // Instantiates a Pedersen hash circuit.
    function new(const parameters: [group; 256]) -> Self {
        return Self { parameters };
    }

    // Returns the Pedersen hash for given message.
    function hash(self, message: [bool; 256]) -> group {
        let digest: group = 0;
        for i in 0..256 {
            if message[i] {
                digest += self.parameters[i];
            }
        }
        return digest;
    }
}
```

Figure 11: A simple 256-bit Pedersen hash circuit.

The PedersenHash circuit has two member functions: new and hash. To instantiate a new instance of PedersenHash, we invoke the new method, which takes as input a *constant* array of 256 group elements as public parameters. To hash a message, we invoke the hash method, which takes as input an instance of the circuit along with a message of 256 bits, and returns a single group element as output.

6.2.2 Bubble sort

We implement the bubble sorting algorithm on an array of u32 elements. Bubble sort is a standard algorithm that works by examining each set of adjacent elements, from left to right, and swapping positions if they are out of order. This process is repeated until there are no more swaps to be made, resulting in a sorted array. Figure 12 shows the code used to instantiate the bubble_sort function.

```
// Executes the bubble sorting algorithm.
function bubble_sort(arr: [u32; 10]) -> [u32; 10] {
    // Traverse the entire array
    for i in 0..9 {
        for j in 0..9-i {
            // Move the smaller elements forward
            if arr[j+1] < arr[j] {
            // Swap the elemets at indexes `j` and `j+1`
            let swap = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = swap;
            }
        }
    }
    return arr;
}
```

Figure 12: A bubble sorting algorithm function.

The bubble_sort function takes as input a *mutable* array of ten u32 elements, and returns a sorted array of the same size. The function uses a double for-loop to perform the compare and swap operations. For convenience, we provide in Fig. 13 an example test function for checking the correctness of the bubble_sort function, which when ran, outputs the results to the console, as shown in Fig. 14.

6.2.3 Least-squares linear regression

We implement the least-squares linear regression algorithm. Linear regression is a popular machine learning algorithm that models the relationship between two variables as linear. In a simple linear regression, the linear regression line has an equation of the form $Y = a + b * X$ where X is an explanatory variable and Y is a dependent variable. To calculate a and b, the following formula is used:

$$a = \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2}, \; b = \frac{1}{n}\left(\sum_{i=1}^{n} y_i - a\sum_{i=1}^{n} x_i\right)$$

While more efficient implementations of least-squares linear regression can easily be achieved, we show a simple implementation for ease of readability. For convenience, we define a Point circuit to abstract the

```
@test
function test_bubble_sort() {
    let unsorted: [u32; 10]
        = [8u32, 2u32, 4u32, 3u32, 5u32, 10u32, 7u32, 1u32, 9u32, 6u32];
    let expected: [u32; 10]
        = [1u32, 2u32, 3u32, 4u32, 5u32, 6u32, 7u32, 8u32, 9u32, 10u32];

    let result = bubble_sort(unsorted);
    console.log("Result is: {}", result);
    console.assert(result == expected);
}
```

Figure 13: An example test function for the bubble sorting algorithm.

```
  Test Running 1 tests
  Test Result is: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
  Test bubble-sort::test_bubble_sort ... ok

  Test Tests passed in 8 milliseconds. 1 passed; 0 failed;
  Done Finished in 10 milliseconds
```

Figure 14: A sample output of the example test function for the bubble sorting algorithm.

notion of a two-dimensional point, as shown in Figure 15. Figure 16 shows the code used to instantiate the least-squares linear regression circuit.

```
circuit Point {
    x: i32,
    y: i32,

    function new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
}
```

Figure 15: A two-dimensional point circuit.

```
circuit LinearRegression {
    points: [Point; 5],

    // Instantiates a linear regression circuit.
    function new(points: [Point; 5]) -> Self {
        return Self { points };
    }

    // Return the slope of the linear regression.
    function slope(self) -> i32 {
        let num_points = 5i32;
        // Calculate the sums.
        let x_sum = 0i32;
        let y_sum = 0i32;
        let xy_sum = 0i32;
        let x2_sum = 0i32;
        for i in 0..5 {
            x_sum += self.points[i].x;
            y_sum += self.points[i].y;
            xy_sum += self.points[i].x * self.points[i].y;
            x2_sum += self.points[i].x * self.points[i].x;
        }
        let numerator = (num_points * xy_sum) - (x_sum * y_sum);
        let denominator = (num_points * x2_sum) - (x_sum * x_sum);
        let slope = numerator / denominator;
        return slope;
    }

    // Return the offset of the linear regression.
    function offset(self, slope: i32) -> i32 {
        let num_points = 5i32;
        // Calculate the sum.
        let x_sum = 0i32;
        let y_sum = 0i32;
        for i in 0..5 {
            x_sum += self.points[i].x;
            y_sum += self.points[i].y;
        }
        return (y_sum - slope * x_sum) / num_points;
    }
}
```

Figure 16: A least-squares linear regression circuit.

6.3 Evaluation of the compiler phases

	PEDERSEN HASH	BUBBLE SORT	LEAST-SQUARES LINEAR REGRESSION
Time from code to AST	0.314 ms	0.398 ms	6.163 ms
Time from AST to ASG	0.084 ms	0.079 ms	0.245 ms
Time from ASG to R1CS	24.882 ms	70.393 ms	1651.012 ms
Total compile time	25.28 ms	70.871 ms	1657.42 ms
Number of constraints	1539	16115	433867

Figure 17: Running time for parsing each compile phase from LEO source code to an R1CS instance.

We separately evaluate the performance of the primary phases of our LEO implementation:

1. **LEO code to abstract syntax tree (AST).** We begin by evaluating the running time for parsing a common set of LEO programs from their source code to their AST representations.
2. **Abstract syntax tree (AST) to abstract semantic graph (ASG).** We then evaluate the running time for converting the same set of Leo programs from an AST representation through their ASG compiler passes.
3. **Abstract semantic graph (ASG) to rank-1 constraint system (R1CS)**. We then evaluate the running time for translating the same set of Leo programs from an ASG representation through circuit synthesis to its R1CS representation.

We evaluated our compiler and established that most compiler passes are quick to execute, and that the primary time spent during compilation was in circuit synthesis.

6.4 Formal verification of Blake2s from LEO CORE

We have applied the Axe Lifter and the Axe Prover for R1CS, described above in section 4.10, to verify a handcrafted R1CS gadget that represents the Blake2s hash function (the full version and also a smaller version limited to a single round). The one-round version is composed of 2,864 variables and 2,896 constraints. We establish that lifting it into logic takes under one second, and the proof (comparing it with an ACL2 specification of one-round of Blake2s) takes under a minute. The full version of Blake2s has 21,728 variables and 22,048 constraints. The proof (comparing it with our full specification of Blake2s) currently takes about 3,100 seconds. Future prover improvements may reduce that time. We are also working to craft robust sets of rewrite rules that suffice to automatically verify not just Blake2s but also a variety of other R1CS relations. For the Blake2s examples, we currently prove only the “forward” direction, that the R1CS implies the specification relation over inputs and outputs. Consideration of the backward direction is future work. Note that cryptographic functions such as Blake2s by design take inputs large enough that they could never be exhaustively tested. A proof provides assurance that the R1CS is correct for all possible combinations of input values.

7 Limitations

While we are excited about formally verified, zero-knowledge applications, our work shows us that both zkSNARKs and theorem provers continue to suffer from fundamental limitations.

First, making the zkSNARK prover more efficient overall remains a challenging open problem. Prior work on scaling zero-knowledge proof systems [WZC+18] enables using zkSNARKs on circuits far larger than what was previously thought possible; however, resorting to using a compute cluster is still very expensive. We see several promising avenues towards enhancing prover efficiency by way of new research as well as explorations into the use of GPUs and FPGAs for speeding up subroutines of the zkSNARK prover, such as for multi-scalar multiplications (MSMs).

Second, industrial-strength theorem provers such as ACL2 have made notable progress in their applicability for real-world software, yet there is room to improve on the scalability and automation of theorem provers. We see a need for faster execution times, and there has been promising work to enable parallel proving, such as with waterfall parallelism [Rag11, RJK13].

Our overall outlook on the areas of efficient proof systems and performant theorem provers is optimistic. Progress in these areas offers LEO tremendous opportunities to scale formally verified, zero-knowledge applications beyond its current capabilities, further expanding the applicability of *verifying compilers*.

8 Future Work

Our work on LEO is just beginning, and there are a number of exciting directions to improve both the LEO language and compiler. Beyond new language features and expanding LEO CORE, we look to expand LEO in the following directions:

Optimizing R1CS. LEO implements IR-level optimizations and could be expanded on the R1CS-level through "intermediary arithmetization" to minimize the number of constraints generated by the circuit synthesizer. In addition, recent work into optimizing R1CS using SMT solvers [OBW20] has shown promising capabilities, such as in the context of loop unrolling.

Symbolic reasoning. It could be promising to further improve the safety of compiled LEO programs through symbolic reasoning to detect and/or rule out semantic errors. Common issues such as overflows and out-of-bound array indexing could then be identified and resolved prior to runtime.

Program registry. As LEO matures, a growing need for an improved (de)serialization is required to encompass the state (or records) of the applications it executes on. Prior work into R1CS serialization formats [Qi19] have shown promising growth. However our purposes necessitate far more state than merely an R1CS instance, such as robust versioning, encoding of universal SRS targets, encapsulation of state transitions in cryptographic records, and checkpointing of program registers to name a few.

Acknowledgements

The authors are grateful to Matthew Green, Akis Kattis, Ian Miers, and Pratyush Mishra for participating in early stages of this work. The authors are thankful to Pranav Gaddamadugu, Kobi Gurkan, Ali Mousa, Konstantin Pandl, and Alex Pruden for their input in the design of the programming language and compiler.

References

[Alea] Aleo Team. ABNF grammar of Leo. `https://github.com/AleoHQ/leo/blob/master/grammar/abnf-grammar.md`.

[Aleb] Aleo Team. Aleo developer documentation. `https://developer.aleo.org`.

[ark] arkworks. `https://github.com/arkworks-rs/snark`.

[Bas20] Baseline. Potential security bug with the zk-snark verifier. `https://github.com/ethereum-oasis/baseline/issues/34`, 2020.

[BBC$^+$17] Eli Ben-Sasson, Iddo Bentov, Alessandro Chiesa, Ariel Gabizon, Daniel Genkin, Matan Hamilis, Evgenya Pergament, Michael Riabzev, Mark Silberstein, Eran Tromer, and Madars Virza. Computational integrity with a public random string from quasi-linear PCPs. In *Proceedings of the 36th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '17, pages 551–579, 2017.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 326–349, 2012.

[BCG$^+$13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Proceedings of the 33rd Annual International Cryptology Conference*, CRYPTO '13, pages 90–108, 2013.

[BCG$^+$14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 459–474, 2014.

[BCG$^+$15] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, S&P '15, pages 287–304, 2015.

[BCG$^+$20] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, SP '20, 2020. `https://eprint.iacr.org/2018/962`.

[BCI$^+$13] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In *Proceedings of the 10th Theory of Cryptography Conference*, TCC '13, pages 315–333, 2013.

[BCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *Proceedings of the 14th Theory of Cryptography Conference*, TCC '16-B, pages 31–60, 2016.

[BCTV14a] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *Proceedings of the 34th Annual International Cryptology Conference*, CRYPTO '14, pages 276–294, 2014. Extended version at `http://eprint.iacr.org/2014/595`.

[BCTV14b] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 781–796, 2014. Extended version at `http://eprint.iacr.org/2013/879`.

[bela] bellman. `https://github.com/zkcrypto/bellman`.

[Belb] Bell Labs. SPIN. `http://spinroot.com/spin/whatispin.html`.

[BFR$^+$13] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 341–357, 2013.

[BGG18] Sean Bowe, Ariel Gabizon, and Matthew D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-SNARK, 2018.

[BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. Cryptology ePrint Archive, Report 2017/1050, 2017.

[Bit15] Bitcoin. Some miners generating invalid blocks. `https://bitcoin.org/en/alert/2015-07-04-spv-mining`, 2015.

[CCF$^+$] Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, and Xavier Rival. Astrée. `https://www.astree.ens.fr`.

[CFH$^+$15] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, S&P '15, pages 250–273, 2015.

[CGP$^+$97] A. Cimatti, F. Giunchiglia, P. Pecchiari, B. Pietra, J. Profeta, D. Romano, P. Traverso, and B. Yu. A provably correct embedded verifier for the certification of safety critical software. In *Computer Aided Verification*, pages 202–213, 1997.

[CHM$^+$19] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. Cryptology ePrint Archive, Report 2019/1047, 2019. `https://eprint.iacr.org/2019/1047`.

[CO08] D. Crocker and P. Overell. Augmented BNF for syntax specifications: ABNF. Request for Comments (RFC) 5234, January 2008.

[Cog] Alessandro Coglio. Leo language formal specification. Available on the Aleo GitHub space.

[Cog18] Alessandro Coglio. A formalization of the ABNF notation and a verified parser of ABNF grammars. In *Proc. 10th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, number 11294 in Lecture Notes in Computer Science (LNCS), pages 177–195, 2018.

[CZK$^+$18] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution. 2018. `arXiv:1804.05141`.

[DFKP13] George Danezis, Cedric Fournet, Markulf Kohlweiss, and Bryan Parno. Pinocchio Coin: building Zerocoin from a succinct pairing-based proof system. In *Proceedings of the 2013 Workshop on Language Support for Privacy Enhancing Technologies*, PETShop '13, 2013.

[DFKP16] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, and Bryan Parno. Cinderella: Turning shabby X.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, S&P '16, pages 235–254, 2016.

[DGK$^+$19] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *CoRR*, abs/1904.05234, 2019. URL: `http://arxiv.org/abs/1904.05234`, `arXiv:1904.05234`.

[EMC19] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. Sok: Transparent dishonesty: front-running attacks on blockchain. *CoRR*, abs/1902.05164, 2019. URL: `http://arxiv.org/abs/1902.05164`, `arXiv:1902.05164`.

[EOS18] EOS. EOS.IO technical white paper, 2018. `https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md`.

[ET18] J. Eberhardt and S. Tai. Zokrates - scalable privacy-preserving off-chain computations. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1084–1091, 2018. `doi:10.1109/Cybermatics_2018.2018.00199`.

[Eth16] Ethereum. I thikn the attacker is this miner - today he made over $50k. https://www.reddit.com/r/ethereum/comments/55xh2w/i_thikn_the_attacker_is_this_miner_today_he_made/, 2016.

[FBK] FBK-irst and Carnegie Mellon University and University of Genoa and University of Trento. NuSMV. https://nusmv.fbk.eu.

[Fil17] Filecoin: A decentralized storage network. https://filecoin.io/filecoin.pdf, 2017.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Proceedings of the 32nd Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '13, pages 626–645, 2013.

[GM16] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. Cryptology ePrint Archive, Report 2016/701, 2016. https://eprint.iacr.org/2016/701.

[GM17] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In *Proceedings of the 37th Conference on the Theory and Applications of Cryptographic Techniques*, CRYPTO '17, pages 581–612, 2017.

[GN20] Hermenegildo García Navarro. Design and implementation of the circom 1.0 compiler. 2020.

[Goo14] L.M. Goodman. Tezos — a self-amending crypto-ledger, 2014. https://tezos.com/static/white_paper-2dc8c02267a8fb86bd67a108199441bf.pdf.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '10, pages 321–340, 2010.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *Proceedings of the 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT '16, pages 305–326, 2016.

[GW11] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing*, STOC '11, pages 99–108, 2011.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://eprint.iacr.org/2019/953.

[INR] INRIA. Coq. http://coq.inria.fr.

[JKS16] Ari Juels, Ahmed E. Kosba, and Elaine Shi. The ring of Gyges: Investigating the future of criminal smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 283–295, 2016.

[JPM17] J.P. Morgan Quorum, 2017. https://www.jpmorgan.com/country/US/EN/Quorum.

[jsn] jsnark. https://github.com/akosba/jsnark.

[KB18] Jae Kwon and Ethan Buchman. Cosmos: A network of distributed ledgers, 2018. https://github.com/cosmos/cosmos/blob/master/WHITEPAPER.md.

[KM] Matt Kaufmann and J Strother Moore. The ACL2 theorem prover: Web site. http://acl2.org.

[KMS$^+$16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, SP '16, pages 839–858, 2016.

[KPP$^+$14] Ahmed E. Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, Mahmoud F. Sayed, Elaine Shi, and Nikos Triandopoulos. TRUESET: Faster verifiable set computations. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 765–780, 2014.

[Kyz14] P. Kyzivat. Case-sensitive string support in ABNF. Request for Comments (RFC) 7405, December 2014.

[Ler09] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):461–492, 2009.

[Lib19] Libra, 2019. `https://developers.libra.org/docs/assets/papers/the-libra-blockchain.pdf`.

[Lip12] Helger Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the 9th Theory of Cryptography Conference on Theory of Cryptography*, TCC '12, pages 169–189, 2012.

[LP19] Matt Luongo and Corbin Pon. The Keep network, 2019. `https://backend.keep.network/whitepaper`.

[LTKS15] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, CCS '15, pages 706–719, 2015.

[Mat18] Matter Inc. whitepaper, 2018. `https://github.com/matter-labs/whitepaper/blob/master/whitepaper.md`.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updateable structured reference strings. Cryptology ePrint Archive, Report 2019/099, 2019. `https://eprint.iacr.org/2019/099`.

[Mic] Microsoft Research. Z3. `https://github.com/Z3Prover`.

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000. Preliminary version appeared in FOCS '94.

[MS18] Izaak Meckler and Evan Shapiro. Coda: Decentralized cryptocurrency at scale, 2018. `https://cdn.codaprotocol.com/v2/static/coda-whitepaper-05-10-2018-0.pdf`.

[Nak09] Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 2009. URL: `http://www.bitcoin.org/bitcoin.pdf`.

[Ner19] The Nervos network positioning paper, 2019. `https://github.com/nervosnetwork/rfcs/blob/master/rfcs/0001-positioning/0001-positioning.md`.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. ACM SIGPLAN Notices, 1998. `doi:10.1145/277652.277752`.

[NT16] Assa Naveh and Eran Tromer. Photoproof: Cryptographic image authentication for any set of permissible transformations. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, SP '16, pages 255–271, 2016.

[OBW20] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. Unifying compilers for snarks, smt, and more. Cryptology ePrint Archive, Report 2020/1586, 2020. `https://eprint.iacr.org/2020/1586`.

[PB17] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. `https://plasma.io/`, 2017.

[PGHR13] Brian Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, Oakland '13, pages 238–252, 2013.

[QED17] QED-it, 2017. http://qed-it.com/.

[Qi19] QED-it. zkinterface. https://github.com/QED-it/zkinterface/blob/master/zkInterface.pdf, 2019.

[Rag11] David L Rager. Proof and program parallelism with acl2 (p). 2011.

[RJK13] David L. Rager, Warren A. Hunt Jr., and Matt Kaufmann. A parallelized theorem prover for a logic with parallel execution. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 435–450. Springer, 2013. doi:10.1007/978-3-642-39634-2_31.

[SA15] M-J. Saarinen and J-P. Aumasson. The blake2 cryptographic hash and message authentication code (mac). RFC 7693, RFC Editor, November 2015.

[SCI] SCIPR Lab. libsnark: a C++ library for zkSNARK proofs. https://github.com/scipr-lab/libsnark.

[Sem19] Semaphore. Vulnerability allowing double spend. https://github.com/appliedzkp/semaphore/issues/16, 2019.

[Smi] Eric W Smith. Axe. https://www.kestrel.edu/research/axe/.

[Spa19] Spacemesh: A fair, secure, decentralized and scalable cryptocurrency and a smart contracts computer, 2019. https://spacemesh.io/assets/built/whitepaper1.2.pdf.

[SRIa] SRI International. PVS. http://pvs.csl.sri.com.

[SRIb] SRI International. Yices. https://yices.csl.sri.com.

[SWB19] Josh Swihart, Benjamin Winston, and Sean Bowe. Zcash counterfeiting vulnerability successfully remediated. https://electriccoin.co/blog/zcash-counterfeiting-vulnerability-successfully-remediated/, 2019.

[TBT19] tbtc: A decentralized redeemable BTC-backed ERC-20 token, 2019. http://docs.keep.network/tbtc/index.pdf.

[Tor19] Tornado19. Tornado.cash got hacked. by us. https://tornado-cash.medium.com/tornado-cash-got-hacked-by-us-b1e012a3c9a8, 2019.

[Unia] University of Cambridge. HOL4. http://hol.sourceforge.net.

[Unib] University of Cambridge and Technical University of Munich. Isabelle. http://isabelle.in.tum.de.

[Vor] Andrei Voronkov. Vampire. https://vprover.github.io.

[Woo17] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2017. http://yellowpaper.io.

[WSR$^+$15] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Proceedings of the 22nd Network and Distributed System Security Symposium*, NDSS '15, 2015.

[WZC$^+$18] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: A distributed zero knowledge proof system. In *27th USENIX Security Symposium (USENIX Security 18)*, USENIX '18, pages 675–692, 2018.

[ZCa16] ZCash parameter generation. https://z.cash/technology/paramgen.html, 2016. Accessed: 2017-09-28.

[ZCa17] ZCash Company, 2017. `https://z.cash/`.

[ZPK14] Yupeng Zhang, Charalampos Papamanthou, and Jonathan Katz. Alitheia: Towards practical verifiable graph processing. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, CCS '14, pages 856–867, 2014.