We introduce Lurk, a new LISP-based programming language for zk-SNARKs. Traditional approaches to programming over zero-knowledge proofs require compiling the desired computation into a flat circuit, imposing serious constraints on the size and complexity of computations that can be achieved in practice. Lurk programs are instead provided as data to the universal Lurk interpreter circuit, allowing the resulting language to be Turing-complete without compromising the size of the resulting proof artifacts. Our work describes the design and theory behind Lurk, along with detailing how its implementation of content addressing can be used to sidestep many of the usual concerns of programming zero-knowledge proofs.

Additional Key Words and Phrases: Lisp, Rust

1 INTRODUCTION

Assume that you query a remote database \( w \) with key \( x \), and want a guarantee that the remote database server is answering the query correctly; or that you need to prove that you have a sufficiently high balance \( w \) in your financial account in order to participate in a given transaction \( x \), but don’t want to reveal how much money you have. These are examples of “proofs” that a (potentially secret) value \( w \) and a public input \( x \) satisfy a given relationship, and play an important role in many secure applications where computations are performed by untrusted servers, including blockchain systems.

Succinct Non-Interactive Arguments of Knowledge (or SNARKs) [Ben-Sasson et al. 2014b] have been a very exciting area of research in the last decade: a SNARK allows one party (the prover) to prove to another party (the verifier) that a certain computation \( F \) has been performed correctly. Specifically, it allows the verifier to prove the existence of a witness \( w \) such that \( y = F(x, w) \) for a publicly known input \( x \). It is easy to see that the above examples can be cast in this framework.

One such proof is the witness \( w \) itself, with the verification procedure being to simply recompute the function \( F \). The crucial property of SNARKs, however, is that they produce a proof \( \pi \) which is shorter than \( w \) and can be verified faster than recomputing \( F \) (particularly, sublinear in either). An additional important property of SNARKs is that they can be zero-knowledge (zk-SNARKs), revealing no information about \( w \) to the verifier.
This paper introduces **Lurk**\(^1\), a new LISP-based programming language which automatically constructs zk-SNARKs for arbitrary programs, avoiding ad-hoc compilation of programs into flat circuits – a process which imposes serious constraints on the size and complexity of computations that can be achieved in practice. Although zk-SNARKs theoretically enable applications like those described above, the possibility of deploying them has so far been impeded by the lack of a practical and general language stack. One author conceived of Lurk after his experience implementing the Filecoin proofs [Fisch et al. 2018]\(^2\), which consist largely of Merkle-inclusion proofs at scale.

Lurk emerged through a design effort to generalize such proofs of knowledge, to exploit recent cryptographic proving-system breakthroughs, and to solve software-engineering usability problems still unaddressed by new cryptography. Claims about computation provable in arithmetic circuits (the implementation language of SNARK statements) generally end with the observation that such circuits are Turing-complete. This theoretical equivalence might lead prospective proof implementers to wrongly believe that proofs of execution of programs written in conventional programming languages can be easily represented in SNARK circuits, but this is not the case. In fact, non-trivial programs expressed in R1CS require that control structures be unrolled and recursive programs be translated into a witnessed form which is unintuitive to audit or author by hand and penalizes performance of general-purpose programs.

Lurk solves this problem by integrating a concise interpreter with its cryptographic backend, to express proofs over the evaluation of a high-level Turing-complete source programming language. In other words, the Lurk interpreter sequentially reduces Lurk programs until a terminal result remains, with no intermediate representation required: the (content-addressed) human-readable program is the input to the arithmetic circuit proving its reduction; and the final result of evaluation is similarly legible.

This approach emerged following the observation that Merkle proofs are isomorphic to *functional* membership checks when data structures are represented as Merkle DAGs. In this model, hash-consed pointers to atomic values, including symbols, allocate compound data on a virtual heap to instantiate a RAM without a linear address space. Expressions encoded in such a content-addressed data language transparently (without need of a compilation step) specify arbitrary computation of flat circuits. The resulting language is a human-readable and *writable* Lisp dialect with code-data equivalence. It can be directly proved with a single universal SNARK circuit iterated by a suitable recursive proving system (e.g. Nova)

2 CRYPTOGRAPHIC BACKGROUND

2.1 Arithmetization

The first step in the construction of a SNARK is to arithmetize the computation \(f\), which for the purpose of this paper, can be thought as expressing a computation into a format that makes easier to prove its correctness. Following the work on Quadratic Span Programs (QSP) [Gennaro et al. 2013], a very popular arithmetization for SNARKs is *Rank 1 Constrained Systems* (R1CS) which are a universal model that can encode any computation \(f\).

Let \(f\) be a function defined over a field \(\mathbb{F}\). We want to show that \(\exists w : f(x, w) = y\) or equivalently via a satisfiability predicate \(f’\), that \(\exists w : f’(x, y, w) = 1\). We call \(x, y\) public input and let \(m = |x| + |y| + |w| + 1\). We can associate to \(f\) three \(m \times m\) matrices \(A, B, C\) defined over \(\mathbb{F}\). The condition

\(^1\)https://github.com/lurk-lang/lurk-rs
\(^2\)https://github.com/filecoin-project/rust-fil-proofs
∃\( w : f'(x, y, w) = 1 \) is equivalent to

\[
(A \cdot z) \circ (B \cdot z) = C \cdot z
\]

where \( z = x \| y \| w \| 1 \) and \( \circ \) is the Hadamard (component wise) vector multiplication.

R1CS are closely related to arithmetic circuits (indeed those matrices can be thought as encoding addition and multiplication gates).

Mapping a computation \( f \) to an R1CS system of constraints can be a tedious effort, but it is also one of the main computational bottlenecks in SNARKs, requiring large overhead for the prover in terms of both computation time and memory. Indeed a circuit verifying the computation of \( f \) must basically "write down" the entire computation trace as a witness to prove its correctness.

### 2.2 Incrementally Verifiable Computation

Starting with the work of Valiant [2008], researchers have been studying alternative ways to construct SNARKs that would not require construction of a circuit for the entire computation \( f \). One approach involves verifying that each step of the computation has been performed correctly and then use recursion to fold the correctness proofs of the first \( i - 1 \) steps and the correctness proof of the \( i \)th step into a proof of correctness of all \( i \) steps.

The reason this is appealing is that it allows to verify a computation as it is executed: in this case the cryptographic verification engine is only applied to the transition function of the machine executing a program.

This gives rise to the notion of Incrementally Verifiable Computation (IVC) where the function \( F \) is executed as the repeated composition of a smaller function \( f \) that at the \( i \)th step takes as input the result of the previous step (the input and output of the computation \( f \) is the state of the machine over which the big computation \( F \) is run).

The most efficient candidate for IVC is currently Nova [Kothapalli et al. 2022]. This scheme arithmetizes the circuit of the small function \( f \) as an R1CS and then it shows how to recursively prove that \( y = F(x, w) \) as the composition of \( \ell \) iterations of the transition function \( f \).

### 2.3 Cryptographic Commitments

A cryptographic commitment is a protocol that can be thought of as the equivalent of an opaque envelope. A sender who has a value \( v \), produces a commitment \( C = \text{Com}(v) \) to \( v \), and later can open the commitment \( C \) to \( v \). A commitment must be binding, i.e. can only be opened to a unique value \( v \); it is usually compressing in the sense that \( |C| < |v| \) and can be hiding, i.e. the value \( C \) reveals no information about \( v \). In practice commitments are built from collision-resistant hash functions \( C = H(v, r) \) for some randomness \( r \). The collision-resistant property guarantees binding, the range of \( H \) is usually smaller than its domain, and under some reasonable assumptions the randomness \( r \) protects the secrecy of \( v \).

Commitments are a crucial tool in SNARKs. For example, note that the efficiency requirement on the SNARK verifier prevents them from even reading a description of the function \( F \). One way to deal with this is to assume some preprocessing phase where the function is committed to a short string that can be handled by the verifier (preprocessing SNARKs). In IVC schemes like Nova, the intermediate computation steps are compressed in order for the state of the recursion not to grow too much.

We will show later how the design of Lurk allows for a natural expression of commitments to both values and functions.
2.4 Some Terminology

The construction of a SNARK is usually divided into two parts [Benarroch et al. 2019]. A cryptographic backend which given a suitable arithmetization of the function $F$ builds a cryptographic proof of the correctness of the computation (the QSP work and its improvement by Groth [2016] are examples of cryptographic backends for R1CS).

A SNARK frontend on the other hand is a way to map a program written on some high-level programming language to a good arithmetization that can then be fed to the correct cryptographic backend. Lurk is a frontend that pairs programs written in a dialect of LISP to R1CS circuits that allow building a meaningful proof about these programs. In the next section we describe the drawbacks of a naive compilation of code into arithmetic circuits, and show how Lurk avoids these pitfalls.

3 GENERALIZING CIRCUIT COMPILATION

3.1 The drawbacks of direct compilation

The overarching goal of Lurk is to express the user’s computation in the form of an R1CS instance – or its close predecessor an arithmetic circuit – that can be used by a cryptographic proof backend. At first glance, one may be tempted to think that a stepwise translation of the instructions of the high-level program, coupled with the definition of an ad-hoc set of combinators [Hughes 1982; Wand 1982b] would suffice. But this direct compilation approach has compounding drawbacks.

First, as mentioned in Section 1, R1CS is a flat structure, which admits no explicit control, akin to SMT formulas. Directly compiling into this language requires flattening the structure of the program, notably unrolling all loops and branches, and inlining functions – transformations that are a staple of the domain-specific languages (DSLs) implementing a direct compilation approach [Bellés-Muñoz et al. 2022; Chin et al. 2021; Eberhardt and Tai 2018; Ozdemir et al. 2022]. As these transformations lengthen the size of the R1CS form of the program, they have a negative impact on the proving time.

Second, as most SNARK prover constructions require space linear in the size of the computation – not to mention that many use space-time trade-offs that worsen this space complexity [Thaler 2013] –, this expansion of the program statement risks making prover memory a practical bottleneck.

Since it only considers a fragment of the program at any given proving step, the approach of IVC bypasses these limitations, albeit at the expense of requiring the source program to reflect the transition function of an abstract state machine. Unfortunately, not all programs natively present such an incremental nature, which is why Lurk eschews the direct compilation approach.

3.2 A generalized approach to design proof languages with IVC

On the other hand, an important and often used method of defining a programming language is to give an interpreter for it. In many cases, this interpreter can be defined by way of an abstract machine, either by design, or using a set of elementary techniques to translate a wide range of formal semantics into the desired form [Danvy 2008].

Once equipped with this definitional abstract machine, we can use it to recover a compiler to the instruction set of our domain-specific “virtual machine” – our cryptographic proving backend. We can model those instructions as combinators, whose names and arguments are defined in our source language, and where their generated low-level code is the arithmetic circuits to which they immediately translate, also known as “gadgets” in the cryptographic literature. The cryptographic proving protocol is the “runtime” operating on our circuit. This combinator-based approach to deriving a compiler from an interpreter was pioneered by Wand [1982a,b, 1983] and later formalized.
by Ager et al. [2003]. Lurk adopts one instance of this general blueprint to achieve the general architecture described by Fig. 1.

![Diagram](attachment:image.png)

Fig. 1. Lurk architecture

The overall approach of Lurk is hence to solve the problem of a difficult compilation task by abstracting it: instead of compiling a potentially large program \(\langle f \rangle\) expressed in a language \(L\) directly to R1CS, we write a small-step abstract machine interpreter \(U\) for \(L\), and derive a step-wise compiler to circuits from that interpreter.

This specific use of a zero-knowledge proof system involves changing the nature of the proof statement, informally from \(\exists w : y = f(x, w)\), where \(f\) is expressed as a circuit, to \(\exists w : y = U(\langle f \rangle, x, w)\), where \(\langle f \rangle\) is expressed as a representation of a program in \(L\). Yet, in a pure model of computation, where the deterministic evaluation of \(f\) on its input data \((x, w)\) is definitionally equivalent to the iteration of our interpreter on \(\langle f \rangle, x, w\), we would claim this is a better approach: we expect that the user will represent their intent more directly using a programming language-based description, steeped in omnipresent assumptions about basic control flow.

### 3.3 The Lurk interpreter

Lurk is an eagerly evaluated, purely functional programming language. Its syntax and semantics are inspired by Lisp and Scheme, and consist of a superset of the lambda calculus with `let` and `letrec`, along with lists and a handful of data types.

We chose to adopt eager evaluation and a small-step abstract machine, by using a variant of the CEK machine [Felleisen et al. 2009; Felleisen and Friedman 1986]. The choice of a small-step semantics [Plotkin 2004], beyond leveraging the simplicity of the nominal CEK machine, avoids duplicating rules and premises that a big-step semantics [Kahn 1987] may require to handle exceptions and divergence, and thus makes for a smaller circuit.

Let’s now give some intuition of the reduction: the transition rules of the CEK machine are all syntax-directed, and hence mutually exclusive if we pattern-match them accurately. For instance, let’s consider the variable rule, in the sub-case where the sought variable isn’t at the head of the environment. In a textbook elaboration of the CEK machine to a Lisp-like language with lists, with \(x, y\) ranging over symbols, \(E\) over environments, \(K\) over continuations, and \(V\) over values, it would look like:

\[
\langle x, (y, V) :: E, K \rangle \rightarrow \langle x, E, K \rangle
\]

As in all reduction rules, the pattern on the left of the arrow is related by the reduction rule to the product on the right. The circuit expression of this single relation consists, at a high level, in:

1. assembling a conjunction of boolean clauses that check the conformance of the input term to the pattern,
2. constructing the production from the input pattern elements, and forming a clause expressing its equality with the output of the step,
(3) relating the two items above through an implies gadget, which links the pattern clause to the product clause in the usual (boolean) sense.

In our example, we would check that the control word is indeed a symbol, that it is a valid variable name (rather than e.g. a reserved word), that the environment is non-empty, and that the first binding in the environment is not to the sought-after symbol. We would then form the output state using the same control word and continuation, but with a smaller environment corresponding to the tail of the initial one.

For the purpose of expressing this, the circuit uses an array of high-level tools:

- convenience gadgets for circuit arithmetic,
- gadgets which encode boolean arithmetic using arithmetic circuits (e.g. \(\text{AND}(x, y)\) can be encoded as \(x \times y\), \(\text{NOT}(x)\) as \(1 - x\), etc),
- a fast hash function with field elements as a codomain, which allows fast equality comparisons,
- further, if the current level of elaboration of the program is not sufficient to pattern-match a case, we can ask the prover to unfold more sub-terms of the program through hidden witnesses.

To elaborate on the last point, we note the full computation is not part of the input of any specific step of proof. Rather, Lurk defines hash-consing along the structure of each term of the language [Goto 1974], and pervasively uses domain-separated hash pointers for redexes of the source language. We will detail this in section 4.3. Those hashes, performed with a cryptographically-secure and field-algebra-friendly hash function [Grassi et al. 2021], embody the notion of a cryptographic commitment, so that a proof starts with just a public commitment to the whole program, and the prover progressively produces the required fragments of the computation as witnesses during the proof creation process.

In the case of Lurk, the technique of hash-consing carries recursively through the whole structure of the program: when unfolding a top-level if construct, the prover only needs to reveal the condition of the if, and a commitment to each of its two subsequent branches, in order to pursue the proof to the next frame. One useful side-effect of this discipline is that the proving process will stop early if the evaluation of the program itself stops early through branching, incurring no penalty from the size of the unused branches.

We can think of the overall circuit, then, as being a disjunction of the individual “pattern-match implies product” clauses elaborated through the process above. The actual circuit implementation differs slightly from this discipline, but only because it aims at achieving maximal sharing between sub-clauses used in several rules, as a performance optimization. The primitive data types, supported by Lurk (e.g. 64-bit unsigned integers and their operations) also generate a special set of constraints (e.g. bound checks), defined as a specific gadget, and that we do not elaborate on here.

Finally, since the application of the Lurk circuit provides proofs of individual evaluation steps, it is the responsibility of a cryptographic backend to link them together in a coherent proof relating the whole sequence of statements: besides the recursive Nova proof system [Kothapalli et al. 2022], Lurk is general enough to also support an aggregative backend called Snarkpack [Gailly et al. 2022], used in legacy applications.

4 A BRIEF OVERVIEW OF LURK

In this section, we describe the surface language and novel features of Lurk, along with providing examples motivating those features’ utility.
4.1 Lurk is a Lisp

Inheriting from typical Lisp tradition, Lurk has no syntax of note. Everything is an expression. Some expressions are self-evaluating; for example, the number 3 evaluates to itself. This can be seen when entered into the Lurk REPL:

```
3
Iterations: 1
Result: 3
```

Note that Lurk explicitly tracks the count of “iterations” (in this case, 1), representing the “cost” or number of “clock cycles” needed to evaluate the expression to normal form.

Lurk data is content-addressed, which means that every expression is identified and may be referred to by a type-tagged, cryptographic hash digest.

4.2 Looping and branching

Lurk supports evaluating only some branches of a program, as well as unbounded loops and recursion. This is a strict improvement on the direct compilation approach, which represents branching as the full evaluation of all sub-clauses of a disjunction, and requires unrolling loops at compilation time. For instance, the number of iterations of a sieve of Erathostenes is only equal to the worst case when the number is prime.

The following Lurk program, which counts the number of iterations of the Syracuse recurrence that an initial argument must go through until reaching 1, is inexpressible in open form in direct compilation DSLs today – as unrolling its iterations for any argument would solve the famous Collatz conjecture:

```
(let ((collatz (lambda (n)
    (letrec ((aux (lambda (acc n)
        (if (= n 1)
            acc
            (let ((x (/ n 2)))
                (if (< x 0) ; odd
                    (aux (+ 1 acc) (+ 1 (* 3 n)))
                    (aux (+ 1 acc) x)))))
            (aux 0 n))))))
(collatz 27))
```

```
Iterations: 4016
Result: 111
```

4.3 Commitments

Lurk has built-in support for cryptographic commitments. For ease of display, all commit hashes in the following examples have been truncated to only the first 8 digits; the Lurk REPL displays (and manipulates) commits by their full 254+-bit hashes.

We can create a commitment to any Lurk expression with commit.

```
(commit 123)
```

```
Iterations: 2
Result: (comm 0x2937881e)
```

Now the Lurk evaluation environment knows that (comm 0x2937881e) is a commitment to 123, which can be recovered via open:

Draft, 2023
Important, this opening only works if the value had indeed been previously committed, thus registering the commitment with the Lurk interpreter. Because Lurk commitments are based on cryptographically-secure hashes – just as all compound data in Lurk is [Grassi et al. 2021] – it is computationally intractable to discover a second preimage to the digest represented by a commitment. For this reason, a commitment can be viewed as an index into a write-once store, such that all uses of the same commitment represent the same underlying value. This property is known as computational binding.

Lurk also allows open to operate on field elements, omitting the (comm . . . ) wrapper. For brevity, we will henceforth use bare field elements to refer to commitments.

Lurk also supports explicit hiding commitments with a salt. When hiding is unimportant, commit creates commitments with a default secret of 0.

However, any field element can be used as the secret, which makes Lurk commitments hiding as well as binding.

By varying the secret salt used for a given commitment at random, the prover can prevent the verifier from gaining information about the committed value by pre-computing the hashes of likely values.

For reproducibility, all commitments in the remaining examples of this report were created with a secret of 0. In real applications, secret salts should be selected at random if data hiding is required.

### 4.4 Functional Commitments

Because Lurk allows commitments to any Lurk expression, we can also commit to functions. Functional commitments, introduced by Libert et al. [2016] and extended to function-hiding commitments by Boneh et al. [2021], enable a prover to commit to a secret function \( f \) and later prove that \( y = f(x) \) for public \( y \) and \( x \) without revealing any other information about \( f \). In Lurk, this notion enjoys native support in the language.
The above is a commitment to a function that squares its input, then adds seven. Prior work on efficient function-hiding commitments [Boneh et al. 2021] could only posit an R1CS description of a function \( f \), and modified commitments built from the indexed relations used in holographic proof systems [Chiesa et al. 2020] to convey both a commitment to the defining binary relation \( R_f \) of \( f \), and a proof that \( R_f \) is total and univalent.

By simply choosing the functional Lurk language itself as a basis of how to describe functions, instead of the relational R1CS, Lurk can represent function-hiding commitments more directly. Lurk’s deterministic semantics, which extend the lambda-calculus, offer a straightforward argument for the universality and well-formedness of function definitions. We can thus construct and evaluate an expression that can only be proven to evaluate to one value: the result of applying the function to a given input.

\[
\left( \text{open} \ 0 \ x1aec2d8c \right) \ 9
\]

More formally, in the above, we rely on our built-in structural, nominal hashing scheme \( c \leftarrow \text{Commit}(\langle f \rangle, r) \) using a secret \( r \) to be binding and hiding on the Lurk description \( \langle f \rangle \) of a function \( f \) and our deterministic evaluator \( U(\cdot, \cdot) \) to build a proof that we know a witness for the following relation:

\[
R := \{(c, x, y; \langle f \rangle, r) : c = \text{Commit}(\langle f \rangle, r) \text{ and } U(\langle f \rangle, x) = y \text{ and } \langle f \rangle \in L\}
\]

where \( L \) is the set of definable Lurk functions, and \( c = 0x1aec2d8c, x = 9, y = 88 \) are public.  
Interestingly, even though the set of primitive operations Lurk supports is quite small, they enable the possibility of higher-order functional commitments for free.

\[
\text{(let } ((\text{secret-data} \ 555) \ (\text{data-interface} \ (\lambda \ (f) \ (f \ \text{secret-data})))) \ \text{commit data-interface)}
\]

As far as we are aware, Lurk is the first extant system enabling this powerful usage in the bare language of the proving system.

\[\text{Draft, 2023}\]
5 EXAMPLE APPLICATIONS

5.1 Credit-Score

This example was first introduced by Boneh et al. [2021]. Consider a credit bureau, who wants to preserve the secrecy of its proprietary rating algorithm while also proving that it applies its algorithm fairly to all parties. This can be accomplished by first committing to a function that implements the secret algorithm, then opening function applications on each individual’s credit data. By only interacting with the function through its commitment, it can be verified that the same function is used in all cases, while only revealing the result.

We demonstrate with a simple (if unrealistic) map-reduce based algorithm (note that the definition of map-reduce itself is elided):

```scheme
(letrec ((plus (lambda (a b) (+ a b)))
    (square (lambda (x) (* x x)))
    (secret-function (lambda (credit-data)
          (map-reduce square plus 0 credit-data))))
    (commit secret-function))
```

Iterations: 15
Result: (comm 0xe0fa1fab)

Now, we can open the committed function 0xe0fa1fab to apply it to some data:

```scheme
((open 0xe0fa1fab) '(2 4 7 10))
```

Iterations: 409
Result: 169

Next, consider the complementary situation, in which the algorithm is public, but consumers’ individual credit data is secret. In this case, an individual commits to their private data wrapped in a functional interface (as in the data-interface example above).

```scheme
(let ((credit-data '(2 4 7 10))
    (data-interface (lambda (f) (f credit-data))))
    (commit data-interface))
```

Iterations: 7
Result: (comm 0x3dd3ad73)

Then, the function implementing the credit algorithm is passed as input to a higher-order functional commitment application opening.

```scheme
(letrec ((plus (lambda (a b) (+ a b)))
    (square (lambda (x) (* x x)))
    (credit-score-function (lambda (credit-data)
          (map-reduce square plus 0 credit-data))))
    ((open 0x3dd3ad73) credit-score-function))
```

Iterations: 426
Result: 169

Here, a consumer can prove that their score was computed using the publicly-known algorithm without revealing the details of their own credit data.
5.2 Zero-Knowledge Type-Certificates (zk-TCs)

Another interesting program to implement in Lurk is a type checker. Modern languages and proof assistants often feature extremely rich type systems, with computationally expensive type-checking algorithms. Next, consider that well-typed program fragments are frequently written once and shared as a library with many users, each of whom must independently re-check the program in order to verify its correctness.

With a Lurk typechecker, however, we can generate zero-knowledge proofs that a specific program corresponds to a given type. For example, consider a hypothetical Lurk function `type-check` which takes a program and a type as inputs (in some well-typed language) and returns a boolean if the typing is correct.

```lurk
(type-check my-program my-type)
```

Iterations: ...
Result: T

A zero-knowledge proof that the above program returns T proves that `my-program` inhabits `my-type`. A user of `my-program` can now cheaply verify this type-signature, without having to recompute the type-checking operation. We call this kind of proof a zero-knowledge type-certificate or zero-knowledge proof of type-correctness (zk-TC).

The Yatima Compiler⁵ implements a Lurk backend for the Lean Theorem Prover and Programming Language [de Moura and Ullrich 2021]. Yatima also includes a self-hosted kernel (or trusted typechecker) of Lean written in itself, extended with content-addressing using Lurk expressions and the Poseidon hash function [Grassi et al. 2021]. Yatima then compiles this content-addressing kernel to Lurk, as well as its input declarations (Lean expressions and types).

An example of a formal proof in Lean is `addComm`, which inductively proves commutativity of addition over `Nat`, the type of natural numbers (defined using the Peano construction).

```lean
theorem addComm : ∀ (n m : Nat) → n + m = m + n
| n, 0 => Eq.symm (Nat.zeroAdd n)
| n, m+1 => by
  have : succ (n + m) = succ (m + n) := by apply congrArg; apply Nat.addComm
  rw [succAdd m n]
  apply this
```

This proof relies on other Lean declarations as dependencies, such as `Eq.symm`, `Nat.zeroAdd`, etc., each of which has its own definition (and may contain further dependencies). Each of these declarations is content-addressed in Lurk as a functional commitment, as described in the previous section, as is the Yatima kernel itself. The Yatima kernel is constructed to allow Lurk proofs that dependencies are type-correct, and therefore can perform IVC across any Lean dependency graph.

A zk-TC not only enables faster verification of types, but also verification of type-signatures where the type, program, or even typechecker itself are private inputs. In other words, one can use a zk-TC to prove that one knows of a program that inhabits a certain type, without revealing the program. For example, suppose one had a proof of Fermat’s Last Theorem in Lean, but did not want to reveal it:

```lean
theorem fermatsLast (n : Nat) (p : n >= 2) :
  (exists a : Nat, exists b : Nat, exists c : Nat,
   (pa : (a = 0) → False) → (pa : (b = 0) → False) →
   (pc : (c = 0) → False) →
```

⁵https://github.com/yatima-inc/yatima

Draft, 2023
With the Yatima compiler one can generate a Lurk zk-TC with `secretProof` as a private input, which is a zero-knowledge proof that one possesses a valid formal proof of the above theorem (revealing only its Poseidon hash). This succinct zk-TC of `secretProof` could even, with an appropriate visual encoding, fit within the margins of a book of Diophantine equations.

6 DISCUSSION
So far, we have described the insights that led to the development of Lurk, and demonstrated that its architecture is adequate to serve real-world programming use cases. In this section, we compare our project to related work, and explore upcoming work.

6.1 Related work
The insight that one could bypass both the challenge of a compilation to R1CS and that of segmenting proofs expressed by very large generated circuits by representing the cryptographic interface of a SNARK as iterations of a specialized virtual machine is not new, and predates the existence of performant cryptography to implement it. For instance, landmark approaches have approached simulating a simple CPU, attesting to the validity of memory accesses and intermediate state representations [Ben-Sasson et al. 2014a,b]. The area has gained renewed interest of late [Bruestle et al. 2023].

Moreover, SNARKs have enjoyed an affinity with programmable blockchains, which accumulate updates to a shared state through the execution of programs expressed in a high-level DSL, also called smart contracts. There, zero-knowledge proofs tackle a scalability problem: the most frequent approach for validators of a blockchain to verify the correctness of state updates is to re-execute each of those and come to agreement on their outcome through a Byzantine fault-tolerant consensus protocol. The computation hence needs repeating roughly as many times as there are validators, which is wasteful. Hence streams of work both academic and industrial have sought to model state update messages as proofs, offering succinct verification of the outcome of state updates, rather than an explicit one (see [Bonneau et al. 2021; Bowe et al. 2020; Gluchowski 2021; Polygon 2022; Starkware 2021; Zhang 2019]).

However, those zk-VMs assume that computation is segmented ex ante, as successive executions of reasonably-sized smart contract invocations. The design of prover machines for public proving platforms must hence tailor the hardware to the largest possible contract’s execution, and bound it explicitly through gating at the protocol level.

To our knowledge, Lurk is the only work that places the iterative nature of computation at the level of the evaluation of a programming language, picking a “Goldilocks” level of granularity between microprocessor emulators (which risk being sent bookkeeping instructions of little relevance in the high-level proof), and blockchain zk-VMs (which express but the incremental nature of a sequence of smart contract updates).

6.2 Future Work
Formal verification. By implementing Lurk as an interpreter, we are reducing the surface area of the complex R1CS conversion step to that of using a simpler and universal reduction circuit, so that the resulting proof process can have simpler semantics with fewer engineering sharp edges (see [Aumasson 2022] for a tour of historic pitfalls in this area).

This approach translates changes the slope of the verification challenge: while some direct compilation DSLs have made notable progress on applying formal methods to the verification of an
R1CS compiler [Chin et al. 2021], verifying Lurk consists more simply in verifying the correctness of a specific circuit. As we have a formal semantics for this circuit, in the form of a CEK machine, this area is ripe for formal verification.

**Backend extensions.** At its core Lurk is an interpreter based on the small-step CEK abstract machine. While this offers the advantage of simplicity, the requirement of the cryptographic interface is more loosely defined as an abstract machine with deterministic transitions. This leaves open the exploration of a big-step abstract machine [Danvy 2008] reducing the number of evaluation steps, or that of other abstract machines, such as the CESK machine should we want to extend the language with e.g. control effects [Felleisen et al. 2009].

On the cryptographic side, Lurk uses the Nova proof backend to generate proofs of its execution trace, which forces the proof process to pay an identical overhead on each step of the proof. The recent SuperNova [Kothapalli and Setty 2022], allows using alternate circuits for each step, while only paying for the cost of the specific circuit invoked by each particular step. This would allow us to precede the reduction operated by our interpreter by optimization steps applicable to our domain (e.g. constant folding, see [Appel 1991]). In effect, this would let us build an optimizing compiler modularly, using the staple of compiler phases.

**Proof-Carrying Data.** A more involved direction in which to extend Lurk’s use of IVC is research towards support for Proof-Carrying Data [Chiesa and Tromer 2010]. PCD is a powerful cryptographic primitive describing computation occurring on the nodes as a directed acyclic graph (DAG) of messages, of which IVC embodies the special case of a path on the graph [Bünz et al. 2020]. Equipped with a cryptographic backend supporting PCD, Lurk would be able to model concurrent programming use cases involving mutually distrustful execution nodes, and build native support for it in the language.

**ACKNOWLEDGMENTS**

We are grateful to Friedel Ziegelmayer for a fast store and seminal conversation, to Jonathan Gross for non-technical wizardry, to Eduardo Morais for braving the circuit, and to all the Lurk contributors. We are grateful to Srinath Setty for Nova. We are grateful to Protocol Labs for supporting this work. Nada Amin and Cameron Wong were partially supported by gifts from Protocol Labs and the Filecoin Foundation.

**REFERENCES**


