Lurk: Lambda, the ultimate recursive knowledge (experience report)

NADA AMIN
Harvard University, USA
(email: namin@seas.harvard.edu*)*

JOHN BURNHAM
Argument Inc., USA
(email: john@argument.xyz*)*

GUSTAVO CAPARICA*
Offchain Labs, USA
(email: gustavo.caparica@gmail.com*)*

FRANÇOIS GARILLOT*
Miden, KY
(email: francois@garillot.net*)*

ROSARIO GENNARO†
The City College of New York, USA
(email: rosario@ccny.cuny.edu*)*

ADRIAN HAMELINK*
Miden, KY
(email: adrian.h@miden.team*)*

CHHI'MÈD KÜNZANG
Lurk Lab, USA
(email: clwk@lurk-lab.com*)*

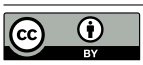
DANIEL ROGOZIN
University College London, UK
(email: d.rogozin@ucl.ac.uk*)*

CAMERON WONG
Harvard University, USA
(email: cwong@g.harvard.edu*)*

Abstract

We introduce Lurk, a 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 an interpreter circuit, allowing the resulting language to be Turing-complete without requiring circuits that grow with the program. We report on the two generations of the system. In the first, Lurk 1.0, evaluation is proved inside a single universal R1CS circuit of roughly twelve thousand constraints, folded recursively with Nova. That design makes proofs

*Work done while at Lurk Lab.
†Work done while at Protocol Labs.



© 2026 Copyright held by the owner/author(s).
This work is licensed under a Creative Commons Attribution 4.0 International License.

Journal of Functional Programming, Volume 36, Article 14 DOI: 10.46298/jfp.17841
Submitted August 2025; accepted August 2026

independent of program size, but every reduction step pays the full cost of the universal circuit and of hashing the machine state, and in early 2024 Lurk 1.0 ran 55 times slower than the SP1 zero-knowledge virtual machine on a shared Fibonacci benchmark. The second generation, Lurk 2.0, keeps the language and the interpreter model but replaces the universal circuit with a DAG of small specialized proof fragments linked by a lookup argument over a STARK backend. On the same benchmark it runs in the same class as general-purpose zkVMs and handles an input one thousand times larger than Lurk 1.0 in the same proving time. Along the way we describe the design and theory behind Lurk, and how its implementation of content-addressing sidesteps many of the usual concerns of programming zero-knowledge proofs.

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) as introduced in their modern form by [10] 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 it has correctly performed a given computation. More precisely, let F be a function (the computation to be verified); then, for publicly known inputs x, y, the prover can convince the verifier of the existence of a *witness* w such that $y = F(x, w)$, i.e. that y is the result of applying F to its arguments. 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 π 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**, 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. Lurk has undergone significant architectural evolution from a folding-based implementation (Lurk 1.0) to the current STARK-based architecture (Lurk 2.0), as detailed below. 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 of [28],

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 (circuits which gates are arithmetic operations over a finite field, 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. Circuits do not yield the intuitive runtime characteristics of a regular program: they represent the trace of a computation, rather than the computation itself. In other terms, circuits fit within a non-uniform model of computation as described by [49], and to compute the unbounded set of functions computed by a Turing machine, a family of circuits is needed. In fact, non-trivial programs translated to arithmetic circuits 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 an 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 random-access memory (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 using the evaluation model outlined in the original Lisp paper by [42]. This requires only a handful of primitive operations, which suffice to resolve the expressiveness problems of flat circuits. The resulting language is a human-readable and *writable* Lisp dialect with code-data equivalence. The first implementation described in this paper, Lurk 1.0, proves programs directly with a single universal SNARK circuit iterated by a recursive proving system (Nova). Measured performance of this design fell well behind contemporary zero-knowledge virtual machines, which motivated the redesign presented in Section 6. Lurk 2.0 keeps the language, replacing only the proof machinery with a DAG of small, specialized proof fragments over a STARK backend.

Because Lurk is a Lisp with code-data equivalence, its content-addressed data language provides an ideal substrate for applications requiring proofs applied to arbitrary data. Lurk operates at a different level of the software stack than Authenticated Data Structures (ADSes) [43, 53]: while ADSes expose a fixed API for verifying specific queries and updates on particular data structures (e.g., set membership, dictionary lookups), Lurk exposes the full expressivity of a Turing-complete Lisp for arbitrary computation. Internally, Lurk does maintain authenticated state using content-addressing and Merkle structures, where the reduction rules of its abstract machine act as the read/write operations. However, as noted by [43], zero-knowledge proof systems like those underlying Lurk trade the asymptotic efficiency of hash-based ADSes for the ability to prove arbitrary computation correctness—a qualitatively different capability that sits at a higher level of abstraction than traditional ADS protocols.

1.1 Relationship to prior publications and version history

This paper extends the Lurk paper presented at ICFP 2023 [3] and published as eprint 2023/369 [2] (updated December 2024). Sections 2–5 closely follow the original conference

version, with minor expansions to cryptographic background (§ 2.2) and limitations (§ 4). New contributions include Section 6 (Lurk 2.0's STARK-based architecture as an experience report), a substantially rewritten Section 7 reflecting lessons from the redesign, and benchmarks comparing Lurk 2.0 with contemporary zkVMs (Fig. 5).

Regarding version naming: **Lurk 1.0** refers to the folding-based architecture using recursive SNARKs (Nova/SuperNova) with a universal R1CS circuit (Sections 2–5), implemented in the `lurk-beta` repository (https://github.com/lurk-lab/lurk-beta) and released as "1.0α" in December 2023. **Lurk 2.0** refers to the current STARK-based architecture using LogUp lookup arguments with a DAG of specialized chips (Section 6), implemented at https://github.com/lurk-lab/lurk. This transition represents a strategic architectural pivot from large-field SNARKs with folding to small-field STARKs with lookup-based composition, detailed in Table 1.

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. A very popular arithmetization for SNARKs is *Rank 1 Constrained Systems* (R1CS) which are a universal model that can encode any computation F, building on the work of [31] on Quadratic Span Programs (QSP).

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 $\exists w : F'(x, y, w) = 1$ 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*

Following the pioneering work of [56], 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.

Consider for example the case of a RAM: a step of the machine consists of some memory addresses, an opcode specifying what type of elementary operation to be applied to the values stored at those addresses, and an address to store the result. In this case the verification of a computation requires only the recursive verification of each step.

This gives rise to the notion of *Incrementally Verifiable Computation* (IVC) in which the result of applying the function F is obtained by 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 each application of f being the state of the machine executing F).

The most efficient candidate for IVC is currently the Nova scheme introduced by [40]. This work arithmetizes the circuit of the small function f as an R1CS and then shows how to recursively prove that $y = F(x, w)$ as the composition of ℓ iterations of the transition function f. IVC is the conceptual impetus behind Lurk 1.0, and Nova its main backend.

Because Nova implements recursion with very low overhead ($\approx$ 20000 constraints), it provides a synergistic proving environment for Lurk's very small, universal circuit. While Nova's recursive circuit is "the smallest in the literature," Lurk's reduction kernel is smaller still ($\approx$ 12000 constraints). Together these efficiencies allow for very low-memory proving. By only performing a single reduction per recursive step, we can reduce the constant memory requirement to that required to fold a circuit of only $\approx$ 32000 constraints. Moreover, this memory efficiency is accomplished in the conceptually simplest way—by directly combining recursive and reduction steps. When space can productively be traded for time by batching inner reductions, we reduce amortized proving cost to that of a single reduction circuit, a remarkable bargain for universality. This bargain carried two costs that only became clear in production. First, every reduction step of Lurk 1.0, however simple, hashes the full interpreter state in circuit. Second, every folded step commits to elements of a 256-bit field, a cryptographic floor on proving time that circuit-size economy cannot lower. Section 5.3 measures both costs, and Section 6 shows how they motivated the move to small-field STARKs.

The first step in Nova is how to show that two satisfying instances z_1, z_2 for the same matrices A, B, C can be folded into a single one. If the R1CS computation was purely linear it would be easy to prove that a linear combination of z_1, z_2 is also a satisfying instance. Because R1CS have a quadratic component coming from the Hadamard product, Nova defines a notion of *relaxed* R1CS for which the above approach works. One issue with the relaxed notion is that at every step some additional terms are added to the system of equations which would increase the size of the system. To "compress" for this additional terms into an R1CS of the same size, Nova uses cryptographic commitments. The end result is that the folding operation is quite efficient, requiring only a few cryptographic operations per step. The correctness of the entire computation is then proven by proving the correctness of the final "folded" R1CS, via a final SNARK.

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 $C = Com(v)$ 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, as described by [11]. 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 [35] 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.[1]

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 as explored by [37, 58] 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 as shown by [6, 20, 25, 45]. 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 as shown by [54] – this expansion of the program statement risks making *prover memory* a practical bottleneck.

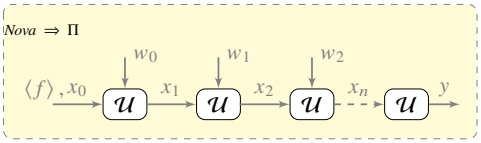
[1] For cryptographic background elements, the reader is invited to consult [19, 55] for a more comprehensive exposition.

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 as discussed by [21].

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 [57–59] and later formalized by [1]. Lurk adopts one instance of this general blueprint to achieve the general architecture described by Fig. 1.



(b)

Symbol	Legend
$\langle f \rangle, x_0$	initial public Lurk program and arguments.
$x_1, \ldots, x_n$	states of the Lurk abstract machine
$w_0 \ldots w_n$	witnesses provided by the prover
y	public output
$\mathcal{U}$	Lurk abstract machine (as a circuit)
Π	proof artifact finally produced by Nova

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 $\mathcal{U}$ 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 $\exists w : y = \mathcal{U}(\langle f \rangle, x, w)$, 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 of [26, 27]. The choice of a small-step semantics as in [46], beyond leveraging the simplicity of the nominal CEK machine, avoids duplicating rules and premises that a big-step semantics à la [38] 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:

$$\langle x, (y, V) :: E, K\rangle \longrightarrow \langle x, E, K\rangle$$

That rule exercises little of the machine: it tests the control word and drops the head of the environment. A rule that exercises more of it is the one that applies an already evaluated closure to an argument value,

$$\langle (\lambda\, x.\, b)\ v, E, K\rangle \longrightarrow \langle b, E[x \mapsto v], K\rangle\,,$$

where the circuit must check the application pattern in the control word, form the extended environment $E[x \mapsto v]$ as a new content-addressed node, and relate output to input through the (unchanged) continuation K. Sibling rules, which begin the evaluation of the sub-terms of an application, additionally push a new frame onto the continuation stack, e.g.

$$\langle e_1\ e_2, E, K\rangle \longrightarrow \langle e_1, E, (\text{arg}\ e_2, E) :: K\rangle\,,$$

so that argument and control flow become part of the proven state as well.

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 as described by [33], 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 of [34], 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[2].

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 of [40], Lurk is general enough to also support an aggregative backend called Snarkpack from [30], used in legacy applications. We have fully implemented and open-sourced the Lurk 1.0 compiler, including runnable tutorials for section 4 of the paper. As of release 1.0α, the size of the circuit for Lurk's evaluation is 12096 R1CS constraints (or roughly half of what it takes to write a SHA-256 function). The same hashing discipline is also the main cost of a proof. Proving one evaluation step means proving the digests that link it to the next, and when reduction steps multiply, the Poseidon gadget dominates the constraint budget. Section 5.3 quantifies this cost; Section 6 shows how Lurk 2.0 amortizes it through proof-level memoization. We provide an engineer's view of the lifecycle of a Lurk proof in Fig. 2[3].

[2]Once complete, the artifact of a succinct and knowledge-hiding proof would no longer contain nor reveal those witnesses.

[3]The cryptographic backend is equipped with Lurk's universal state transition circuit and generates prover & verifier programs of correct Lurk evaluation, as described in Fig. 1. The choice of portions of the program that are kept private is up to the user, see section 4.4 for a choice different from the one exemplified here.

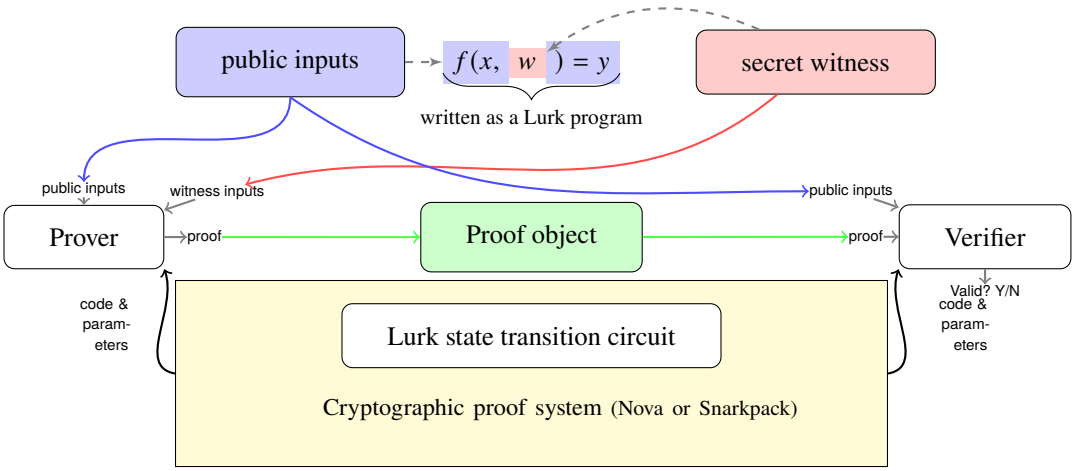


Fig. 2. Lifecycle of a Lurk proof

3.4 Fibonacci function benchmark

We ran a small benchmark focused on computing evaluations of the Nth-term of the Fibonacci function, written recursively in Lurk with the native addition defined within the same language. This naive form of benchmarking does not reflect production usage, [4] but we hope for it to be instructive w.r.t Lurk's evaluation engine. Computing one incremental term of the Fibonacci function takes 16 iterations of the Lurk evaluation algorithm with an additional 11 iterations required for the setup of the algorithm, as well as some padding to account for folding thresholds. The recursion count parameter (rc) represents the number of Lurk evaluation iterations performed before Nova folding is engaged. Results are shown in Fig 3 for varying Fibonacci terms and rc parameters. [5]

rc	term (N)	median runtime (s)	Iter./s (Hz)
600	100	16.675	107.95
700	100	19.695	106.63
800	100	20.943	114.60
900	100	13.980	128.76
600	200	39.098	92.08
700	200	36.885	94.89
800	200	39.295	113.33
900	200	35.920	100.22

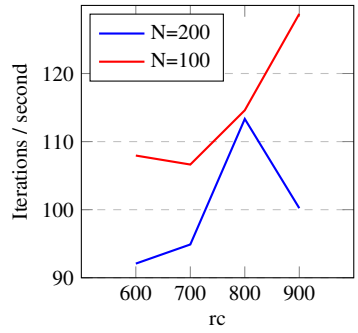


Fig. 3. Benchmark results: iterations per second vs rc for different Fibonacci terms alongside raw data.

[4] Lurk 1.0α allows augmenting the reduction engine with custom circuits for performance-sensitive functions.
[5] The benchmark machine was an AMD Ryzen Threadripper 3970X 32-Core Processor with 256GB RAM and an Nvidia Geforce 2080Ti GPU. Benchmark source: https://github.com/lurk-lab/lurk-rs/tree/icfp-benchmarks.

3.5 SuperNova "pay-as-you-go" folding

In December 2023, the Lurk team implemented and upstreamed support for SuperNova, from [39][6], a more advanced folding system enabling a "pay-as-you-go" model: the prover registers a disjunction of circuits that may be used at any point, and only folds the subset of reduction rules actually used. In practice, this architecture reduced prover memory pressure but did not markedly change overall throughput on our benchmarks. We retain these results in Section 3.4.

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)))))
```

[6] via PR https://github.com/microsoft/Nova/pull/283

```
(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`:

```
(open (comm 0x2937881e))
```

```
Iterations: 4
Result: 123
```

Importantly, 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, thanks to [34] – 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`.

```
(hide 0 123)
```

```
Iterations: 3
Result: (comm 0x2937881e)
```

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

```
(hide 999 123)
```

```
Iterations: 3
Result: (comm 0x3cb2f966)
```

Note that the returned commitment is different from the one returned by both `(commit 123)` and `(hide 0 123)`, since the used salt differs from those cases. However, both commitments open to the same value.

```
(= (open 0x2937881e)
(open 0x3cb2f966))
```

```
Iterations: 7
Result: T
```

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 [41] and extended to *function-hiding* commitments by [13], 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.

```
(commit (lambda (x) (+ 7 (* x x))))
```

```
Iterations: 2
Result: (comm 0x1aec2d8c)
```

The above is a commitment to a function that squares its input, then adds seven. Prior work on efficient function-hiding commitments by [13] could only posit an R1CS description of a function f, and modified commitments built from the indexed relations used in holographic proof systems introduced in [17] 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.

```
((open 0x1aec2d8c) 9)
```

```
Iterations: 12
Result: 88
```

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

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

where $\mathcal{L}$ is the set of definable Lurk functions, and $c = \texttt{0x1aec2d8c}, x = 9, y = 88$ are public. [7]

Interestingly, even though the set of primitive operations Lurk supports is quite small, they enable the possibility of *higher-order functional commitments* for free.

```
(let ((secret-data 555)
(data-interface (lambda (f) (f secret-data))))
(commit data-interface))
```

```
Iterations: 7
Result: (comm 0x03836e2e)
```

```
((open 0x03836e2e) (lambda (data) (+ data 111)))
```

```
Iterations: 14
Result: 666
```

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

5 Example applications

5.1 *Credit-Score*

This example was first introduced by [13]. 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):

```
(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 0x0ea21fab)
```

Now, we can open the committed function `0x0ea21fab` to apply it to some data:

```
((open 0x0ea21fab) '(2 4 7 10))
```

```
Iterations: 409
Result: 169
```

[7] Unfortunately, some implementation details of Nova today do not quite make this proof hiding, as Nova requires the number of iterations of the reduction circuit to be a public input, which leaks some extra information about f. We leave fixing this flaw to future work.

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).

```
(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*.

```
(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)

Type-checking, a computationally intensive operation in numerous programming languages and proof assistants, becomes more manageable with the application of a type-checker constructed in Lurk. With this, we can yield zero-knowledge proofs affirming the type-correctness of a given program, sparing us repeated computation:

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

```
Iterations: ...
Result: T
```

The above Lurk execution provides proof that `my-program` correctly fits within `my-type`, allowing for swift, repeatable verification without re-executing the type-checking process. This kind of proof is what we term a zero-knowledge type-certificate, or zk-TC.

Exemplifying this concept, the Yatima Compiler[8] integrates a Lean Theorem Prover type-checker (see [22]), written in Lurk. Yatima compiles the evaluation trace of this type-checker, when checking Lean expressions and types, into Lurk proofs, providing cryptographic validation for their type correctness.

In addition to its speed benefits, zk-TCs also support secure validation of private inputs, such as the type, program, or even the type-checker itself. For example, a hidden proof of Fermat's Last Theorem in Lean can be safely attested:

[8] https://github.com/yatima-inc/yatima

```
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) ->
a ^ n + b ^ n = c ^ n) -> False
:= secretProof
```

By using Yatima's compiler to generate a zk-TC, with `secretProof` as a private input, we can create a compact zero-knowledge proof confirming possession of a valid proof for the theorem. This zk-TC of `secretProof`, given an apt visual encoding, could neatly fit within a book's margins. As we work towards addressing current performance limitations in the type-checker, the generation of zk-TCs for broader Lean programs will become more feasible.

5.3 Limitations of Lurk 1.0

While Lurk 1.0 delivered a first working zk-Lisp, it exhibited fundamental bottlenecks:

(1) **Cryptographic Backend performance.** The cryptographic backend of Lurk was based on a large-field SNARK using elliptic curves. To summarize salient aspects of SNARK performance with caricatural brevity, much of the proving speed of cryptographic proving is determined by the cost to perform an operation called a polynomial commitment over what the cryptographic proving system considers its "native" elements. While Lurk is a Lisp interpreter which deals with values which can be represented in 64 bits or less, the native elements elliptic curve based systems are most often 256 bits field elements (at the 128 bit security level): committing to these values naively induces a blowup factor as part of their commitment scheme, which can translate to an 8-fold worse in performance for this key operation[9]. Modern proof systems built on small-field STARKs, including ethSTARK from [7] and Plonky2 described by [48], allow much more efficient arithmetic closely matching the size of user data.

(2) **Commit and Open Overhead.** The initial Lurk architecture was meant to leverage folding, a way to combine full-fledged R1CS instances. Designing a "universal circuit" for this use therefore required making sure that the objects of the proof were of a fixed size, despite Lurk programs being potentially unbounded in size. As explained in Section 3.3, Lurk used cryptographic commitments to link each element of the state of the interpreter —the Control word, Environment and (K/C)ontinuation of the abstract machine— to the next step of reduction. Every CEK step thus began by "opening" (verifying) the previous program state hash and ends by "committing" (hashing) the new state. Each hash invocation, whether proving knowledge of a pre-image to a fixed hash, or proving the hash of a fixed state, costs a significant amount of constraints enforcing the Poseidon gadget from [34], sometimes dominating the prover's time when the number of reduction steps grows. Moreover, without memoization, identical subcomputations were re-hashed from scratch each time

[9]While some works, such as [50], argue that some elliptic-curve based commitment schemes can be implemented in ways that reduce the computational impact of the bit-width difference between committed elements and native representation, the careful engineering required to achieve this result in practice was not widespread at the time of Lurk's initial release. See [5, 44] for recent progress on making bit-width awareness a more widespread engineering concern in the world of elliptic-curve based SNARKs and folding.

they appeared, multiplying this overhead. Lurk 2.0 addresses this through the lookup-based memoization described in § 6.5: each unique subcomputation (and its associated content-addressed hash) is computed once and reused via the lookup bus, dramatically reducing redundant hashing. While content-addressing remains central to the architecture, the DAG structure and proof-level memoization ensure that hash costs are amortized across all uses of shared subterms. Workloads with minimal structural reuse—such as linear array scans or dense matrix operations—do not benefit from this amortization and pay the hash overhead without compensating gains.

(3) **Monolithic CEK Circuit and Folding overhead.** Lurk 1.0 used a single universal R1CS circuit for all reduction rules. Noticing that in the Nova paradigm, even trivial reduction steps (e.g. integer comparisons) paid the full circuit cost, the Lurk team moved to an implementation of SuperNova (§ 3.5), which limited the cost of unused gates. However that non-uniformity support was limited to a disjunction of *folding step* circuits, which perhaps counter-intuitively should not be *reduction steps*. Indeed, to not introduce spurious inefficiencies by way of folding itself, the work done in each *folding step* circuit should be larger than the recursive verifier circuit ensuring the soundness of the construction (see § 2.2), which required Lurk to use *sequences of reduction steps* as atoms of folding (the length of those *reduction steps* is represented by the `rc` parameter in Fig. 3). For a reduction engine with 15 small-step reduction rules such as the CEK machine, there would be 15^{10} (about half a trillion) possible sequences of steps of length 10, making the use of non-uniform circuits unwieldy.

These factors led to sub-optimal performance (see Section 3.4), motivating the redesign in Section 6.

6 Lurk 2.0: from universal circuits to zkVM architecture

This section presents Lurk 2.0's architecture, which fundamentally reimagines zero-knowledge proof construction for functional languages. We begin by contrasting the key architectural differences between Lurk 1.0 and 2.0 (§ 6.1), then explain how computation is structured as a DAG rather than a linear chain (§ 6.2). We detail LAIR's role in coupling witness generation with constraint specification (§ 6.3), show how lookup arguments replace continuation-passing (§ 6.4), and explain the soundness mechanisms preventing overflow and cycles (§ 6.5). A worked example demonstrates memoization benefits (§ 6.7), followed by discussions of memory management (§ 6.8), LOAM's declarative layer (§ 6.9), and performance results (§ 6.10).

6.1 Overview: key architectural changes

The original Lurk system, which we now call Lurk 1.0, was built on a single, monolithic "universal" circuit, where every reduction step (no matter how simple) was handled by the same cryptographic machinery and combined into proofs using recursive SNARKs and folding schemes. This approach had strong theoretical elegance: by auditing and trusting the correctness of just one circuit (plus the folding protocol), users could verify any Lurk computation. However, this design came with major performance costs. Every computation, however straightforward, had to pay some overhead due to the mechanics that glued each step together: folding and hashing.

Lurk 2.0 adopts a fundamentally new approach, inspired by modern zkVMs but tailored to the unique strengths of functional programming and abstract machine semantics. Rather than packaging all of computation into a universal circuit, or piecing together CPU-style instruction circuits, we instead split the computation into several small, specialized proof fragments (technically, spans of trace rows). Each fragment is responsible for the reduction of a local subterm. These fragments are connected as nodes inside a *Directed Acyclic Graph* (DAG) of computations, reflecting the true structure of the program's reductions.

To *glue* these proof fragments together, Lurk 2.0 uses a *lookup argument*—a cryptographic sub-protocol, introduced in [29], that emulates the API of a key-value store. Each fragment declares what reduction results it *provides* and what results it *requires* from the rest of the computation. The lookup argument acts as a "bus" that tracks every requirement and provision throughout the proof graph, guaranteeing (by the Schwartz-Zippel lemma) that every computation a proof fragment depends on was actually computed somewhere else. This forms a sound assembly for the whole computation within a single proof instance, giving us an atomic building block that is both modular and reusable—ready to be integrated with recursive proofs, folding, or more advanced composition techniques if needed.

Table 1 summarizes the key differences between the two architectures.

6.2 A shift from linear chains to computation DAGs: an overview

Unlike Lurk 1.0's folding-based design, which pieced together a chain of identical circuits, Lurk 2.0's architecture decomposes computation into a DAG of small, specialized sub-circuits of reduction (sometimes called "coroutines"[10]). Each node in the DAG corresponds to a familiar piece of computation (e.g., a function call). Directed edges in this graph encode dependencies: if one computation requires the result of another, we draw an edge from the provider to the consumer.

A key advantage of this approach is *memoization at the sub-proof level*. When the same subcomputation is needed in multiple places—such as in recursive or dynamic programming algorithms—Lurk 2.0 proves it just once, memoizes it, and reuses its proof anywhere else it's needed. This greatly reduces redundant work and allows massive parallelism: sub-proofs (nodes at the same DAG layer) can be proved at the same time.

Soundness paradigms: abstract machines versus CPU-based zkVMs. This shift also changes the way soundness and trust are handled in the system. In the old Lurk design (Lurk 1.0), the main trusted component was the universal circuit: auditing it, plus verifying the safe use of the folding machinery, provided security. In contrast, most zkVMs operating today—such as those that emulate CPUs—require users to trust that the constraints generated for each instruction correctly emulate the semantics of that CPU (its computational and memory model). Each fragment of the computation corresponds to a circuit that acts out a CPU instruction. So the trusted computing base (TCB) is essentially the set of constraint-generating functions for the emulated hardware.

In Lurk 2.0, although we use the same idea of generating constraints per computational fragment, there is a crucial difference. Our proofs are not based on CPU instruction emulation, but on the reduction rules of a minimal functional abstract machine. While Lurk 1.0 explicitly

[10] In Lurk's STARK-based architecture, a *coroutine* or *chip* is a modular constraint-generating unit that implements a specific reduction rule (e.g., function application, arithmetic operation, or memory access) with its own specialized AIR (Algebraic Intermediate Representation) constraints (see § 6.3).

Aspect	Lurk 1.0	Lurk 2.0
Proof Backend	Recursive SNARKs (Nova/SuperNova) with folding over cycles of elliptic curves	Small-field STARKs (32-bit BabyBear field) with lookup-based composition
Circuit Structure	Single monolithic universal R1CS circuit handling all reduction rules	DAG of specialized chips (coroutines), one per reduction type
Proof Composition	Linear folding chain: each step folds into accumulator	DAG of atomic proofs connected via LogUp lookup argument
State Linking	Explicit CEK state (Control, Environment, Continuation) passed via cryptographic commitments and hash openings	Implicit dependencies via provide/require on lookup bus; content-addressed pointers
Field Size	Large prime field (~256-bit, e.g., Pallas)	Small prime field (~32-bit BabyBear)
Intermediate Rep.	Direct R1CS constraint generation from Lurk semantics	LAIR (two-level IR coupling witness and constraints) + LOAM (declarative Datalog-style semantics)
Trusted Computing Base	Universal circuit correctness + folding protocol soundness	Specialized chip correctness + lookup argument soundness + acyclicity enforcement + recursion
Performance	~20 s to prove the 100-th Fibonacci number (§ 3.4); 55× slower than the SP1 alpha on Fibonacci in early 2024 [4]	19.4 s to prove the 10^5-th Fibonacci number, comparable to SP1's 18.6 s on the same workload (Fig. 5)

Table 1. Comparison of Lurk 1.0 and Lurk 2.0 architectures. The transition from folding-based SNARKs to lookup-based STARKs enabled specialization, yielding orders-of-magnitude performance improvements.

traced CEK machine states (Control, Environment, Continuation), Lurk 2.0's abstract machine semantics are specified in LOAM and realized through LAIR chips, with dependencies tracked implicitly via the lookup bus rather than explicit continuation passing. This abstract machine is far simpler, with only explicit and transparent state. Operations are pure and stateless, save for explicit changes to the environment or function call dependencies. Unlike CPU-emulating machines, the impact of each fragment is clear and limited—there is no hidden state or side-effect-laden memory to track.

The functional advantage: simplicity and memoization. The main benefit of this functional interpretation is that it makes both memoization and soundness far easier to achieve. In CPU-based zkVMs, even if you see "the same" expression in two different places, you can't simply reuse the proof—there could be differences in memory, registers, or program context that change its meaning or side effects. Common sub-expression elimination and proof reuse are hard to get right.

But in Lurk 2.0, because the computation is fundamentally functional and state effects are explicit, encountering the same expression and environment in more than one place will always yield the same result. This lets us share the work of proving each unique fragment or application: each is computed and proved once, even if it appears in many branches of the reduction. The result is not just a computation DAG but a minimal one, where repeated work has been eliminated.

This approach is built into the architecture: memoization comes for free, and more generally, it allows for highly efficient, parallel, and reliable proof generation, with a clear audit trail for each computational step.

6.3 LAIR: coupling witness generation with constraint specification

The shift to specialized circuits required a new intermediate representation managing the interplay between execution (witness generation) and verification (constraint generation). Lurk addresses this with the *Lurk Algebraic Intermediate Representation* (LAIR), a two-level IR tightly coupling these aspects.

At the top level, LAIR provides a human-readable representation resembling a conventional intermediate language. Programs decompose into functions with named variables, control flow, and operations expressed as relations. This layer captures structure suitable for reasoning, optimization, and audit.

Beneath this, LAIR grounds in a shared bytecode representation. Crucially, *the same bytecode underpins both witness generation and constraint generation*. For each bytecode instruction, two routines exist: one computes concrete field values during execution (building the witness trace), and the other emits algebraic relations that must hold (specifying STARK constraints). These routines are strictly aligned by shared bytecode, ensuring the trace and constraint system remain synchronized by construction.

As a concrete example, consider computing `(+ 2 3)` in LAIR bytecode:

```
Const(2)          // Push 2 to stack[0]
Const(3)          // Push 3 to stack[1]
Add(0, 1)         // Push stack[0] + stack[1] to stack[2]
Return(id, [2])   // Return stack[2] as output
```

During witness generation, `Add(0, 1)` computes the concrete value 5 and appends it to the trace. During constraint generation, the same `Add` bytecode emits the algebraic relation `stack`$[2] =$ `stack`$[0] +$ `stack`$[1]$, which is checked during STARK verification via batched quotient evaluation. Memory operations like `Store` and `Load` similarly generate both concrete memory traces and consistency constraints.

This design eliminates "semantic drift" bugs plaguing systems where execution and proof semantics are defined separately, as documented by [60]. Because both derive from the same representation, any semantic change automatically reflects in both. Auditing reduces to verifying bytecode semantics and the two derived routines.

For example, a `cons` operation (allocating a pair) in LAIR involves calling a hasher chip (typically Poseidon2) to compute the content-addressed pointer[11] from the car, cdr, and a type tag; issuing a `Store` bytecode operation to record the allocation; and tracking the resulting

[11] *Content-addressing* means data is referenced by its cryptographic hash rather than by a memory location. A content-addressed pointer is computed deterministically as `Hash(data)`, ensuring two structurally identical values have the same pointer and enabling efficient memoization and data deduplication.

memory relation in LOAM's content-addressed store. During constraint generation, the same bytecode emits algebraic relations enforcing that the pointer equals the hash output and that memory accesses are consistent. During witness generation, the hasher chip computes concrete field values and the store operation records the pair with its pointer. Both the witness trace and the constraint system derive from the shared LAIR bytecode, ensuring they remain synchronized by construction.

LAIR bridges Lurk's high-level language and the AIR (Algebraic Intermediate Representation), as pioneered by [8], defining the STARK constraints, ensuring correctness, modularity, and auditability.

The relationship between LAIR and the broader system architecture is clarified by considering the role of LOAM (Lurk Ontological Abstract Machine), a declarative layer that sits conceptually above LAIR. While LAIR specifies *how* proofs are constructed—the operational details of witness generation, constraint emission, and bytecode execution—LOAM specifies *what* should be proved: the reduction semantics and memory management rules expressed as Datalog-style inference rules. The current implementation hand-compiles LOAM specifications into LAIR bytecode, but the long-term vision is an automated LOAM-to-LAIR compiler that would make the semantic specification the sole source of truth. This separation ensures that the high-level semantics remain backend-agnostic and auditable, while LAIR handles the low-level details of efficient proof generation. LOAM's declarative nature makes reduction rules easier to reason about and evolve, and is discussed further in § 6.9.

6.4 From continuation-passing to memory arguments via lookups

The prior Lurk architecture relied on explicit stack and continuation manipulation, faithfully tracing the CEK abstract machine's bookkeeping for argument and control flow. Lurk 2.0 revisited this foundation in light of proof-system design.

It became clear that, for the purposes of zero-knowledge proof construction, certain aspects of the abstract machine's bookkeeping (such as stack or continuation management) can be derived or inferred, rather than being proved per se. The critical requirement is to maintain a faithful, content-addressed representation of the state, and to prove that subcomputations and data accesses are valid according to the reduction semantics.

This led to the adoption of a *memory argument* approach originally built by [12] based on an explicit *lookup* protocol: each time a reduction step or function call depends on the output of a prior computation, this dependency is modeled as a lookup (or *require*) into a logical memory or communication bus; when a step produces a value or result, it is modeled as a corresponding "write" (or *provide*) to this logical memory. This architecture has a direct operational analog: each time the Lurk evaluator observes an allocation or reduction that produces a new pointer, symbol, or closure, it records a *provide*; each read, application, or dereference is a *require*.

For example, when computing `(+ (fib 3) (fib 3))`, the first call to `(fib 3)` issues a *provide* with the result 2 and a content-addressed key (the hash of the expression and environment). The second call issues a *require* for the same key. The lookup argument cryptographically ensures this require matches the earlier provide, enabling memoization soundness.

The lookup argument both models memory and mediates all functional and cryptographic dependencies in the proof. The best-suited cryptographic primitive for our use was the LogUp

argument, a *logarithmic derivative* based lookup protocol as pioneered by [24, 36], which maintains a running accumulator that tracks the balance between provides and requires across the entire execution trace. The accumulator is constrained by the following equation:

$$s_{i+1} = s_i + \sum_k \frac{m_k}{r + \mathrm{RLC}(\mathrm{value}_k, \gamma)} \tag{1}$$

where s_i is the accumulator at trace row i, m_k the (signed) multiplicity of the k-th provide or require at that trace row, and RLC (Random Linear Combination) is a randomized encoding parameterized by verifier randomness γ and an additional global challenge r. The constraint system enforces $s_0 = 0$ and that the sum over all fragments returns to zero; by the Schwartz-Zippel lemma and field size bounds, this enforces global consistency of the lookups with overwhelming probability. The soundness of this LogUp argument is proven in Section 3.2 of [36], though with a critical limitation: soundness only holds when the total multiplicity (the sum of all lookup multiplicities) is bounded by the field characteristic. For small fields like BabyBear ($p \approx 2^{31}$), this bound can be reached in practice, necessitating the additional soundness mechanisms described in § 6.5.

6.5 Ensuring soundness in small fields: the nonce and depth arguments

Operating in small prime fields introduces a subtle and critical soundness challenge for lookup-based arguments: it becomes possible for a malicious prover to exploit field overflows, for example, by completing p-many requires with a single provide when p is the field order, thus bypassing genuine proof of computation[12]. Similarly, complex dependency patterns can create cycles in the proof DAG, undermining termination and correctness.

Lurk 2.0 addresses these problems with a two-pronged solution:

Nonce-based write-once, read-many (WORM) lookups. To thwart multiplicity overflow, every lookup interaction is parameterized by a unique *nonce* and an incrementing *counter* (inspired by memory checking protocols of [12]). Each provide initializes a new value at $(nonce = 0, count = 0)$. Every subsequent require consumes a prior state $(prev_nonce, prev_count)$ and produces a new state with counter increased by 1 and a fresh nonce (e.g., current trace row). The final state generated this way gets consumed by the initial provide call to ensure every state is used exactly once, forming a closed chain. While the counter alone prevents field overflow (even if a prover issues p requires where p is the field order, the counter increments prevent the lookup accumulator from returning to zero via overflow), the nonce prevents a distinct attack: a malicious prover reusing the same provide multiple times by rewinding to an earlier counter state. The nonce (typically the current trace row index) ensures Write-Once semantics by making each state unique. The circuit enforces that $new_count \neq 0$, forcing the prover to explicitly witness the chain of dependencies and ensuring no lookup is ever shortcut via overflow or state rewind.

This nonce-counter approach bypasses the characteristic bound with approximately 2× overhead (each provide and require becomes two lookup operations). Alternative approaches exist: [23] solves the same characteristic bound problem using a different technique based on abstracting pole multiplicities with algebraic basis units. In their approach, each lookup pole is assigned a polynomial unit (e.g., Y^i for univariate monomials or multilinear polynomials), which is then evaluated at a random challenge to produce scalar units that prevent pole

[12]This issue was investigated in parallel work by [23]

cancellation even when the number of lookups exceeds the field characteristic. While their algebraic units approach is elegant and particularly well-suited for binary fields and wide-trace zkVMs, Lurk's nonce-counter approach was chosen for its simplicity and direct fit with our functional abstract machine's memory-checking semantics. Both solutions achieve the same security goal—preventing multiplicity overflow—through different cryptographic techniques.

Depth-tracking for acyclic proof DAGs. To prevent cyclic reasoning, each function call result carries a *depth* parameter—effectively recording the "generation number" or causal distance from the root computation. When a function depends on subcomputations with depths $d_1, d_2, \ldots, d_n$, its result receives depth:

$$d = \max(d_1, d_2, \ldots, d_n) + 1 \tag{2}$$

The AIR enforces that each dependency's depth is strictly less than the depending computation's depth via LessThan constraints (implemented as range-checked byte decompositions). The resulting proof dependency graph is acyclic by construction—any cycle would require $d < d$ for some depth d, which is impossible. This mechanism is reminiscent of Lamport clock orderings in distributed systems and eliminates the opportunity for circular proofs.

6.6 Computation as a DAG of proofs: memoization and common subexpression elimination

With the above mechanisms in place, Lurk 2.0's computation is naturally reified as a *Directed Acyclic Graph* of reduction steps or proof fragments. Each node corresponds to a function call, memory allocation, or arithmetic operation. Edges represent *require* dependencies on prior provides.

This architecture yields an immediate advantage: *memoization*. If a given subcomputation (for example, a recursive call with particular arguments) arises in multiple places, its proof need only be constructed once. Other fragments of the computation that depend on it simply refer to the same node, avoiding re-proving and redundancy. This approach can be understood as "common subexpression elimination" lifted from the state to the proof layer, and is particularly effective for functional algorithms (e.g., dynamic programming) and workloads with significant sharing.

Proof generation thus admits topological ordering (from leaves to root), parallelization, and on-demand evaluation: proofs can be constructed for any fragment independently, so long as their dependencies are satisfied.

6.7 Example: Fibonacci computation as a memoized proof DAG

To illustrate Lurk 2.0's memoization, consider naive recursive Fibonacci. Computing `(fib 5)` generates 15 total calls in the recursive tree, but only 6 are unique: `(fib 5)` through `(fib 0)`. The duplicate calls—`(fib 3)` appears twice, `(fib 2)` three times, and base cases repeatedly—represent wasted proving effort in traditional systems.

Lurk 2.0 transforms this 15-node call tree into a 6-node proof DAG. When the prover first encounters `(fib 3)`, it generates the proof fragment and issues a *provide* to the lookup bus with the content-addressed hash of the query. Later, when `(fib 4)` requires `(fib 3)`, the prover issues a *require* for the same hash. The lookup argument ensures soundness: every require matches a corresponding provide, but the *same* provide satisfies multiple requires—precisely common subexpression elimination at the proof layer.

The depth tracking mechanism ensures acyclicity: each provide has strictly greater depth than its dependencies. Proving proceeds in topological order from leaves to root. The prover generates only one proof fragment per unique subcomputation, yielding a 2.5× reduction for `(fib 5)`. For `(fib n)`, savings scale to $O(2^n/n)$, making functional recursion competitive with imperative implementations.

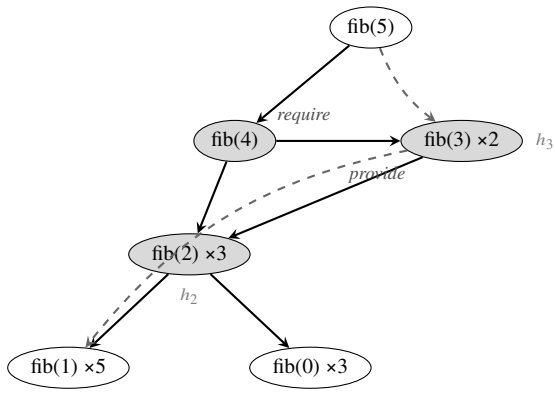


Fig. 4. Memoized proof DAG for `(fib 5)`. The naive call tree has 15 nodes, but memoization reduces this to 6 unique computations. Nodes shaded gray appear multiple times in the original tree (multiplicity shown as $\times N$). Content-addressed hashes (e.g., h_3, h_2) identify each unique subcomputation on the lookup bus. Solid edges represent the first computation establishing each provide. Dashed edges represent subsequent requires reusing the same memoized result. One exemplar provide/require pair is annotated. This DAG structure enables proving each unique subcomputation once, regardless of how many times it appears in the recursive call tree.

6.8 Memory management and content addressing

Memory management in Lurk 2.0 fully leverages the lookup argument framework. All heap-allocated data—symbols, conses, closures—are committed via content-addressed pointers, computed via deterministic cryptographic hashes (e.g., Poseidon by [34]), ensuring computational binding and privacy. Each allocation issues a *provide* in the proof bus; dereferencing or interacting with an object is a *require*. Thus, pointer dereferences, function calls, or environment lookups are uniformly expressed as buses of dependency, with their validity ensured by lookup constraints as above.

6.9 LOAM: future work in declarative VM specification

Above LAIR sits the *Lurk Ontological Abstract Machine* (LOAM), a declarative layer for specifying reduction semantics and memory management. Inspired by Datalog, LOAM expresses the abstract machine's behavior as inference rules and logical predicates rather than imperative control flow.

LOAM serves as the semantic architect, specifying *what* computations should happen without prescribing *how* they are implemented. This separation makes reduction rules easier to reason about, audit, and evolve. For example, cons allocation in LOAM is specified as:

```
// Allocate cons in content-addressed memory
cons_mem(car, cdr, addr) <--
```

```
        cons(car, cdr),
        let addr = alloc_addr(Tag::Cons);

        // Register pointer value relation
        ptr_value(ptr, value) <--
        cons_digest_mem(value, addr),
        let ptr = Ptr(Tag::Cons, addr);
```

These rules declare that when a `cons` operation occurs, memory is allocated and the pointer-value relation is established. The actual address allocation and hash computation are delegated to LAIR's bytecode layer. This declarative style enables formal reasoning about reduction semantics independently of proof system details.

LOAM remains under active development. While LAIR has reached production stability, the current Lurk implementation uses LOAM as a design and documentation layer, with abstract machine logic hand-compiled to LAIR. As the LOAM compiler matures, the vision is for the entire VM semantics to be specified declaratively, providing a high-assurance foundation. We defer detailed exposition to future work.

6.10 Lurk 2.0 benchmarks

Figure 5 reports proving times for Lurk 2.0 and four RISC-V-based zkVMs on four workloads, measured in the Lurk 0.5 benchmark report [4][13] and reproduced here in full rather than by reference. The comparison with Sphinx matters most for interpretation. Sphinx is our fork of the SP1 prover, and Lurk 2.0 runs on the same prover backend, so the difference between the Lurk and Sphinx bars is the price of Lurk's language semantics relative to a RISC-V target on identical proving machinery. The Fibonacci benchmark computes the 10^5-th Fibonacci number. The fast Fibonacci benchmark computes the $(2^{64} - 1)$-th Fibonacci number by 2×2 matrix exponentiation. The sum benchmark adds the integers $0..10^5$ passed as input. The longest common subsequence benchmark compares two strings of 142 and 121 characters; the Lurk program is a naive recursive transcription of the textbook algorithm, and the Rust baselines come in two variants, one that builds the dynamic programming table explicitly and a simpler one. The implementations compared are not the same source, by design. The RISC-V systems run idiomatic Rust, the high-level language those zkVMs are built to compile, in the simplest form that expresses each algorithm, while Lurk runs the natural Lisp program, which benefits from memoization. Each language thus implements the same algorithm in its own best way, so the algorithms may reflect a workload bias, but the languages do not. Fig. 6 quotes both Fibonacci implementations as an example.

All measurements are single-machine runs on an AWS r7iz.metal-32xl instance (Intel Xeon at 3.9 GHz, 128 vCPUs, 1024 GiB of DDR5 RAM) with no GPU acceleration, using commit `97d8ac6` of the benchmark repository. Lurk, Sphinx, and SP1 share the same backend architecture and were run with standard FRI parameters at a conjectured 116-bit security level. RiscZero was run with its default parameters, and Jolt is the one system in the comparison that does not use FRI, since it bases its security on the discrete logarithm problem. Each system has up to three bars in Fig. 5. The default bar uses each prover's default parameters. The Lite bar uses smaller and more numerous shards, which gives the prover more parallelism at the cost of more verifier work, while the Turbo bar uses larger and fewer shards, with the opposite

[13] Benchmark sources: https://github.com/lurk-lab/zkvm-benchmarks

trade. RiscZero and Jolt do not expose these shard controls, so they appear with the default bar alone.[14] All five systems agree on the computed outputs, e.g., 2754320626097736315 for the Fibonacci benchmark and 4999950000 for the sum benchmark. Table 2 reports the setup, verification, cycle, and memory measurements behind Fig. 5 for the Default configuration.

Lurk 2.0 is fastest on three of the four workloads. The lead is largest on the vector sum, 9.6 times faster than the next system, reflecting Lurk's native handling of content-addressed input data. On longest common subsequence, a twelve-line naively recursive Lurk program proves faster than both Rust baselines, including the one that builds the dynamic programming table explicitly, because the lookup bus memoizes the proof of each unique subcomputation at the proof level. The fast Fibonacci runs show the same effect on a workload with much larger inputs.

The plain Fibonacci workload is where Lurk 2.0 loses. Jolt proves it in 15.9 s and SP1 in 18.6 s, against 19.4 s for Lurk, and Sphinx reaches 10.8 s in its Lite configuration. Two factors explain this result. First, the benchmark is inherently sequential, so memoization has little repeated structure to exploit beyond the linear chain of calls itself. Second, each recursive call allocates and hashes content-addressed state, the overhead discussed in § 5.3, which the imperative loop of the Rust programs avoids entirely. Lurk 2.0 also loses on measures other than proving time. Verification takes 0.235 s for Lurk against 0.027 s for RiscZero, and default proving of the Fibonacci workload uses 12.1 GiB against 8.4 GiB for RiscZero, though Jolt uses more at 29.6 GiB.

A workload without reusable structure, e.g., a linear scan over private data, would instead pay Lurk the hash overhead discussed in § 5.3 without compensation. We do not measure hash-dominated workloads, e.g., repeated SHA-256 or Keccak evaluations, because such a benchmark would not be informative. A byte-oriented cryptographic hash does not run as virtual-machine code on any system compared here; on every one of them, the only performant implementation is a precompile, a direct arithmetization of the hash in the prover's constraint language. That arithmetization, an AIR gadget, would be shared by Sphinx, SP1, RiscZero, and Lurk, so the benchmark would measure the shared precompile rather than the systems, and only Jolt, whose arithmetization differs, could set itself apart.

The redesign was motivated by a measured gap on this shared workload. In February 2024 the Lurk 1.0 prover ran about 55 times slower than the SP1 alpha on the same Fibonacci benchmark [4], which prompted the move to a STARK backend and to the lookup-based architecture of this section. Lurk 1.0 took about 20 s to prove the 100-th Fibonacci number with this program (§ 3.4, measured on different hardware), while Lurk 2.0 proves the 10^5-th Fibonacci number in 19.4 s, in the same class as SP1 at 18.6 s. The two generations prove the same recursive program, but Lurk 2.0 handles an input one thousand times larger in the same proving time.

[14] The Lite configuration (configuration A in the report) sets `SHARD_SIZE=1048576`, `SHARD_CHUNKING_MULTIPLIER=1`, `SHARD_BATCH_SIZE=0`, and `RECONSTRUCT_COMMITMENTS=false`. The Turbo configuration (configuration B) sets `SHARD_SIZE=4194304` and `SHARD_CHUNKING_MULTIPLIER=32`, leaving the other two unchanged.

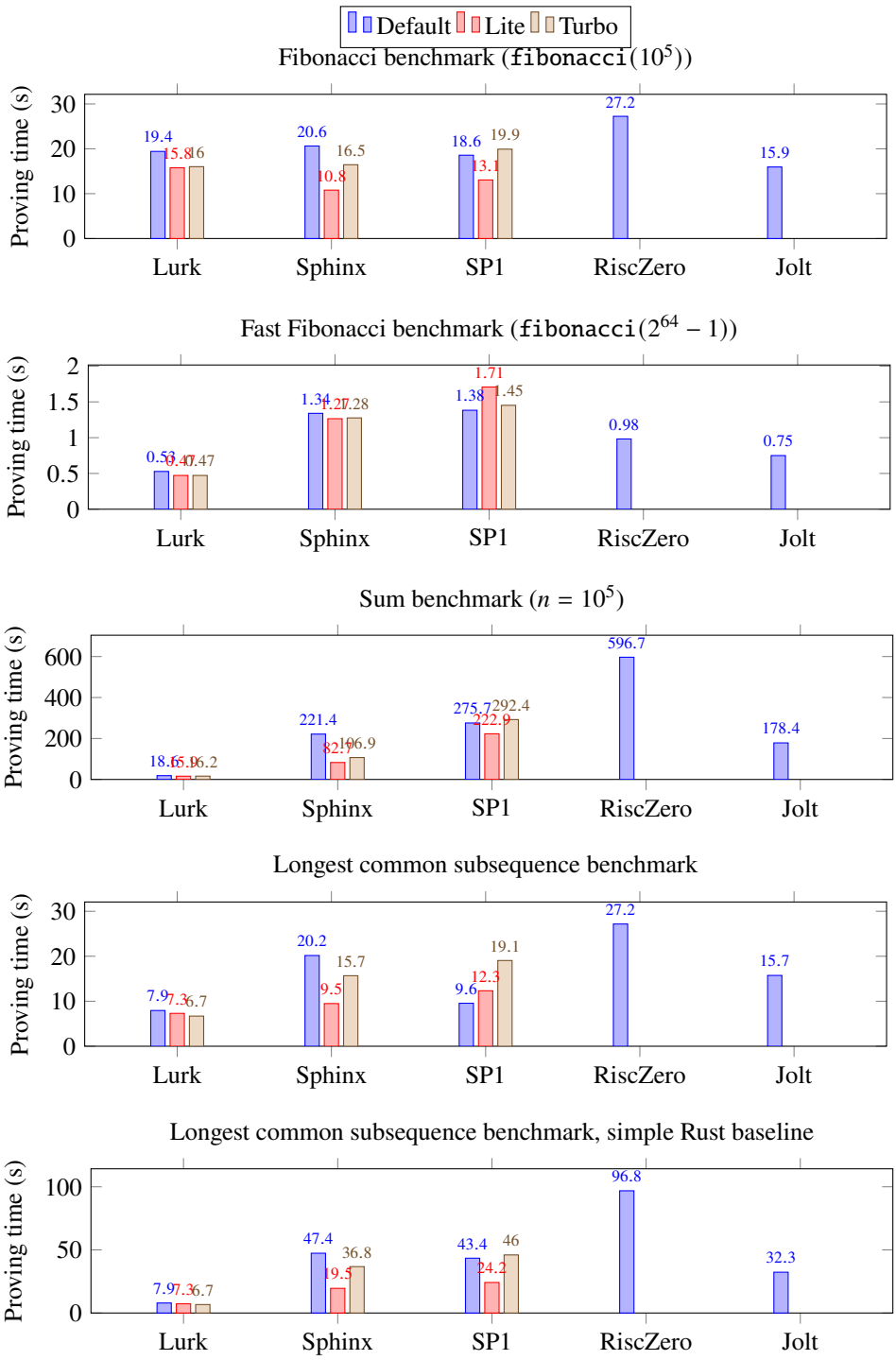


Fig. 5. Proving time comparison for the four workloads of § 6.10 (lower is better). The Default, Lite, and Turbo bars are the prover configurations described there. Data from the Lurk 0.5 benchmark report [4].

Lurk program:

```
(letrec ((fib
          (lambda (n)
            (if (<= n 1) n
              (+ (fib (- n 1)) (fib (- (- n 1) 1)))))))
  (fib 100000))
```

Rust program (Sphinx):

```
pub fn main() {
    let n: u64 = sphinx_zkvm::io::read::<u64>();
    sphinx_zkvm::io::commit(&n);
    // Compute the n'th fibonacci number, using normal Rust code.
    let mut a: u64 = 0;
    let mut b: u64 = 1;
    for _ in 0..n {
        // Naturally overflow at 64 bits
        let c: u64 = a.wrapping_add(b);
        a = b;
        b = c;
    }
    sphinx_zkvm::io::commit(&a);
}
```

Fig. 6. The two implementations of the Fibonacci benchmark, quoted from `lurk/fib/src/main.rs` and `sphinx/fib/program/src/main.rs` of the benchmark repository [4]. The Lurk program is the natural Lisp transcription of the algorithm; the Rust program is idiomatic Rust, the high-level language the RISC-V systems compile.

Workload	System	Setup (s)	Prove (s)	Verify (s)	Cycles	Memory (GiB)
Fibonacci	Lurk	0.004	19.418	0.235	1,000,003	12.149
	Sphinx	2.809	20.625	0.208	1,501,090	12.887
	SP1	3.899	18.572	0.264	1,748,347	12.643
	RiscZero	0.000	27.243	0.0268	1,702,759	8.414
	Jolt	5.713	15.948	0.378	1,500,191	29.604
Fast Fibonacci	Lurk	0.005	0.528	0.290	4,821	0.211
	Sphinx	2.790	1.340	0.196	7,387	2.301
	SP1	3.913	1.383	0.249	11,245	3.278
	RiscZero	0.000	0.982	0.011	9,379	0.708
	Jolt	5.609	0.750	0.108	6,549	0.632
Sum	Lurk	1.938	18.612	0.237	700,008	12.474
	Sphinx	2.882	221.433	0.581	17,951,154	55.584
	SP1	3.989	275.728	2.295	71,630,120	319.32
	RiscZero	0.000	596.713	0.567	38,965,402	8.456
	Jolt	0.959	178.446	13.118	3,884,636	192.89
LCS (tabulated baseline)	Lurk	0.005	7.943	0.238	359,759	5.432
	Sphinx	2.790	20.168	0.207	1,060,217	12.697
	SP1	3.954	9.553	0.264	1,199,115	12.696
	RiscZero	0.000	27.168	0.026	1,252,841	8.432
	Jolt	7.866	15.741	0.454	1,088,186	32.502
LCS (simple baseline)	Lurk	0.005	7.943	0.238	359,759	5.432
	Sphinx	2.829	47.354	0.209	3,622,427	26.966
	SP1	3.958	43.420	0.386	5,033,647	25.210
	RiscZero	0.000	96.829	0.101	4,773,346	8.434
	Jolt	0.967	32.335	0.698	3,566,186	65.846

Table 2. Full measurements behind Fig. 5 for the Default configuration; lower is better in every column. Cycle counts have per-system meanings: Lurk counts calls to its evaluation function, Sphinx and SP1 report RISC-V cycles at the end of proving, RiscZero reports user cycles, and Jolt reports its trace length. The Lurk rows repeat across the two longest common subsequence baselines because the same Lurk program ran against both. Data from the Lurk 0.5 benchmark report [4].

6.11 Summary: modular, high-performance, and auditable zkVMs

Altogether, Lurk 2.0 points in a new direction for general-purpose zero-knowledge computation. By replacing monolithic universal circuits and recursive folds with a modular, small-field STARK-based architecture, unifying evaluation and constraint generation in LAIR, mediating all proof dependencies with a cryptographic lookup bus using nonce and depth arguments, and structuring computation as a memoized proof DAG, the system achieves both substantial performance improvements and new expressiveness. The architectural shift does introduce a compositional Trusted Computing Base: whereas Lurk 1.0's TCB comprised a single universal circuit plus the folding protocol, Lurk 2.0's TCB includes each specialized chip's LAIR bytecode, the LogUp lookup argument, the nonce-counter overflow prevention mechanism, the depth-based acyclicity enforcement, and the underlying small-field STARK security assumptions. While this represents more components, each is simpler, more focused, and more amenable to formal analysis than the monolithic alternative. The net effect is a system that trades a single large trust assumption for a collection of smaller, independently auditable components—a trade-off that aligns with modern secure system design principles. These innovations, and the design patterns they embody, point toward a scalable, transparent, and formally auditable standard for future zero-knowledge VMs.

7 Discussion

In this section, we reflect on Lurk's evolution in the broader context of zero-knowledge virtual machines (zkVMs), and discuss future directions for the language and its verification stack. While earlier iterations of Lurk (1.0) centered around a monolithic, universal circuit model, the architectural advances realized in Lurk 2.0 allow us to reconsider prior limitations not only for Lurk, but for zkVMs in general.

7.1 Related work

As described in Section 6, recent years have seen rapid development in the implementation of zkVMs, both in academic work and industrial efforts. Early systems approached the general-purpose proof problem by simulating a simple CPU and proving correct memory and register updates, with SNARKs of [9, 10] and, more recently, STARKs underpinning the constraint systems. At the same time, the blockchain community motivated the need for succinct, verifiable program execution as an alternative to redundant validator re-execution (see [14, 15, 32, 47, 51, 52, 61]), driving widespread zkVM adoption.

Lurk 2.0 fully adopts a zkVM structure, but is distinguished by an explicit choice of input language: a high-level, functional Lisp dialect with code-data equivalence, whose reduction semantics are specified at the language level (see [42]). This choice enables Lurk to leverage strong, language-level invariants about data structures, evaluation order, and functional purity, translating these into an efficient segmentation of computation. Each subterm in a Lurk program naturally maps to its own proof fragment, and the lookup argument ensures that dependencies between these fragments are cryptographically enforced and efficiently memoized.

This approach can be seen as navigating a "Goldilocks" path: unlike microprocessor-emulator zkVMs that work at too low a level (paying the cost of simulating hardware instructions), Lurk 2.0 expresses iteration and dependencies at the level of a high-level language's reduction semantics. The result is a system that is true to the original spirit of

Lurk: programming the proof system via language evaluation, while benefiting from the performance, modularity, and auditability of modern zkVM techniques.

7.2 Future work

Formal verification. By structuring proof generation around the evaluation of a reduction semantics specified in a high-level language, Lurk opens promising new opportunities for formal verification. The closer alignment of program interpretation, witness trace, and proof constraints means that correctness of Lurk evaluations can, in principle, be reduced to checking properties of a more comprehensible specification, such as a CEK machine realization (see [1, 21, 27, 38]). In contrast to direct-compilation or monolithic circuit approaches, this explicit semantic layer allows leveraging powerful tools from programming language theory to reason about correctness, consistency, and security. We plan to contribute further to the formal study of Lurk's semantics and to develop mechanized verification tools at the LOAM and LAIR levels.

Language adoption. To address the practical barrier of Lisp adoption in communities that have coalesced around mainstream languages, the Lurk team has developed a JavaScript-like syntax surface ("LurkScript") that compiles to Lurk. The transpiler accepts a strict subset of JavaScript syntax—leveraging existing JS parsers for IDE support—but targets Lurk's functional, immutable evaluation semantics. This means that while developers can write familiar JS-style expressions and object literals, the underlying language remains purely functional: mutation is not supported, and control flow maps to Lurk's expression-oriented reduction model. The system is actively used for Lurk development, demonstrating its practical viability while maintaining clear semantic boundaries—LurkScript is best understood as a syntactic facade over Lurk, not a faithful JS implementation.

Looking forward, we plan to target general-purpose zkVMs via SP1 and RISC-V rather than bespoke backends. This "LISP-1" approach specializes LOAM's fully general lookup-oriented strategy to the simpler case of expression reduction: we extract a verifier from the interpreter, compress the witness through memoization and content addressing, then prove each reduction step as a small RISC-V program. Because reduction admits a natural topological ordering, we can sidestep Memoset's provenance tracking and rely instead on RISC-V memory and sequential verification. We anticipate releasing the full language stack—including the actor-model extensions (processes, message passing, and resource-based state) for LurkOS—in 2026.

Backend extensions. The shift from a monolithic circuit to a modular, STARK-based backend enables many new optimizations and architectures. A major open problem is the design and selection of abstract machines that faithfully represent high-level language semantics and are also optimal for zero-knowledge proof generation. The exploration of big-step vs small-step abstract machines, effect systems (such as the CESK machine for control operators of [26]), and novel IRs remains an active area of research. Lurk's experience underscores the importance of explicitly recognizing both the operational requirements of language evaluation and the particulars of the zero-knowledge proof activity. Bridging these domains, while maintaining auditability and efficiency, is a compelling challenge, and we hope Lurk will serve as a testbed for these ideas.

Proof compression. A trade-off of STARK-based proving systems is that they generate much larger proofs than corresponding SNARK-based systems. Modern zkVMs such as SP1 and RiscZero work around this issue by implementing a STARK proof verifier in a SNARK-based proving system (such as Groth16 [35]) that can then generate a corresponding proof of the verification of a larger proof, resulting in a much smaller total size at the cost of extra prover work. Lurk's verifier could adopt similar strategies to enable applications of Lurk proofs in domains where proof sizes must be minimal, such as on-chain verification in resource-constrained environments like Ethereum.

Proof-carrying data and distributed computation. Another direction of growth is toward proof-carrying data (PCD) of [16, 18], which extends IVC to computation graphs (beyond simple chains) and enables secure, audited distributed computation. Lurk's modular, inner proof structure offers a natural conceptual foundation for such applications. The integration of interoperability with diverse proof systems, recursive proof aggregation, and sound composition (inspired by modern work in PCD and privacy-preserving computation) is likely to be an area of both theoretical and practical importance for future Lurk development.

In summary, while the architecture of Lurk 2.0 builds on and incorporates the advances of modern zkVM design, its opinionated stance on language and semantics makes it well-positioned to exploit the strengths of both abstract machine theory and zero-knowledge proof engineering. We believe this synthesis will inform and inspire continued advances in both fields.

Acknowledgements

We would like to thank Matej Penciak, Arthur Paulino, Nicolas Ramsrud, and Thomas Chateignier for significant feedback and contributions to the design and engineering work that led to this paper.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. From Interpreter to Compiler and Virtual Machine: A Functional Derivation. *BRICS* 10, 14 (March 2003). doi:10.7146/brics.v10i14.21784

[2] Nada Amin, John Burnham, François Garillot, Rosario Gennaro, Chhi'mèd Künzang, Daniel Rogozin, and Cameron Wong. 2023. LURK: Lambda, the Ultimate Recursive Knowledge. Cryptology ePrint Archive, Paper 2023/369. doi:10.1145/3607839

[3] Nada Amin, John Burnham, François Garillot, Rosario Gennaro, Chhi'mèd Künzang, Daniel Rogozin, and Cameron Wong. 2023. LURK: Lambda, the Ultimate Recursive Knowledge (Experience Report). *Proc. ACM Program. Lang.* 7, ICFP, Article 197 (Aug. 2023), 16 pages. doi:10.1145/3607839

[4] Argument Computer Corporation. 2024. Lurk 0.5 Benchmarks. https://blog.lurk-lab.com/posts/perf-2024/.

[5] Suyash Bagad, Quang Dao, Yuval Domb, and Justin Thaler. 2025. Speeding Up Sum-Check Proving. Cryptology ePrint Archive, Paper 2025/1117. https://eprint.iacr.org/2025/1117

[6] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. 2022. Circom: A Circuit Description Language for Building Zero-knowledge Applications. *IEEE Transactions on Dependable and Secure Computing* (2022), 1–18. doi:10.1109/TDSC.2022.3232813

[7] Eli Ben-Sasson. 2021. ethSTARK Documentation. Cryptology ePrint Archive, Paper 2021/582. https://eprint.iacr.org/2021/582

[8] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046. https://eprint.iacr.org/2018/046

[9] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Scalable Zero Knowledge via Cycles of Elliptic Curves. In *Advances in Cryptology – CRYPTO 2014*, Juan A. Garay and Rosario Gennaro (Eds.).

Vol. 8617. Springer Berlin Heidelberg, Berlin, Heidelberg, 276–294. doi:10.1007/978-3-662-44381-1_16

[10] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, USA, 781–796.

[11] Daniel Benarroch, Kobi Gurkan, Ron Kahat, Aurélien Nicolas, and Eran Tromer. 2019. zkInterface, a Standard Tool for Zero-Knowledge Interoperability. In *2nd ZKProof Workshop*. https://docs.zkproof.org/pages/standards/accepted-workshop2/proposal--zk-interop-zkinterface.pdf

[12] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. 1991. Checking the Correctness of Memories. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science (SFCS '91)*. IEEE Computer Society, USA, 90–99. doi:10.1109/SFCS.1991.185352

[13] Dan Boneh, Wilson Nguyen, and Alex Ozdemir. 2021. Efficient Functional Commitments: How to Commit to a Private Function. Cryptology ePrint Archive, Paper 2021/1342. https://eprint.iacr.org/2021/1342

[14] Joseph Bonneau, Izaak Meckler, and Vanishree Rao. 2021. *Mina: Decentralized Cryptocurrency at Scale*. Technical Report.

[15] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. 2020. ZEXE: Enabling Decentralized Private Computation. In *2020 IEEE Symposium on Security and Privacy (SP)*. 947–964. doi:10.1109/SP40000.2020.00050

[16] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. 2020. Recursive Proof Composition from Accumulation Schemes. In *Theory of Cryptography (Lecture Notes in Computer Science)*, Rafael Pass and Krzysztof Pietrzak (Eds.). Springer International Publishing, Cham, 1–18. doi:10.1007/978-3-030-64378-2_1

[17] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. 2020. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *Advances in Cryptology – EUROCRYPT 2020 (Lecture Notes in Computer Science)*, Anne Canteaut and Yuval Ishai (Eds.). Springer International Publishing, Cham, 738–768. doi:10.1007/978-3-030-45721-1_26

[18] Alessandro Chiesa and Eran Tromer. 2010. Proof-Carrying Data and Hearsay Arguments from Signature Cards. In *Innovations in Computer Science*. Tsinghua University, Beijing, China, 310–331.

[19] Alessandro Chiesa and Eylon Yogev. 2024. *Building Cryptographic Proofs from Hash Functions*. https://github.com/hash-based-snargs-book

[20] Collin Chin, Howard Wu, Raymond Chu, Alessandro Coglio, Eric McCarthy, and Eric Smith. 2021. Leo: A Programming Language for Formally Verified, Zero-Knowledge Applications. In *4th ZKProof Workshop*. https://eprint.iacr.org/2021/651.pdf

[21] Olivier Danvy. 2008. Defunctionalized Interpreters for Programming Languages. *SIGPLAN Not.* 43, 9 (Sept. 2008), 131–142. doi:10.1145/1411203.1411206

[22] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28 (Lecture Notes in Computer Science)*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 625–635. doi:10.1007/978-3-030-79876-5_37

[23] Liam Eagen and Ulrich Haböck. 2024. Bypassing the Characteristic Bound in logUp. Cryptology ePrint Archive, Paper 2024/2067. https://eprint.iacr.org/2024/2067

[24] Liam Eagen, Sanket Kanjalkar, Tim Ruffing, and Jonas Nick. 2022. Bulletproofs++: Next Generation Confidential Transactions via Reciprocal Set Membership Arguments. Cryptology ePrint Archive, Paper 2022/510. doi:10.1007/978-3-031-58740-5_9 Published at EUROCRYPT 2024.

[25] Jacob Eberhardt and Stefan Tai. 2018. 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)*. 1084–1091. doi:10.1109/Cybermatics_2018.2018.00199

[26] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press.

[27] Matthias Felleisen and Daniel P. Friedman. 1986. Control Operators, the SECD-machine, and the λ-Calculus. In *Proceedings of The Conference on Formal Description of Programming Concepts*. Ebberup, Denmark.

[28] Ben Fisch, Joseph Bonneau, Juan Benet, and Nicola Greco. 2018. Proofs of replication using depth robust graphs. *Blockchain Protocol Analysis and Security Engineering* 2018 (2018).

[29] Ariel Gabizon and Zachary J. Williamson. 2020. Plookup: A Simplified Polynomial Protocol for Lookup Tables. Cryptology ePrint Archive, Paper 2020/315.

[30] Nicolas Gailly, Mary Maller, and Anca Nitulescu. 2022. SnarkPack: Practical SNARK Aggregation. In *Financial Cryptography and Data Security*, Ittay Eyal and Juan Garay (Eds.), Vol. 13411. Springer International Publishing, Cham, 203–229. doi:10.1007/978-3-031-18283-9_10

[31] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. 2013. Quadratic Span Programs and Succinct NIZKs without PCPs. In *Advances in Cryptology – EUROCRYPT 2013 (Lecture Notes in Computer Science)*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer, Berlin, Heidelberg, 626–645. doi:10.1007/978-3-642-38348-9_37

[32] Alex Gluchowski. 2021. Introducing zkSync. https://blog.matter-labs.io/introducing-zk-sync-the-missing-link-to-mass-adoption-of-ethereum-14c9cea83f58

[33] Eiichi Goto. 1974. *Monocopy and Associative Algorithms in an Extended LISP*. Information Science Laboratory 74–03. Tokyo, University of.

[34] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. POSEIDON: A New Hash Function for Zero-Knowledge Proof Systems. In *30th Usenix Security Symposium*. Virtual.

[35] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016 (Lecture Notes in Computer Science)*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer, Berlin, Heidelberg, 305–326. doi:10.1007/978-3-662-49896-5_11

[36] Ulrich Haböck. 2022. Multivariate Lookups Based on Logarithmic Derivatives. Cryptology ePrint Archive, Paper 2022/1530. https://eprint.iacr.org/2022/1530

[37] R. J. M. Hughes. 1982. Super-Combinators a New Implementation Method for Applicative Languages. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming (LFP '82)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/800068.802129

[38] Gilles Kahn. 1987. Natural Semantics. In *Symposium on Theoretical Aspects of Computer Science*. doi:10.1007/BFb0039592

[39] Abhiram Kothapalli and Srinath Setty. 2022. SuperNova: Proving Universal Machine Executions without Universal Circuits. Cryptology ePrint Archive, Paper 2022/1758. https://eprint.iacr.org/2022/1758.pdf

[40] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. 2022. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In *Advances in Cryptology – CRYPTO 2022 (Lecture Notes in Computer Science)*, Yevgeniy Dodis and Thomas Shrimpton (Eds.). Springer Nature Switzerland, Cham, 359–388. doi:10.1007/978-3-031-15985-5_13

[41] Benoît Libert, Somindu C. Ramanna, and Moti Yung. 2016. Functional Commitment Schemes: From Polynomial Commitments to Pairing-Based Accumulators from Simple Assumptions. In *43rd International Colloquium on Automata, Languages and Programming (ICALP 2016)*. doi:10.4230/LIPIcs.ICALP.2016.30

[42] John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM* 3, 4 (April 1960), 184–195. doi:10.1145/367177.367199

[43] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. 2014. Authenticated Data Structures, Generically. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. Association for Computing Machinery, New York, NY, USA, 411–423. doi:10.1145/2535838.2535851

[44] Wilson Nguyen and Srinath Setty. 2025. Neo: Lattice-based Folding Scheme for CCS over Small Fields and Pay-per-Bit Commitments. Cryptology ePrint Archive, Paper 2025/294. https://eprint.iacr.org/2025/294

[45] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. 2022. CirC: Compiler Infrastructure for Proof Systems, Software Verification, and More. In *2022 IEEE Symposium on Security and Privacy (SP)*. 2248–2266. doi:10.1109/SP46214.2022.9833782

[46] Gordon D. Plotkin. 2004. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.* 60-61 (2004), 17–139.

[47] Polygon. 2022. Polygon zkEVM. https://polygon.technology/solutions/polygon-zkevm

[48] Polygon Zero team. 2022. *Plonky2: Fast Recursive Arguments with PLONK and FRI*. Technical Report. 16 pages.

[49] John E Savage. 1998. *Models of computation*. Vol. 136. Addison-Wesley Reading, MA.

[50] Srinath Setty, Justin Thaler, and Riad Wahby. 2023. Unlocking the Lookup Singularity with Lasso. Cryptology ePrint Archive, Paper 2023/1216. https://eprint.iacr.org/2023/1216

[51] Starkware. 2021. StarkNet. https://starkware.co/starknet/

[52] Succinct Labs. 2024. Introducing SP1: A Performant, 100% Open-Source, Contributor-Friendly zkVM. https://blog.succinct.xyz/introducing-sp1/.

[53] Roberto Tamassia. 2003. Authenticated Data Structures. In *Algorithms - ESA 2003 (Lecture Notes in Computer Science)*, Giuseppe Di Battista and Uri Zwick (Eds.). Springer, Berlin, Heidelberg, 2–5. doi:10.1007/978-3-540-39658-1_2

[54] Justin Thaler. 2013. Time-Optimal Interactive Proofs for Circuit Evaluation. In *Advances in Cryptology – CRYPTO 2013 (Lecture Notes in Computer Science)*, Ran Canetti and Juan A. Garay (Eds.). Springer, Berlin, Heidelberg, 71–89. doi:10.1007/978-3-642-40084-1_5

[55] Justin Thaler. 2022. Proofs, arguments, and zero-knowledge. *Foundations and Trends® in Privacy and Security* 4, 2–4 (2022), 117–660. doi:10.1561/3300000030

[56] Paul Valiant. 2008. Incrementally Verifiable Computation or Proofs of Knowledge Imply Time/Space Efficiency. In *Theory of Cryptography (Lecture Notes in Computer Science)*, Ran Canetti (Ed.). Springer, Berlin, Heidelberg, 1–18. doi:10.1007/978-3-540-78524-8_1

[57] Mitchell Wand. 1982. Deriving Target Code as a Representation of Continuation Semantics. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982), 496–517. doi:10.1145/357172.357179

[58] Mitchell Wand. 1982. Semantics-Directed Machine Architecture. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*. Association for Computing Machinery, New York, NY, USA, 234–241. doi:10.1145/582153.582179

[59] Mitchell Wand. 1983. Loops in Combinator-Based Compilers. In *The 10th ACM SIGACT-SIGPLAN Symposium*. ACM Press, Austin, Texas, 190–196. doi:10.1145/567067.567086

[60] Hongbo Wen, Jon Stephens, Yanju Chen, Kostas Ferles, Shankara Pailoor, Kyle Charbonnet, Isil Dillig, and Yu Feng. 2023. Practical Security Analysis of Zero-Knowledge Proof Circuits. The accepted version of USENIX Security '24.

[61] Ye Zhang. 2019. Scroll Overview. https://scroll.io/blog/zkEVM