LURK: Lambda, the Ultimate Recursive Knowledge (Experience Report)

NADA AMIN, Harvard University, USA
JOHN BURNHAM, Lurk Lab, USA
FRANÇOIS GARILLOT, Lurk Lab, Canada
ROSARIO GENNARO*, The City College of New York, USA
CHHI'MÈD KÜNZANG, Lurk Lab, USA
DANIEL ROGOZIN, University College London, UK
CAMERON WONG, Harvard University, USA

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

CCS Concepts: • **Software and its engineering → Functional languages**.

Additional Key Words and Phrases: Lisp, Rust

ACM Reference Format:
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 (August 2023), 16 pages. https://doi.org/10.1145/3607839

1 INTRODUCTION

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

Succinct Non-Interactive Arguments of Knowledge (or SNARKs) [Ben-Sasson et al. 2014b] have been a very exciting area of research in the last decade: a SNARK allows one party (the *prover*) to

*work done while at Protocol Labs

Authors' addresses: Nada Amin, namin@seas.harvard.edu, Harvard University, USA; John Burnham, john@lurk-lab.com, Lurk Lab, USA; François Garillot, francois@lurk-lab.com, Lurk Lab, Canada; Rosario Gennaro, rosario@ccny.cuny.edu, The City College of New York, USA; Chhi'mèd Künzang, clwk@lurk-lab.com, Lurk Lab, USA; Daniel Rogozin, d.rogozin@ucl.ac.uk, University College London, UK; Cameron Wong, cwong@g.harvard.edu, Harvard University, USA.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/8-ART197
https://doi.org/10.1145/3607839

prove to another party (the *verifier*) that a certain computation F has been performed correctly. Specifically, it allows the verifier to prove the existence of a *witness* w such that $y = F(x, w)$ for a publicly known input x. It is easy to see that the above examples can be cast in this framework.

One such proof is the *witness* w itself, with the verification procedure being to simply recompute the function F. The crucial property of SNARKs, however, is that they produce a proof 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**[1], a new LISP-based programming language which automatically constructs zk-SNARKs for arbitrary programs, avoiding ad-hoc compilation of programs into flat circuits – a process which imposes serious constraints on the size and complexity of computations that can be achieved in practice. Although zk-SNARKs theoretically enable applications like those described above, the possibility of deploying them has so far been impeded by the lack of a practical and general language stack. One author conceived of Lurk after his experience implementing the Filecoin proofs [Fisch et al. 2018][2], which consist largely of Merkle-inclusion proofs at scale.

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

Lurk solves this problem by integrating 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 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 McCarthy's original Lisp paper [McCarthy 1960]. 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. It can be directly proved with a single universal SNARK circuit iterated by a suitable recursive proving system (e.g. Nova)

[1] https://github.com/lurk-lang/lurk-rs
[2] https://github.com/filecoin-project/rust-fil-proofs

2 CRYPTOGRAPHIC BACKGROUND

2.1 Arithmetization

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

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

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

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

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

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

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 [Benarroch et al. 2019]. A cryptographic backend which given a suitable arithmetization of the function F builds a cryptographic proof of the correctness of the computation (the QSP work and its improvement by Groth [2016] are examples of cryptographic backends for R1CS).

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

3 GENERALIZING CIRCUIT COMPILATION

3.1 The Drawbacks of Direct Compilation

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

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

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

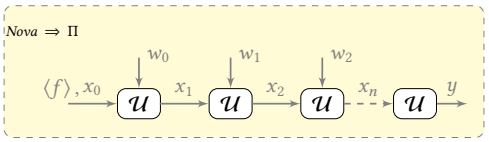
Since it only considers a fragment of the program at any given proving step, the approach of IVC bypasses these limitations, albeit at the expense of requiring the source program to reflect the transition function of an abstract state machine. Unfortunately, not all programs natively present such an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

3.2 A Generalized Approach to Design Proof Languages with IVC

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

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

(a)



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

Let's now give some intuition of the reduction: the transition rules of the CEK machine are all syntax-directed, and hence mutually exclusive if we pattern-match them accurately. For instance,

let's consider the *variable* rule, in the sub-case where the sought variable isn't at the head of the environment. In a textbook elaboration of the CEK machine to a Lisp-like language with lists, with x, y ranging over symbols, E over environments, K over continuations, and V over values, it would look like:

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

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

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

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

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

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

To elaborate on the last point, we note the full computation is not part of the input of any specific step of proof. Rather, Lurk defines hash-consing along the structure of each term of the language [Goto 1974], and pervasively uses domain-separated hash pointers for redexes of the source language. We will detail this in section 4.3. Those hashes, performed with a cryptographically-secure and field-algebra-friendly hash function [Grassi et al. 2021], embody the notion of a cryptographic commitment, so that a proof starts with just a public commitment to the whole program, and the prover progressively produces the required fragments of the computation as witnesses during the proof creation process[3].

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.

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

Finally, since the application of the Lurk circuit provides proofs of individual evaluation steps, it is the responsibility of a cryptographic backend to link them together in a coherent proof relating the whole sequence of statements: besides the recursive Nova proof system [Kothapalli et al. 2022], Lurk is general enough to also support an aggregative backend called Snarkpack [Gailly et al. 2022], used in legacy applications. We have fully implemented and open-sourced the Lurk compiler[4], 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). We provide an engineer's view of the lifecycle of a Lurk proof in Fig. 2.

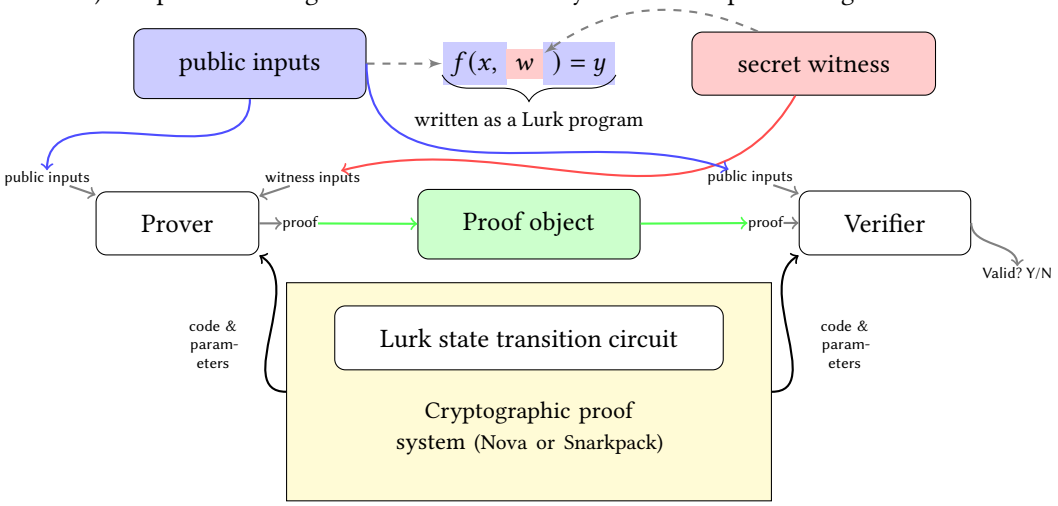


Fig. 2. Lifecycle of a Lurk proof : the cryptographic backend is equipped with Lurk's universal state transition circuit and generates prover & verifier programs of correct Lurk evaluation, as described in Fig. 1. The choice of portions of the program that are kept private is up to the user, see section 4.4 for a choice different from the one exemplified here.

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, [5] 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. [6]

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] https://github.com/lurk-lab/lurk-rs

[5] Lurk 1.0α allows augmenting the reduction engine with custom circuits for performance-sensitive functions.

[6] The benchmark machine was an AMD Ryzen Threadripper 3970X 32-Core Processor with 256GB RAM and an Nvidia Geforce 2080Ti GPU. These results are preliminary; the CPUs were under-utilized and the GPU was not fully exploited. During the 4-week review period, we engineered a significant raw proving throughput increase, indicating that better performance is to be expected: we do not consider comparison with similar platforms meaningful at this stage. Benchmark source: https://github.com/lurk-lab/lurk-rs/tree/icfp-benchmarks.

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

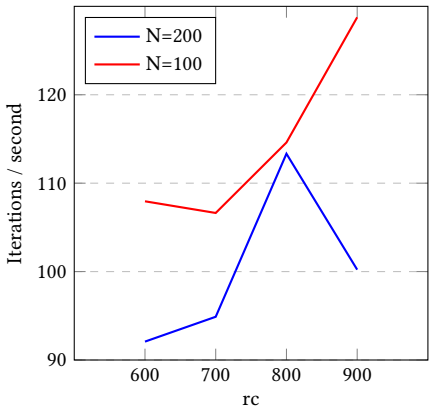


Fig. 3. Benchmark Results: Iterations per second vs rc for different Fibonacci terms alongside raw data.

4.1 Lurk is a Lisp

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

```
3
```

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

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

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

4.2 Looping and Branching

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

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

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

```
  (collatz 27))
```

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

4.3 Commitments

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

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

```
(commit 123)
```

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

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

```
(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 [Grassi et al. 2021] – it is computationally intractable to discover a second preimage to the digest represented by a commitment. For this reason, a commitment can be viewed as an index into a write-once store, such that all uses of the same commitment represent the same underlying value. This property is known as computational binding.

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

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

```
(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 Libert et al. [2016] and extended to *function-hiding* commitments by Boneh et al. [2021], enable a prover to commit to a secret function f and later prove that $y = f(x)$ for public y and x without revealing any other information about f. In Lurk, this notion enjoys native support in the language.

```
(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 [Boneh et al. 2021] could only posit an R1CS description of a function f, and modified commitments built from the indexed relations used in holographic proof systems [Chiesa et al. 2020] to convey both a commitment to the defining binary relation R_f of f, and a proof that R_f is total and univalent.

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

```
((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 $c = \mathtt{0x1aec2d8c}, x = 9, y = 88$ 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.

[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.

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

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

```
(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
```

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 [de Moura and Ullrich 2021] type-checker, 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:

```
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.

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

6 DISCUSSION

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

6.1 Related Work

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

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

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

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

6.2 Future Work

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

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

Backend Extensions. At its core Lurk is an interpreter based on the small-step CEK abstract machine. While this offers the advantage of simplicity, the requirement of the cryptographic interface is more loosely defined as an abstract machine with deterministic transitions. This leaves open the exploration of a big-step abstract machine [Danvy 2008] reducing the number of evaluation steps, or

that of other abstract machines, such as the CESK machine should we want to extend the language with e.g. control effects [Felleisen et al. 2009].

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

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

ACKNOWLEDGMENTS

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

REFERENCES

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). https://doi.org/10.7146/brics.v10i14.21784

Andrew W. Appel. 1991. *Compiling with Continuations.* Cambridge University Press, Cambridge. https://doi.org/10.1017/CBO9780511609619

Jean-Philippe Aumasson. 2022. The Security of ZKP projects: same but different. ZK Summit 7 workshop. https://www.aumasson.jp/data/talks/zksec_zk7.pdf

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. https://doi.org/10.1109/TDSC.2022.3232813

Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014a. 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. https://doi.org/10.1007/978-3-662-44381-1_16

Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014b. 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.

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

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

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

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. https://doi.org/10.1109/SP40000.2020.00050

Jeremy Bruestle, Paul Gafni, and RiscZero team. 2023. RISC Zero zkVM: Scalable, Transparent Arguments of RISC-V Integrity. Technical Report. https://www.risczero.com/proof-system-in-detail.pdf

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. https://doi.org/10.1007/978-3-030-64378-2_1

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. https://doi.org/10.1007/978-3-030-45721-1_26

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.

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

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

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. https://doi.org/10.1007/978-3-030-79876-5_37

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. https://doi.org/10.1109/Cybermatics_2018.2018.00199

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

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.

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).

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. https://doi.org/10.1007/978-3-031-18283-9_10

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. https://doi.org/10.1007/978-3-642-38348-9_37

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

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

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.

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. https://doi.org/10.1007/978-3-662-49896-5_11

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. https://doi.org/10.1145/800068.802129

Gilles Kahn. 1987. Natural Semantics. In *Symposium on Theoretical Aspects of Computer Science*.

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

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. https://doi.org/10.1007/978-3-031-15985-5_13

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)*.

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

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. https://doi.org/10.1109/SP46214.2022.9833782

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

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

Starkware. 2021. StarkNet. https://starkware.co/starknet/

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. https://doi.org/10.1007/978-3-642-40084-1_5

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. https://doi.org/10.1007/978-3-540-78524-8_1

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

Mitchell Wand. 1982b. 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. https://doi.org/10.1145/582153.582179

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

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

Received 2023-03-01; accepted 2023-06-27