Proofbase: A Proof-Based Blockchain for Low-Latency, Scalable, Multichain, Zero-Knowledge Applications

Lurk Lab Systems

Febuary 7, 2025

Low-latency, trustless, and cost-efficient validation of zero-knowledge proofs unlocks unbounded innovation in privacy, functionality, user-experience, and interconnectivity for blockchain-based applications. Blockchains today impose application design limitations: expensive on-chain compute, indiscriminately public processed information, and the ability to reason only about their own state. However, although zero-knowledge proofs are widely accepted as the solution, the complexity, cost, and rapid evolution of zero-knowledge protocols compounded by the limitations of today's blockchains make integration and deployment challenging.

We propose a new sovereign proof-of-stake Layer-2 network, called Proofbase with native support for Lurk, a safe, performant, and upgradable zero-knowledge abstract machine, compatible with many high-level functional languages, including JavaScript, Lisp, and OCaml.

Proofbase serves as a universal settlement layer for proofs, being proving system-agnostic, while supporting Lurk as its recommended framework. With Lurk, a high-performance programming language specifically designed to efficiently generate interrelated proofs, Proofbase provides a fast and cost-effective solution comparable to a smart contract language for proof generation.

To enable scalable, low-latency, private state applications, Proofbase employs a sharded state actor-based network model that allows the asynchronous progression of single-owner proofchains without requiring global consensus. The result is a network topology that neatly aligns with proof-based application design to provide a platform for highly scalable, low-latency private applications.

Unlike other blockchain layers, Proofbase does not require asset migration or associated infrastructure. Instead, it acts as a command-and-control layer for Web3, issuing proof-based certificates that facilitate asset movement on other chains. This approach ensures the consistent and secure management of the interconnected state histories that justify each cross-chain action, with Proofbase operating as a high performance stateful verification layer for multichain settlement.

In this paper, we also reintroduce Lurk not only as a minimalistic and secure model for proof generation but also as a polyglot framework that provides the tools necessary to reason about privacy and concurrency for application design. With adaptability across programming languages, Lurk surpasses traditional ISA-based zkVMs in both security and performance and can be reimplemented on any cryptographic prover, making it the best safe and futureproof approach to decentralized proof generation.

Contents

The Problem With Proofs 3
Lurk: The Optimal Zero-Knowledge Language Stack 5
Content-Addressed Code And Data 6
Abstract Machine: an Optimal Model 7
Loam & Memoset: An Optimal Lookup-Centric Architecture 7
Implementation & Portability 8
Performance 9
Irreproachability & Formalization 9
The Problem With Using Proofs 11
Fragmentation 12
Cost 13
State Management 13
Proofbase: A Proof-Based Asynchronous Blockchain 14
Proof Certificates: Indirect Versus Direct Proof Verification 17
Why a ZK Rollup: Bootstrapping Meaningful Security 18
Why the Proofbase Protocol is a Good Fit for Proof Verification 19
Manipulating State with Proofs 19
Proofbase's Ad-hoc Sequencing Service 20
The Proofbase Protocol Allows Pricing Transaction Storage Differently 21
Conclusion: Applications of Proofbase 22
Use Cases 22

The Problem With Proofs

Zero-knowledge proof technology has the potential to revolutionize the blockchain application design space and user experience. It allows developers to create offchain application logic of any complexity while providing privacy to users at the cost of adding proof verification to their onchain application smart contract. This potential is widely recognized, which begs the question: Why don't we see ZK proofs used widely today for onchain applications? Simply put, it boils down to the difficulty in writing and deploying proof-based applications, which has kept the barrier of use out of reach for most developers and application types.

There are 3 primary problems to solve in the domain of proofs:

- Proofs are hard to write
- Proofs are hard to agree on
- Proofs are hard to make safe

The first point has been the primary focus of many in the space. The creation of many "zkDSL"s has sought to lower the entry barrier for developers by abstracting domain-specific concepts behind common languages and semantic practices. The core issue of this is that it requires a buy-in to both the language and the particular back-end, as well as the prover stack that supports it. As cryptographic proving systems seem to be on a progress curve not unlike hardware progression in the heyday of Moore's law, these backends quickly fall behind the state-of-the-art wave.[1]

[1] Another issue is that those DSLs do not yield the intuitive runtime characteristics of a regular program: circuits represent the trace of a computation, rather than the computation itself. In other terms, circuits fit within a non-uniform model of computation [Savage, 1998], and to compute the unbounded set of functions computed by a Turing machine, a family of circuits is needed. The consequences of this mismatch are explored in the Lurk paper [Amin et al., 2023].

To solve the language issue, others have built general-purpose zkVMs which model CPU architectures, such that users can compile commonly used languages into an assembly language consumable and provable by this simulated machine. This enables generality and customizability in the proving system to obtain lowest barriers for onboarding users. Where those zkVMs allow defining custom "precompiles", this also offers a manageable overhead for performance-critical operations.

ISA	VM
RISC-V	Jolt, RiscZero, Succinct SP1, Nexus
MIPS	01 Labs 01VM, ZKM
WASM	Polygon/Near zkWASM, Delphinus zkWASM
custom CPU-style	Cairo, Valida, zkSync Era, Polygon Miden

Table 1: ISAs supported by various zkVM projects

This has the benefit in that it reduces the requirement on users to learn new languages and programming paradigms but creates subsequent critical issues.

First, modeling a CPU requires designing proving systems to emulate the entire instruction set, including the hardware architecture. As modern CPUs become more complex, their instruction sets expand to allow developers to leverage the underlying hardware features. However, concrete performance of zkVM proof systems directly depends on the complexity of modeling these instructions. For example, register and memory accesses represent distinct hardware paths with different performance profiles. Proving systems that emulate this distinction, by separately proving operand storage in registers or memory, may introduce unnecessary complexity, especially when the relevance of these minutiae to the end-user's proof is questionable. Moreover, this approach undermines the role of the compiler. The performance characteristics of cryptographic hardware emulation diverge significantly from those of the physical hardware being modeled, making it uncertain whether traditional hardware-specific optimizations will carry over effectively to the zkVM. This disconnect complicates auditing, formal verification, and the assurance of prover correctness, all of which are critical for security-sensitive applications. Even as methods to reduce instruction costs improve, hardware emulation will remain fundamentally ill-suited for proof generation.

Second, a proof alone provides little insight into its validity. For a verifier to trust the result, the proof must be verified within the context of the statement it supports. In many SNARK systems, this involves generating a verifier key during a preprocessing step, which serves as a cryptographic commitment to the program's statement. The verifier must be able to independently reproduce this key to confirm that the proof corresponds to the original program.

However, in zkVMs that use CPU architectures, the process is more complex. These systems depend on the correctness of the compiled bytecode *as well as* the CPU's execution, which can introduce a gap in verifiability. The verifier must not only *reproduce the compilation process*, but also *ensure that the compiler has correctly translated the high-level code into the low-level representation*. This makes safe and reproducible compilation essential across the entire tool chain, from the high-level source code to the final proof.

Common compilers, such as LLVM, are optimized for hardware performance but do not prioritize reproducibility and have frequent correctness gaps [Lopes et al., 2015, Zhou et al., 2021]. These long-lived compilation bugs[2] are extremely hard to detect in the proving context: the point of zero-knowledge proofs is often to hide the inputs and outputs that would allow us to sanity-check the compiled code. On the reproducibility front, factors like the target OS or even the compiler user's home path can influence the compiled output, making it difficult to consistently reproduce binaries without a controlled environment. Ensuring that a compiler produces correct and reproducible code is a significant challenge, particularly when modern compilers are not designed for formal verification. Certified compilers [3] can offer stronger assurances of correctness but are often slower and less accessible[4], and they typically do not address

[2] On average [...] developers take 11.16 months for GCC and 13.55 months for LLVM to fix an optimization bug.
(Zhou et al. [2021])

[3] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016

[4] Compcert is licensed for non-commercial use only.

Table 2: Comparison of VM Models

	Abstract machine based VM (Lurk)	**ISA-based VM**	**Compilation to circuit**
Trusted Computing Base [a]	Abstract machine	Compiler + ISA VM	Circuit compiler
User interface	Specific language (Lurk)	High-level language compilable to the ISA	Specific DSL (Noir, Circom, Leo)
Perf. overhead [b]	Medium	High	Low
Developer productivity	High	High	Low

[a] Cryptographic prover excepted. [b] Precompiles excepted.

nondeterminism in the compilation process.

In the context of zkVMs, ensuring that the low-level assembly matches the high-level program's intent is critical. The verifier must reproduce the prover's verification key to confirm the proof's accuracy, which requires a reliable and reproducible compilation process. This is complicated by variations in compiler behavior [Bjäreholt, 2017], configurations, and environmental factors, all of which can affect the final binary.

Addressing these challenges involves using tools for formal verification and reproducibility, such as certified compilers [Leroy et al., 2016], reproducible containerization [5], and advanced package managers [Batten et al., 2022]. Although these tools can help, they come with trade-offs, particularly between performance and reproducibility. Achieving reliable zkVMs requires careful management of the entire toolchain to ensure that the high-level code is faithfully represented in the final proof.

[5] Existing container technologies (like Docker) do not provide reproducibility: they are neither deterministic nor portable, as many details of the host OS and processor microarchitecture are directly visible inside the container.
(Navarro Leija et al. [2020])

Lurk: The Optimal Zero-Knowledge Language Stack

This section establishes the following key points:

- Lurk content-addresses data and programs, and proves the validity of program evaluations by directly processing the high-level program, making proof interpretation straightforward.
- Lurk implements a functional reduction machine, abstracting away hardware-specific behavior, a general, minimalistic model for programming language semantics. Consequently, Lurk can be extended to support other programming languages.
- Lurk maximizes the use of lookup arguments and minimizes arithmetization, enabling efficient proof implementation and facilitating lightweight VM reimplementation in different cryptographic proving systems.
- Lurk employs a systematic, rule-based approach to proof design, programmatically deriving proofs from clear rules, linking specification and implementation to enable auditability and formal verification.

Lurk presents a refined approach to zkVM design through its use of an abstract functional reduction machine, the CEK machine, which underpins the source language semantics. This abstract machine abstracts hardware-specific behavior to focus on the essential operations of the language [6]. Unlike hardware-based models, Lurk's abstract machine uses a minimal and consistent set of state transitions to describe program execution, making it hardware independent, simple to formalize, and portable across backends. The size and simplicity of the machine, consisting of only 12 rewrite rules, make it easy to reimplement, easy to audit, and efficient to optimize. These qualities are central to Lurk's auditability and support for formal verification.

Lurk's functional reduction model offers further advantages. Programs are interpreted directly as a cascade of function "reductions," constructing navigable data structures such as linked lists or trees of dependent expressions. This enables semantic correctness from code to evaluation, with a one-to-one relationship between the source program and its execution. The verifier can thus independently confirm the correctness of the execution by checking only the program's content, rather than relying on complex hardware instructions or multiple layers of translation. Lurk's abstract machine follows in the tradition of abstract machines, such as the SECD and CEK machines [Landin, 1964, Felleisen and Friedman, 1983], which have provided foundational models for evaluating functional programs.

[6] Nada Amin, John Burnham, François Garillot, Rosario Gennaro, Chhi'mèd Künzang, Daniel Rogozin, and Cameron Wong. Lurk: Lambda, the ultimate recursive knowledge (experience report). *Proc. ACM Program. Lang.*, 7(ICFP), aug 2023. DOI: 10.1145/3607839. URL `https://doi.org/10.1145/3607839`

Indeed, most functional programming languages are modeled using closely related minimalistic abstract machines, specific to their language (see table 3). These machines are not radically different, but they do reflect the particular semantics of each language, making the transition to new language models straightforward and efficient. Supporting a new language does not require building an entirely new abstract machine from scratch but rather extending the existing model to accommodate different high-level constructs. This modularity ensures that new languages can be integrated into Lurk with minimal overhead.

Lurk itself is implemented as a Turing-complete dialect of Lisp [7], chosen for its expressiveness, simplicity, and its well-established status in functional programming. In Lurk, as in Lisp, code and data are interchangeable, which enables seamless programmatic interaction and data sharing. The language's emphasis on composability means that proofs and outputs from one Lurk program can be reused as inputs in another, supporting the construction of complex, interrelated proof systems.

[7] Lisp was chosen as the initial target for its simplicity and core commonality with other functional languages and is not the final target language.

Content-Addressed Code And Data

Lurk introduces content addressing for both code and data, using cryptographically secure hash functions to create constant-sized commitments of 256 bits. These cryptographic commitments are integral to the language, enabling succinct zero-knowledge proofs while ensuring data privacy and interoperability. By allowing commitments to be blinded with secrets, Lurk provides robust mechanisms for securely managing private data, all while minimizing storage overhead for large or complex structures.

Language	Abstract Machine
Lisp	SECD [Landin, 1964], CEK [Felleisen and Friedman, 1983]
Ocaml	ZINC [Leroy, 1990]
Haskell	STG [Jones, 1992]
μLua	CESK variant [Midtgaard et al., 2013]
Prolog	WAM [Warren, 1983]
JavaScript	JAM [Van Horn and Might, 2011]
Erlang	BEAM [Hausman, 1994]
Coq	KAM variant [Coquand and Huet, 1986]

Table 3: Some programming languages and their abstract machine models

Abstract Machine: an Optimal Model

Lurk's abstract machine, the CEK machine, is a functional reduction machine that evaluates each input expression into a corresponding output expression in a predictable and deterministic manner. The machine's evaluation process is defined by a small set of reduction rules, which are applied iteratively until a fixed point is reached, ensuring that each function application is resolved. The simplicity of these rules makes the evaluation both efficient and easy to verify.

Moreover, this approach is a standard for authoritatively defining the semantics of functional languages. It follows in the lineage of abstract machines proposed by P.J. Landin [8], which introduced a hardware-independent framework to understand how programs evaluate. The CEK machine in Lurk builds upon this tradition, adapting the model to support zero-knowledge proofs and cryptographic commitments.[9]

One of the defining characteristics of Lurk's abstract machine is its composability. As a content-addressed functional expression language, Lurk naturally supports the composition of proofs, allowing complex systems of proofs to be constructed from smaller, independent components. This makes Lurk an ideal platform for building scalable and efficient zero-knowledge applications.

[8] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966. ISSN 0001-0782, 1557-7317. DOI: 10.1145/365230.365257

[9] *Why not develop an abstract machine for Python or Rust straight away?* Lurk's abstract machine supports Lisp-style functional reductions today.
In functional languages, abstract machines often define authoritative semantics by convention: If the implementation diverges, the machine's semantics prevail. For non-functional languages like Rust or Python, however, an abstract machine represents no such consensus, and must match the reference implementation through extensive conformance testing, maintained as the language evolves.
As Lurk evolves to support new languages, it builds on previous capabilities, making extending to complex languages in the future a simpler task, moving step by step from Lisp toward more intricate language constructs.

Loam & Memoset: An Optimal Lookup-Centric Architecture

Lurk embraces a lookup-centric approach at every level, aiming to transform arbitrary computer programs into circuits that perform mostly lookups[10]. Central to this architecture are two key components: the Lurk Algebraic Intermediate Representation (*Lair*) and the Lurk Ontological Abstract Machine (*Loam*).

Lair is a finite-field-oriented language that helps generate arithmetization, the low-level language of zero-knowledge (ZK) proofs, by translating high-level expressions into field element-based constraints. This arithmetization ensures that Lurk's performance remains competitive while providing a clear path for compiling programs to cryptographic backends.

[10] The lookup singularity seeks to transform arbitrary computer programs into "circuits" that only perform lookups. See [Whitehat, 2022]

Building on Datalog [Green et al., 2013], *Loam* is a rules description language that allows the decomposition of computations corresponding to complex relations into smaller atomic relations that comprise them. These atomic (or irreducible) relations correspond to steps in a proof and serve multiple purposes:

- They *require* that other steps be fulfilled elsewhere.
- They lay out constraints that must be verified for their own proof fragment to be meaningful (using Lair).

- They explicitly *provide* (support) a proof statement.

Loam encodes high-level programs as formal relations corresponding to the tuples provided and required by the Memoset lookup [Caparica et al., 2025].[11]

Not only is the Lurk VM itself implemented as a minimal weave of lookups implementing its reduction rules, but those lookups (in memoized form) become a feature accessible to user programs via implicit memoization[12]. Even the most naive implementation of the Fibonacci function achieves performance on the order of best-in-class:

```
!(defrec fib (lambda (n)
    (if (< n 2) 1 (+ (fib (- n 2))
        (fib (- n 1))))))
```

[11] Memoset generalizes lookup arguments to extend beyond shallow table or function evaluations. Its *provide* and *require* primitives precisely define semantics, allowing programs to instantiate finite instances of infinite virtual circuits.

[12] Memoset is so named because it allows transparent memoization of function calls, eliminating redundant computation. Memoset's provenance annotation ensures the soundness of arbitrary mutually recursive function calls.

Loam implements optimal, externalizable, functional memory and compiles high-level reduction rules to lookup rules using Memoset's *provide* and *require* primitives. Because Loam memory relies on Memoset, it radically generalizes the idea of immutable memory: Any pure function invocation can serve as an 'address'. Loam unifies this generalized memory with content-addressing so that purely functional data structures [13] can be externalized as Merkle DAGs when necessary, but represented internally as cheaply allocated and manipulated references.

[13] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, 1998. ISBN 978-0-521-63124-2. DOI: 10.1017/CBO9780511530104

Implementation & Portability

The Lurk Stack is designed to work with any modern backend proving system, leveraging the flexibility of both *Loam* and *Lair*. *Loam*, relying on Memoset and minimal arithmetization, can be compiled to pure lookups, requiring only a trivial subset of Lair's sophisticated arithmetization capabilities. However, when dense arithmetic or complex circuit-level optimizations are needed, *Lair* can be embedded within Loam programs and inlined into the chips that implement the leaf rules of the final Loam program.

Loam includes:

Data-definition specification which declaratively specifies the characteristics of Loam memory.

Data-transition specification which defines the provable transition rules for a Datalog program representing Lurk reduction.

Since the data-transition language is fully relational, it effectively simulates Lair's unidirectional functional semantics. Lair compiles to the data transition language and serves as the top-level program source, supplemented by data definition. This setup allows the compiler to adjust the arithmetization versus lookup trade-off based on the program's performance characteristics (see Performance below).

To port Lurk to a new backend, the Lurk team only needs to ensure:

- **Lair** extends to support the backend's novel arithmetization.
- **Memoset** is implemented efficiently, taking advantage of Lair where possible.
- **Loam Memory** operates through constrained memory chips that prevent prover equivocation.

Performance

A Lurk implementation, comprising the evaluator, witness generator, prover, and verifier, compiles a program specification into components that seamlessly interact with lookup (*Loam*) and arithmetization (*Lair*). This architecture allows heavily optimized coprocessor circuits to interact without the need for hand-authoring error-prone arithmetic circuits. While most circuits can translate into pure lookups, Lurk manages these lookups with sophistication.

However, decomposing dense arithmetization into lookups can become inefficient beyond a certain point. This tradeoff arises naturally from the cost model of virtual circuits[14].

[14] For example, matrix multiplication of statically-sized matrices will likely cost less if the arithmetic operations are inlined directly. NxN matrix multiplication should therefore be implemented as a primitive operator, even if invocations use lookups

Lurk supports efficient implementations of lookup relation providers through *Lair*, a uniform high-level expression language whose compilation pipeline requires auditing only once. When performance, development time, or audit confidence demand, Lurk programs can also wrap efficient circuits written in a more primitive arithmetization-specification language.

Lurk version 0.5 is the second implementation of Lurk[15].

[15] The latest implementation of Lurk is based on Sphinx, a small-field STARK prover [Ben-Sasson et al., 2018] equipped with a LogUp-based lookup argument [Eagen et al., 2022, Haböck, 2022], derived from a modified open-source zkVM project [Succinct Labs, 2024]

and includes a benchmark suite to compare the performance of Lurk with different proving systems. The top-line benchmark results are reproduced in figure 1 on the next page. Despite using a relatively simple prover implementation, these results already demonstrate that Lurk reduction is competitive with other provers in terms of prover time. Avoiding work modeling irrelevant hardware details takes less time.

Irreproachability & Formalization

Lurk's self-contained and deterministic design guarantees that every phase of the VM setup remains reproducible (see table 4 on page 11).

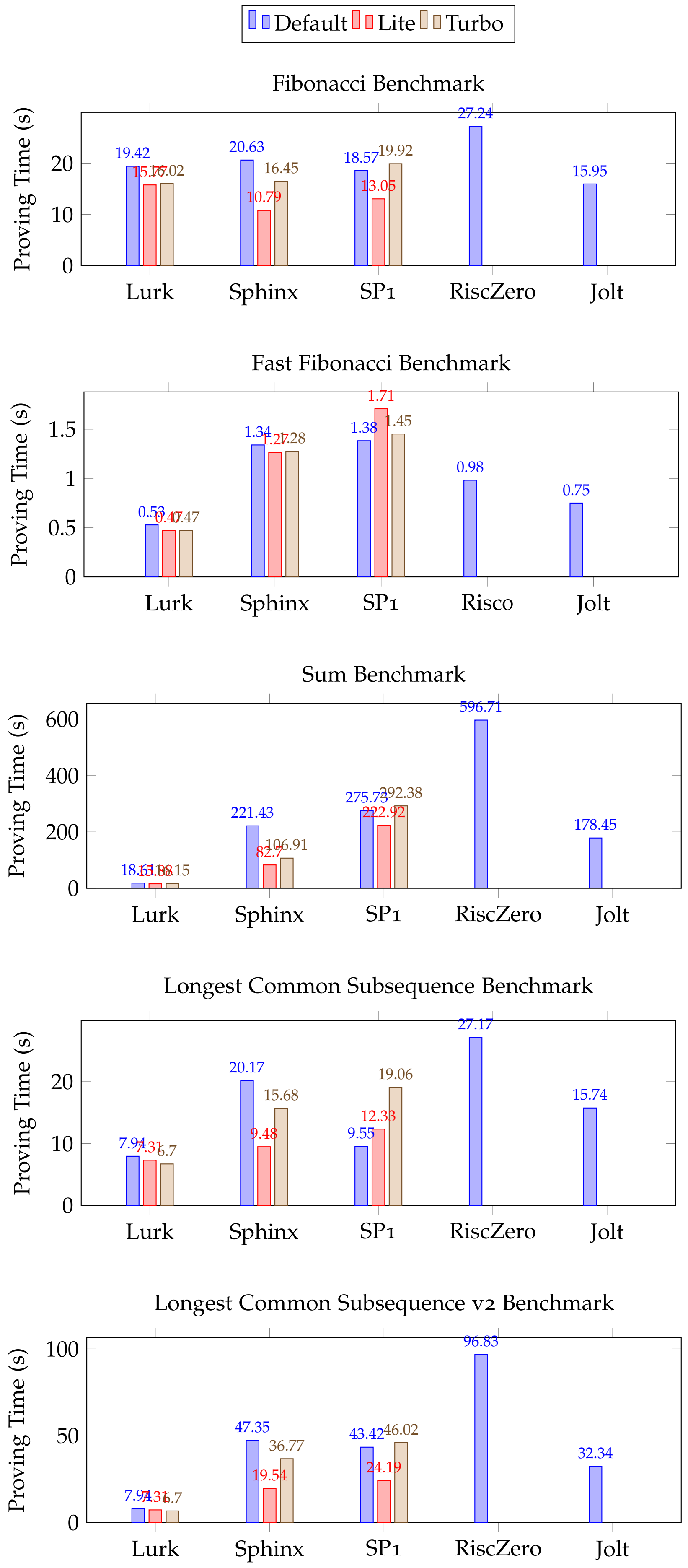

Figure 1: Proving time comparison for Lurk, Sphinx, SP1, RiscZero, and Jolt across four benchmarks. Benchmarks are detailed in the Lurk 0.5 benchmark blog post [Argument Computer Corporation, 2024]. In all benchmarks, lower is better. The Fibonacci benchmark is a simple iterative implementation of the Fibonacci sequence. The Fast Fibonacci benchmark is an implementation of the same as a 2x2 matrix multiplication. The Sum benchmarks a sum of 1000 integers passed as an input. In all longest common subsequence benchmarks, we measure against an implementation where the RISC-V based VMs implement memoization in the high-level source.

The benchmark parameters primarily manage shard size and memory-performance trade-offs. Both "Lite" and "Turbo" configurations disable memory-saving mechanisms to boost performance, storing intermediate data in memory instead of recalculating or writing to disk. This setup requires more memory but accelerates proof generation, as long as the prover has sufficient RAM.

The main difference lies in shard parallelism: "Lite" uses smaller, more numerous shards, increasing parallelism and prover efficiency but at the cost of higher verifier workload. "Turbo," with larger shards, reduces verifier costs and limits parallelism. Essentially, "Lite" trades higher verifier overhead for more parallelism, while "Turbo" optimizes for fewer shards with less impact on the verifier.

Audit req.	**Description**
Proving system	Although this falls outside the scope of the language itself, Lurk facilitates correct usage of the backend proving system through higher-level constructs like Loam and Lair, simplifying the audit process.
Program-specific circuit	Lurk simplifies this by breaking it down into an audit of the *language semantics* (without cryptographic components) and the cryptographic correctness of Loam, Lair, and MemoSet.
Proof statement	The specific proof statement associated with public inputs requires auditing. In Lurk, the program source and low-level representation are identical, so only the source needs auditing.
Provenance & correctness of Public inputs	This is a property of a system of interoperating application programs. Lurk's composability allows systems of interoperating programs to be treated as single programs, streamlining audits in environments like blockchains.

Table 4: Auditability requirements and their descriptions in Lurk.

Furthermore, because the VM directly proves Lurk programs, verifying the equivalence between the source program and the artifact used as VM input becomes trivial. Lurk can also adopt more efficient internal representations and implement a compilation pass. Importantly, these optimizations maintain irreproachability by treating them as provable transformations from the source, implemented as first-class operations within the VM.

Finally, formal verification provides the strongest guarantees of auditability and irreproachability. Lurk's design, grounded in well-defined and well-understood theory, makes it particularly suited to formalization. Each stage of the compiler pipeline leading to the prover and verifier, as well as individual application programs, remains self-contained and modular.

This simplicity allows not only one-time formalization, but it reduces the maintenance required when the language evolves, and supports incremental formalization for new backends. Lurk Lab values *strong proofs* and commits to eventually formalizing Lurk.

The Problem With Using Proofs

Even with an optimal proving stack, deploying proof-based applications as consumer products remains nontrivial, particularly in the context of blockchain-based applications. Blockchains, by design, are resource-constrained distributed systems, making applications slow, costly, and cumbersome. Zero-knowledge (ZK) proofs offer a mechanism to separate application state into onchain and offchain components, enabling the complex and expensive computations to happen offchain, with only the verification happening onchain, significantly reducing costs. This has made ZK a critical technology for scaling blockchains by adding verifiable execution layers, where the state is proven valid and settled on the Layer-1 blockchain. However, perhaps the larger opportunity lies in ZK enabling novel applications and UX in blockchain application design. Using ZK, new applications in privacy, compliance, AI and Web2 interaction, can verifiably interact with blockchains, pulling application UX away from the blockchains they are built upon while increasing the capabilities of these systems. This potential could massively amplify blockchain's role as society's permissionless, trustless financial layer of the Internet. Despite this potential, the realization of these opportunities remains limited.

Using proofs for blockchain applications faces three key challenges:

- Proving systems and verifiers are not standardized and are evolving rapidly,
- Verifying proofs onchain remains expensive,
- Applications require interaction with a chain-managed state.
- Synchronous blockchains limit the practical scalability and UX of proof-based applications.

Fragmentation

Proving systems have evolved rapidly over the last eight years, driving innovation in cryptography, system design, and applications. However, this rapid pace of iteration has often resulted in incompatible systems, requiring full-stack redesigns of verifiers. Blockchains, being slow to add functionality, have largely stuck with the 2016 State-of-the-Art: Groth16 proofs [Groth, 2016] over the BN254 pairing-based curve. The small proof size of Groth16 (256 bytes) remains attractive for resource-constrained blockchains.

Although modern proving systems have outpaced Groth16 in performance and usability, the need for Groth16 compatibility onchain neutralizes much of this progress. Modern proving systems, having diverged from R1CS [Benarroch et al., 2019] and elliptic curve-based commitments, require expensive "wrapping" of newer proofs to fit within Groth16, adding significant latency and stalling innovation in proof-based applications.[16]

[16] Additionally, many blockchains do not even support Groth16 verification, forcing developers to deeply understand various proving systems and decide which blockchain to target for building and verifying proofs.

Cost

Even when a blockchain supports the chosen verification method, the challenges persist. Verifying any ZK proof remains prohibitively expensive for many use cases, even though it is cheaper than executing applications directly onchain. For example, verifying a Groth16 proof on Ethereum or other blockchains typically incurs a cost in the range of tens of USD per instance, which is significantly more expensive than simple transactions.

This expense may not seem like a major issue at first glance, but many applications require frequent proof verifications as part of their regular user interactions. For instance, an application that needs to verify one proof every 5 minutes would face an annual cost exceeding a million dollars, a burden too high for many business uses. This cost barrier often makes proof-based applications commercially unviable, either forcing these costs onto users or requiring heavy subsidization by developers, stifling innovation in the space.

State Management

Most applications require managing a use case rather than executing a one-off transaction. This means they need to create a stream of valid states that represents the evolving state of the use case as reflected on the blockchain. For example, this could be the state of redemptions in an airdrop, successive block headers validated by a verifying bridge, or the board state in a game like chess or battleship.

This continuous stream of state updates cannot be handled effectively by stateless proof verification (such as ZK coprocessors) or recursive proofs. While recursive proofs allow building on top of existing proofs, they would require proving, with every interaction, that the new proof is the legitimate successor of the previous one, a task that is essentially the role of a blockchain: managing state sequencing.

The better approach is to allow users to manage the state directly on the blockchain. However, traditional blockchains treat the state as shared among all users, which complicates state management. If two users submit a proof of a valid state transition based on the same initial state, the second proof to be processed will be considered invalid because it conflicts with the earlier transition.[17]

[17] This issue creates severe limitations on proof-driven applications in some blockchains, which limit the state to just a few field elements worth of storage and require applications state update to be commutative, see `https://github.com/o1-labs/o1js/issues/265`.

To address this, we introduce a blockchain that eliminates this problem allowing for conflict-free state management while still leveraging the benefits of zero-knowledge proofs.

Proofbase: A Proof-Based Asynchronous Blockchain

We present Proofbase, a proof-based execution layer designed as a Proof-of-Stake (PoS) zero-knowledge (ZK) layer 2 blockchain built on Ethereum. Taking advantage of its lineage from the Linera and FastPay protocols [Baudet et al., 2020, Linera team, 2023], Proofbase offers several distinct advantages over other proof-based blockchains, especially in latency, reactivity, scalability, and cost-efficiency.

- **Low latency:** As proofs in proof-based applications can only operate on state they can change, Proofbase can execute most transactions in a BFT-resistant manner without requiring consensus. This allows simple transactions to be processed almost instantaneously, making Proofbase ideal for applications that demand fast and efficient state updates.

- **Highly reactive and cost-effective:** As proof-based applications do not inherently interact with one another, neither do their states, so Proofbase supports *parallel transaction processing*, ensuring that the verification of one proof does not delay the execution of others. This parallelism leads to lower fees and more predictable pricing for users. Furthermore, Proofbase introduces the concept of *ephemeral transactions*, where transaction data are stored for a customizable amount of time, enabling more efficient proof verification and avoiding the high costs associated with long-term data retention seen in other blockchains. Furthermore, by moving proof verification outside of blockchain execution, proof verification costs are not subject to execution cost or data storage cost models that make proof verification in other blockchains prohibitive.

- **Anchoring proofs in state:** Proofbase is designed to anchor proofs within a continuous chain of state manipulations, making it particularly well suited for applications with stateful management of blockchain-based assets utilizing complex offchain logic. By encapsulating proofs in a verifiable blockchain state, Proofbase simplifies the deployment and maintenance of proof-based applications, enhancing both the capabilities of blockchain-based applications and their scalability.

- **Externally verifiable application state:** As a result of Proofbase's unique design, users submit transactions and proofs directly validators which attest to the validity of proofs and the application states, sending attestation directly back to the user. As each proofchain's state is verifiable independent of the rest of the network, these attestations can be made without any validator-to-validator communication or consensus. Not only does this result in a very low time-to-finality, and highly scalable network, but these attestations are aggregated user-side to result in a "proof certificate" which is verifiable externally to the network without needing a global state oracle, enabling applications on Proofbase to cheaply interoperate with external blockchains without adding latency.

Proofs in Proofbase can be Lurk proofs, which have a richer interaction with the underlying protocol of the chain, or they can be a proof using any proof system supported by Proofbase[18]. The Proofbase protocol enables low-latency proof verification, even for proofs that are too expensive for most blockchains.

[18] Proofbase aims to support the Plonky3 ecosystem (Succinct SP1, Lita Valida, and others) as the initial first-class citizen.

Proof-based applications on Proofbase operate on the model of *bulwarked state*, inherited from the Linera [Linera team, 2023] protocol. In this model, the state of the blockchain is sharded into low-granularity fragments that are shared between carefully managed sets of users, and the "shards", called "proofchains", can communicate through asynchronous message passing. This enables the validators to parallelize the verification of proofs relating to different proofchains, lowering the latency of the sequencing protocol, and gives users strong guarantees that their proofs will not be invalidated through sequencing conflicts. We'll touch more on this on page 20.

This results in a model where each proofchain is owned by a single owner [19]. This owner can be either the application or a single user. In the case of the application owner, state update sequencing would be conducted offchain. In the case of single-owner chains, global application state is sharded across user-owned proofchains for that application type. Cross-proofchain interaction is accomplished via asynchronous messaging (as outputs of proofs and are therefore verifiable), enabling asynchronous, multi-user, trust-gapped applications. This model introduces novel application design to yield new possibilities in user-controlled, highly scalable, private state applications without introducing latency.

[19] This doesn't have to be the case, as the protocol does allow for the proofchain to be owned by a collection of users with the requirement that the proposer order is defined as part of the application, resulting only 1 owner at any given time.

As each proofchain's state and transactions are processed and maintained independently from others in the network, Proofbase is highly horizontally scalable, only requiring validators to increase hardware capabilities to add additional capacity. This network architecture has been shown to scale to support sustained throughput above 100,000 transactions per second in laboratory settings [Baudet et al., 2020] [Blackshear et al., 2024]. This level of scalability makes Proofbase uniquely suited for mass-adoption scale private applications.

In Proofbase, proofchain owners submit transactions containing the proposed state transitions and proof directly to the validators. Validators independently verify and attest to the validity of the proofs and the application states (see figure 2 on the next page), sending attestations back to the submitter. This process, which occurs in one round trip of communication (less than 1 second in practice), enables users to aggregate attestations to construct "proof certificates" that are cheaply verifiable as a substitute for direct proof verification, allowing applications on Proofbase to interact with blockchains and assets externally. Developers can use Proofbase certificates –whose authenticity is easy to demonstrate– to "settle" application state updates across multiple chains, eliminating the need to build entire applications within Proofbase or have users interact directly with smart contracts. This model supports native multichain applications, offering a seamless and blockchain-less user experience without sacrificing access to existing user groups, assets, and liquidity.

By incorporating Lurk as the smart contract and data language, proof-based applications have well-defined semantics for program I/O and messages, allowing applications in Proofbase to interact asynchronously across proofchains. This feature is not only a requirement for multi-user, user-controlled applications, but also creates novel possibilities in concurrent proving, permissionless and configurable privacy and access control, as well as composable ZK applications. As a result of Lurk being natively provable, all proofs, smart contract logic and state transitions are also provable in the aggregate: Proofbase periodically compresses the global blockchain state into succinct verifiable checkpoints through proof recursion. This allows Proofbase to function as a full ZK layer-2, with its state verifiably settled on Ethereum.

A common concern of ZK users is the ability to keep up with the latest state-of-the-art proving system and not be locked into any one language or VM. Because of this, Proofbase will support verifiers of the most popular zkDSL and zkVM, allowing maximum flexibility for developers. As proof verification is handled at the validator client level in Proofbase, incorporating new proving systems, prover networks and maintaining pace with the developments in proving system backends, is of low complexity. As Lurk is designed to be portable to proving system backends, these developments do not pose any risk for Proofbase to lag in performance to newer generation networks. While we are confident that the power of Lurk to simplify and abstract the complexity of developing ZK applications will result in Lurk being the preferred ZK stack, this agnostic approach minimizes touch points with Lurk to just those for managing onchain state, which ensures Proofbase's success is not dependent on the wider adoption of Lurk.

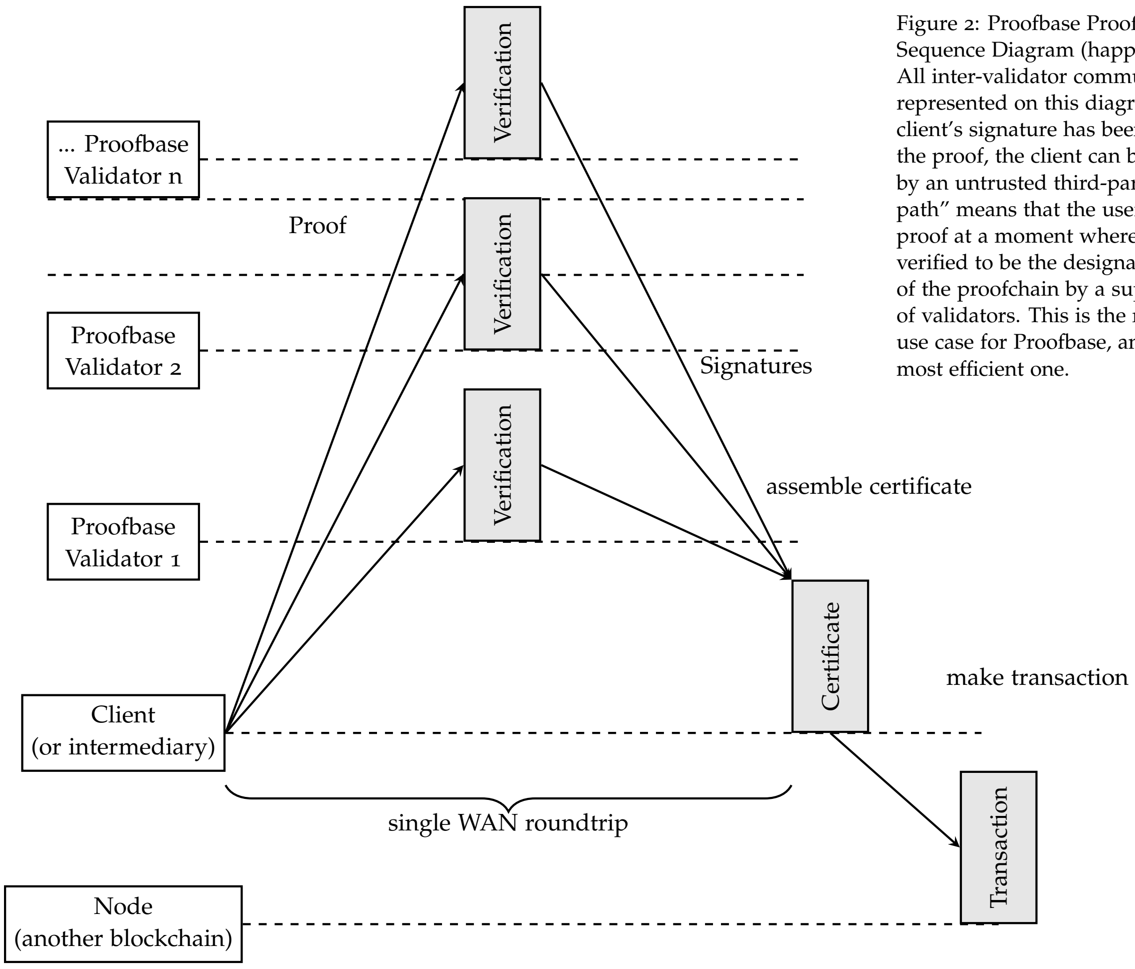

Figure 2: Proofbase Proof Verification Sequence Diagram (happy path).
All inter-validator communication is represented on this diagram. Once the client's signature has been apposed to the proof, the client can be replaced by an untrusted third-party. "happy path" means that the user submits the proof at a moment where they can be verified to be the designated next writer of the proofchain by a super-majority of validators. This is the most common use case for Proofbase, and it is also the most efficient one.

Proof Certificates: Indirect Versus Direct Proof Verification

In Proofbase, proof certificates play a crucial role in optimizing proof verification while maintaining security and interoperability across blockchains. These certificates distinguish between indirect and direct proof verification, offering flexibility in proof-based applications.

Direct proof verification involves validating proofs by executing the underlying cryptographic algorithms. While this ensures the highest security level, it is computationally expensive and time-consuming, making it impractical for many use cases, especially in resource-limited environments. The costs are particularly burdensome when frequent or rapid proof validation is required, such as in complex dApps or applications where proof generation happens at the rate of user interaction.

To address this, Proofbase introduces *proof certificates*, a more efficient alternative. A decentralized verification network (DVN), consisting of independent validators, produces these certificates.

Validators verify proofs and generate certificates that attest to their validity, allowing the certificates to be used in place of direct proof verification. This approach provides a cost-effective and time-efficient solution, while enabling the use of Proofbase as a stateful verifier layer in parallel with other blockchains.

The key advantages of proof certificates include:

1. **Cost Efficiency:** Certificates significantly reduce onchain costs compared to direct proof verification, lowering the computational burden and transaction fees, thus lowering the barrier for applications.

2. **Low latency:** The expected overhead of getting a certificate for the validity of a proof is less than half a second (over the milliseconds needed to run the native proof verification). This metric is resistant to Proofbase congestion, as a Proofbase validator tasked with parallel proof verification can scale horizontally.

3. **Verifier Standardization:** Certificates offer a standardized mechanism for proof verification, simplifying proof-based application development, and ensuring consistent security across the network.

4. **Economic Security Model:** Besides the guarantees provided by their BFT-resistant blockchain protocol, certificates also operate within a larger crypto-economic security model, leveraging economic incentives and a base chain (L1) bonding to maintain their integrity. This ensures that potential attacks are economically unfeasible.

Unlike traditional proof aggregation, where multiple proofs are combined to reduce verification costs, Proofbase's proof certificates focus on cost and latency reduction through attestations. These certificates are portable across blockchains, providing developers with a versatile tool for multichain applications without introducing latency. But unlike other proof attestation layers, Proofbase's proof certificates attest to both the state and proof, enabling the stateful, cost-effective interaction from Proofbase to external domains.

In summary, Proofbase's proof certificates strike a balance between cost, efficiency, and security, making them essential for scalable, interoperable applications while managing the trade-offs between direct and indirect verification.

Why a ZK Rollup: Bootstrapping Meaningful Security

Proof certificates in Proofbase unify the computational capabilities and assets of multiple blockchains into a single application layer, potentially serving as the cornerstone of security for applications managing large sums of value across various blockchains with private, offchain data and logic. This requires robust economic guarantees from the Proofbase network, ensuring that proof certificates remain secure.

Building Proofbase as a PoS Layer 1 blockchain using a native asset for security could achieve this, but it also introduces risks for early users dependent on a new asset.

However, with innovations such as EigenLayer, economic security can be bootstrapped from established assets and validator sets. This method enables the security growth to outpace the security risks of applications built on Proofbase and proof certificates. Furthermore, by using Ethereum to verify check-pointed Proofbase state, economic security becomes time-windowed from the issuance of the proof certificate to its verification on Ethereum.

Thus, we plan to build Proofbase as a ZK layer 2 on Ethereum, using ETH-denominated assets to establish the Proofbase validator set using EigenLayer, ensuring both security and compatibility with Ethereum's robust ecosystem.

Why the Proofbase Protocol is a Good Fit for Proof Verification

Manipulating State with Proofs

The Proofbase protocol is a fully Byzantine-resistant blockchain protocol that allows for the manipulation of state. This capability is often underestimated in the zero-knowledge (ZK) proof ecosystem, yet it significantly expands the potential applications of ZK proofs. Many examples of ZK proof usages have thus far been limited to one-round protocols, where the exchange concludes after a single message: proof of identity (zkPassport), proof of humanity (Worldcoin), variants of anonymous credentials (zkLogin, keyless login), and proof of ownership of a credit score.

However, most applications are multi-user and require a regular stream of proofs operating over complex state, such as payments, rollups, social applications, games, or maintaining a portfolio of financial assets in a privacy-preserving yet compliant manner. The Proofbase blockchain enables the proving of state updates involved in this stream of proofs, tying the proofs to an onchain sharded state.

For example, proving the 50th move in a game requires:

1. Proving the state after the first 49 moves.
2. Proving that one's current move is valid.
3. Agreeing with a counterpart that the first 49 proven moves correspond to the same game they have played so far.

A recursive proof can efficiently handle the first two elements, but leaves the last point as an exercise to the reader. In other words, a ZK rollup could prove the valid sequencing and execution of their blocks from their genesis to the latest block height. However, it is up to the underlying Layer 1 (L1) that receives that proof to tie it to the ZK rollup state it verified until then if it wants to ensure there was no reversion or hard fork. Proofbase serves as that Layer 1, not only for rollups, but for games, portfolio management, and other applications that require sequencing states over time.

Proofbase's Ad-hoc Sequencing Service

The Proofbase protocol enables ad-hoc sequencing through independently sequenced proofchains, involving only the participants of each proofchain. This approach eliminates concerns about block fill rates and congestion, focusing instead on whether validators can scale transaction processing power to handle punctual loads. Decades of server management practices affirm the feasibility of this approach, as soon as we accept the idea of moving beyond home staking setups.

This scalability comes with a trade-off: proofchain participants operate segmented from the rest of the network. While they can asynchronously transmit assets and messages to and from other proofchains, external users cannot directly modify a proofchain's state where they are not a valid writer. In some cases, such as maintaining an auditable key directory, a single participant may suffice. However, larger use cases, like operating a fully public automated market maker (AMM) based on proofs, introduce more complexity [20].

[20] In order to have a shielded chain, you need to have some kind of composable state. You need to have the the state of the chain split up into all these little fragments [. . .] When you look at what people actually like to do with blockchains [. . .], they need to have some kind of late binding capability. In a sense, this is a similar problem as having a kind of cross-rollup communication: If every user is doing their own independent state transition on their own user device, [. . .] we need to have some model to perform asynchronous ZK execution via message passing.
(de Valence [2023])

This is unavoidable, as proofs have always faced the challenge of managing shared state when multiple provers attempt to edit it simultaneously. In such cases, both provers might try to transition the same state, but only one proof can be sequenced, rendering the other invalid. The Proofbase protocol addresses this by encoding a designated-participant model within a proofchain, for instance through an exclusive participant configuration or a strict round-robin system among authorized proposers. This structure ensures no proofchain becomes stale involuntarily, as each proposer is linked to the next transaction with precise granularity, unlike traditional blockchains, which update global protocol states.

Additionally, Proofbase equips proofchains with a messaging model that enables communication with other chains or users, supporting batch flows for admitting new users or transactions.

The Proofbase Protocol Allows Pricing Transaction Storage Differently

The Proofbase protocol introduces a unique approach to pricing transaction storage. Blockchains typically store both the continually updated state and historical transactions, with the latter often consuming the majority of storage.

In traditional blockchains, validators and full nodes exchange transactions rather than the current state. Full nodes re-execute transactions to verify correctness, which prevents tampering and holds validators accountable to the blockchain's protocol, but necessitates storing vast amounts of transaction data. High-throughput chains and Layer 2s (e.g., Optimism, Arbitrum, Avalanche, Solana) typically run out of storage quickly and must either increase hardware requirements or shed transaction data. As a result, these chains introduce an "event horizon," after which older data becomes available only through archival nodes, creating an artificial limit on data accessibility.

This model of "forever" re-executable transactions is thus a flawed security promise: full nodes inherently limit throughput or restrict data availability past a certain point. In fact, the cost of storage for placing a transaction remains the same, but the transaction's "shelf life" shrinks: access to the data on full nodes diminishes over time, requiring users to depend on archival nodes for historical information. [21]

[21] See `https://etherscan.io/chartsync/chaindefault`.

Classical blockchains price transaction storage uniformly and cannot trade this against reduced data availability while managing the security implications of such decisions. Proofbase, however, can. Using asynchronous message passing between proofchains, Proofbase allows proofchains to accept messages only from proofchains with equivalent or better transaction storage durations. This flexibility enables Proofbase to offer ephemeral transaction storage at a lower cost while ensuring that the security boundaries between proofchains are not violated.

This model not only makes sense for short-lived applications (e.g., ZK battleship) but *also allows larger proofs, such as STARKs, to function practically as transactions*. STARK proofs are faster to produce and equally fast to verify than SNARKs, though typically larger (40-100KB minimally compressed). In most blockchains, using them as transactions is prohibitively expensive. Proofbase overcomes this limitation by configuring proofchains to accommodate such proofs, bypassing the need for costly recursive proving steps.

Conclusion: Applications of Proofbase

Proofbase functions as a decentralized stateful verification network that provides users with certificates (effectively multisigs) after each transaction. These certificates are not limited to the Proofbase ecosystem; they are designed to be portable and usable across any blockchain. Using the decentralization of Proofbase and its Ethereum backstop, the certificates are secured in a two-tiered system: Proofbase has the decentralization of an L1, on top of which Ethereum acts as a backstop, which will make Proofbase a full ZK rollup. Unlike traditional scalability-focused L2s, however, Proofbase's focus is on creating certificates that are widely acceptable across different blockchain environments. This broad acceptance will be supported by the EigenLayer restaking mechanism.

All Proofbase transactions are themselves proofs, making Lurk the ideal smart contract programming language for Proofbase. Lurk's flexibility enables developers to write proofs for various applications, but verification on Proofbase is not limited to Lurk-based proofs. The system is designed to verify and issue certificates for a range of specialized proofs, such as zkOAuth proofs based on Groth16 [22].

[22] Foteini Baldimtsi, Konstantinos Kryptos Chalkias, Yan Ji, Jonas Lindstrøm, Deepak Maram, Ben Riva, Arnab Roy, Mahdi Sedaghat, and Joy Wang. zkLogin: Privacy-Preserving Blockchain Authentication with Existing Credentials, January 2024

This opens up powerful new use cases. On any blockchain that supports multisig verification, developers can write contracts that accept Proofbase certificates as proof of validity. For instance, a contract can accept a Proofbase certificate verifying a zkOAuth proof, making Sui's zkLogin app-—where you send funds to a user who doesn't have a blockchain account—-available on all chains, even those that do not natively support proof verification or provide precompiles for it.

The broad applicability of Proofbase is further enhanced by Lurk's programmability. Developers can write proofs for entirely new applications, such as complex onchain games, user-controlled private applications, and providing mechanisms for the web to interact with blockchains.

Use Cases

The flexibility of Proofbase's verification network makes it suitable for a range of high-impact use cases:

- **High-volume proofs**: Proofbase 's scalable verification network can handle large volumes of proofs efficiently, making it ideal for applications that require continuous or frequent proof generation.
- **Extensible Anonymous Payment**: the Proofbase model natively supports anonymous credentials [23], and can correspondingly provide high levels of privacy to its users.

[23] Mathieu Baudet, Alberto Sonnino, Mahimna Kelkar, and George Danezis. Zef: Low-latency, Scalable, Private Payments. *arXiv:2201.05671 [cs]*, January 2022

- **Private state applications**: Proofbase enables private, decentralized applications, with private payments being one of the most promising use cases. These applications maintain confidentiality while ensuring verifiability through Proofbase certificates.
- **Revolutionizing blockchain UX across all blockchains**: By acting as an "OpenID for all blockchains," Proofbase can simplify user onboarding and interaction across multiple chains. Users can interact with any blockchain securely, without needing blockchain-specific credentials, thanks to Proofbase certificates.
- **Trustless onchain games**: Proofbase enables a world of onchain gaming with verifiability without UX tradeoffs. Games like poker, which rely on the trust of the randomness, privacy on user's cards and the ability to use assets in-game, can be implemented as asynchronous applications with verifiable and private state. Trust-gapped actors (through trusted execution environment) isolate shared private information without requiring application-level cryptographic design. With Lurk on Proofbase, these applications become a simple encoding of rules as the primitives for privacy and verifiability exist today.

References

Nada Amin, John Burnham, François Garillot, Rosario Gennaro, Chhi'mèd Künzang, Daniel Rogozin, and Cameron Wong. Lurk: Lambda, the ultimate recursive knowledge (experience report). *Proc. ACM Program. Lang.*, 7(ICFP), aug 2023. DOI: 10.1145/3607839. URL `https://doi.org/10.1145/3607839`.

Argument Computer Corporation. Lurk 0.5 Benchmarks. https://argument.xyz/blog/perf-2024/, September 2024.

Foteini Baldimtsi, Konstantinos Kryptos Chalkias, Yan Ji, Jonas Lindstrøm, Deepak Maram, Ben Riva, Arnab Roy, Mahdi Sedaghat, and Joy Wang. zkLogin: Privacy-Preserving Blockchain Authentication with Existing Credentials, January 2024.

Christopher Batten, Pjotr Prins, Efraim Flashner, Arun Isaac, Jan Nieuwenhuizen, Ekaitz Zarraga, Tuan Ta, Austin Rovinski, and Erik Garrison. The Case for Using Guix to Enable Reproducible RISC-V Software & Hardware. In *Sixth Workshop on Computer Architecture Research with RISC-V (CARRV)*, June 2022.

Mathieu Baudet, George Danezis, and Alberto Sonnino. Fast-Pay: High-Performance Byzantine Fault Tolerant Settlement. *arXiv:2003.11506 [cs]*, November 2020.

Mathieu Baudet, Alberto Sonnino, Mahimna Kelkar, and George Danezis. Zef: Low-latency, Scalable, Private Payments. *arXiv:2201.05671 [cs]*, January 2022.

Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity, 2018.

Daniel Benarroch, Kobi Gurkan, Ron Kahat, Aurélien Nicolas, and Eran Tromer. zkinterface, a standard tool for zero-knowledge interoperability. In *2nd ZKProof Workshop. https://docs. zkproof. org/pages/standards/acceptedworkshop2/proposal–zk-interop-zkinterface. pdf*, 2019.

Johan Bjäreholt. *RISC-V Compiler Performance:A Comparison between GCC and LLVM/Clang*. Biekinge Institute of Technology, 2017.

Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, Brandon Williams, and Lu Zhang. Sui Lutris: A Blockchain Combining Broadcast and Consensus, August 2024.

Gustavo Caparica, François Garillot, Adrian Hamelink, Chhi'mèd Künzang, and Winston Zhang. Memoset: Sound functional lookups for mutual recursion. *in preparation*, 2025.

T. Coquand and Gérard Huet. *Concepts mathematiques et informatiques formalises dans le calcul des constructions*. report, INRIA, 1986.

Henry de Valence. Shielded Transactions Are Rollups, July 2023. URL `https://www.youtube.com/watch?v=VWdHaKGrjq0`.

Liam Eagen, Sanket Kanjalkar, Tim Ruffing, and Jonas Nick. Bulletproofs++: Next Generation Confidential Transactions via Reciprocal Set Membership Arguments, 2022.

Matthias Felleisen and Daniel P. Friedman. Control Operators, the SECD Machine, and the Lambda-calculus. *Formal Description of Programming Concepts*, III:193–219, 1983.

Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *Found. Trends Databases*, 5(2):105–195, nov 2013. ISSN 1931-7883. DOI: 10.1561/1900000017. URL `https://doi.org/10.1561/1900000017`.

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

Ulrich Haböck. Multivariate lookups based on logarithmic derivatives, 2022.

Bogumił Hausman. Turbo Erlang: Approaching the Speed of C. In Evan Tick and Giancarlo Succi, editors, *Implementations of Logic Programming Systems*, pages 119–135. Springer US, Boston, MA, 1994. ISBN 978-1-4615-2690-2. DOI: 10.1007/978-1-4615-2690-2_9.

Simon L Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine Version 2.5. *Journal of Functional Programming*, 2:127–202, July 1992. DOI: 10.1017/S0956796800000319.

P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, January 1964. ISSN 0010-4620, 1460-2067. DOI: 10.1093/comjnl/6.4.308.

P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966. ISSN 0001-0782, 1557-7317. DOI: 10.1145/365230.365257.

Xavier Leroy. *The ZINC Experiment : An Economical Implementation of the ML Language*. Report, INRIA, 1990. URL `https://inria.hal.science/inria-00070049/`.

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

Linera team. Linera: A Blockchain Infrastructure for Highly Scalable Web3 Applications version 2. Technical report, Linera, August 2023.

Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably Correct Peephole Optimizations with Alive. *ACM SIGPLAN Notices*, 50, March 2015. DOI: 10.1145/2813885.2737965.

Jan Midtgaard, Norman Ramsey, and Bradford Larsen. Engineering definitional interpreters. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, pages 121–132, New York, NY, USA, September 2013. Association for Computing Machinery. ISBN 978-1-4503-2154-9. DOI: 10.1145/2505879.2505894.

Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. Reproducible Containers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–182, Lausanne Switzerland, March 2020. ACM. ISBN 978-1-4503-7102-5. DOI: 10.1145/3373376.3378519.

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, 1998. ISBN 978-0-521-63124-2. DOI: 10.1017/CBO9780511530104.

John E Savage. *Models of computation*, volume 136. Addison-Wesley Reading, MA, 1998.

Succinct Labs. Introducing SP1: A performant, 100% open-source, contributor-friendly zkVM. https://blog.succinct.xyz/introducing-sp1/, February 2024.

David Van Horn and Matthew Might. Pushdown Abstractions of JavaScript, December 2011.

D. H. Warren. An Abstract Prolog Instruction Set. *Technical Report*, 1983.

Barry Whitehat. Lookup Singularity, November 2022. URL `https://zkresear.ch/t/lookup-singularity/65`.

Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software*, 174:110884, April 2021. ISSN 0164-1212. DOI: 10.1016/j.jss.2020.110884.