zkFold

zkFold: UPLC Converter

zkfold.io

Name of project and Project URL on IdeaScale/Fund

zkFold: UPLC Converter

https://milestones.projectcatalyst.io/projects/1200257

Project Number

Project ID: 1200257

Name of project manager

zkFold team

Date project started

12 August 2024

Date project completed

28 July 2025

Objectives

The objective of this project is to enable seamless zero-knowledge smart contract development across the diverse ecosystem of Cardano on-chain languages by building a UPLC Converter and accompanying CLI tool. zkFold Symbolic introduces a new paradigm for smart contract execution on Cardano, where contract logic is executed off-chain and only a succinct zero-knowledge proof is verified on-chain. This drastically reduces on-chain computation costs while enabling highly complex applications to run within the platform's resource constraints. However, since smart contracts on Cardano are ultimately compiled into Untyped Plutus Core (UPLC), our goal is to build a bridge: a toolchain that translates UPLC into zero-knowledge circuits, allowing existing tools like PlutusTx, Aiken, Plutarch, and others to benefit from zkFold's advantages without requiring developers to change how they write contracts. The UPLC Converter project therefore aims to (1) implement a UPLC-to-ZK transpiler that supports an increasingly complete subset of UPLC constructs and built-ins, (2) create arithmetic circuits for both basic and complex built-in operations, (3) provide a CLI tool to compile UPLC into circuits, generate verifier scripts, and output zk prover metadata, and (4) ensure that these tools are documented, tested, and ready for integration into broader language ecosystems. Ultimately, this initiative strives to make the creation of ZK smart contracts not only possible but effortless for any developer working within the Cardano ecosystem.

Status of the Project

The project is completed and has already met several critical technical milestones. A working prototype of the UPLC converter has been developed, supporting a selected subset of UPLC constructs along with a comprehensive test suite to ensure correctness and coverage. We have successfully generated arithmetic circuits for both basic and complex UPLC built-in operations, including cryptographic functions and higher-order logic, forming the foundation for general-purpose ZK contract support. In parallel, a CLI tool prototype has been implemented with the core functionality: it can load UPLC scripts in various formats, generate zero-knowledge circuits, produce Plutus verifier scripts, and export prover metadata. The architecture of the CLI tool is modular and extensible, enabling easy integration into existing developer workflows. Documentation has been completed for the specification, ensuring that third-party developers can use the tools effectively and with minimal friction. Each component has been tested in isolation and in combination, validating the pipeline from UPLC input to zero-knowledge proof output. The project now moves toward full CLI completion and broader integration with Cardano language toolchains, supported by active collaboration with key developers across the ecosystem.

List of challenge KPIs and how the project addressed them

1. Create documentation

KPI Challenge

The key challenge for this milestone was to produce technical documentation that accurately explains both the high-level function and low-level mechanics of the UPLC Converter CLI tool, which serves as a bridge between traditional smart contract development and zero-knowledge proof systems on Cardano. Given the novelty and complexity of arithmetic circuit compilation in the context of zero-knowledge smart contracts, it was critical to describe the tool’s behavior in a way that is both technically rigorous and accessible to developers with varying levels of expertise. The documentation needed to clearly define inputs, outputs, transformation stages, and command-line functionality, all while avoiding ambiguity. Another important challenge was ensuring that the documentation remained aligned with the tool's actual implementation, minimizing the risk of miscommunication or developer error during integration or usage.

How the KPI was Addressed

- **Detailed UPLC Converter Specification**
  A comprehensive specification was produced in PDF format, explaining the internal logic of the arithmetic circuit compiler and its role in the broader zero-knowledge contract workflow. This included step-by-step descriptions of how the converter processes input code and generates circuit-compatible output, alongside explanations of design decisions and limitations.

- **Command-Level API Documentation**
  The API documentation clearly outlines all CLI commands available within the tool, including their syntax, parameters, input/output formats, and expected behaviors. Examples and usage patterns were included to guide developers in both standard and advanced use cases.

- **Consistency with Tool Behavior**
  The documentation was written in close collaboration with the development team to ensure accuracy and consistency with the tool’s actual implementation. Regular

cross-checks with the codebase ensured that the docs reflected the latest command set and compiler logic.

- **Focus on Developer Usability**
  Particular attention was paid to making the documentation accessible to smart contract developers who may be new to zero-knowledge proofs. Terminology was clarified, diagrams and flow descriptions were added where helpful, and the documents were structured for ease of navigation and quick reference.

The documentation created for the UPLC Converter marks a critical step toward making zero-knowledge smart contract development accessible within the Cardano ecosystem. By clearly describing the tool's function and how it integrates with familiar smart contract languages, the documentation enables developers to adopt zero-knowledge technology without needing to master low-level cryptographic concepts. The specification explains the behavior of the arithmetic circuit compiler, detailing how inputs are transformed into zero-knowledge-compatible outputs. Complementing this, the API documentation provides a structured overview of the CLI commands, usage syntax, and data expectations, empowering developers to use the tool effectively. Special care was taken to ensure clarity, completeness, and alignment with the underlying code, reducing potential friction for users. Examples and use cases were included to support both novice and advanced users. The result is a set of developer-friendly documents that bridge theoretical functionality with practical implementation. This milestone improves the accessibility and usability of Cardano's infrastructure, paving the way for broader adoption of privacy-preserving smart contracts.

2. Implement a prototype for a limited subset of UPLC

KPI Challenge

The main challenge for this milestone was to design and implement a working prototype of a UPLC (Untyped Plutus Core) converter that supports a limited but functional subset of the language. UPLC is a low-level, expressive language used in Cardano smart contracts, and adapting it for use in zero-knowledge circuits required careful abstraction, especially given the constraints imposed by ZK systems. Only a specific set of language constructs and built-in functions could be supported in this phase, so a key challenge was to select and implement a subset that was both meaningful and representative of real-world contract logic. Additionally, the prototype needed to be reliable and correct from the outset, making comprehensive testing essential. Creating a robust test suite that could validate both simple and edge-case behavior in the limited UPLC subset added an additional layer of complexity, especially in terms of coverage and automation. The goal was to strike a balance between delivering functional core features and laying a foundation for broader language support in the future.

How the KPI was Addressed

- **Implementation of the Prototype for a Limited UPLC Subset**
  A functioning Haskell-based prototype was developed to process and convert a carefully selected subset of UPLC constructs into arithmetic circuits. This subset excluded more complex or less-used language features to maintain simplicity and ensure early-stage stability. Supported constructs and built-ins were implemented with clear, maintainable logic and aligned with constraints of zero-knowledge proof systems.

- **Design of a Comprehensive Test Suite**
  A detailed and automated test suite was created alongside the prototype to validate the supported UPLC subset. The test cases were written to cover a wide variety of input scenarios, including expected behavior, invalid code handling, and corner cases. This ensured that the converter produces correct and predictable outputs, even within the subset limitations.

- **Incremental and Modular Development**
  The implementation followed a modular design, allowing future extensions to the UPLC subset without requiring major rewrites. This modularity also helped isolate feature logic for targeted testing and debugging.

- **Validation Through Iteration**
  Frequent testing and incremental integration helped ensure that each supported construct worked correctly before expanding the feature set. This approach helped maintain code quality while progressively increasing coverage.

Summary

This milestone successfully delivered a working Haskell prototype that supports a limited subset of UPLC, providing a foundation for converting smart contract logic into zero-knowledge circuits. The team carefully selected the most relevant UPLC constructs to support at this stage, ensuring that the prototype could demonstrate meaningful functionality while remaining stable and focused. The implementation was structured with extensibility in mind, allowing for gradual expansion of the supported language subset in future iterations. A comprehensive test suite was developed in parallel, providing strong validation and automated coverage for a wide range of scenarios. The tests ensured that edge cases and invalid input were handled correctly, giving confidence in the prototype's reliability. By narrowing the focus to essential features, the project avoided unnecessary complexity while still achieving practical utility. This approach allowed the team to learn and iterate rapidly without sacrificing code quality. The milestone represents a key step in enabling developers to use familiar smart contract tools within zero-knowledge environments on Cardano.

3. **Circuit generation for basic builtin operations**

KPI Challenge

The key challenge for this milestone was to design and implement arithmetic circuits that accurately represent the behavior of basic UPLC built-in operations, excluding cryptographic and serialization functions. Since zero-knowledge circuits require precise mathematical modeling of computation, each operation - such as arithmetic comparisons, logic gates, and numeric calculations - had to be translated into low-level constraints that could be evaluated inside a zk-SNARK or zk-STARK system. One difficulty was preserving semantic correctness: even small differences in how operations like addition, subtraction, or equality are handled can lead to incorrect or unverifiable proofs. Moreover, the team had to account for performance: poorly optimized circuits can quickly become too large or inefficient for real-world proving systems. Ensuring compatibility with the existing UPLC converter architecture while preparing for future support of more complex operations added another layer of complexity. Lastly, all circuits needed to be implemented in Haskell, demanding careful attention to functional purity, composability, and maintainability.

How the KPI was Addressed

- **Precise Modeling of Builtin Operations**
  The team implemented arithmetic circuits in Haskell to cover core UPLC built-ins, including operations like addition, subtraction, multiplication, division, and equality. Each operation was modeled with arithmetic constraints that align with the behavior expected by the UPLC specification, ensuring semantic fidelity

between source code and circuit output.

- **Modular Circuit Design**
  The circuits were structured in a modular way, allowing each operation to be composed into larger contract logic easily. This modularity supports both maintainability and extensibility, enabling future inclusion of more advanced operations such as cryptographic primitives or data serialization.

- **Optimization for Performance and Constraint Efficiency**
  Particular attention was given to optimizing constraint count and circuit depth to ensure that generated circuits remain efficient for real-world zero-knowledge proof generation. Operations were reviewed and benchmarked for performance to avoid introducing unnecessary computational overhead.

- **Alignment with UPLC Converter Architecture**
  Circuit logic was integrated into the broader UPLC converter pipeline, maintaining compatibility with the tool's existing input/output expectations. This ensured that the circuits could be immediately applied in real scenarios involving smart contract compilation.

Summary

The team successfully delivered arithmetic circuits in Haskell for a core set of UPLC built-in operations, marking a significant step toward zero-knowledge smart contract support in Cardano. These circuits cover essential logic such as arithmetic calculations and basic conditionals, which form the backbone of most smart contracts. Each operation was precisely modeled to maintain compatibility with UPLC semantics and integrated smoothly into the converter toolchain. By excluding cryptographic and serialization features at this stage, the project was able to focus on correctness and constraint efficiency. The modular structure of the circuits allows for future expansion and ease of composition in more complex workflows. Performance optimizations were incorporated from the outset, ensuring that the generated circuits remain practical for real-world zk applications. Integration with the CLI tool ensures immediate usability for developers. This milestone lays the groundwork for transforming high-level smart contract logic into provable, privacy-preserving computations on Cardano.

4. **Circuit generation for more complex builtin operations**

KPI Challenge

This milestone focused on generating arithmetic circuits for the remaining and more complex UPLC built-in operations, including cryptographic functions. The main challenge was the inherent complexity and performance sensitivity of cryptographic operations, which require precise mathematical modeling and efficient constraint systems to remain feasible within zero-knowledge proof environments. Operations such as hashing, signature checks, and advanced data manipulation involve non-trivial logic and often require breaking down high-level primitives into optimized, circuit-compatible components. Ensuring correctness while keeping circuit size and proof generation time within acceptable limits was critical. In addition, these advanced operations had to align with the UPLC specification and be integrated into the existing architecture of the UPLC converter tool without disrupting previous functionality. Maintaining readability, testability, and extensibility of the Haskell implementation while handling such low-level cryptographic logic presented both technical and architectural challenges.

How the KPI was Addressed

- **Development of Cryptographic Circuits in Haskell**
  The team successfully implemented arithmetic circuits for cryptographic built-ins, such as hash functions and other advanced operations defined in the UPLC spec. These circuits were designed using optimized patterns to keep constraint counts minimal, ensuring practical performance for zero-knowledge applications.

- **Support for Complex Data Manipulations**
  In addition to cryptographic functions, circuits were developed for more complex non-cryptographic built-in operations, such as advanced control structures and higher-order functions, where applicable. These were carefully mapped to arithmetic constraints while preserving semantic integrity.

- **Integration and Compatibility with UPLC Converter**
  All new circuit logic was integrated into the existing converter infrastructure, preserving modularity and maintaining compatibility with earlier development phases. The architecture allowed seamless invocation of both basic and complex operations during the translation of UPLC code into circuit form.

- **Performance and Correctness Validation**
  The circuits were tested extensively with edge-case scenarios and reference inputs to validate both correctness and efficiency. Benchmarks were used to evaluate constraint size and proving time, ensuring that even complex operations remain usable in practical zero-knowledge workflows.

Summary

This milestone delivered arithmetic circuits for the full range of complex UPLC built-in operations, including cryptographic primitives essential for real-world smart contract use cases. The team implemented these circuits in Haskell with a focus on constraint efficiency, correctness, and compatibility with the broader UPLC converter pipeline. Cryptographic functions, such as hashing, were modeled carefully to ensure they work reliably within zk-compatible arithmetic constraint systems. In parallel, complex non-cryptographic operations were supported, enabling the converter to handle a much broader range of UPLC programs. Integration with the CLI tool remained seamless, supporting a unified experience for developers building zero-knowledge contracts. Rigorous testing and benchmarking helped confirm both the correctness and performance of the circuits under realistic usage conditions. This milestone significantly enhances the expressiveness and utility of the UPLC converter, enabling developers to write privacy-preserving smart contracts with a much richer set of operations. It marks a key step toward full-featured zk-support within the Cardano ecosystem.

5. **CLI tool prototype with the core functionality**

KPI Challenge

The key challenge for this milestone was to implement a working CLI tool prototype that brings together the core components of the UPLC converter system into a cohesive, functional interface. This meant supporting multiple input formats for UPLC scripts, compiling them into arithmetic circuits, and enabling further downstream actions such as zero-knowledge circuit generation, Plutus verifier script generation, and prover metadata extraction. Building a CLI that integrates all these components required aligning diverse subsystems - from the parser and circuit generator to ZK tooling - into a streamlined and user-friendly command-line experience. Achieving this in a prototype phase demanded a strong focus on modular design and clear separation of concerns to ensure that the CLI could be extended later without needing major rework. Additionally, each command had to be implemented with enough robustness to handle errors gracefully, support edge cases, and return meaningful output for developers. Ensuring consistency across commands, as well as correctness in how the CLI reflected the tool's internal logic, was vital for developer trust and usability.

How the KPI was Addressed

- **Implementation of Core CLI Functionality**
  The CLI tool was developed in Haskell to support its core purpose - compiling UPLC scripts into arithmetic circuits. It supports loading UPLC in different formats, providing flexibility for developers using various tooling and workflows.

- **Support for Key ZK Workflow Commands**
  Core commands were implemented to support zero-knowledge circuit generation, Plutus verifier script generation, and ZK prover metadata output. Each command was mapped to internal components of the converter and designed to produce reliable, developer-usable outputs in a standardized structure.

- **Modular and Extensible Architecture**
  The CLI architecture was designed to be modular, enabling each command to operate as a standalone component while sharing core infrastructure like input parsing and error handling. This approach supports future growth and simplifies testing and maintenance.

- **Usability and Developer-Focused Design**
  Attention was given to clear command syntax, helpful error messages, and consistent behavior across all commands. The CLI interface was shaped to support realistic developer workflows and provide clear insight into what the tool is doing at each step of execution.

- **End-to-End Integration and Testing**
  All implemented features were tested end-to-end within the CLI to ensure proper interaction with the UPLC loading, circuit generation, and output stages. This confirmed that the prototype is not only functionally complete but also reliable in practice.

Summary

This milestone marks the successful delivery of a fully functional CLI tool prototype, integrating the core capabilities of the UPLC converter system. The tool can load UPLC scripts in various formats and compile them into arithmetic circuits, forming the basis of the zero-knowledge transformation pipeline. Key commands were implemented to generate zk circuits, export Plutus verifier scripts, and provide necessary metadata for proof generation workflows. The CLI was designed with modularity and extensibility in mind, allowing future enhancements without sacrificing code clarity or usability. Each command provides meaningful developer output, with clear error handling and structured formats. Testing confirmed smooth operation across all core functions, validating the tool's readiness for real-world use cases. This prototype provides a command-line interface that developers can use to begin integrating zero-knowledge capabilities into their Cardano smart contracts. It lays the foundation for the final milestone, where the tool will be completed and refined for production readiness.

List of project KPIs and how the project addressed them

1. Functional UPLC-to-ZK Circuit Conversion

KPI: Deliver a working transpiler that can convert UPLC code into zero-knowledge circuits.

Addressed: A prototype was developed in Haskell that compiles a meaningful subset of UPLC into arithmetic circuits. This subset was carefully chosen to support common contract logic while maintaining compatibility with zk circuit constraints. The converter was validated through a comprehensive test suite and integrated into the CLI tool.

2. Support for Basic and Complex Built-in Operations

KPI: Implement arithmetic circuits covering all necessary built-in operations, including both simple arithmetic and complex cryptographic functions.

Addressed: The team implemented arithmetic circuits for a broad range of built-ins. First, basic operations like arithmetic and comparison were built and tested. Then, more complex built-ins, including cryptographic functions (e.g., hash functions), were added to support real-world smart contracts. Circuits were optimized for constraint efficiency and verified for correctness.

3. CLI Tool for Developer Usability

KPI: Provide a developer-facing CLI tool that integrates with existing Cardano language workflows.

Addressed: A modular CLI tool was created with core commands, including UPLC loading, zero-knowledge circuit generation, Plutus verifier script creation, and prover metadata output. The CLI was designed to support multiple UPLC formats and return structured, developer-friendly outputs. Usability and extensibility were prioritized to ensure it integrates smoothly into real workflows.

4. Documentation and Developer Onboarding

KPI: Produce clear technical documentation for the UPLC converter, CLI commands, and underlying architecture.

Addressed: Comprehensive documentation was delivered in the form of a formal specification for the converter and detailed CLI API docs. These materials include input/output formats, examples, and command descriptions to facilitate easy integration and reduce support overhead.

5. Test Coverage and Reliability

KPI: Achieve robust testing for all implemented features, including edge cases.

Addressed: A full test suite was implemented alongside feature development to validate each supported UPLC construct and CLI command. Tests covered valid and invalid inputs, ensuring reliability and correctness in both standalone and integrated usage.

6. Integration Readiness with Other Language Toolchains

KPI: Ensure that the tool can be integrated with other Cardano on-chain language ecosystems (e.g., PlutusTx, Aiken, Plutarch).

Addressed: The tool accepts UPLC as its input, which is the common compilation target of all major Cardano smart contract languages. This design enables seamless integration with other language stacks without requiring developers to switch tools or learn new syntax, thus meeting the goal of transparent ZK smart contract support.

7. Modularity and Extensibility of Architecture

KPI: Design components in a way that supports future feature expansion and maintenance.

Addressed: Both the converter and the CLI tool were built using modular Haskell architecture. Each core component (e.g., circuit generation, command execution, prover info output) is isolated and testable, allowing future teams to extend the functionality (e.g., more builtins, new output formats) with minimal disruption.

Key achievements

Delivered a Working UPLC-to-ZK Circuit Converter
Successfully implemented a Haskell-based transpiler that converts a subset of Untyped Plutus Core (UPLC) into arithmetic circuits compatible with zero-knowledge proof systems. This core functionality enables developers using any Cardano on-chain language to access zkFold Symbolic capabilities without changing their development workflow.

Implemented Arithmetic Circuits for Full Builtin Coverage
Developed and optimized arithmetic circuits for both basic and complex UPLC built-in operations, including cryptographic functions. This achievement ensures the tool can support real-world smart contract logic within zero-knowledge constraints.

Built a CLI Tool Integrating Core ZK Workflow
Created a modular CLI tool with commands for loading UPLC scripts, generating zero-knowledge circuits, producing Plutus verifier scripts, and outputting ZK prover metadata. This makes zk smart contract development accessible through a developer-friendly interface.

Established Integration Path for All Cardano Language Toolchains
By targeting UPLC as the input, the project ensured compatibility with all major Cardano on-chain languages (e.g., PlutusTx, Aiken, Plutarch, plu-ts). This positions the tool for broad adoption across the ecosystem with minimal integration friction.

The project successfully delivered a working prototype of the UPLC Converter, enabling the transformation of Untyped Plutus Core (UPLC) into zero-knowledge circuits. This foundational achievement allows developers using Cardano's various smart contract languages to access zkFold Symbolic without changing their existing development workflows. A complete set of arithmetic circuits was implemented to support both basic and complex built-in operations, including cryptographic functions, ensuring compatibility with real-world contract logic. The team also developed a modular and extensible CLI tool that integrates all core functionality - loading UPLC scripts, generating circuits, producing Plutus verifier scripts, and exporting prover metadata. Each CLI command was designed to be developer-friendly and consistent, streamlining adoption and reducing onboarding time. The tool was thoroughly tested with a wide range of input scenarios to ensure correctness, stability, and performance across all components. Comprehensive documentation was produced, detailing both the converter specification and CLI API, to support independent use and third-party integration. Overall, the project laid a solid foundation for zero-knowledge smart contracts on Cardano and positioned the tooling for wide adoption across the ecosystem.

Key learnings

Technical

- Translating UPLC into zero-knowledge circuits is feasible but requires careful abstraction of both basic and complex built-in operations to maintain semantic correctness.

- Cryptographic operations pose unique challenges in circuit design due to their complexity and performance impact, requiring low-level optimizations for constraint efficiency.

- Modular architecture in both the converter and CLI proved essential for maintainability, extensibility, and reliable integration across multiple smart contract languages.

- Handling various UPLC formats effectively broadened compatibility and highlighted the need for clear, consistent internal representations to support diverse language toolchains.

Operational

- Developing and testing multiple components in parallel (transpiler, circuits, CLI) required strong coordination and modular development practices to avoid integration bottlenecks.
- Automating a robust test suite early in development was critical to validating correctness and allowed for safer iteration as the feature set expanded.
- Documentation must evolve continuously alongside code to remain useful; integrating it early in the development process minimized confusion and onboarding delays.
- End-to-end testing across CLI commands was necessary to catch integration-level issues that wouldn't surface in isolated module tests.

Strategic / Ecosystem

- Targeting UPLC as the common interface enabled compatibility with the entire Cardano smart contract ecosystem, reinforcing the value of low-level abstraction.
- Developer usability is essential for adoption - offering familiar workflows through a CLI tool and maintaining compatibility with existing language tools greatly lowers the barrier to entry.
- Community and ecosystem alignment (e.g., support from other language maintainers) increases the likelihood of downstream integration and long-term success.
- Building developer trust through predictable behavior, meaningful error messages, and transparent output is as important as raw functionality.

Summary

The project revealed that translating UPLC to zero-knowledge circuits is technically viable with careful abstraction and performance-conscious design. A modular, test-driven development approach was essential to manage complexity and support smooth integration across toolchain components. Operationally, documentation and test coverage proved just as critical as code in supporting usability and adoption. Strategically, focusing on UPLC compatibility allowed the solution to serve the entire Cardano smart contract ecosystem with minimal friction. By aligning technical execution with developer needs and ecosystem standards, the project is well-positioned for widespread adoption and future expansion.

Next steps for the product or service developed

1. Finalize the CLI Tool for Production Use

The next stage is to complete the CLI tool by refining its interface, improving performance, and ensuring full stability across supported commands. This includes enhancing error handling, adding command-line flags for advanced use cases, and streamlining output formats. A key focus will be usability for developers with varying levels of ZK and Cardano experience. Once finalized, the tool can be integrated directly into existing Cardano development pipelines.

2. Expand Support for Full UPLC Coverage

To broaden functionality, the converter will be extended to support the remaining UPLC constructs and edge-case patterns not yet covered in the prototype. This will involve developing additional arithmetic circuits, improving parsing logic, and optimizing the internal representation of UPLC terms. Full coverage is necessary for compatibility with all smart contracts written in Cardano-based languages. This step ensures the tool's relevance for a wider range of developers and applications.

3. Initiate Ecosystem Integration and Developer Adoption

With the core tooling in place, the focus will shift toward integration with prominent Cardano language ecosystems like PlutusTx, Aiken, and Plutarch. Outreach and collaboration with these communities will ensure compatibility and ease of use. Additionally, workshops, tutorials, and developer guides will be produced to promote adoption and streamline onboarding. Early feedback loops will guide iterative improvements based on real-world usage.

Summary

The project has reached a strong technical foundation and is now ready to evolve from prototype to production-grade tooling. Finalizing the CLI tool will ensure developers can confidently use it in real-world projects. Expanding UPLC coverage will unlock broader compatibility across Cardano smart contracts. Ecosystem integration will help drive adoption by embedding the converter into familiar language environments. By combining robust technology with strong community alignment, the next phase will accelerate real-world usage. These next steps will solidify the UPLC Converter's role as a key enabler of zero-knowledge smart contracts on Cardano.

Final thoughts/comments

The UPLC Converter represents a major step forward in empowering developers across the Cardano ecosystem to build next-generation, privacy-preserving, and scalable decentralized applications. By enabling seamless integration of zero-knowledge circuits into existing smart contract workflows, it removes a critical barrier for developers, allowing them to create secure applications without requiring deep expertise in cryptographic engineering. This innovation not only enhances privacy - for use cases like finance, healthcare, and identity - but also improves scalability by moving heavy computation off-chain while keeping on-chain verification lightweight and predictable. As a result, it significantly reduces development costs while opening the door to a broader range of high-performance dApps.

With increased developer access and capability, Cardano becomes more competitive and attractive to a global user base seeking advanced, secure blockchain solutions. The UPLC Converter will help catalyze a wave of innovation, driving real utility and adoption on the Cardano platform. Its impact extends beyond technical efficiency - by strengthening developer experience and user trust, it reinforces Cardano’s position as a leader in the evolving Web3 space. This tool lays the foundation for a more versatile, accessible, and high-value blockchain ecosystem.

Links to other relevant project sources or documents

Design UPLC Converter specification:

https://docs.zkfold.io/symbolic/tools/uplc-converter/

Create UPLC Converter API docs:

https://docs.zkfold.io/symbolic/tools/uplc-converter/#cli-commands

A prototype for a limited subset of UPLC (Haskell):

https://github.com/zkFold/zkfold-base/blob/main/zkfold-uplc/src/ZkFold/Symbolic/UPLC/Converter.hs#L33

A comprehensive test suite (Haskell)

https://github.com/zkFold/zkfold-base/pull/359

Hashing algorithms:

https://github.com/zkFold/symbolic/tree/main/symbolic-base/src/ZkFold/Symbolic/Algorithm/Hash

Signature verification:

https://github.com/zkFold/symbolic/blob/main/symbolic-base/src/ZkFold/Symbolic/Algorithm/ECDSA/ECDSA.hs

Conversions between integers and bytestrings are here:

https://github.com/zkFold/symbolic/blob/main/symbolic-base/src/ZkFold/Symbolic/Data/UInt.hs

Operations on builtin Data:

https://github.com/zkFold/symbolic/blob/main/symbolic-uplc/src/ZkFold/Symbolic/UPLC/Data.hs

BLS12-381 definitions are here:

https://github.com/zkFold/symbolic/blob/main/symbolic-base/src/ZkFold/Algebra/EllipticCurve/BLS12_381.hs https://github.com/zkFold/symbolic/blob/main/symbolic-base/src/ZkFold/Symbolic/Data/EllipticCurve/BLS12_381.hs

Link to Close-out video

Close out video:

https://youtu.be/OEnyaXMN0Oc