Slovak University of Technology in Bratislava
Faculty of informatics and information technologies

Lukáš Častven

Designing Zero-Knowledge Proof Solutions in Ethereum ecosystem

Progress report on DP2 solution

Thesis supervisor: Ing. Kristián Košťál PhD.

January 2026

Slovak University of Technology in Bratislava

Faculty of informatics and information technologies

Lukáš Častven

Designing Zero-Knowledge Proof Solutions in Ethereum ecosystem

Progress report on DP2 solution

Study programme: Intelligent software systems

Study field: Computer Science

Training workplace: Institute of Computer Engineering and Applied Informatics

Thesis supervisor: Ing. Kristián Košťál PhD.

January 2026

SLOVENSKÁ TECHNICKÁ
UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY
A INFORMAČNÝCH TECHNOLÓGIÍ

Návrh zadania diplomovej práce

Finálna verzia do diplomovej práce [1]

Študent:

Meno, priezvisko, tituly:	Lukáš Častven, Bc.
Študijný program:	Inteligentné softvérové systémy
Kontakt:	xcastven@stuba.sk

Výskumník:

Meno, priezvisko, tituly:	Kristián Košťál, doc. Ing. PhD.

Projekt:

Názov:	Návrh riešení využívajúcich dôkazy s nulovým vedomím v Ethereum ekosystéme
Názov v angličtine:	Designing Zero-Knowledge Proof Solutions in Ethereum ecosystem
Miesto vypracovania:	Ústav počítačového inžinierstva a aplikovanej informatiky, FIIT STU
Oblasť problematiky:	blockchain, kryptografia, kryptomeny, dôkazy s nulovým vedomím

Text návrhu zadania[2]

Zero-knowledge proofs (ZKPs) are a new cryptographic primitive in applied cryptography with applications in multiple industries, including Web3, supply chains, and the Internet of Things. By verifying the authenticity of information without disclosing its content, ZKPs improve privacy, security, and efficiency in digital systems. Current use cases include decentralized identity (Worldcoin), private transactions (stealth address schemes or blockchains like Zcash and Monero), secure and scalable Layer-2s (ZkSync, Scroll) voting systems, IoT networks, and supply chain management.

Examine existing solutions, proposals, and trends in this domain. Analyse a specific challenge discovered through the related work. Design a solution to address the challenge. Implement and test the solution on Ethereum (or a Layer-2) blockchain network. Evaluate and compare results with existing approaches. Discuss findings and contributions. Conclude with novelty, scientific findings, and future research directions.

[1] Vytlačiť obojstranne na jeden list papiera

[2] 150-200 slov (1200-1700 znakov), ktoré opisujú výskumný problém v kontexte súčasného stavu vrátane motivácie a smerov riešenia

Literatúra[3]

- Ulrich Haböck, David Levit, Shahar Papini. Circle STARKs. Cryptology ePrint Archive, 2024, https://eprint.iacr.org/2024/278.
- Jeremy Bruestle, Paul Gafni, and the RISC Zero Team. RISC Zero zkVM: Scalable, Transparent Arguments of RISC-V Integrity. Risc0. Retrieved January 11, 2024 from https://dev.risczero.com/proof-system-in-detail.pdf.

Vyššie je uvedený návrh diplomového projektu, ktorý vypracoval(a) Bc. Lukáš Častven, konzultoval(a) a osvojil(a) si ho doc. Ing. Kristián Košťál, PhD. a súhlasí, že bude takýto projekt viesť v prípade, že bude pridelený tomuto študentovi.

V Bratislave dňa 1.6.2025

Podpis študenta	Podpis výskumníka

Vyjadrenie garanta predmetov Diplomový projekt I, II, III

Návrh zadania schválený: áno / nie[4]

Dňa: .....

Podpis garanta predmetov

[3] 2 vedecké zdroje, každý v samostatnej rubrike a s údajmi zodpovedajúcimi bibliografickým odkazom podľa normy STN ISO 690, ktoré sa viažu k téme zadania a preukazujú výskumnú povahu problému a jeho aktuálnosť (uveďte všetky potrebné údaje na identifikáciu zdroja, pričom uprednostnite vedecké príspevky v časopisoch a medzinárodných konferenciách)

[4] Nehodiace sa prečiarknite

Čestné prehlásenie

Čestne vyhlasujem, že som túto prácu vypracoval(a) samostatne, na základe konzultácií a s použitím uvedenej literatúry.

V Bratislave, 1.1.2026 .....

Lukáš Častven

Anotácia

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Študijný program: Inteligentné softvérové systémy

Autor: Lukáš Častven
Priebežná správa o riešení DP2: Návrh riešení využívajúcich dôkazy s nulovým vedomím v Ethereum ekosystéme
Vedúci bakalárskej práce: Ing. Kristián Košťál PhD.
Január 2026

Táto práca skúma návrh a potenciálnu implementáciu riešení využívajúcich dôkazy s nulovým vedomím (ZKP) v rámci ekosystému Ethereum, s cieľom riešiť problémy škálovateľnosti a dosiahnuť post-kvantovú bezpečnosť. Dôkazy s nulovým vedomím sú kryptografické metódy, ktoré umožňujú overenie integrity výpočtov bez odhalenia citlivých dát, čo ich robí mimoriadne vhodnými pre zlepšenie súkromia a efektivity v blockchainových systémoch. Ústredným zameraním tejto práce je prieskum Zero-Knowledge Virtual Machine (ZkVM) prispôsobenej pre inštrukčnú sadu RISC-V, s využitím princípov kryptografie založenej na mriežkach. Kľúčovou aplikáciou takéhoto ZkVM by bolo dokázateľne správne vykonanie Ethereum blokov cez ZkEVM.

Výskum zahŕňa analýzu architektúry Ethereum, konceptu "snarkifikácie" pre konsenzuálnu aj exekučnú vrstvu, evolúciu a prirodzené výzvy ZkEVM (vrátane rekurzívnych dôkazov a skladacích schém), a preskúmanie kryptografických primitív založených na mriežkach, ako sú Ajtaiho záväzky, spolu so súčasnými systémami LaBRADOR, Greyhound a LatticeFold/LatticeFold+.

Cieľom tejto diplomovej práce je prispieť do odboru zhodnotením poten-

ciálnej výkonnosti ZKP systémov postavených na kryptografii založenej na mriežkach pre úlohu dokazovania správneho vykonania Ethereum blokov. Práca sa ďalej snaží porovnať tieto nové riešenia s etablovanými klasickými ZKP prístupmi, pričom zachováva post-kvantovú odolnosť.

Annotation

Slovak University of Technology in Bratislava
Faculty of informatics and information technologies
Degree Course: Intelligent software systems

Author: Lukáš Častven
Progress report on DP2 solution: Designing Zero-Knowledge Proof Solutions in Ethereum ecosystem
Supervisor: Ing. Kristián Košťál PhD.
January 2026

This thesis explores the design and potential implementation of solutions utilizing Zero-Knowledge Proofs (ZKPs) within the Ethereum ecosystem, aiming to address scalability challenges and achieve post-quantum security. ZKPs are cryptographic methods that allow the verification of computational integrity without revealing sensitive data, making them suitable for improving privacy and efficiency in blockchain systems. The central focus of this work is the exploration of a Zero-Knowledge Virtual Machine (ZkVM) for the RISC-V instruction set, leveraging lattice-based cryptography. A key application of such a ZkVM would be to enable creating proofs of correct execution of EVM.

The research includes an analysis of Ethereum's architecture, the concept of "snarkification" for both consensus and execution layers, the evolution and challenges of ZkEVMs (recursive proofs and folding schemes), and an examination of lattice-based cryptographic primitives, such as Ajtai commitments, along with contemporary systems like LaBRADOR, Greyhound, and LatticeFold/LatticeFold+.

This engineer’s thesis aims to contribute to the field by evaluating the potential performance of ZKP systems built on lattice-based cryptography for the task of proving the correct execution of Ethereum blocks. The work further seeks to compare these emerging solutions with established classical ZKP approaches, while maintaining post-quantum resistance.

Table of contents

List of Figures

List of abbreviations used

EVM	Ethereum Virtual Machine
ISA	Instruction Set Architecture
NIFS	Non-Interactive Folding Scheme
NTT	Number Theoretic Transform
L1	Layer 1
L2	Layer 2
PCS	Polynomial Commitment Scheme
PoS	Proof of stake
p2p	peer-to-peer
SNARK	Succinct Non-interactive Argument of Knowledge
STARK	Scalable Transparent ARgument of Knowledge
R1CS	Rank-1 Constraint System
ZK	Zero Knowledge
ZkEVM	Zero Knowledge Ethereum Virtual Machine
ZkVM	Zero Knowledge Virtual Machine
ZKP	Zero Knowledge Proof

Chapter 1

Introduction

Ethereum, a decentralized blockchain platform launched in 2015, provides a Turing-complete execution environment for smart contracts and decentralized applications [1]. Its modular architecture comprises a proof-of-stake consensus layer (Gasper [2]) and an execution layer governed by the Ethereum Virtual Machine (EVM [3]). A primary challenge for Ethereum is scalability, as the need for all nodes to validate and re-execute every transaction limits throughput and can centralize the network by increasing resource demands.

"Snarkification," the integration of Zero-Knowledge Proofs (ZKPs) like SNARKs, offers a path to mitigate these scalability issues. Initiatives such as the 'Lean consensus' [4] aim to snarkify consensus, allowing validators to verify compact proofs instead of entire state components. Similarly, Zero-Knowledge Ethereum Virtual Machines (ZkEVMs) seek to make EVM execution provable, enabling nodes to verify block execution via a single proof. Enshrining a ZkEVM into Ethereum's Layer 1 could further enhance scalability and simplify the creation of native rollups [5].

Zero-Knowledge proofs allow proving a statement's truth without revealing underlying information. These statements are about an element membership in NP language, thus an entire computations can be proven with this primitives, for example like computations done by the EVM, turning it into a ZkEVM. However, developing ZkEVMs is challenging due to the EVM's original design: complex opcodes, large word sizes requiring range proofs, its stack-based nature, and ZK-unfriendly storage mechanisms. Early solutions utilized recursive proofs to aggregate computation steps, leading to the first generation of Layer 2 ZkEVMs. Folding schemes further optimized this by compressing instances. The current trend involves Zero-Knowledge Virtual Machines (ZkVMs), especially those based on the simpler, register-based RISC-V ISA [6], which is more amenable to ZK proof generation.

While many ZKP systems rely on assumptions vulnerable to quantum attacks, lattice-based cryptography provides a foundation for post-quantum secure ZK solutions, with security often based on hard problems like SVP or SIS [7].

This thesis investigates the design and implementation of a RISC-V ZkVM using lattice-based cryptography. A key application is proving Ethereum block execution, contributing to the network's scalability and post-quantum security. This work explores the practical construction of such a system and aims to assess whether lattice-based cryptography can offer performance comparable to classical ZKP methods while ensuring quantum resistance.

Document Structure

This work is structured as follows. Chapter 2 provides the necessary background and analysis of the current state of ZkVMs and lattice-based cryp-

tography. It begins with an overview of Ethereum's consensus and execution mechanisms 2.1, followed by an analysis of Snarkification initiatives 2.2. The chapter then examines ZkEVMs 2.3, discussing the evolution from recursive proofs to modern folding schemes and RISC-V based ZkVMs. Finally, it introduces lattice-based cryptography 2.4, covering Ajtai commitments and recent developments in lattice-based zero-knowledge systems.

Chapter 3 defines the research gap regarding post-quantum secure ZkVMs and establishes the specific scientific questions and objectives that guide the solution design and implementation. It outlines the problem of the lack of lattice-based RISC-V ZkVM designs and specifies the feasibility, performance, and long-running computation questions to be addressed.

Chapter 4 details the architectural and cryptographic design of the proposed lattice-based RISC-V ZkVM. It presents the high-level architecture 4.1 describing the proof lifecycle from Rust code to succinct argument, specifies the cryptographic primitives 4.2 including LatticeFold, SuperSpartan, and Greyhound, and describes the construction of the RISC-V step circuit 4.3 and the IVC-augmented circuit 4.4.

Chapter 5 presents the prototype implementation of the lattice-based RISC-V ZkVM. It describes the technology stack 5.1 and implementation details, including the RISC-V emulator, Poseidon2 integration, and circuit synthesis with CCS constraints for the IVC step commitments.

Chapter 6 provides an initial evaluation of the implemented components. It presents baseline performance measurements from the Fibonacci benchmark on consumer hardware, discusses the limitations of the incomplete implementation, and addresses the research questions from both design and implementation perspectives.

Chapter 7 concludes the work by summarizing the design and implementation achievements and outlining directions for future work, including formal verification, LatticeFold+ integration, GPU acceleration, and lookup incorporation.

Chapter 2

Analysis

2.1 Ethereum

Ethereum is a decentralized, open-source blockchain with smart contract functionality [1]. Conceptualized by Vitalik Buterin in 2013/2014, it went live in 2015, aiming to build a platform for decentralized applications that adds Turing complete execution environment to blockchain [8]. Ethereum enables developers to create and deploy decentralized applications and crypto assets. The Ethereum network is comprised of two layers, two peer-to-peer networks, consensus layer and execution layer.

2.1.1 Consensus

Consensus on Ethereum is achieved with proof-of-stake (PoS) consensus mechanism called Gasper [2]. PoS system relies on crypto-economic incentives, rewarding honest stakers (people who put economic capital in the network) and penalizing malicious ones.

Stakers, or also called validators, propose new blocks. Validator is selected for a block proposal pseudo-randomly from the pool in each slot. A slot is some time amount (as of writing of this work it is 12 seconds on Ethereum mainnet), in which the pseudo-randomly chosen validator can propose new block. The software creating the new block is called consensus client. Proposer's consensus client requests a bundle of transactions from the execution layer 2.1.2, wraps them into a block and gossips (sends) the new block to other participants over the consensus p2p network. The rest of the validator pool can in 32 slot (one epoch) attest to that new block's validity. In order for a block to be finalized, it must be attested by a super-majority, which is 66% of the total balance of all validators. [3].

2.1.2 Execution

Software operating on the execution layer is called execution client. Nodes on execution layer hold the latest state and database of all Ethereum data. These clients gossip incoming transactions over the execution layer p2p network, and each stores them in a local mempool. Once they are put inside a block, each transaction is executed in Ethereum-Virtual-Machine (EVM). EVM is a stack based virtual machine operating 256 bit words which executes smart contract code. Ethereum is a decentralized state machine, with rules defined by the EVM. EVM can be thought of as a function. Given a valid state and valid set of transactions[1], outputs next valid state:

$$F(state_n, T) = state_{n+1}$$

[1] validity of transactions, and thus transitively validity of state, is guaranteed by the consensus layer 2.1.1

Thus, Ethereum's state transition function is described by the EVM. This function must be executed by each execution node for each new block in order to keep up with the current chain state. [9]

2.2 Snarkification of Ethereum

As discussed in sections on consensus (2.1.1) and execution layer (2.1.2), nodes on each layer must expend compute resources to validate the consensus data of new blocks or re-execute all transactions within them to verify and maintain the blockchain's state. These computational demands limit the scalability of the entire network.

Ethereum's decentralized node network includes not only participants with capable servers, but also hobbyists, who use home internet connections and less powerful machines. Naively attempting to scale the network, for instance, by decreasing slot time, or increasing block size may raise bandwidth and node requirements, thereby ousting less capable nodes and thus hindering Ethereum's decentralization.

Snarkification refers to the process of integrating a specific type of Zero Knowledge Proofs (ZKPs) called SNARK (Succinct Non-interactive Argument of Knowledge) in order to offload a computation to one entity (one node, or a cluster that is a small subset of the whole network), which performs the computation and generates a proof of its validity. This proof can then be verified by the rest of the network at fractional compute cost.

2.2.1 Zero Knowledge Proofs

Zero-Knowledge Proofs are cryptographic primitives that allow a prover to convince a verifier of a statement's truth without revealing any information beyond the statement's validity itself. The statement typically concerns membership in an NP language/relation [10]. The membership statements are usually named ZK circuits. Traditionally, ZKPs are interactive protocols where a prover, who knows a private witness for a statement (e.g., a solution to a problem), aims to convince a verifier of this knowledge. Through a series of back-and-forth interactions, the verifier becomes convinced of the prover's claim (if true) without learning the witness itself.

Interactive proofs where the verifier's messages consist only of random challenges are known as public-coin protocols. The Fiat-Shamir heuristic [11] can transform such public-coin interactive proofs into non-interactive proofs by replacing the verifier's random challenges with challenges derived from a cryptographic hash function applied to the protocol's transcript. This non-interactivity allows a single generated proof to be validated by any number of verifiers.

A SNARK is a specific type of ZKP. The 'Succinct' aspect implies that proof sizes are very small (e.g., polylogarithmic in the size of the witness or statement) and verification is faster than re-executing the original computation, often polylogarithmic in the computation's complexity [12].

2.2.2 Snarkifying consensus

The 'Lean consensus' [4] is a research initiative, introduced by Justin Drake at Devcon 2024 [13], aimed at redesigning and improving Ethereum's consensus mechanism. A key aspect of this proposal involves leveraging SNARKs

to make consensus layer state transitions provable, in near real-time. Instead of each validator independently re-validating all components of a new consensus state, they would verify a single, compact SNARK. Based on this proof's validity, they can accept or reject the proposed state update.[2]

2.2.3 Snarkifying execution

Ethereum's execution state transitions are dictated by the rules of the EVM. To reduce the computational burden of verifying execution using ZKPs, the EVM's execution logic needs to be provable with a ZKPs, creating a ZkEVM. This would change the state transition function (2.1.2) to:

$$F(state_n, T) = (state_{n+1}, \pi)$$

Where π is the proof of the state transition's validity. Therefore, verifying a block's execution state transition would involve verifying this single proof, rather than re-executing all transactions within the block. Furthermore, an increase in block size would have a little to no impact on verification times for nodes, as proof verification complexity grows much slower than the computation being proven. This could allow for larger block sizes without risking the exclusion of nodes with modest computational capabilities due to their weaker computational capabilities, thereby supporting scalability.

Another approach is to 'enshrine' a ZkEVM directly into the Ethereum protocol (L1). This offers similar benefits in terms of L1 state transition verification, as the native EVM could delegate execution proving to this enshrined ZkEVM. In addition, an enshrined ZkEVM could support and simplify the

[2]The entity proposing the state update, such as a block proposer, would be responsible for generating and submitting this proof.

creation of what are sometimes termed 'native rollups' [5]. These would be Layer 2 solutions that could leverage the L1's enshrined ZkEVM for verifying their own state transitions, which are themselves batches of EVM-compatible transactions. The correctness of these L2 transaction batches would thus be directly enforced by the L1 execution layer through ZK proof verification. This could lead to faster, settlement for these L2s on L1, which in turn could simplify composability between L1 and these native L2s, as well as among different native L2s.

2.3 ZkEVM

ZkEVM, as mentioned in 2.2.3, is a modified EVM, which given state and list of transactions produces next state and a proof π attesting to correctness of that execution. The primary use case of ZkEVMs today is enabling "ZK rollups", L2s which execute a bundle of transactions offchain, generate a proof and submit the bundle with the proof on the L1, where only the proof is validated. A second use case, though not used today, is to enshrine ZkEVM into EVM as described in 2.2.3.

2.3.1 Challenges of EVM

The EVM was not designed with ZK taken into consideration:

1. Many EVM opcodes are complex and expensive to prove with ZK. This has led to different types of EVM compatibility among ZkEVMs [14].
2. 256 bit word size means that ZK systems working over prime fields must include range proofs, which increases the ZkEVM complexity.
3. It is harder to prove a stack based VM. For example, Starkware de-

veloped Cairo [15], a register based model in order to implement its ZkEVM. This requires a custom smart contract compiler.

4. EVM storage uses Merkle Patricia Tries with Keccak, which are not ZK friendly and have huge proving overhead.

Proving the entire EVM within a single circuit (one proof) is computationally and economically unfeasible. Such a proof would be several megabytes in size, making it too expensive to store on L1, let alone prove.

2.3.2 Recursive proofs

Recursive proofs combine benefits of ZKP systems with fast prover times (e.g., those based FRI [16]) and systems with short proofs (like Groth16 [17]). The idea is to produce a proof of a knowledge of a proof.

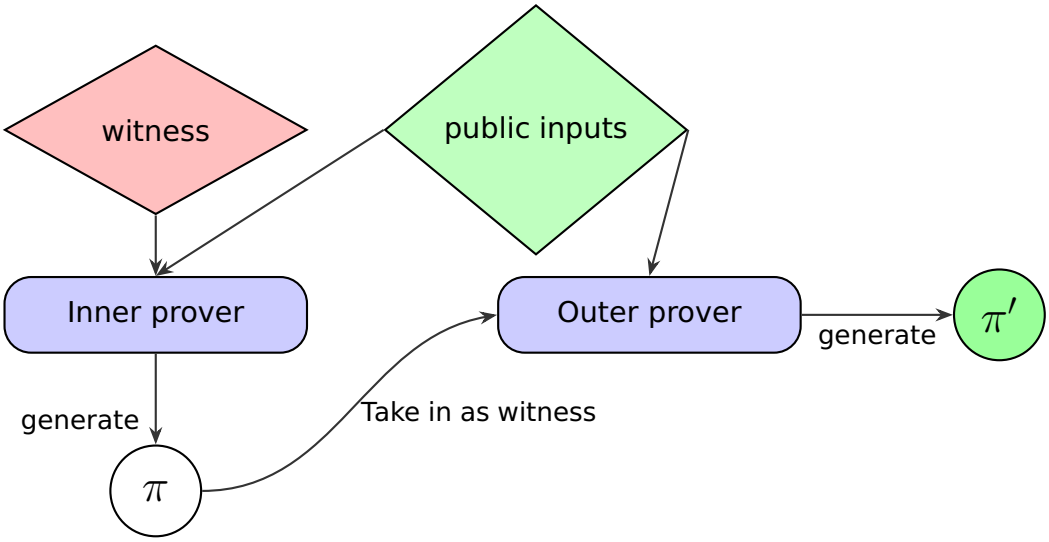

Figure 2.1: Recursive proving

The recursive ZKP is an instantiation of Incrementally Verifiable Computa-

tion (IVC) [18]. IVC is a cryptographic primitive for proving the correctness of an iterated and incremental computation (such as an EVM or RISC-V microprocessor execution).

Let F be a step function (e.g., representing the execution of a single instruction). The output state of step i is fed as input into step $i + 1$, along with a proof π_i attesting to the validity of the history up to that point.

Formally, at each step i, the prover demonstrates that:

1. **State transition correctness:**

$$F(z_{i-1}, u_i) = z_i$$

   This proves that executing the step function F with the previous public state z_{i-1} (e.g., program counter, registers, memory root) and the current non-deterministic advice u_i (e.g., private witness data, memory values) correctly produces the new state z_i.

2. **Recursive verification:**

$$V((i-1, z_0, z_{i-1}), \pi_{i-1}) = \text{true}$$

   This proves that π_{i-1} is a valid proof for the computation path from the initial state z_0 to the previous state z_{i-1}.

Where the variables are defined as:

- i: The current step counter.
- z_0: The immutable initial state.
- z_{i-1}: The output state of the previous step.

- u_i: The private witness or non-deterministic advice required for the current step.
- π_{i-1}: The proof generated at the previous step, which is verified recursively inside the current step.

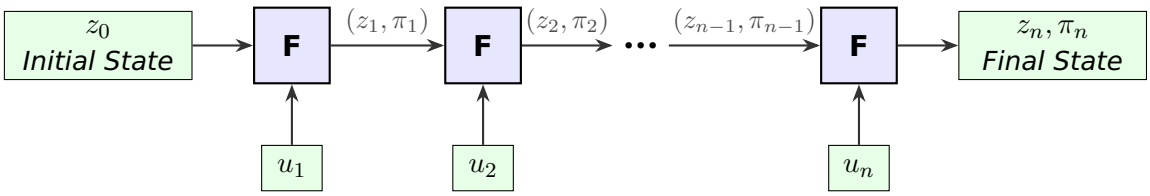

Figure 2.2: Visualization of the IVC chain [18]

The final proof π_n serves as a cryptographic certificate. It attests that the prover possesses a valid sequence of private witnesses (non-deterministic advice) $u_1, \ldots, u_n$, such that applying the transition function F sequentially starting from the initial state z_0 results in the final public state z_n. The first practical implementation of IVC was achieved using recursive SNARKs [19].

2.3.3 First generation of ZkEVMs

In the 2022 to early 2024 first generation of ZkEVMs was successfully deployed. Thanks to recursive proving, it was feasible for these ZkEVMs to be created. They first prove the EVM execution, and then prove it inside another circuit. This enabled parallelization of proving and reduced the size and complexity of the final proof submitted to L1. ZK systems like Plonky2 [20], or Halo2 [21], are examples of recursive ZKP systems. Notable references include Scroll, zkSync Era, Polygon ZkEVM.

2.3.4 Folding schemes

Even though recursion proofs enabled teams to build ZkEVMs, they are not without their flaws. The prover needs to have a circuit which contains the whole verification algorithm of another proof system. It must verify expensive evaluation proofs for polynomial commitments.

Folding removes the need to verify a full SNARK inside the circuit, replacing it with a cheap evaluation of a random linear combination (linearization). It still requires a verifier circuit, but a much cheaper one. There are no FFTs, only multi-exponentiations, which do not require large memory overhead. Also no embedded elliptic curve pairings are needed, because there is no need to switch curves like in recursive proofs.

It compresses two instances into one. Folding prover will fold two instances, with corresponding witnesses, and produce single instance, s.t., if it is correct, it is implied that also the two original instances were correct [22].

Given a relation R, which defines valid instance-witness pairs (u, w) such that $R(pp, u, w) = 1$:

- pp = public parameters,
- u = instance (public inputs, commitments),
- w = witness (private data).

For example, R is an equation, u are coefficients and w are concrete values that satisfy the equation.

A folding scheme for a relation R is an interactive protocol where the prover P and verifier V reduce two instances u and U into a single instance U'.

Let the incoming step be denoted by the pair (u_i, w_i) and the running accumu-

lator be denoted by (U_i, W_i). The folding protocol proceeds as follows:

1. P has $(u_i, w_i) \in R$ and $(U_i, W_i) \in R$.
2. V has u_i and U_i (but does not know the witnesses).

The result of their interaction is a new folded instance U_{i+1} (known to both) and a new folded witness W_{i+1} (known only to P), such that $(U_{i+1}, W_{i+1}) \in R$.

Thus, instead of verifying two distinct instances (u_i and U_i), the verifier only needs to check the single folded instance U_{i+1} at the very end of the computation.

Folding scheme can be made non-interactive by using Fiat-Shamir heuristic [11].

2.3.5 Instantiating IVC with folding scheme

IVC instantiation via folding schemes requires wrapping the base computation F within an augmented function F', where F is the "application" circuit [22, 23]. The function F' is structured to perform two primary tasks:

1. **Base computation:** F' executes the original state transition $F(z_{i-1}, u_i) \to z_i$, where z_{i-1} is the state from the previous step and u_i is non-deterministic advice for step i. This corresponds to verifying the correct execution of one step of the underlying computation (e.g., a single RISC-V instruction cycle).
2. **IVC maintenance:** This task ensures the integrity of the recursive proof chain. F' takes as non-deterministic advice the full state from the previous step: $(i, z_0, z_{i-1}, U_{i-1})$. It then performs two critical checks:

- **Consistency check:** F' re-computes the hash of the input state and asserts that it matches the public input h_{i-1}. This binds the current step's execution to the correct history.
- **Recursive verification:** F' executes the verifier logic of a Non-Interactive Folding Scheme (NIFS). It takes the running accumulator U_{i-1} and the instance u_{i-1} (an instance from the step $i-1$) and outputs a new folded accumulator U_i.

Finally, F' outputs a new hash h_i, committing to the updated state $(i+1, z_0, z_i, U_i)$, which serves as the public input for the next step [22, 23]. Figure 2.3 illustrates this data flow.

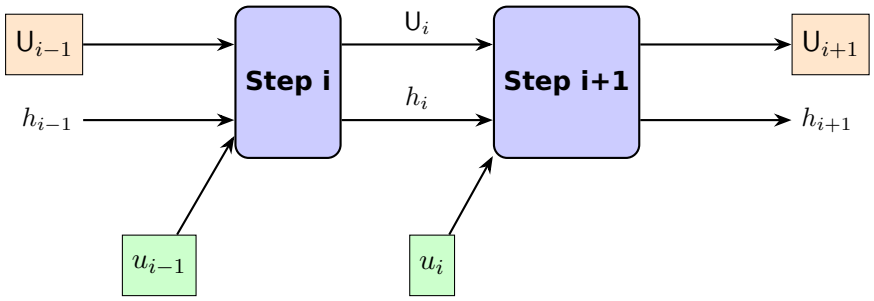

Figure 2.3: IVC instantiation with folding scheme [22, 23]

2.3.6 New generation of ZkEVMs

Currently, the industry is moving towards ZkVMs, dominantly RISC-V ZkVMs. RISC-V is an open-source Instruction Set Architecture (ISA) [6]. RISC-V's simpler, reduced instruction set is better suited for ZK proof computations. Its register-based architecture is generally more efficient for proving than the EVM's stack-based architecture. Another benefit is the mature tooling ecosystem, including compilers like GCC and LLVM, debuggers, and li-

braries. This allows developers to write provable programs in familiar languages like Rust, C, and C++. RISC-V is a general purpose ISA, which opens possibilities for proving not just blockchain applications. Due to this, a EVM written in a language that can be compiled into RISC-V, can be executed inside a ZkVM and together this combination creates a ZkEVM.

RISC-0

First such ZkVM was RISC-0 [24], a general purpose RISC-V VM. It implements the RISC-V RV32IM specification (the RV32I base with the multiplication extension). It uses STARKs [25] with FRI [16], for the inner provers, and the final output proof is generated with Groth16 [17]. Proving process of RISC-0 is as follows:

1. From execution of a program a collection of segments is collected,
2. Each segment is proven with STARK based circuit [26],
3. Pairs of these proofs are recursively proven until only one proof remains,
4. This proof is proven with Groth16.

SP1

SP1 by Succinct is also a RISC-V RV32IM, proving programs in a recursive STARK [25] environment using FRI [16], with final proof being wrapped by Groth16 [17], or PlonK [27] SNARK for small proof sizes [28].

Jolt

Another RISC-V RV32I ZkVM. Uses a technique termed as *lookup singularity*. This technique produces circuits that perform only lookup into massive precomputed lookup tables of size more than 2^{128} [29]. Underlying lookup argument protocol is Lasso [30].

OpenVM

OpenVM introduces a RISC-V ZkVM with novel "no-CPU" design. This decouples opcode implementation and let's developers build custom VM extensions. Using FRI [16] and DEEP-ALI [31]. The overall proving process follows previous designs [24], but enables aforementioned modularization [32].

2.3.7 Comparison

The only sensible comparison method for ZkEVMs is to benchmark them on proving Ethereum blocks. However, this is not doable in this analysis because of two reasons. First, only recently was there some effort to create standardized benchmarks. EthProofs [33] is a portal where ZkEVM teams can join and be measured on the most important thing, the proving times of Ethereum blocks. The average time for proving a Ethereum L1 block is (as of writing this work) around 15.5 seconds, and median being 7.6 seconds.

2.4 Lattice-based cryptography

Lattice-based cryptography has emerged as a significant area of research, particularly due to its potential for constructing cryptographic primitives re-

sistant to attacks by quantum computers. Unlike traditional public-key cryptosystems such as those based on elliptic curves, which are vulnerable to Shor's algorithm on a quantum computer, many lattice-based constructions are believed to maintain their security in a post-quantum world. The security of these schemes often relies on the presumed hardness of certain computational problems on lattices, such as the Shortest Vector Problem (SVP) [34, 35, 7].

A lattice L is a discrete subgroup of $\mathbb{R}^n$, typically represented as the set of all integer linear combinations of a set of linearly independent basis vectors $B = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n\}$ where $\mathbf{b}_i \in \mathbb{R}^n$. That is:

$$L(B) = \left\{ \sum_{i=1}^{n} x_i \mathbf{b}_i \mid x_i \in \mathbb{Z} \right\}$$

The dimension of the lattice is n. Many cryptographic constructions are built upon the difficulty of solving problems like finding the shortest non-zero vector in a lattice (SVP) or finding the lattice point closest to a given target vector (CVP), especially in high dimensions.

One of the pioneering works in this field was by Miklós Ajtai in 1996, who introduced a cryptographic construction whose security could be based on the worst-case hardness of lattice problems [34]. This provided a stronger theoretical foundation for cryptosystems using lattices.

Lattice-based cryptography offers several advantages:

- **Post-quantum security:** As mentioned, this is a primary driver for its adoption.
- **Efficiency:** Some lattice-based operations can be computationally ef-

ficient, particularly those involving matrix-vector multiplications over small integers [35].

- **Versatility:** Lattices have been used to construct a wide array of cryptographic primitives, including public-key encryption, digital signatures, and importantly for this thesis, vector commitment schemes [35].

2.4.1 Rings instead of matrices

In lattice-based cryptography, it is more efficient to use polynomial ring elements, instead of the traditional matrix representations. In these constructions, operations are performed in the quotient ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$, where polynomials are considered modulo $x^n + 1$. This polynomial modulus induces a negacyclic structure. Multiplying by x corresponds to a right rotation of coefficient vectors, with a sign inversion when elements wrap around [36]. The ring-based approach provides significant advantages over unstructured lattices, storing a ring element requires only n coefficients rather than n^2 matrix entries, yielding key sizes that are an order of magnitude smaller [37]. Furthermore, polynomial multiplication can be computed efficiently in $O(n \log n)$ time using the Number Theoretic Transform (NTT), compared to the $O(n^2)$ complexity of matrix vector multiplication [38]. These efficiency gains, combined with the worst-case hardness guarantees inherited from Ring-LWE [38], have made ring-based schemes the preferred approach for practical lattice cryptography.

2.4.2 Ajtai commitments

A commitment scheme is a cryptographic primitive that allows a party (the prover) to commit to a value while keeping it hidden from another party (the

verifier), with the ability to reveal the committed value later. The scheme must satisfy two main properties:

- **Hiding:** The verifier cannot learn the committed value from the commitment itself before the reveal phase.
- **Binding:** The prover cannot change the committed value after the commitment phase.

The Ajtai commitment scheme from Ajtai's 1996 work [34], is a lattice-based commitment scheme. Its security is based on the hardness of the Shortest Integer Solution (SIS) problem [34, 35, 39].

The SIS problem is defined as follows: Given a random matrix $A \in \mathbb{Z}_q^{n \times m}$ (where q is a modulus, n is the row dimension, and m is the column dimension, with $m > n \log q$), find a non-zero integer vector $\mathbf{s} \in \mathbb{Z}^m$ with small norm (i.e., $s_i \in s; ||s_i|| \leq \beta$, for some bound β) such that:

$$A\mathbf{s} = \mathbf{0} \pmod{q}$$

Finding such a short vector $\mathbf{s}$ is believed to be computationally hard for appropriate parameters.

- **Binding:** The binding property relies on the SIS assumption. If a prover could find two different openings (e.g., $\mathbf{s}$ and $\mathbf{s}'$ where $\mathbf{s} \neq \mathbf{s}'$) for the same commitment $\mathbf{t}$, such that $A\mathbf{s} = \mathbf{t} \pmod{q}$ and $A\mathbf{s}' = \mathbf{t} \pmod{q}$, then $A(\mathbf{s} - \mathbf{s}') = \mathbf{0} \pmod{q}$. If $\mathbf{s} - \mathbf{s}'$ is a short non-zero vector, this would imply a solution to the SIS problem for matrix A.
- **Hiding:** The hiding property is based on the Learning With Errors (MLWE) problem or related assumptions. Informally, given A and $A\mathbf{s}$

$(\mathrm{mod}\ q)$ where s is short and random, it is computationally difficult to recover s. The distribution of $A\mathbf{s}\ (\mathrm{mod}\ q)$ is computationally indistinguishable from a uniform random vector over $\mathbb{Z}_q^n$ [40].

A useful feature of some Ajtai commitments is that the size of the commitment t does not grow with the dimension of the message s, though the message components themselves must be small [40].

2.4.3 Lattices in ZK

Lattice-based cryptography presents a compelling avenue for advancing zero-knowledge proof systems, due to its strong security foundations against quantum computers. This makes them a promising candidate for future-proofing ZK solutions.

LaBRADOR: Compact proofs for R1CS from Module-SIS

LaBRADOR, introduced by Beullens and Seiler [41], is a lattice-based proof system designed for Rank-1 Constraint Systems (R1CS). An advantage of LaBRADOR is its ability to generate very compact proof sizes, particularly for large circuits, making it more efficient in terms of proof transmission and storage compared to many other post-quantum approaches. For instance, LaBRADOR can prove knowledge of a solution for an R1CS modulo $2^{64} + 1$ with 2^{20} constraints with a proof size of only 58 KB.

However, a notable drawback of LaBRADOR is that its verifier runtime is linear in the size of the R1CS instance. This means that while the proofs are small, the verification process is not succinct, which can limit its applicability in scenarios requiring fast, sublinear verification. Despite this, LaBRADOR's compact proof generation makes it a candidate as an "outer

layer" or wrapper. It can be used to prove the correctness of a (potentially much larger) proof generated by another succinct, but perhaps less compact, proof system, thereby achieving overall succinctness with post-quantum security.

Greyhound: Fast Polynomial Commitments from Lattices

Building upon the strengths of LaBRADOR, Nguyen and Seiler proposed Greyhound, an efficient polynomial commitment scheme (PCS) based on standard lattice assumptions [42]. Greyhound leverages LaBRADOR as a core component to achieve succinct proofs of polynomial evaluation.

The construction involves a three-round protocol for proving evaluations for polynomials of bounded degree N with a verifier time complexity of $O(\sqrt{N})$. By composing this with the LaBRADOR proof system, Greyhound obtains a succinct proof of polynomial evaluation that also has a sublinear verifier runtime. For large polynomials (e.g., degree up to $N \approx 2^{30}$), Greyhound produces evaluation proofs of approximately 53KB. This demonstrates the practical utility of using LaBRADOR to compress proofs for more complex cryptographic primitives like polynomial commitments.

LatticeFold and LatticeFold+

In the domain of folding schemes, which are techniques for building efficient recursive SNARKs, Boneh and Chen introduced LatticeFold. LatticeFold addressed the challenge of maintaining low-norm witnesses through multiple rounds of folding by employing a sumcheck-based range proof to ensure extracted witnesses remained valid [43].

More recently, Boneh and Chen proposed LatticeFold+ [44], an improved

lattice-based folding protocol. LatticeFold+ enhances its predecessor in several key aspects: the prover is substantially faster (five to ten times), the verification circuit is simpler, and the folding proofs are shorter. These improvements are achieved through two main lattice techniques:

1. A new, purely algebraic range proof that is more efficient than the bit-decomposition approach used in LatticeFold.
2. The use of double commitments (commitments of commitments) to further shrink proof sizes, along with a new sumcheck-based transformation for folding statements about these double commitments.

LatticeFold+ aims to be competitive with, or even outperform, pre-quantum folding schemes like HyperNova [23] while offering plausible post-quantum security.

Chapter 3

Problem statement and research objectives

Based on the analysis of the current state of ZkVMs and the theoretical properties of lattice-based cryptography, this chapter defines the specific problems addressed by this thesis. It outlines the research gap regarding post-quantum secure ZkVMs and establishes the specific scientific questions and objectives that guide the subsequent solution design and implementation.

3.1 Problem statement

The integration of ZKPs into blockchain architectures, particularly Ethereum, offers a path toward massive scalability. As detailed in the previous chapter, the industry standard has converged on ZkVMs that execute instruction sets like RISC-V. These systems allow developers to write programs in more familiar languages (such as Rust or C++) and prove their correct execution.

However, the majority of currently deployed ZkEVMs and ZkVMs rely on cryptographic hardness assumptions, specifically the Discrete Logarithm Problem in elliptic curve groups, that are vulnerable to attacks by sufficiently powerful quantum computers. While hash-based STARKs offer a post-quantum alternative, they often incur significant costs in terms of proof size or verification complexity compared to SNARKs based on specific algebraic structures.

Lattice-based cryptography presents a promising third path. It offers post-quantum security based on the hardness of the Shortest Vector Problem and the Module Short Integer Solution [34, 35]. Furthermore, lattice-based commitment schemes possess additive homomorphic properties that make them theoretically suitable for highly efficient folding schemes, such as LatticeFold [43].

Despite these theoretical advantages, a significant gap exists in the literature and open-source ecosystem, **there is currently no practical solution of a general-purpose RISC-V ZkVM built entirely upon lattice-based primitives.**

The problem addressed by this thesis is the lack of a concrete instantiation of a RISC-V ZkVM that leverages lattice-based folding schemes. It remains unproven whether the theoretical efficiency of lattice folding translates to practical performance for complex computations like a VM. Without such an implementation, it is impossible to accurately compare the performance trade-offs between lattice-based approaches and the elliptic curve architectures.

3.2 Research objectives

The primary objective of this work is to design and implement a functional prototype of a RISC-V ZkVM using lattice-based cryptography, specifically leveraging the LatticeFold protocol for instantiating Incrementally Verifiable Computation of a RISC-V VM. This implementation aims to serve as a proof-of-concept for post-quantum execution environment for proving Ethereum blocks. The specific objectives are defined as follows:

- **Architectural Design:** Design a system architecture for a ZkVM that maps the RISC-V instruction set architecture to the constraints of the LatticeFold folding scheme. This involves defining the arithmetization of 32 bit CPU logic into a Customizable Constraint Systems (CCS) [45] structure, suitable to be used by LatticeFold.
- **Implementation:** Implement the proposed design using the Rust programming language, integrating with the available LatticeFold libraries.
- **Comparative evaluation:** Evaluate the performance of the implemented lattice-based ZkVM. The goal is to benchmark the system against establishing metrics for proving time and memory usage, comparing these results with theoretical expectations and, where applicable, existing non-post-quantum baselines.

3.3 Scientific Questions

To achieve the stated objectives, this thesis seeks to answer the following scientific questions:

1. **Feasibility of lattice-based instantiation:** *Is it feasible to instantiate*

a full RISC-V execution environment using CCS compatible with the LatticeFold protocol?

2. **Performance trade-offs:** *How does the performance of a lattice-based folding scheme compare to traditional elliptic curve-based folding schemes when applied to general-purpose VM execution?*

3. **Suitability for long-running computations:** *Does the noise growth inherent in lattice-based cryptography hinder the ability to prove long execution traces, such as those required for Ethereum block validation?*

Chapter 4

Solution Design

This chapter details the architectural and cryptographic design of the proposed lattice-based RISC-V ZkVM. It is structured to first establish the system's high-level architecture 4.1, describing the lifecycle of a proof from high-level Rust code to a finalized succinct argument. Following this, the specific cryptographic primitives are defined 4.2, selecting the LatticeFold folding scheme [43] as the core recursion engine, while identifying Greyhound [42] as the necessary wrapping SNARK for final verification. The design then proceeds to the construction of the RISC-V step circuit F 4.3, detailing how 32 bit integer arithmetic is arithmetized into the Goldilocks ring elements required by lattice schemes using CCS [45]. Finally, the recursive mechanism F' is defined 4.4, outlining how IVC [18] is enforced to prove the integrity of long running computations.

4.1 High level Architecture

The proposed system architecture implements a post-quantum ZkVM capable of proving the correct execution of arbitrary RISC-V programs. The design leverages IVC to break down the execution of a program into individual steps. To instantiate IVC a folding scheme LatticeFold [43] is used, which recursively folds steps into a single, constant-sized cryptographic object.

The architecture is designed to support the 32 bit RISC-V instruction set (specifically the `rv32imac` extensions). Due to the proof-of-concept nature of this implementation, a subset of opcodes is supported, sufficient to test the viability of a lattice-based ZkVM.

4.1.1 The ZkVM IVC

The lifecycle of a proof within this system proceeds through a pipeline of transformation, execution, and cryptographic accumulation, as illustrated in Figure 4.1. This workflow is defined by the following stages:

1. **Compilation and loading:** A program written in a high-level language (e.g., Rust) is compiled into a `rv32imac` ELF binary. This binary is loaded into the ZkVM emulator.
2. **Execution and trace generation:** The emulator executes the program one instruction at a time. For every step i, the system captures a comprehensive *execution trace*. This trace encapsulates the exact state transition of the machine. Mathematically, the trace for step i contains:
    - **Cycle count:** The step counter.
    - **Input state:** The Program Counter (PC) and the state of all 32

general-purpose registers before execution.

- **Output state:** The PC and register states after execution.
- **Decoded instruction:** The raw binary instruction broken down into its opcode, operands, and immediate values.
- **Side effects:** A set of flags indicating arithmetic overflows, branching behavior (jump targets), and memory operations (address, value, and read/write flags).

3. **Arithmetization and witness generation:** The raw data is transformed into a vector of Goldilocks ring elements $\mathbb{Z}_q$. This vector serves as the *witness* for the constraint system. The logic verifying the transition from input state to output state based on the decoded instruction defines the **Step circuit**, denoted as F.
4. **The augmented circuit F':** To achieve IVC, the step circuit F is embedded inside a larger augmented circuit, F'. This circuit is responsible for the recursive logic. For a step i, F' performs the following operations:
   (a) **IVC integrity checks:** It recalculates the commitment to the previous step's data, ensuring the chain of computation is unbroken. This follows the IVC patterns established in Nova [22] and HyperNova [23].
   (b) **Folding verification:** It constrains the LatticeFold verifier. This enforces that the folding of step $i-1$ into the running accumulator was performed correctly.
   (c) **Step verification:** It enforces the constraints of F, guaranteeing

that the RISC-V instruction at step i was executed correctly according to the ISA.

5. **Cryptographic folding:** The prover executes the non-interactive folding protocol off-circuit. It takes the current step's instance-witness pair (u_i, w_i) and the running accumulator from the previous step (U_{i-1}, W_{i-1}). Using the LatticeFold scheme, these are compressed into a new updated accumulator (U_i, W_i). This process generates a `folding proof`, which serves as a private input for the augmented circuit checks.

6. **Memory logging:** If the instruction execution triggers a memory operation (load or store), this side effect is captured and appended to a global vector of memory operations. Unlike register transitions, memory consistency is not fully proven within F', instead, it is verified via a deferred memory consistency check after the execution concludes.

7. **Final wrapping:** Once the program halts, the system holds a single folded instance-witness pair. To produce a succinct proof suitable for blockchain verification, this large lattice-based artifact is wrapped in a final SNARK as depicted in Figure 4.2. This wrapper utilizes the SuperSpartan IOP [23] to prove the validity of the folded CCS instance, combined with the Greyhound [42] PCS to ensure post-quantum security and succinct verification.

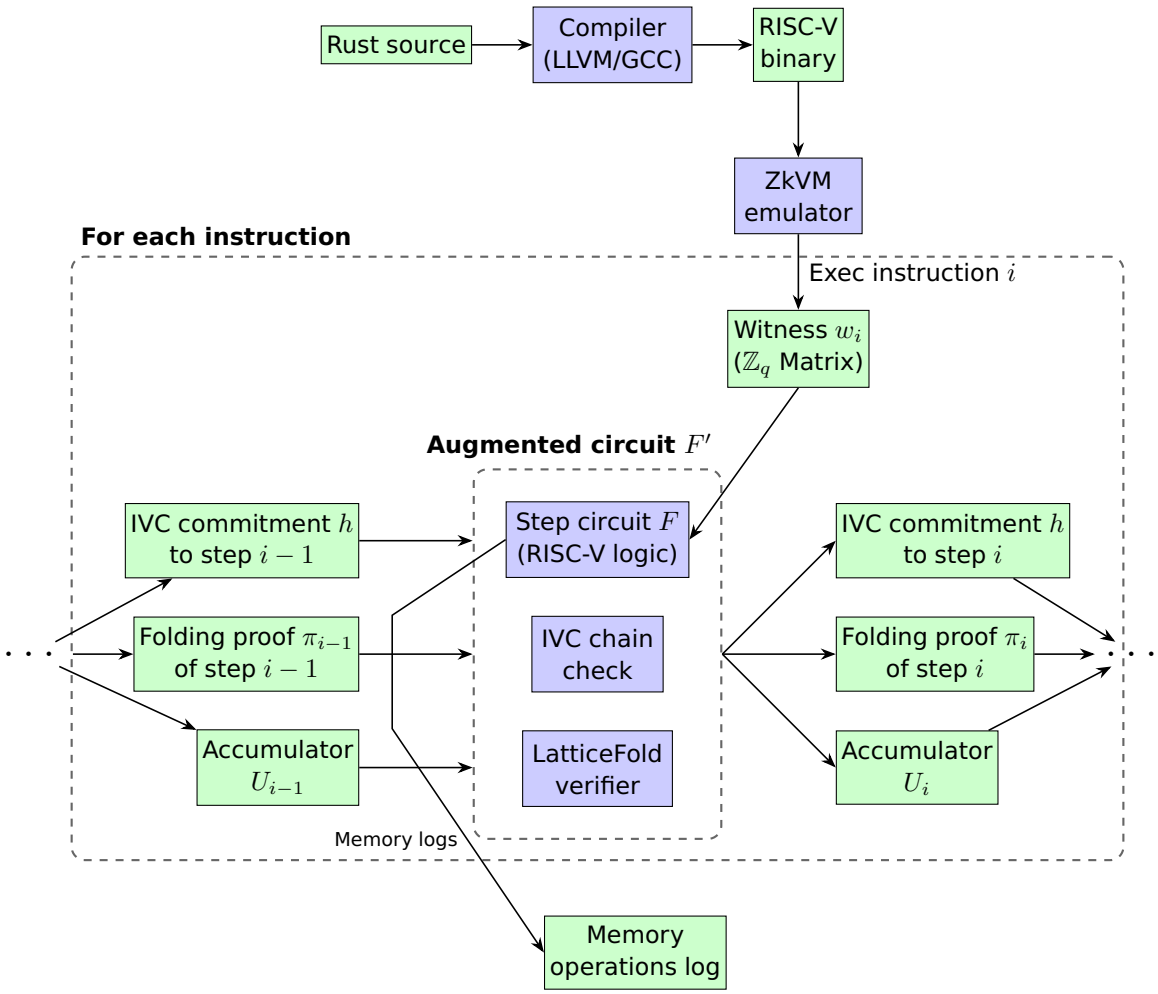

Figure 4.1: Execution flow of the lattice-based RISC-V ZkVM IVC instantiation.

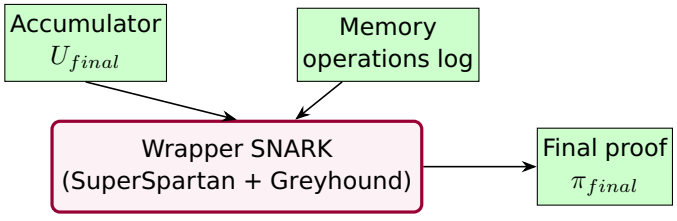

Figure 4.2: Wrapping last folding step into SNARK.

4.1.2 Prover and Verifier Roles

The system distinguishes between the computational roles of the Prover and the Verifier, particularly regarding the recursive steps versus the final verification.

The Prover

The Prover is responsible for the entire computational workload of the VM. Its duties include:

- **Witness generation:** Running the RISC-V emulator to produce execution traces.
- **Folding:** Computing the non-interactive folding steps to compress the current step's witness into the running accumulator.
- **Proof generation:** Generating the final wrapping proof once the recursion is complete.

The Verifier

The verification logic exists at two distinct levels:

1. **The circuit verifier (in circuit):** This is the logic embedded within F' (the augmented circuit). It does not run on an external machine but is simulated by the Prover to generate the proof. It verifies the mathematical correctness of the folding mechanism at every step.
2. **The final verifier:** This is the external entity (e.g., an Ethereum smart contract). It receives the final proof. Its role is to supply initial public arguments and verify the succinct proof of the whole RISC-V VM execution.

4.2 Cryptographic Primitives

ZkVM integrates several zero-knowledge cryptography primitives to achieve IVC via folding. This compositional approach allows the system to benefit from the efficiency of folding in recursive steps while maintaining post-quantum security under lattice-based assumptions. The following section details the selected cryptographic building blocks.

4.2.1 Incrementally Verifiable Computation (IVC)

The foundational theoretical model for this ZkVM is IVC, introduced by Valiant [18]. IVC enables a single proof to attest to the correct execution of a long-running computation $y = F^{(n)}(x)$ by recursively proving that the state at step i is valid and that a valid proof exists for step $i - 1$. RISC-V execution traces of unbounded length perfectly align with this model.

4.2.2 Folding schemes and Nova

To make IVC practical, this work utilizes the concept of *Folding schemes* as introduced in the Nova system [22]. Unlike traditional recursion, where a circuit must fully verify a SNARK at every step, folding allows the prover to compress two instances of the same relation (one representing the current step and one representing the accumulator) into a single instance. The cryptographic cost of folding is negligible compared to SNARK verification.

4.2.3 Customizable Constraint Systems (CCS)

CCS generalizes R1CS [45], allowing for high-degree gates without overhead. This is advantageous for representing IVC constraints (such as con-

straining Poseidon2's S-Box, which is exponentiation to the 7th degree [46]). A CCS structure S consists of:

- Size bounds $m, n, N, \ell, t, q, d \in \mathbb{N}$ where $n > \ell$.
- A sequence of matrices $M_0, \ldots, M_{t-1} \in \mathbb{F}^{m \times n}$.
- A sequence of q multisets $[S_0, \ldots, S_{q-1}]$, where an element in each multiset is from the domain $\{0, \ldots, t-1\}$ and the cardinality of each multiset is at most d.
- A sequence of q constants $[c_0, \ldots, c_{q-1}]$, where each constant is from $\mathbb{F}$.

A CCS instance consists of a public input $x \in \mathbb{F}^{\ell}$. A CCS witness consists of a vector $w \in \mathbb{F}^{n-\ell-1}$. A CCS structure-instance tuple (S, x) is satisfied by a CCS witness w if:

$$\sum_{i=0}^{q-1} c_i \cdot \bigcirc_{j \in S_i} (M_j \cdot z) = \mathbf{0},$$

where $z = (w, 1, x) \in \mathbb{F}^n$, $M_j \cdot z$ denotes matrix-vector multiplication, $\bigcirc$ denotes the Hadamard (entry-wise) product between vectors, and $\mathbf{0}$ is an m-sized vector with entries equal to the additive identity in $\mathbb{F}$.

Informally, for each constant c_i, multiply it by the element-wise product of all $(M_j \cdot z)$ vectors where j comes from multiset S_i. Sum all these results together, and check if the final vector equals zero. If all constraints are satisfied, the result is zero, indicating the witness correctly follows the rules defined by the system.

4.2.4 Lattice-based hardness assumptions

To achieve post-quantum security, the cryptographic primitives in this work rely on hardness assumptions over lattices rather than the Discrete Logarithm Problem (DLP) used in Nova or HyperNova. The security relies on the hardness of the Shortest Integer Solution (SIS) and Module-SIS problems. These problems remain hard even in the presence of quantum computers [34, 7, 35, 40].

4.2.5 LatticeFold

LatticeFold [43] can be though of as a HyperNova [23], but with post-quantum guarantees. It utilizes the Module-SIS [39] problem to construct Ajtai vector commitment scheme. Unlike DLP-based folding, the challenge in LatticeFold is the growth of the witness norm during folding steps. To maintain the binding property of the commitments, the protocol incorporates a sum-check range proof to ensure the folded witness remains within a small, predefined norm bound. This allows the scheme to operate over small prime fields while providing $O(\log n)$ proof size and achieving transparency, as no trusted setup is required.

4.2.6 The wrapping SNARK SuperSpartan + Greyhound

While LatticeFold allows for efficient recursion, the final artifact is a large lattice-based accumulator that is expensive to verify directly. A wrapping SNARK is used to compress this into a succinct proof.

This wrapper is constructed by combining the **SuperSpartan** [23] IOP, which verifies the satisfiability of the CCS relation, with **Greyhound** [42], a lattice-based PCS with polylogarithmic verification complexity.

SuperSpartan is agnostic to the underlying algebraic structure (field vs. ring), allowing instantiation of the Sum-Check protocol over the lattice ring R_q. The final polynomial evaluation query required by SuperSpartan is then proven using the Greyhound PCS, which supports the same ring structure [23, 42].

4.2.7 Memory consistency check

The correctness of the RISC-V execution depends on the integrity of the Random Access Memory (RAM). To avoid the massive overhead of embedding a Merkle Tree or cryptographic accumulator within the step circuit F, this work employs **Offline memory checking**, a technique formalized for SNARKs by Ozdemir, Laufer, and Boneh [47].

Instead of verifying every memory access immediately, the prover maintains a transcript tr of all memory operations (reads and writes) in the order they occurred. Each access is represented as a tuple $_i = (a_i, v_i, w_i)$ where $a_i \in \mathbb{F}$ is the address, $v_i \in \mathbb{F}$ is the value, and $w_i \in \{0, 1\}$ is a write-bit (1 for write, 0 for read).

To prove that these operations are consistent with read-over-write semantics, the system constructs a second transcript tr' by sorting tr lexicographically by (a, t), where each access is augmented with a time field $t_i = i$. The prover must demonstrate three properties:

1. tr' is a **permutation** of tr (same accesses, different order);
2. tr' is (a, t)**-ordered**: accesses are grouped by address and time-ordered within each group;
3. tr' respects **read-over-write (RoW)** semantics: reads return the most

recent written value to each address.

The full memory consistency proof consists of three components, as described by Ozdemir, Laufer, and Boneh [47]:

1. **Permutation proof:** demonstrates that tr' is a permutation of tr. Both transcripts contain the same multiset of memory accesses [47].
2. (a, t)**-ordering proof**: verifies that tr' is correctly sorted by address and time within each address group, using conditional ordering and conditional uniqueness sub-proofs [47].
3. **RoW proof**: enforces read-over-write semantics, ensuring that each read returns the most recent written value to its address [47].

for the detailed construction and security analysis of these protocols, refer to the original work [47].

4.2.8 Future optimization - LatticeFold+

It is noted that a newer iteration of the folding scheme, LatticeFold+ [44], has been proposed. LatticeFold+ introduces algebraic range proofs and double commitments, which reduce the prover overhead by eliminating the need for bit decomposition during witness norm management. However, at the time of system design, the available libraries for LatticeFold+ did not support CCS constraints. Thus, this work utilizes the standard LatticeFold protocol. The architecture is modular, allowing for the backend to be upgraded to LatticeFold+ in future iterations to further optimize proving time.

4.3 Designing the RISC-V circuit (F)

This section details the construction of the step circuit F, which encodes the transition function of the RISC-V VM according to RISC-V ISA [6]. The design targets a 32 bit architecture.

4.3.1 Field representations and arithmetization

The target architecture is a 32 bit RISC-V machine. The state of this machine within the cryptographic proof system is represented by elements from the Goldilocks field. The Goldilocks prime is defined as $p = 2^{64} - 2^{32} + 1$. Since $p > 2^{32}$, a single field element can losslessly represent any 32 bit register value or memory address without overflow during standard representation.

The LatticeFold protocol operates over cyclotomic rings. A Goldilocks field element is mapped into the ring R_q using a Number Theoretic Transform (NTT) representation, where the base field is a 3rd degree extension of the Goldilocks field, and the NTT form consists of 8 components.

Arithmetization is the process of converting numbers handled by the VM, captured in the execution trace, into cyclotomic ring elements over the Goldilocks field.

4.3.2 State representation

The execution state at any step i is encapsulated by the tuple z_i, which contains the following components:

- **Program counter (pc_i):** A 32 bit integer indicating the address of the next instruction.

- **Registers:** An array of 32 general purpose 32 bit registers.
- **Memory state:** Represented by the root of a Merkle tree covering the machine's RAM.
- **Memory log:** A commitment to the vector of memory operations performed up to step i.

The initial state z_0 serves as a public anchor for the computation. Its commitment z_{0_comm} binds the proof to a specific ELF binary (represented by a `code_comm` Merkle root), the entrypoint, and the initial zeroed state of memory and registers. All of these values are public, known to both prover and verifier, this means, they do not have to be recomputed or constrained by the prover within the circuit.

4.3.3 Instruction decoding and execution

For each step, the circuit F enforces valid transitions between z_{i-1} and z_i. This process involves the following constraints:

1. **Fetch:** The circuit verifies that the instruction executed corresponds to the value stored at pc_{i-1}. This is enforced by verifying a Merkle proof against the static `code_comm` root derived from the ELF binary.
2. **Validation:** To ensure that the claimed decoded instruction corresponds to exactly one valid instruction, the circuit enforces three conditions on the set of selector flags $S = \{z_{\texttt{IS\_ADD}}, z_{\texttt{IS\_MUL}}, \dots\}$:
   (a) **Booleanity (are 0 or 1):** $\forall s \in S, s \cdot (1 - s) = 0$.
   (b) **Mutual exclusion (only one is 1):** $\sum_{s \in S} s = 1$.

(c) **OPCODE consistency:** Each instruction word contains a 7 bit OPCODE[1]. To enforce consistency between the active selector flag and the instruction, opcode bits $z_{\texttt{INSTRUCTION}}[0:7]$ are constrained to match the weighted sum of opcode constants:

$$\sum_{s \in S} (s \cdot \texttt{OPCODE}_s) = z_{\texttt{INSTRUCTION}}[0:7]$$

Where $\texttt{OPCODE}_s$ is the corresponding constant to the instruction s[2]. Since only one flag s is active, this sum collapses to the constant $\texttt{OPCODE}_s$ of the selected instruction.

(d) **functX consistency:** Certain instructions need additional fields from their binary format to be uniquely decodable. Instruction flags subset of those that require funct3 is defined as $S_{\texttt{F3}} \subset S$ and subset for those that require funct7 as $S_{\texttt{F7}} \subset S$[3]. The circuit enforces:

$$\sum_{s \in S_{\texttt{F3}}} s \cdot (z_{\texttt{INSTRUCTION}}[12:14] - \texttt{F3}_s) = 0$$

$$\sum_{s \in S_{\texttt{F7}}} s \cdot (z_{\texttt{INSTRUCTION}}[25:31] - \texttt{F7}_s) = 0$$

Instructions that do not require these fields, are not in the subsets, and these terms equal to 0.

3. **Range checks:** Based on the decoded instruction, source registers ($rs1, rs2$), destination register (rd), and/or immediate values are checked inside the circuit. Register indices must be bounded $rs1, rs2, rd \in [0, 31]$ and immediate values are constrained to the valid bit width for each

[1] E.g. 0110011 for R type, register-to-register, instructions
[2] ADD and SUB are R type instruction, which means, for them $\texttt{OPCODE}_s = 0110011$
[3] Instruction ADD and SUB have the same OPCODE and funct3, but their funct7 differ

instruction type.

4. **Execution logic:** Based on the decoded instruction, the circuit enforces it's semantic logic.

5. **State update:** The circuit constrains the output state z_i. For arithmetic instructions, this involves updating the destination register. For branch instructions, it involves updating the pc.

4.3.4 Memory consistency checks

Memory consistency ensures that a load operation at step i returns the value written by the most recent store operation to that address. The verification of the memory consistency is deferred until the end, when all memory access logs have been collected. Done this way, the circuit inside IVC is less complex and does not need to keep track of the history of memory accesses in state.

When a load or store instruction is executed:

1. **Operation capture:** The circuit captures the operation, which contains the cycle count i, the address, the value, and a boolean flag indicating a write operation.

2. **Log update:** This operation is appended to the memory log, and new commitment (from the previous one and the parts of the new memory operation) is passed as public input. Within F, this new commitment is recalculated and compared to the supplied public input.

3. **Page validation:** For store operations, the circuit additionally verifies a Merkle proof for the memory page being modified, calculates the new page root, and updates the memory Merkle root in the state z_i.

The memory log is not fully verified against the register state at every step. Instead, the correctness of the read/write history is verified in the final wrapping SNARK (discussed in subsection 4.2.6) using checks described in previous subsection 4.2.7.

4.4 IVC (F')

To achieve incrementally verifiable computation, the step circuit F is embedded within a larger augmented circuit, denoted as F'. This circuit is responsible for maintaining the integrity of the proof chain and verifying the correct folding of previous steps [22].

4.4.1 Enforcing IVC integrity

The validity of the recursion relies on linking the current step i to the verified history of the computation terminating at step $i-1$. This linkage is enforced through the hash commitment chain. The augmented circuit F' accepts a public input h_{i-1}, which is a Poseidon2 [46] hash commitment to the state of the IVC scheme after the previous step. The private witness supplied by the prover contains preimage of the h_{i-1}. The preimage consists of:

1. **Previous step counter** $i-1$
2. **Initial state commitment** $z_{0_{\text{comm}}}$
3. **Previous state commitment** z_{i-1_comm}
4. **Previous accumulator commitment** U_{i-1_comm}.

Inside the circuit, the integrity check is performed by first recomputing the Poseidon2 step commitment h'_{i-1} from preimage elements from private wit-

ness. A constraint asserts that the recomputed candidate h'_{i-1} is equal to the provided public input h_{i-1}.

Secure all the way down

The IVC construction in Nova [22] enforces a security model with only a "one layered step commitment" h. However, in the context of a ZkVM, the step commitment h is not comprised of only raw values but of more cryptographic commitments.

If the internal structure of z_{i-1_comm}'s and U_{i-1_comm}'s preimages are not constrained within the circuit, a malicious prover could forge a h_{i-1} to commit to fake z^*_{i-1} or fake U^*_{i-1}.

To prevent such attacks and ensure the integrity of the virtual machine state transitions, the augmented circuit F' must recompute the Poseidon2 U_{i-1_comm} commitment from its preimage U_{i-1} supplied through the private witness. And also it must enforce constraints on the z_{i-1_comm}'s preimage:

1. **Code integrity:** The preimage contains the Merkle root of the program's binary code. This root enables the verification of instruction decoding via Merkle opening proofs. Since the program code is static and immutable, this root must not change during execution. To enforce this, the circuit accepts the expected code Merkle root as a public input (supplied by the verifier). The circuit constrains that the code root contained within the preimage of z_{i-1_comm} matches this global input, preventing the prover from executing instructions from a different program.

2. **Program counter:** The program entry point and current execution pointer are tracked. The circuit logic F determines the next pc based

on the current instruction (e.g., sequential execution or branching). The circuit constrains that the pc component in the preimage of z_{i_comm} matches this calculated value.

3. **Memory pages:** The VM memory is represented by a Merkle root. When a store instruction is executed, the circuit requires a Merkle opening proof for the target page against the root in z_{i-1_comm}'s preimage.

4. **Register state:** The state includes a Poseidon2 commitment to the 32 general purpose registers. The values of these registers before and after the instruction are provided as part of the private witness for the step circuit F. The circuit recalculates the commitment to the output register values and asserts equality with the register component of the z_{i_comm}'s preimage.

5. **Memory operations log:** The log represents a history of all memory operations. The commitment to this log must be updated incrementally. If a memory operation occurs during step i, the circuit computes the new log commitment by hashing the previous commitment (from z_{i-1_comm}) together with the details of the current operation. This result is constrained to match the log commitment in z_{i_comm}'s preimage.

4.4.2 The on circuit NIFS verifier

Following the integrity check, F' executes the verification logic for the LatticeFold Non Interactive Folding Scheme (NIFS). This verifies the mathematical correctness of the folding process without re executing the prior computation.

The circuit accepts the folding proof π_{i-1} as part of the witness. It then executes the LatticeFold verifier with the previous accumulator U_{i-1} and proof π_{i-1}, both taken from the private witness. Upon successful verification of the folding, the circuit executes the RISC-V step circuit F, as detailed in Section 4.3.

4.4.3 Output generation

The execution of F' concludes by generating the artifacts required for the subsequent step. This includes the new folded accumulator instance U_i, the new folding proof π_i, and the new IVC commitment h_i. h_i is calculated by hashing the updated step counter, the initial state commitment z_{0_comm}, the new state commitment z_{i_comm}, and the new accumulator commitment U_{i_comm} using Poseidon2. This h_i becomes the public input for the next iteration of the recursive loop.

Chapter 5

Implementation

This chapter presents the prototype implementation of the lattice-based RISC-V ZkVM architecture described in the Chapter 4. As of writing this work, the implementation covers a subset of RISC-V instructions sufficient to run simple programs such as an Nth Fibonacci number computation, the CCS constraints for these instructions, and the CCS constraints for the Poseidon2 IVC step commitment. The circuit NIFS verifier and the final SuperSpartan+Greyhound wrapper are under development.

Regarding the choice of folding scheme, this work utilizes the LatticeFold protocol rather than its successor LatticeFold+, as the latter's library implementation was not available at the time of system design and development. In future migration to LatticeFold+ or other folding schemes is possible.

5.1 Technology stack

The prototype implementation of the lattice-based RISC-V ZkVM is developed using the Rust programming language (`nightly-2025-08-19`). Rust was selected due to its dominance in the cryptography and zero-knowledge ecosystem.

The cryptographic foundations and virtual machine components rely on several crates:

- **Plonky3**[1]**:** The `Plonky3` meta crate is utilized to provide the core arithmetization primitives. The `p3-field` and `p3-goldilocks` crates are used for Goldilocks field arithmetic. `p3-poseidon2` for the construction of Poseidon2 hashes and `p3-merkle-tree` for memory and code commitments.
- **LatticeFold**[2]**:** The core folding mechanism is based on the `latticefold` implementation provided by Nethermind. This crate implements the Ajtai commitment scheme and the NIFS prover and verifier logic. It is the primary component of the IVC loop, enabling the recursive accumulation of execution steps.
- **Cyclotomic-rings**[3]**:** From the same source as LatticeFold crate, the `cyclotomic-rings` crate is used to handle the specific algebraic structures required for lattice-based cryptography over ring $R_q = \mathbb{Z}_q[X]/(X^d + 1)$. It provides the necessary NTT implementations and ring arithmetic for polynomials over the Goldilocks field.

[1] https://github.com/Plonky3/Plonky3
[2] https://github.com/NethermindEth/latticefold
[3] https://github.com/NethermindEth/latticefold

5.2 Cryptography

Cryptographic crates (and majority of all cryptographic libraries, not just for Rust) expose high level API. Developers provide inputs to this API and get back the desired hash, encrypted text, key, etc. In context of ZkVM, also the intermediate states are needed to populate the witness. Main hurdle during implementation was to reuse exposed parts of used cryptographic crates, in order to collect internal states of these primitives.

5.2.1 Poseidon2

The Poseidon2 hash function is used to commit to the IVC state. While the `Plonky3` library provides a implementation of Poseidon2, its standard interface treats the permutation as a black box, returning only the final digest. To correctly constrain the Poseidon2 execution within a CCS requires access to the intermediate states of the permutation. This is because Poseidon2 has a 7th degree S-box, which is dependent on previous rounds, and those on ones before them. Thus, the computation is broken down into smaller steps, where each step verifies a transition of a one computational segment.

A custom implementation of the Poseidon2 permutation was constructed, structurally identical to the `Plonky3` implementation but modified to capture and return the state vector at desired boundaries:

- The state after the initial MDS matrix application.
- The state after each of the external initial full rounds.
- The state after each of the internal partial rounds.
- The state after each of the external terminal full rounds.

5.2.2 LatticeFold Integration

The LatticeFold library is utilized to perform the non-interactive folding of the witness-instance pairs. The internal logic of the LatticeFold prover was not modified to expose intermediate states, because this step was not implemented as of writing this work.

5.3 RISC-V Emulator

A custom, lightweight RISC-V interpreter was developed to execute the program and generate the necessary cryptographic witnesses. The emulator supports the `rv32imac` instruction set architecture. Since it being only a PoC, not all instruction handlers are implemented, but sufficient amount of them to execute Rust binaries compiled for the `riscv32imac-unknown-none-elf` target.

The core of the emulator is structured around an execution loop that fetches, decodes, and executes instructions. At each instruction step, the execution halts and the full machine state, including the program counter, registers, memory pages, and raw instruction bytes, is exposed to an interceptor.

This way, the instruction semantics and the proof generation are decoupled. The emulator is responsible solely for correct state transitions and the generation of the execution trace, while the ZkVM utilizes this trace to generate witnesses.

5.4 Circuit synthesis

The constraints defining the validity of the VM execution and the IVC chain are synthesized into a CCS.

5.4.1 Layout Management

A central component of the synthesis is the `CCSLayout` structure. This structure dynamically maps semantic variables (such as `is_add`, `val_rs1`, or specific Poseidon2 round constants) to column indices within the CCS witness vector z.

This dynamic allocation allows for the easy addition of new constraints. When a new variable is registered in the layout (e.g., adding support for a new instruction), the layout automatically shifts all subsequent indices. The witness generation logic in arithmetization and the constraint definition logic in remain synchronized without manual recalculation of column offsets.

5.4.2 Poseidon2 constraint example

This section illustrates the construction of CCS constraints for Poseidon2 hash. Poseidon2 permutation in Goldilocks field on 12 elements (rate is 12, with capacity 4, total width then being 16) consists of:

1. Applying a maximum distance separable MDS matrix.
2. 4 external initial rounds a 7th degree S-box on all elements.
3. 22 internal rounds with a partial S-box applied only to the first element.
4. 4 external terminal rounds a 7th degree S-box on all elements.

Initial MDS application. The first operation applies the MDS matrix to the absorbed state. The input 12 elements and 4 zeroes must be in the witness. Here this input will be denoted as $s = (s_0, s_1, \ldots, s_{11}, 0, 0, 0, 0)$. Then the output of the MDS application is captured into the witness, denoted as s_{after_mds}. The constraint enforces

$$s_{after\_mds\_i} - s'_i = 0$$

where $i \in (0, 1, \ldots, 15)$ and $s' = \mathsf{MDS} \cdot s$.

External initial rounds. Next are external rounds where a round constant is added and S-box is applied is applied to all state elements. There are 4 external initial rounds, after each the state of 16 elements must be captured (denoted as $s_{\text{ext\_init\_r\_i}}; \quad r \in (0, 1, 2, 3); \quad i \in (0, 1, \ldots, 15)$). Then in circuit, for each round there is a constraint:

$$\mathsf{MDS}^{-1} \cdot s_{\text{ext\_init\_r\_i}} - (s_{\text{in\_r\_i}} + c_{\text{ext\_init\_r\_i}})^7 = 0 \tag{5.1}$$

where $c_{\text{ext\_init\_r\_i}}$ is the specific constant at index i for round r, and the $s_{\text{in\_r\_i}}$ is the input to the round, in first one, it is the s_{after_mds}, otherwise it is the output of previous round.

Internal rounds. Following the external initial rounds, the permutation executes 22 partial internal rounds where the S-box is applied only to the first state element. The state after each round is captured into witness, denoted as $s_{\text{inter\_r\_i}}; \quad r \in (0, 1, \ldots, 21); \quad i \in (0, 1, \ldots, 15)$. Each internal round computes for each first element:

$$M_I^{-1} * s_{\text{inter\_r\_0}} - (s_{\text{in\_r\_0}} + c_{\text{inter\_0}})^7 = 0 \tag{5.2}$$

For other than first element ($i \in (1, 2, \ldots, 15)$ it does:

$$M_I^{-1} * s_{\text{inter\_r\_i}} - s_{\text{in\_r\_i}} = 0 \tag{5.3}$$

The $s_{\text{in\_0}}$ is the output of the last external initial round, other $s_{\text{in\_i}}$ are outputs of the previous internal rounds.

External terminal rounds. The permutation concludes with 4 external terminal rounds, mirroring the external initial rounds but using terminal round constants and different inputs $s_{\text{ext\_term\_r\_i}}; \quad r \in (0, 1, 2, 3); \quad i \in (0, 1, \ldots, 15)$):

$$\mathtt{MDS}^{-1} \cdot s_{\text{ext\_term\_r\_i}} - (s_{\text{in\_r\_i}} + c_{\text{ext\_term\_r\_i}})^7 = 0 \tag{5.4}$$

where $c_{\text{ext\_term\_r\_i}}$ is the specific constant at index i for round r, and the $s_{\text{in\_r\_i}}$ is the input to the round, in first one, it is the last internal round, otherwise it is the output of previous external terminal round.

Output extraction. After the final external terminal round, the output digest is extracted from first 4 elements of the state

$$h = (s_0, s_1, s_2, s_3) \tag{5.5}$$

And compared to the claimed result.

Complexity analysis. For a single Poseidon2 sponge pass, the constraint system captures 388 witness variables across the permutation computation. The system requires 44 sparse matrices to encode the linear transformations, round constants, and S-box operations. The number of matrices remains constant regardless of the number of sponge passes, though their dimensions scale with the permutation state size. The highest degree constraint is a 7th degree multiset constraint corresponding to the Goldilocks S-box, and this degree does not increase with additional sponge passes.

5.5 High level overview

Figure 5.1 illustrates the complete execution flow. The setup phase prepares the VM by loading the ELF binary and constructing the CCS constraint system. Within the execution loop, four main modules work together: the `riscvm` module handles instruction execution and trace generation, the `commitments` module creates Poseidon2 hashes for state components, the `ivc` module arithmetizes the trace into the CCS witness vector, and the `main` module performs the folding operations that compress each step into the accumulator. The dashed line on the right shows the recursive nature of the computation, where the `IVCStepOutput` from step i becomes the `IVCStepInput` for step $i + 1$.

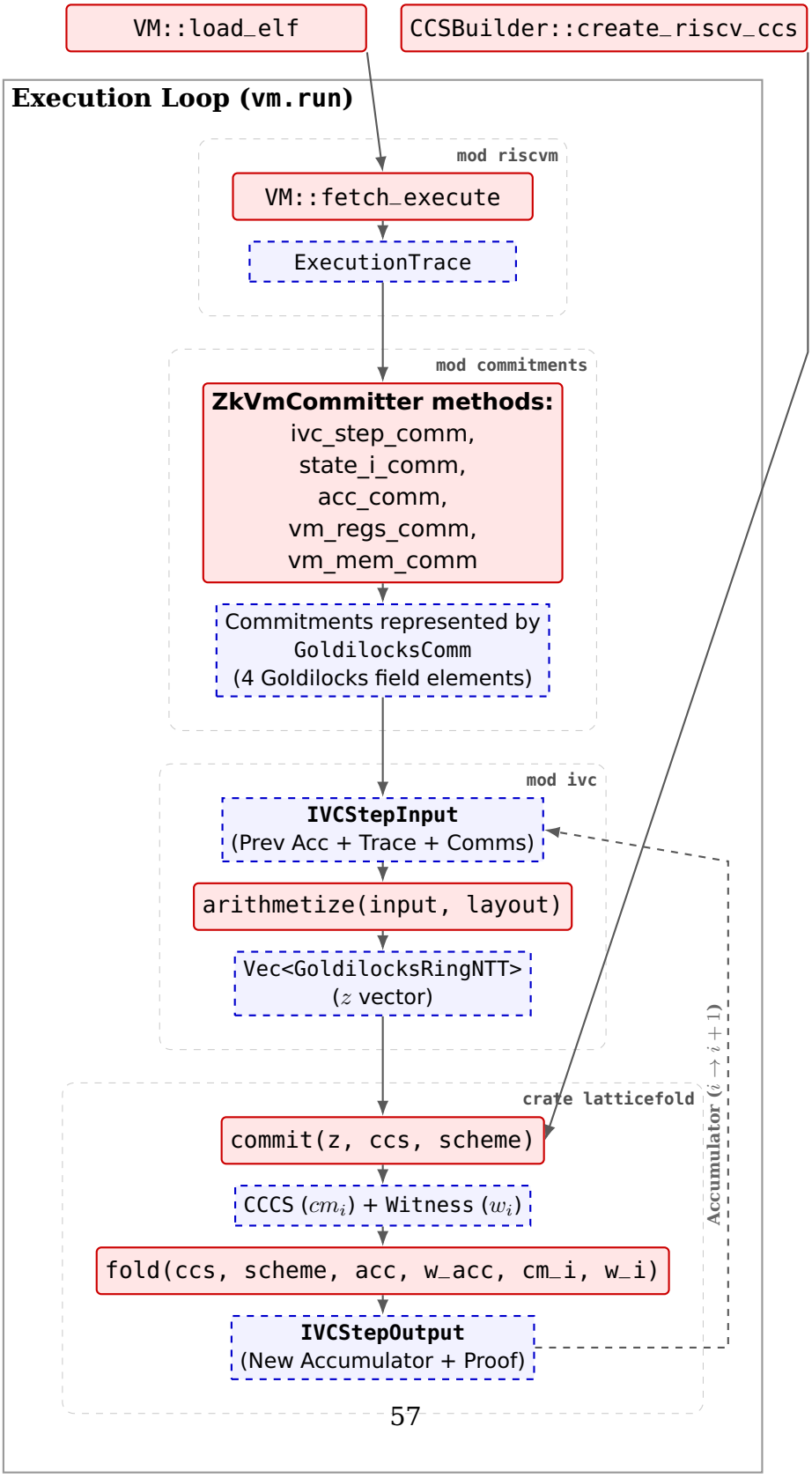

Figure 5.1: High-level overview of the implemented execution flow.

Chapter 6

Evaluation

This chapter provides an initial evaluation of the lattice-based RISC-V ZkVM implementation. It is important to note that the implementation is currently incomplete and does not represent a fully functional system. The results presented here establish baseline performance metrics for the implemented components rather than providing a comprehensive performance comparison with existing ZkVMs.

The performance measurements were conducted on consumer-grade hardware using the following system configuration:

- **CPU:** 11th Gen Intel i7
- **OS:** Arch Linux
- **Memory:** 16 GB RAM

6.1 Limitations

This evaluation is subject to several important limitations:

- **Incomplete implementation:** The in-circuit NIFS verifier, memory consistency checks, and final wrapping SNARK are not yet implemented. The measured performance does not reflect the full proving pipeline.
- **Small test case:** The evaluation uses a simple Fibonacci computation (16 execution traces), which is insufficient to assess performance on realistic workloads like Ethereum block validation.
- **Consumer hardware:** The benchmarks were run on consumer hardware. Production deployments of other ZkVMs use GPU clusters.

6.2 Fibonacci benchmark results

A simple benchmark was conducted using a program that computes the 100th Fibonacci number. This program was chosen because it does not need many RISC-V instructions to calculate the result.

The Fibonacci program execution resulted in:

- **Execution traces:** 16 instructions
- **Memory operations:** 2 operations (loads/stores)
- **Total proving time:** 41.95 seconds

6.2.1 Comparison with State-of-the-Art ZkVMs

For context, benchmarks from Fenbushi Capital report proving times for the 100,000th Fibonacci number on an EC2 g5.x16xlarge instance (64 vCPUs, AMD EPYC 7R32, 256 GB RAM, NVIDIA A10G GPU with 24 GB VRAM) [48]:

ZkVM	Time (seconds)
SP1	3.4
RISC Zero	3.7
OpenVM	7.6

The measured proving time of approximately 42 seconds for 16 traces provides does not sound very "performant" nor nowhere close to real time proving of Ethereum blocks. And compared to established ZkVMs it seems miserable. However, the interpretation of this measurement does not have to be so negative. Firstly, the implementation is not complete, is not optimized and is running only on CPU, no GPU acceleration. There is not enough evidence to rule out this design.

6.3 Addressing the research objectives

Feasibility of lattice-based instantiation

Is it feasible to instantiate a full RISC-V execution environment using CCS compatible with the LatticeFold protocol?

Design perspective: The proposed architecture provides a complete theoretical blueprint. The mapping of RISC-V state transitions to CCS constraints and integration with LatticeFold folding is specified, demonstrating architectural feasibility.

Implementation perspective: Partial feasibility demonstrated. The RISC-V emulator and Poseidon2 CCS constraints are functional and integrated with LatticeFold. In-circuit NIFS verifier and complete instruction set coverage remain to be implemented.

Performance trade-offs

How does the performance of a lattice-based folding scheme compare to traditional elliptic curve-based folding schemes when applied to general-purpose VM execution?

Design perspective: The design leverages lattice-based commitments with additive homomorphic properties, theoretically suitable for efficient folding. Performance trade-offs are architecturally addressed through optimization opportunities such as LatticeFold+ and parallelization.

Implementation perspective: No answer can be provided yet, as there is not direct comparison yet. When the implementation will be completed, then benchmarks can be created and compared to existing ZkVMs.

Suitability for long-running computations

Does the noise growth inherent in lattice-based cryptography hinder the ability to prove long execution traces, such as those required for Ethereum block validation?

Design perspective: LatticeFold addresses witness norm growth through decomposition at each folding step. By decomposing the witness during every fold, the system maintains bounded norms, making execution of long-running computations secure and feasible.

Implementation perspective: Inconclusive. The 100th Fibonacci benchmark with 16 traces does not exercise long-running computation paths. Testing with larger programs is required to observe noise growth effects in practice.

Chapter 7

Conclusion

This thesis presented the design and initial implementation of a post-quantum ZkVM based on lattice-based cryptography. The proposed architecture integrates a RISC-V execution environment with the LatticeFold folding scheme, supported by CCS for arithmetization. The design provides a theoretical blueprint, addressing the challenges of post-quantum security and incrementally verifiable computation.

The implementation validates core component integration. The RISC-V emulator successfully generates execution traces, CCS constraints for Poseidon2 hashes are synthesized and integrated with LatticeFold, and the IVC step commitment mechanism is functional. The performance baseline of approximately 42 seconds for a 16 cycles long Fibonacci computation provides a positive outlook.

7.1 Future Work

Several directions for future research and development emerge from this work:

- **Formal verification:** Formal verification of the implementation would provide strong guarantees of correctness.
- **LatticeFold+ integration:** Migrating the folding backend to LatticeFold+ could yield significant performance improvements through its more efficient algebraic range proofs and double commitment techniques.
- **GPU acceleration:** The LatticeFold prover operations, including NTT transformations and matrix-vector multiplications can be sped up with GPU acceleration, thus bringing proving times down.
- **Lookup incorporation:** Integrating lookup arguments could optimize constraint generation for certain operations, potentially reducing circuit size and improving performance for complex instructions.

References

[1] *Ethereum Whitepaper — ethereum.org*. https://ethereum.org/en/whitepaper/. [Accessed 28-05-2025].

[2] Vitalik Buterin et al. *Combining GHOST and Casper*. 2020. DOI: 10.48550/ARXIV.2003.03052. URL: https://arxiv.org/abs/2003.03052.

[3] *Consensus mechanisms — ethereum.org*. https://ethereum.org/en/developers/docs/consensus-mechanisms/. [Accessed 29-05-2025].

[4] *Lean Consensus Roadmap — leanroadmap.org*. https://leanroadmap.org/. [Accessed 10-12-2025].

[5] *Native rollups—superpowers from L1 execution — ethresear.ch*. https://ethresear.ch/t/native-rollups-superpowers-from-l1-execution/21517. [Accessed 31-05-2025].

[6] *RISC-V Ratified Specifications — riscv.org*. https://riscv.org/specifications/ratified/. [Accessed 01-06-2025].

[7] Miklós Ajtai. "The shortest vector problem in L2 is NP-hard for randomized reductions (extended abstract)". In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing - STOC '98*. STOC '98. ACM Press, 1998, 10–19. DOI: 10.1145/276698.276705. URL: http://dx.doi.org/10.1145/276698.276705.

[8] Vitalik Buterin. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf. [Accessed 28-05-2025].

[9] *Ethereum Virtual Machine (EVM) — ethereum.org*. https://ethereum.org/en/developers/docs/evm/. [Accessed 29-05-2025].

[10] Oded Goldreich, Silvio Micali, and Avi Wigderson. "Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems". In: *Journal of the ACM (JACM)* 38.3 (1991), pp. 690–728.

[11] Amos Fiat and Adi Shamir. "How To Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pp. 186–194. ISBN: 9783540180470. DOI: 10.1007/3-540-47721-7_12. URL: http://dx.doi.org/10.1007/3-540-47721-7_12.

[12] Nir Bitansky et al. *From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again*. Cryptology ePrint Archive, Paper 2011/443. 2011. URL: https://eprint.iacr.org/2011/443.

[13] - *YouTube — youtube.com*. https://www.youtube.com/watch?v=lRqnFrqpq4k. [Accessed 31-05-2025].

[14] *The different types of ZK-EVMs — vitalik.eth.limo*. https://vitalik.eth.limo/general/2022/08/04/zkevm.html. [Accessed 01-06-2025].

[15] Lior Goldberg, Shahar Papini, and Michael Riabzev. *Cairo – a Turing-complete STARK-friendly CPU architecture*. Cryptology ePrint Archive, Paper 2021/1063. 2021. URL: https://eprint.iacr.org/2021/1063.

[16] Eli Ben-Sasson et al. “Fast Reed-Solomon Interactive Oracle Proofs of Proximity”. In: *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Ed. by Ioannis Chatzigiannakis et al. Vol. 107. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, 14:1–14:17. ISBN: 978-3-95977-076-7. DOI: 10.4230/LIPIcs.ICALP.2018.14. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ICALP.2018.14.

[17] Jens Groth. *On the Size of Pairing-based Non-interactive Arguments*. Cryptology ePrint Archive, Paper 2016/260. 2016. URL: https://eprint.iacr.org/2016/260.

[18] Paul Valiant. “Incrementally Verifiable Computation or Proofs of Knowledge Imply Time/Space Efficiency”. In: *Theory of Cryptography*. Springer Berlin Heidelberg, 1–18. ISBN: 9783540785248. DOI: 10.1007/978-3-540-78524-8_1. URL: http://dx.doi.org/10.1007/978-3-540-78524-8_1.

[19] Eli Ben-Sasson et al. *Scalable Zero Knowledge via Cycles of Elliptic Curves*. Cryptology ePrint Archive, Paper 2014/595. 2014. URL: https://eprint.iacr.org/2014/595.

[20] *plonky2/plonky2/plonky2.pdf at main · 0xPolygonZero/plonky2 — github.com*. https://github.com/0xPolygonZero/plonky2/blob/main/plonky2/plonky2.pdf. [Accessed 01-06-2025].

[21] *halo2 - The halo2 Book — zcash.github.io*. https://zcash.github.io/halo2/index.html. [Accessed 01-06-2025].

[22] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. *Nova: Recursive Zero-Knowledge Arguments from Folding Schemes*. Cryptology ePrint

Archive, Paper 2021/370. 2021. URL: https://eprint.iacr.org/2021/370.

[23] Abhiram Kothapalli and Srinath Setty. *HyperNova: Recursive arguments for customizable constraint systems*. Cryptology ePrint Archive, Paper 2023/573. 2023. URL: https://eprint.iacr.org/2023/573.

[24] *dev.risczero.com*. https://dev.risczero.com/proof-system-in-detail.pdf. [Accessed 01-06-2025].

[25] Eli Ben-Sasson et al. *Scalable, transparent, and post-quantum secure computational integrity*. Cryptology ePrint Archive, Paper 2018/046. 2018. URL: https://eprint.iacr.org/2018/046.

[26] *The RISC Zero STARK Protocol | RISC Zero Developer Docs — dev.risczero.com*. https://dev.risczero.com/proof-system/proof-system-sequence-diagram. [Accessed 01-06-2025].

[27] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Paper 2019/953. 2019. URL: https://eprint.iacr.org/2019/953.

[28] *Succinct Docs — docs.succinct.xyz*. https://docs.succinct.xyz/. [Accessed 01-06-2025].

[29] Arasu Arun, Srinath Setty, and Justin Thaler. *Jolt: SNARKs for Virtual Machines via Lookups*. Cryptology ePrint Archive, Paper 2023/1217. 2023. URL: https://eprint.iacr.org/2023/1217.

[30] Srinath Setty, Justin Thaler, and Riad Wahby. *Unlocking the lookup singularity with Lasso*. Cryptology ePrint Archive, Paper 2023/1216. 2023. URL: https://eprint.iacr.org/2023/1216.

[31] Eli Ben-Sasson et al. *DEEP-FRI: Sampling Outside the Box Improves Soundness*. Cryptology ePrint Archive, Paper 2019/336. 2019. URL: https://eprint.iacr.org/2019/336.

[32] *openvm.dev*. https://openvm.dev/whitepaper.pdf. [Accessed 01-06-2025].

[33] *Ethproofs | Ethproofs — ethproofs.org*. https://ethproofs.org/. [Accessed 02-06-2025].

[34] M. Ajtai. "Generating hard instances of lattice problems (extended abstract)". In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, 99–108. ISBN: 0897917855. DOI: 10.1145/237814.237838. URL: https://doi.org/10.1145/237814.237838.

[35] Yang Li, Kee Siong Ng, and Michael Purcell. *A Tutorial Introduction to Lattice-based Cryptography and Homomorphic Encryption*. 2022. DOI: 10.48550/ARXIV.2208.08125. URL: https://arxiv.org/abs/2208.08125.

[36] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On Ideal Lattices and Learning with Errors over Rings". In: *Journal of the ACM* 60.6 (Nov. 2013), 1–35. ISSN: 1557-735X. DOI: 10.1145/2535925. URL: http://dx.doi.org/10.1145/2535925.

[37] Sedat Akleylek et al. *An Efficient Lattice-Based Signature Scheme with Provably Secure Instantiation*. Cryptology ePrint Archive, Paper 2016/030. 2016. URL: https://eprint.iacr.org/2016/030.

[38] Chris Peikert. *Public-Key Cryptosystems from the Worst-Case Shortest Vector Problem*. Cryptology ePrint Archive, Paper 2008/481. 2008. URL: https://eprint.iacr.org/2008/481.

[39] Adeline Langlois and Damien Stehle. *Worst-Case to Average-Case Reductions for Module Lattices*. Cryptology ePrint Archive, Paper 2012/090. 2012. URL: https://eprint.iacr.org/2012/090.

[40] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plancon. *Lattice-Based Zero-Knowledge Proofs and Applications: Shorter, Simpler, and More General*. Cryptology ePrint Archive, Paper 2022/284. 2022. URL: https://eprint.iacr.org/2022/284.

[41] Ward Beullens and Gregor Seiler. *LaBRADOR: Compact Proofs for R1CS from Module-SIS*. Cryptology ePrint Archive, Paper 2022/1341. 2022. URL: https://eprint.iacr.org/2022/1341.

[42] Ngoc Khanh Nguyen and Gregor Seiler. *Greyhound: Fast Polynomial Commitments from Lattices*. Cryptology ePrint Archive, Paper 2024/1293. 2024. DOI: 10.1007/978-3-031-68403-6_8. URL: https://eprint.iacr.org/2024/1293.

[43] Dan Boneh and Binyi Chen. *LatticeFold: A Lattice-based Folding Scheme and its Applications to Succinct Proof Systems*. Cryptology ePrint Archive, Paper 2024/257. 2024. URL: https://eprint.iacr.org/2024/257.

[44] Dan Boneh and Binyi Chen. *LatticeFold+: Faster, Simpler, Shorter Lattice-Based Folding for Succinct Proof Systems*. Cryptology ePrint Archive, Paper 2025/247. 2025. URL: https://eprint.iacr.org/2025/247.

[45] Srinath Setty, Justin Thaler, and Riad Wahby. *Customizable constraint systems for succinct arguments*. Cryptology ePrint Archive, Paper 2023/552. 2023. URL: https://eprint.iacr.org/2023/552.

[46] Lorenzo Grassi, Dmitry Khovratovich, and Markus Schofnegger. *Poseidon2: A Faster Version of the Poseidon Hash Function*. Cryptology ePrint Archive, Paper 2023/323. 2023. URL: https://eprint.iacr.org/2023/323.

[47] Alex Ozdemir, Evan Laufer, and Dan Boneh. *Volatile and Persistent Memory for zkSNARKs via Algebraic Interactive Proofs*. Cryptology ePrint Archive, Paper 2024/979. 2024. URL: https://eprint.iacr.org/2024/979.

[48] Fenbushi Capital |. *Benchmarking zkVMs: Current State and Prospects — fenbushicapital.medium.com*. https://fenbushicapital.medium.com/benchmarking-zkvms-current-state-and-prospects-ba859b44f560. [Accessed 29-12-2025].

Appendix A

Project task schedule

A.1 DP1

1st-4th week	Researching recursive proofs, folding schemes, lattice based cryptography
5th-7th week	Researching Ethereum's architecture
8th-9th week	Researching Lean Consensus initiative
10th-11th week	Researching ZkEVMs, ZkVMs
12th-13th week	Writing analysis

A.2 DP2

1st-3rd week	Designing an IVC instantiation with LatticeFold
4th-6th week	Designing integration between IVC loop and RISC-V VM
7th-9th week	Designing memory consistency checks and wrapping SNARK
10th-12th week	Implementing RISC-V VM, RISC-V constraints, IVC loop
13th week	Finishing DP2 document

A.3 DP3

1st-2nd week	Implementing in-circuit NIFS verifier, completing IVC loop
3th-7th week	Implementing memory consistency checks and wrapping SNARK
8th-9th week	Benchmarking against different ZkVMs
10th-11th week	Optimizing implementation
12th-13th week	Finishing DP3 document