Lova: A Novel Framework for Verifying Mathematical Proofs with Incrementally Verifiable Computation

Noel Elias
University of Texas at Austin
Austin, Texas, USA
nelias@utexas.edu

ABSTRACT

Efficiently verifying mathematical proofs and computations has been a heavily researched topic within Computer Science. Particularly, even repetitive steps within a proof become much more complex and inefficient to validate as proof sizes grow. To solve this problem, we suggest viewing it through the lens of Incrementally Verifiable Computation (IVC). However, many IVC methods, including the state-of-the-art Nova recursive SNARKs, require proofs to be linear and for each proof step to be identical. This paper proposes Lova, a novel framework to verify mathematical proofs end-to-end that solves these problems. Particularly, our approach achieves a few novelties alongside the first-of-its-kind implementation of Nova: (i) an innovative proof splicing mechanism to generate independent proof sequences, (ii) a system of linear algorithms to verify a variety of mathematical logic rules, and (iii) a novel multiplexing circuit allowing non-homogeneous proof sequences to be verified together in a single Nova proof. The resulting Lova pipeline has linear prover time, constant verifying capability, dynamic/easy modification, and optional zero-knowledge privacy to efficiently validate mathematical proofs. Code is available at https://github.com/noelkelias/lova.

KEYWORDS

Incrementally verifiable computation, mathematical proof verification, Nova, SNARKs

1 INTRODUCTION

The efficient verification of proofs and computations has been a long-term foundational problem within Computer Science. Proof systems that efficiently verify computations in different contexts have become key components in many new domains. For example, with machine learning models it is absolutely essential to be able to provide a proof of accurate training and verifiable inferencing [21]. Using such a certificate promotes model transparency as users can verify a model's parameters and features instead of utilizing its predictions through a black box. Additionally, in a Blockchain context, the verification of proofs and computations can also be used to authenticate different transactions while maintaining a decentralized and anonymous network [13].

With the rise of Zero-Knowledge proofs [14], many efficient solutions have been proposed to solve this problem. One of the solutions commonly used are SNARKs (Succinct Non-interactive Argument of Knowledge) [5]. In essence, SNARKs enable a "prover" to succinctly prove a statement to a "verifier" in an efficient manner. More specifically, SNARKs enable untrusted provers to be able to demonstrate knowledge of some witness ω to the verifier. This witness ω could be any statement that can be converted to a computational trace. In addition, SNARKs not only offer succinct proofs but also non-interactive and optionally zero-knowledge proofs.

These SNARKs can even be taken a step further and be utilized for verifying the proofs of proofs. Thus a verification circuit of another inner SNARK can be converted to be a witness ω for an outer SNARK: recursive SNARKs [10]. By utilizing compatible SNARKs, one can significantly increase the efficiency and memory costs of such a proof generation.

So at an initial glace, SNARKs seem to solve the problem of verifying mathematical proofs. However, the problem complexity increases when such proofs become dynamic and are simply repeated executions of the same function over and over again. Iterative functions in these proofs might take the current state S and produce some output based on that state as shown in Equation 1.

$$s_i = F(s_{i-1}) = F(F(s_{i-2})) = F(F(...F(s_0))) \tag{1}$$

If we want to prove the correctness of such a state s_i's execution, we would have to verify the entire proof of the computation from s_0 onwards. In addition, proofs for longer executions would be much larger on the verifier's side. This is not ideal and is the motivation for the cryptographic idea of Incrementally Verifiable Computation (IVC) [28].

At a high level, IVC suggests breaking a proof for such a program into its respective iterative sub-steps. To verify each sub-step i we show the following proof π_i: (1) we can verifiably arrive at s_{i-1} starting from the original state s_0 AND (2) $F(s_{i-1}) = s_i$ is computed correctly. Utilizing combinations of these cryptographic checks we can efficiently verify recursive proofs.

1.1 Problem Overview

As discussed, it is essential to be able to verify proof correctness to facilitate trustworthy computation. Many computations within computer science can easily be translated into mathematical proofs or checkable logical proofs. Each step of these logical proofs can be verified using simple axioms including first-order logic (FOL) [2] and the rules of inference. Specifically, we focus on the rules of addition, conjunction, simplification, resolution, modus ponens, modus tollens, disjunctive syllogism, and hypothetical syllogism. Generally, these mathematical/logic proofs not only need to be verified at each independent step for axiomatic correctness but also cumulatively verified for sequential correctness and order.

Throughout the years, optimizations in formal logic and cryptography have dramatically decreased the overhead for the verification of individual steps of proofs to sub-linear times. However, being able to simultaneously compute checks over the entire proof and between proof steps drastically increases the complexity of proof

verification. An alternate approach might be to verify each proof step independently and then aggregate all these verified steps with a cryptographic aggregator. However, this is also problematic as sequential checks to the entire proof are extremely difficult to compute.

In addition, most existing constructions of proof verification are not dynamic. In other words, a verification proof must be recomputed if new proof steps are added to the existing mathematical proof. As a result, once verification proofs are generated, they must remain static and cannot be easily utilized as sub-proofs for proving larger computations. This dramatically decreases the efficiency of many proof verification schemes as each proof must be verified from scratch.

In essence, this is a problem solved by Incrementally Verifiable Computation (IVC). However at the present, there is a lack of viable implementations and general methodologies to be able to efficiently and dynamically verify a basic mathematical logic proof. In particular, there are currently no common-place IVC implementations to verify mathematical proofs for individual correctness and sequential order. In addition, most IVC tools do not work for non-linear systems or for verifying proofs with differing non-repeated steps. So, state-of-the-art IVC protocols like Nova [19] are not immediately applicable to verifying mathematical proofs which can be non-linear and contain many different proof steps. Thus, Lova proposes a general framework to solve these problems while dynamically and efficiently verifying logic proofs using the state-of-the-art Nova system for incrementally verifiable computation [1].

1.2 Literature Review

The field of formal verification offers many solutions to verify computations. Such solutions often work by breaking down the steps of a program into states that are verified based on logical formulas [12]. However, in this paper, we particularly look at an implementation to efficiently verify, aggregate, and append to mathematical proofs utilizing incrementally verifiable computation.

The foundation for most cryptographic solutions to the verifiable computation problem are Interactive Proofs [15]. Interactive proofs are simply n-message protocols between an efficient verifier and an unbounded prover that are both complete and sound as defined below

- Completeness: A prover can convince a verifier to accept a true statement.
- Soundness: If a statement is false, no prover can convince a verifier to accept such a statement with non-negligible probability.

A classic example of such a proof is the Sum-Check protocol [22]. In this protocol, the Prover uses a n-variate polynomial g to convince the Verifier that:

$$\beta = \sum_{x_1 \in \{0,1\}} \sum_{x_2 \in \{0,1\}} \dots \sum_{x_n \in \{0,1\}} g(x_1, x_2, \dots, x_n) \tag{2}$$

This is done by providing the Verifier with oracle access to g and sending a univariate polynomial $g_i(x_i)$ to the Verifier in each round. However the proof size of this protocol is not constant and requires a much less efficient Verifier [29].

[1] see Appendix A for Lova's applications

As an alternative, the Probabilistically Checkable Proof (PCP) theorem [3] is utilized to create much more efficient probabilistic verifiers. These verifiers are required to have oracle access to the proof π and query the proof at a random number of points to either accept or reject the proof statement in efficient time. A new property that is introduced is defined below:

- Perfect Completeness: For every true statement, there exists a proof π that the prover can submit to the verifier to accept.

The most simple example of a PCP is a Linear PCP where the proof π is a linear function of the form:

$$\lambda(a \times x + b \times y) = a \times \lambda(x) + b \times \lambda(y) \tag{3}$$

More complex PCP proofs can be generated alongside an Interactive Oracle Proof (IOP) [7] or even a Polynomial IOP where a verifier checks low-degree polynomials. This is particularly important as polynomials can be utilized to express computations as constraints. We can use this mechanism to mathematically verify that computations are equivalent to their expected value with polynomial properties. One such theorem is the Schwartz-Zippel lemma which states that "two different polynomials p and q of degree d agree on at most d points". In other words, polynomials that are the same are guaranteed to evaluate to the same value at random points while different polynomials will have drastically different outputs for random queries.

Polynomials are often used in cryptographic commitment schemes which maintain proof integrity and proof secrecy (binding and hiding) [18]. Such commitment schemes, like homomorphic commitment schemes [16], provide a mechanism to evaluate a polynomial at different points without revealing its coefficient terms. In practice, these are implemented as arithmetic circuits [27] with multiplication and addition gates representing the evaluation of a polynomial. A collection of these arithmetic circuits with satisfiable variables is known as a constraint system. The current standard are Rank-1 Constraint System (R1CS) instances [6]. An R1CS instance is satisfiable if there exists some private variables or values (witness w) such that $Aw + Bw = Cw$. These matrices A, B, and C represent the program and its circuit.

These tools have been combined to create one of the current state-of-the-art solutions for verifiable computation: SNARKs [29]. As defined earlier, SNARKs are succinct non-interactive arguments of knowledge where every prover producing a proof must be correlated to a known and extractable witness w. More concretely, a SNARK can be described by three algorithms (G,P,V). G is a generator that outputs a public reference string and a verification state. The honest prover P uses a valid witness w to generate a proof π which can be verified by V. Some SNARKs can even be zero-knowledge indicating that the provided proof π does not leak any additional information about the claim. SNARKs are not only succinct but are also complete and sound.

However, we run into a problem when applying SNARKs to an IVC setting. Particularly we can examine the case where a verifier should be able to verify incremental steps of a proof at different states. The trivial approach is to construct SNARKs for local updates so that at step i we only need to show that (1) the output of step $i-1$ is correct and (2) the program is correctly applied to the output

of step $i - 1$. This approach however is infeasible [19] due to its large proof size. While different SNARK-IVC approaches have been constantly proposed, the required trusted set-up and consistent SNARK creation remain extremely inefficient. In addition, while Incrementally Verifiable PCPs [28] pose another solution to IVC, they also still rely on traditional verifiable computation methods such as SNARKs [24]. Thus, solutions to the IVC problem without using SNARKs are essential in increasing efficiency and improving the practicality of these systems. This literary background is important to understand the benefits of the utilized IVC construction and the importance of immediate practical implementations of this IVC construction such as the one proposed in this paper.

2 METHODOLOGY

2.1 Contributions

To solve these problems with IVC, we specifically propose utilizing Nova: Recursive Zero-Knowledge Arguments from Folding Schemes [19]. Nova is a protocol that utilizes folding schemes instead of SNARKs to accomplish IVC. In doing so, Nova allows for a constant-sized verifier circuit, an efficient prover, and a deferred verification proof.

The innovation of Nova lies in the ability to use folding schemes. These primitives work by combining or "folding" two NP (nondeterministic polynomial time) instances of a problem into a single NP instance. This single folded instance holds the property that it is only satisfiable if the original NP instances are also satisfiable. These folded instances are also dynamic and can be combined with new instances to create new verification proofs without starting from scratch. As a result, Nova folding schemes can efficiently verify proof states within an IVC context.

However, to be able to achieve the feat of using Nova for mathematical proof verification, mathematical proofs must be linear and contain homogeneous (identical) proof steps to be compatible for Nova folding. Currently, there are very few working end-to-end implementations of Nova and no methods support the ability to verify non-trivial proofs. The lack of literature and implementations in this topic illustrate the necessity of the novelty of this work for the real-world usability of such IVC-proof systems.

At a high level, Lova introduces a novel method to accomplish this verification of non-linear and non-homogeneous mathematical proofs utilizing the following pipeline. The mathematical proofs are first formatted and sliced into independent sections: each with the statements, logic rules, and previous lines that were utilized. Next, the mathematical proof slices are converted into a system of linear constraints based on a proof circuit. Utilizing a novel multiplexing circuit, each instance calculates the necessary sums for all logic rules before conducting the necessary checks for the current logic step as indicated by a private input signal. Lastly, each of these linear instances is folded utilizing Nova and converted into a "recursive SNARK". To provide further proof compression, this SNARK is then utilized to form another compressed SNARK using the Spartan SNARK proof system [26]. A diagram of this workflow is found in Figure 1. Formal analysis of the correctness of this framework requires (i) analysis of proof verification logic (Appendix B) (ii) analysis of Nova and Spartan proof systems [19]. The full code implementation can be found as follows [2].

To summarize, Lova provides an end-to-end efficient IVC framework as a solution to mathematical proof verification (see Appendix A for Lova's applications). Particularly, Lova expands upon the existing Nova proof system to allow for the efficient verification of mathematical proofs that are non-linear and heterogeneous (non-identical proof steps). More formally, the novelty of the Lova framework can be described as follows:

- A mechanism to splice and encode mathematical proofs into independent, verifiable, and Nova-friendly proof sequences. These proof sequence are self-contained with the necessary public and private inputs.
- An method to create conditional code segments without breaking the linearity of R1CS instances.
- A system of algorithms to verify a variety of linear and non-linear mathematical logic rules.
- A novel multiplexing circuit design which even allows for non-homogeneous (non-identical) proof steps to be efficiently verified together in a single succinct Nova proof.
- The incorporation of Nova recursive SNARKs with existing SNARK technologies resulting in an end-to-end proof generation pipeline.

The attached Lova framework implementation addresses the current lack of IVC implementations utilizing Nova SNARKs and provides solutions for Nova's theoretical shortcomings. Thus, the Lova framework can be used as a stepping stone to understand how to adapt Nova-based IVC techniques for universal succinct proof verification [3].

2.2 Mathematical Proofs

Mathematical proofs are defined as deductive arguments that argue the correctness of a statement via certain assumptions and mathematical/logical theorems. In particular, logical proofs or formal proofs are mathematical proofs that strictly rely on more rigorous predicate logic and logical axioms. Thus, all mathematical proofs point towards more formal logical proofs that can be computationally verified [17].

To generate these mathematical proofs or more precisely, logic proofs, we need a few different logical operators as well as the Rules of Inference (ROI). Specifically, we rely on the logical operators $\wedge$ (and), $\vee$ (or), $\rightarrow$ (if then), and $\neg$ (not). A line in the proof can thus be formatted as "*statement (logic, [reasoning_line1, reasoning_line2])*". The format of the rules of inference that were utilized is defined in Table 1.

Each proof is first parsed to check that the different logic rules follow the syntax of the definitions in Table 1. The resulting proof is then sliced into different independent sections. Each section is self-contained with the statements to prove, the predicate logic utilized, and a copy of the lines referenced for the logic reasoning. To ensure proof serialization, each section can easily check that only previous lines in the proof were referenced and no lines after the current line were used as justification. Finally, the modified proof slices are encoded numerically utilizing simple numeric substitution.

[2] https://github.com/noelkelias/lova
[3] see Appendix A for Lova's applications

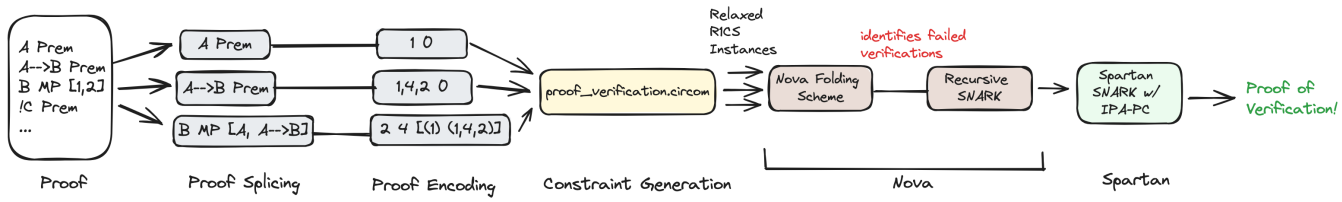

Figure 1: Lova's Pipeline for Mathematical Proof Verification

Table 1: Rules of Inference Definitions

Name	Rule of Inference
Modus Ponens	$\dfrac{p \quad p \to q}{q}$
Modus Tollens	$\dfrac{\neg q \quad p \to q}{\neg p}$
Hypothetical Syllogism	$\dfrac{p \to q \quad q \to r}{p \to r}$
Disjunctive Syllogism	$\dfrac{p \vee q \quad \neg p}{q}$
Addition	$\dfrac{p}{p \vee q}$
Simplification	$\dfrac{p \wedge q}{p}$
Conjunction	$\dfrac{p \quad q}{p \wedge q}$
Resolution	$\dfrac{p \vee q \quad \neg p \vee r}{q \vee r}$

2.3 Linear Verification

As mentioned earlier, one of the key drawbacks of most IVC proof systems including Nova is that proof steps must be validated linearly. To get around this requirement and validate the independent proof slices produced by the last step, a system of linear equations was developed to validate each possible logic step.

As shown by Table 1, there are three possible lines for any proof step: two lines of reasoning and a final statement. We can divide these lines into three groups of sums: the reasoning sum, the statement sum, and the entire proof statement sum. Simply by using certain values of the inputted proof slice to compute these sums, we can ensure through a series of linear equations that the inputted proof slice is bound to the required syntax of the logic step. An example can be demonstrated for the Modus Ponens rule as shown by Equation 4 [4].

[4]Formal analysis of algorithm in Appendix B

$$
\begin{aligned}
&line1 = [a, b, c] \\
&line2 = [d, e, f] \\
&line3 = [g, h, i] \\
&modusPonens_statement_sum = f \\
&modusPonens_reason_sum = a + a + arrow_val + g \\
&modusPonens_proof_sum = 2 \cdot d + 2 \cdot f + arrow_val
\end{aligned}
\tag{4}
$$

By computing the sums of the statement, reasoning, and entire proof sum in a non-traditional method, we can verify that this inputted proof slice contains the correct values. Particularly, in the case of Modus Ponens, we can check that the reasoning sum of all values in lines 1 and 2 is the same as 2a+arrow_val+g. This inadvertently ensures that the necessary values of the reasoning lines are empty and the proof splice follows the syntax where no other values except for the presumed p and q values are present in the reasoning lines. Similarly, the Modus Ponens statement sum checks if the presumed q value of the statement is in both line 2 of the reasoning and the statement line (line 3). Lastly, the proof sum does one final check to ensure that the same p and q values occur twice throughout the entire proof splice. Thus, using this linear system, we can validate the syntax and format of any inputted proof step that is supposed to show Modus Ponens. An example formal analysis of this algorithm can be seen in Appendix B. Similar linear equations to check the other logic steps found in Table 1 are found in the code implementation.

2.4 Circuit Generation

Zero-knowledge proof systems like SNARKs rely on arithmetic circuits or R1CS constraint systems to represent statements and their corresponding witnesses. To satisfy this requirement, a circuit was designed utilizing the Circom domain-specific language [4]. Circom's parameterizable circuits, or templates, allow for the valid assignment and computation of all wires within a circuit.

The circuit was also bounded by the following experimental constraints. First, no quadratic expressions were allowed to be part of the circuit as these Circom circuits needed to be able to be converted to R1CS instances. Second, all variables and signals within the circuit needed to be known at compile-time to provide an R1CS instance. Most importantly, the circuit itself needed to have a homogeneous (each iteration is identical) architecture with no conditional calculations for the Nova folding scheme to work. This was essential to be able to verify dissimilar steps of proofs and combine them into a single Nova folded instance.

Thus, circuits where the proof validation computation drastically changed based on the inputted predicate logic could not be utilized. To solve this requirement for homogeneity, a novel multiplexed circuit design was developed to replace any conditional logic within each circuit. At a high level, the validation of all possible rules of inference was calculated on each iteration of the circuit and a signal was utilized to decide which outputs to compare for the final public output. An outline of this circuit is shown in Algorithm 1.

The encoded proof slices served as private inputs to the Circom circuit. The circuit in turn calculated check-sums on the predicate logic and the lines used for proof reasoning that were both passed into the circuit as private inputs. These check-sums were systems of linear equations that were calculated on the encoded statement and lines of reasoning to be compared to the expected sums as shown in the previous section. Each check-sum was based on the logic signal that was inputted and makes sure that the inputted statement and reasoning signal follow Table 1. As a result, each iteration of the circuit checked whether or not the proof slice was valid and outputted the validated proof sum as a public output/witness.

Algorithm 1 Proof Verification Circuit Pseudocode

Input: statement, reasoning, logic

var **proof_sums** = []
proof_sums[0] ← Premise(statement,reasoning)
proof_sums[1] ← ModusPonens(statement,reasoning)
...
proof_sums[8] ← Resolution(statement,reasoning)

var **statement_sum** = sum of statement values
var **reason_sum** = sum of reasoning values
var **proof_sum** = sum of entire proof slice
assert(**statement_sum** == proof_sums([logic][0])
assert(**reasoning_sum** == proof_sums([logic][1])
assert(**proof_sum** == proof_sums([logic][2])

Output: proof_sum

The resulting multiplexed Circom circuit was then compiled into an R1CS instance with 10 template instances, 1 public input, 1 public output, 10 private inputs, 3 wires, 121 labels, and close to 10000 constraints per step. Afterward, utilizing the Nova-Scotia middleware [9], these R1CS instances are converted directly to a Nova-compatible homogeneous relaxed R1CS instance for folding.

2.5 Nova

As mentioned, we often represent a set of correct execution code in the form of R1CS constraints. However, these instances are not trivially compatible with folding schemes as they are less convenient for linear combinations. This leads to potential problems for the Nova Scheme which has to verify the correctness of each previous state (proof π_{N-1}) utilizing R1CS instances [20].

To solve this problem, Nova introduces the concept of NP-complete relaxed R1CS instances with a scalar term u and a slack or error vector E as shown below:

$$Az \times Bz = uCz + E \tag{5}$$

As mentioned, each relaxed R1CS instance can be described by its public and private vector values: $z_i = (w_i, x_i)$. Using folding schemes, Nova combines these equations into a new representation z=(w,x) which implies that each original $z_i = (w_i, x_i)$ holds with the system of equations given by the matrices of A,B, and C. More formally, this is done by having the verifier select some random value r so that z can be written as a linear combination of z_1 and z_2, particularly z=$z_1 + rz_2$. In doing so, the error vector E and scalar u of the relaxed R1Cs instance encompasses the extra terms generated by such a linear combination to result in the following equation:

$$E = E_1 + r(Az_1 \times Bz_2 + Az_2 \times Bz_1 - u_1Cz_2 - u_zCz_1) + r^2E_2 \tag{6}$$

Now these values of E and u are also added to the representation of the folded relaxed R1CS instance. Particularly, to hide their values we represent them in commitment schemes like Pedersen commitments which are homomorphic and allow for computations. Thus, an instance of a committed relaxed R1CS can be described by the tuple (x, $Commit(w, r_w)$, $Commit(E, r_E)$, u) and is satisfied by the witness tuple (E, r_E, w, r_w). Note that we can shorten $Commit(x)$ as $com(x)$.

Now suppose the verifier and prover have access to multiple instances $(x_1, com(w_1), com(E_1), u_1)$ and $(x_2, com(w_2), com(E_2), u_2)$ and the prover needs to show knowledge of the witnesses $(E_1, r_{E_1}, w_1, r_{w_1})$ and $(E_1, r_{E_1}, w_1, r_{w_1})$. The Nova protocol occurs as follows:

- The prover computes $T = Az_1 \times Bz_2 + Az_2 \times Bz_1 - u_1Cz_2 - u_2Cz_1$ and sends a commitment $com(T)$. The verifier responds with a random challenge r.
- The prover and verifier create the folded instance: $com(E) = com(E_1) + r^2com(E_2) + rcom(T)$ with the associated public and private variables $(com(E), u, com(w), x)$ defined as linear combinations of both instances with the value r.
  - $com(E) = com(E_1) + r^2com(E_2) + rcom(T)$
  - $u = u_1 + ru_2$
  - $com(w) = com(w_1) + rcom(w_2)$
  - $x = x_1 + rx_2$
- The prover also updates the corresponding witnesses (E, r_E, w, r_w) in a similar fashion utilizing r.
  - $E = E_1 + rT + r^2E_2$
  - $r_E = r_{E1} + rr_T + r^2r_{E2}$
  - $w = w_1 + rw_2$
  - $r_w = r_{w1} + rr_{w2}$

With this protocol, the prover is able to update the witness parameters after each folding step - IVC. Then, utilizing a SNARK the prover can easily demonstrate knowledge of the secret witness (E, r_E, w, r_w) to the verifier who has the folded instance of the last step of the IVC. Particularly, we can use a Polynomial interactive oracle proofs[8] where we can convert the matrices A,B, C as well as the vectors E,w,z,y=(x,u) as multilinear extensions (ML). We simply compute the function F(t):

$$\begin{aligned} F(t) = \sum_y A_{ML}(t,y)z_{ML}(y)) \times \sum_y V_{ML}(t,y)z_{ML}(y)) \\ -u\sum_y C_{ML}(t,y)z_{ML}(y) + E_{ML}(t)) \end{aligned} \tag{7}$$

We can check if the function holds by determining if $\sum_x g(\tau, x)F(x) = 0$ for a random value of τ. This can be checked by applying the sum-check protocol [23] to the polynomial defined by $p(t) = g(r, t)F(t)$ where g(x,y)=1 if x=y and 0 otherwise. Thus, using Nova we can succinctly and efficiently verify proofs in an IVC format by utilizing folding schemes on relaxed R1CS instances and SNARKing the final state to demonstrate knowledge of the corresponding folded witnesses. With this Nova protocol, independent sections of the proof can each be verified in a distributed setting and later combined to form a new folded Nova instance which can be proved. This promotes proof dynamicity, allowing existing proof instances to be verified independently and combined efficiently without starting from scratch.

To implement this protocol, we specifically relied on the Rust Recursive SNARK implementation on relaxed R1CS instances as implemented by Microsoft in the original Nova paper [19].

2.6 Spartan

Most zero-knowledge SNARK protocols require a trusted setup which creates some"toxic waste" that can be utilized by a malicious verifier to break the soundness requirement of a SNARK and generate valid fake proof. The Spartan-proof system is one solution to this problem by that avoids the trusted setup phase. At a high level, Spartan [26] works by first converting the statement to an arithmetic circuit and then a low-degree polynomial. The polynomial is then formatted so that the sum of the polynomial over the boolean hypercube is zero. Lastly, the polynomial is submitted for the sum-check protocol.

More precisely, after the arithmetic circuit is converted into a low-degree polynomial via arithmetization [25], each function in the polynomial is replaced with its multi-linear extension (MLE) as shown in Equation 8 where Z is a function and $\tilde{Z(t)}$ is its multi-linear extension:

$$\tilde{Z(t)}(x_1, ..., x_n) = \sum_{e \in \{0,1\}^n} Z(e) \cdot \prod_{i=1}^{n} (x_i + e_i + (1 - x_i) \cdot (1 - e_i)) \quad (8)$$

Then, the resulting MLE polynomial is formatted to be suitable for the sum-check protocol [23] where polynomial commitments are used to verify the witness. This is done to make sure that nonzero terms do not cancel out when running the sum-check protocol of resulting polynomial $\tilde{Z(t)}$. Particularly, we use a modified polynomial Q(x) utilizing $\tilde{Z(t)}$ by multiplying the sum of $\tilde{Z(t)}$ by different powers of x, Q: $Q(x) = \sum \tilde{Z(t)} \cdot x^y$. Now, Q(x) can be directly fed into the sum-check protocol to prove and verify the statement much more efficiently and without a trusted setup.

For this reason, the recursive SNARK of the Nova folded instance is once again inputted into the Spartan proof system for an even smaller and more efficient proof. Specifically, we utilized the Spartan proof system with inner product arguments (IPA) and polynomial commitments (PC) as implemented in Rust by Microsoft in the original Spartan paper [26]. Another reason for this inclusion was to demonstrate the Lova's cross-compatibility between the resulting Nova proofs and existing SNARK implementations like Spartan.

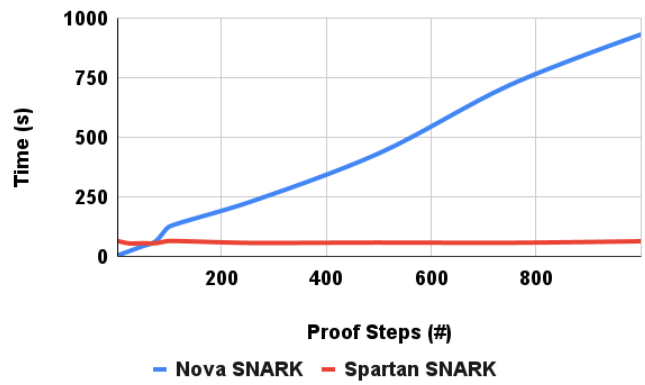

Figure 2: Lova's Performance in SNARK Generation

3 RESULTS

The resulting Lova proof verification pipeline successfully spliced the proof, encoded the proof slices, generated relaxed R1CS constraints, folded these instances into a single verifiable instance with Nova, and finally compressed all this computation into a succinct Spartan proof. A few different experiments were conducted to further understand the capabilities of Lova's proof verification pipeline. Particularly, these experiments were conducted to get a better understanding of the Lova framework's capabilities with varying proof sizes and different proof parameters. These benchmarks can be compared to other external approaches for mathematical proof verification that may or may not be included in this paper. The datasets used for these benchmarks are included in the implementation [5].

The first experiment experimented on Lova's performance with different proof sizes. Particularly, the proof generation and verification time for the recursive Nova SNARK and the compressed Spartan SNARK were measured and analyzed for efficiency and performance in regards to the proposed proof verification Circom circuit. Proofs ranged from having just a few steps to proofs with 1000+ steps. The results are the averages of 20 trials that were conducted with the exact same parameters for each different input proof. All computations were run on a 2.3 GHz Dual-Core Intel Core i5 processor without a GPU. The results of this experiment is shown in Figures 2 and 3.

As demonstrated by the data, the resulting recursive Nova SNARK grows linearly by the number of proof steps in the mathematical proof. On the other hand, the Spartan proof system with IPA-PC maintains a constant proof generation time even with a varying number of proof steps. This makes sense as the Recursive Nova SNARK includes the step of folding more relaxed R1CS instances together as the number of proof steps increases. As a result, the generation time for the Nova Recursive SNARK only grows linearly or in $O(n)$ time where n is the number of proof steps! However, the compressed Spartan SNARK works on a constant size input which is simply the output of the Nova recursive SNARK whose output remains constant with variable proof steps. As such, the

[5] https://github.com/noelkelias/lova/tree/main/misc/proof_tests

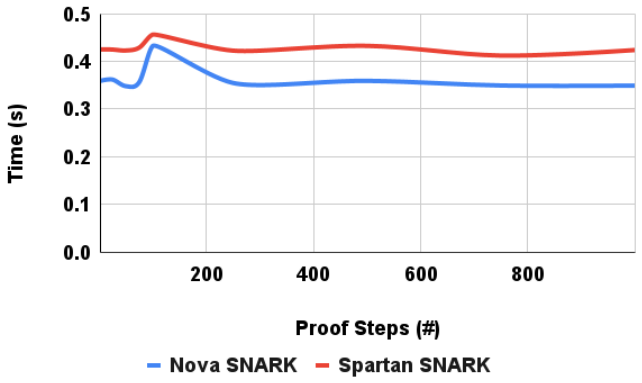

Figure 3: Lova's Performance in SNARK Verification

generation time for the Spartan SNARK remains $O(1)$ or constant. In addition, Figure 3 demonstrates that the proposed Lova pipeline gets the added benefit of efficient and short verification times no matter the size of the inputted mathematical proof. Thus, the proposed verification circuit did not differ at all from Nova's expected performance.

Another experiment that was conducted included testing Lova's performance with the implemented Circom circuit on different cycles of elliptic curves. Within the Nova implementation, two curves are utilized for folding. One curve is utilized for the inputted private and public points of the relaxed R1CS instance while the other curve is used for Nova's secondary circuit. As such this experiment analyzes the performance of utilizing different combinations of curves for both Nova inputs and the Nova secondary circuit. Specifically, the pasta (Pallas/Vasta) curves and the bn256/grumpkin curves as defined by the arkworks [1] and implemented by the original Nova paper [19] were tested. Each curve was tested with a 50-step proof and then compared the proof generation and verification time for the recursive Nova SNARK and the compressed Spartan SNARK.

Table 2 illustrates that the overall best combination of curves was the Pasta curve where the Nova inputs are assigned as points on the Pallas curve and Nova's secondary circuit runs on the Vesta curve. This resulted in the shortest amount of time needed for Nova folding for the recursive SNARK creation and contained the shortest verification times for both the recursive and compressed SNARKs. In fact, the Pasta curves are much more efficient than the bn256_grumpkin curves taking close to 1/4 less time to fold instances together for Nova and generate public parameters for the recursive SNARK. This suggests that for the proof verification circuit, the Pasta curves contain the best field for the corresponding low-value integer arithmetic of the implemented mathematical proofs verification Circom circuit.

4 CONCLUSION

Utilizing incrementally verifiable computation we were able to develop Lova, a novel framework to efficiently verify mathematical/-logic proofs. Lova addresses the short comings of current state-of-the-art IVC methods by providing a general approach and concrete implementation to efficiently verify non-linear and non-homogeneous mathematical proofs. Particularly, Lova introduces a unique proof splicing methodology, a system of linear equations for logic verification, and a novel multiplexing circuit to generate Nova friendly proof instances from incompatible mathematical proofs. Then using the Nova folding scheme, we can combine these resulting relaxed R1CS instances of statements within a proof into a single folded instance. This instance can easily and efficiently be proved utilizing a recursive SNARK and further compressed utilizing the Spartan proof system. As a result, we have shown that we can achieve linear-time proof validation of mathematical proofs and constant time verification of these validation proofs via IVC.

Now not only is Lova efficient and effective, but also allows the prover to enable many more features for such a validation proof. The Nova folding scheme is in fact amendable and dynamic. Relaxed R1CS instances can be calculated in a distributed setting and folded individually. Later, if a proof needs to be appended, the resulting Nova folded state of the existing proof can simply be combined with the new proof steps to create a new valid validation proof. SNARKs can additionally be used as zero-knowledge tools and provide a validation proof of a mathematical proof without revealing any additional information of the input (see Appendix A for expanded applications). Thus, utilizing IVC methods such as Nova folding marks the start of more efficient and robust technologies for mathematical proof validation.

5 ACKNOWLEDGEMENTS

I would like to sincerely thank Dr. Sriram Vishwanath for introducing me to this research problem. I would also like to thank Dr. David Wu for his helpful discussions, specifically for providing helpful pointers in the domain of IVC.

REFERENCES

[1] arkworks. 2023. curves. https://github.com/arkworks-rs/curves
[2] Jon Barwise. 1977. An introduction to first-order logic. In *Studies in Logic and the Foundations of Mathematics*. Vol. 90. Elsevier, 5–46.
[3] Mihir Bellare, Shafi Goldwasser, Carsten Lund, and Alexander Russell. 1993. Efficient probabilistically checkable proofs and applications to approximations. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. 294–304.
[4] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. 2022. Circom: A Circuit Description Language for Building Zero-knowledge Applications. *IEEE Transactions on Dependable and Secure Computing* (2022).
[5] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*. Springer, 90–108.
[6] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P Ward. 2019. Aurora: Transparent succinct arguments for R1CS. In *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part I 38*. Springer, 103–128.
[7] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. 2016. Interactive oracle proofs. In *Theory of Cryptography: 14th International Conference, TCC 2016-B, Beijing, China, October 31-November 3, 2016, Proceedings, Part II 14*. Springer, 31–60.
[8] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. 2016. Interactive Oracle Proofs. Cryptology ePrint Archive, Paper 2016/116. https://eprint.iacr.org/2016/116 https://eprint.iacr.org/2016/116.
[9] Nalin Bhardwaj. 2023. Nova Scotia. https://github.com/nalinbhardwaj/Nova-Scotia
[10] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2013. Recursive composition and bootstrapping for SNARKs and proof-carrying data. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*.

Curve	Nova SNARK Generation	Spartan SNARK Generation	Nova SNARK Verification	Spartan SNARK Verification
Pasta: Pallas	**45.70s**	**65.06s**	**0.35s**	**0.43s**
Pasta: Vesta	51.68s	65.80s	0.419s	0.43s
bn256_grumpkin: bn256s	208.71s	127.81s	5.71s	5.28s
bn256_grumpkin: grumpkin	219.21s	113.38s	5.34s	4.06s

Table 2: Different Curve Combinations

111–120.

[11] Robert S Boyer and Yuan Yu. 1996. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM (JACM)* 43, 1 (1996), 166–192.

[12] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. 2008. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 7 (2008), 1165–1178. https://doi.org/10.1109/TCAD.2008.923410

[13] Kai Fan, Qiang Pan, Kuan Zhang, Yuhan Bai, Shili Sun, Hui Li, and Yintang Yang. 2020. A secure and verifiable data sharing scheme based on blockchain in vehicular social networks. *IEEE Transactions on Vehicular Technology* 69, 6 (2020), 5826–5835.

[14] Uriel Fiege, Amos Fiat, and Adi Shamir. 1987. Zero knowledge proofs of identity. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 210–217.

[15] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. 2015. Delegating computation: interactive proofs for muggles. *Journal of the ACM (JACM)* 62, 4 (2015), 1–64.

[16] Jens Groth. 2011. Efficient zero-knowledge arguments from two-tiered homomorphic commitments. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 431–448.

[17] Michael Hutchings. 2007. Introduction to mathematical arguments. *Online. Tersedia* (2007).

[18] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. 2010. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16*. Springer, 177–194.

[19] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. 2022. Nova: Recursive zero-knowledge arguments from folding schemes. In *Annual International Cryptology Conference*. Springer, 359–388.

[20] LambdaClass. 2023. Incrementally verifiable computation: Nova. https://blog.lambdaclass.com/incrementally-verifiable-computation-nova/

[21] Seunghwa Lee, Hankyung Ko, Jihye Kim, and Hyunok Oh. 2024. vcnn: Verifiable convolutional neural network based on zk-snarks. *IEEE Transactions on Dependable and Secure Computing* (2024).

[22] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. 1992. Algebraic methods for interactive proof systems. *Journal of the ACM (JACM)* 39, 4 (1992), 859–868.

[23] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. 1992. Algebraic Methods for Interactive Proof Systems. *J. ACM* 39, 4 (oct 1992), 859–868. https://doi.org/10.1145/146585.146605

[24] Moni Naor, Omer Paneth, and Guy N. Rothblum. 2019. Incrementally Verifiable Computation via Incremental PCPs. Cryptology ePrint Archive, Paper 2019/1407. https://eprint.iacr.org/2019/1407 https://eprint.iacr.org/2019/1407.

[25] Anca Nitulescu. 2020. zk-SNARKs: a gentle introduction.

[26] Srinath Setty. 2019. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. Cryptology ePrint Archive, Paper 2019/550. https://eprint.iacr.org/2019/550 https://eprint.iacr.org/2019/550.

[27] Amir Shpilka, Amir Yehudayoff, et al. 2010. Arithmetic circuits: A survey of recent results and open questions. *Foundations and Trends® in Theoretical Computer Science* 5, 3–4 (2010), 207–388.

[28] Paul Valiant. 2008. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Theory of Cryptography: Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008. Proceedings 5*. Springer, 1–18.

[29] Jim Woodcock, Mikkel Schimdt Andersen, Diego F. Aranha, Stefan Hallerstede, Simon Thrane Hansen, Nikolaj Kuhne Jakobsen, Tomas Kulik, Peter Gorm Larsen, Hugo Daniel Macedo, Carlos Ignacio Isasa Martin, and Victor Alexander Mtsimbe Norrild. 2023. State of the Art Report: Verified Computation. arXiv:2308.15191 [cs.CR]

6 APPENDIX

6.1 A - Lova Applications

There are many immediate applications that the Lova framework can be utilized for. The experiments and datasets described in this paper are just a sample of one possible configuration of the Lova framework. Developers and researchers can generalize this proof-of-concept implementation to validate proofs with rules and logic of their choice.

A simple and immediate use case of Lova could be to validate the correctness of basic mathematical proofs for mathematical theorems. These proofs could be first-order-logic proofs that a student or program has computed for an exam or an assignment. Normally, each step of the proof is different and thus cannot be verified using traditional IVC schemes like Nova. However, Lova solves this problem exactly by allowing the efficient/succinct verification of heterogeneous proofs. So, the grader could simply input the student's proof into the Lova framework. Utilizing, the Nova folding scheme, the linear verification algorithms, and multiplexing circuit, Lova can check the inputted proof for syntax and logic correctness. This determines whether or not the input is a valid proof for the assigned question utilizing FOL and the rules of inference. At the end of this run, Lova outputs a succinct SNARK proof that shows how the inputted proof was evaluated for correctness and on which step it failed, if any. Thus, the grading/evaluation of the correctness of a proof is transparent and utilizing the outputted SNARK properties can be easily verified by the student themselves to ensure grading fairness.

Another use case for the Lova framework could be for integrated circuit design and verification [11]. Many companies use proof checking as a method to verify that certain operations (especially floating point units) within their processors work as expected. This is often done by converting the gate logic into FOL proofs. These proofs then undergo formal analysis to determine if the operations correctly returns the output of a sample program. As these formal specifications (proofs) are in FOL, they can be verified using the Lova framework. Just as before, Lova will break the formal specification of the processor's operations into self-contained chunks and utilize the implemented linear verification algorithms to ensure the proof contains valid first-order-logic. Utilizing Nova alongside Lova's multiplexing circuit design (allows for non-homogeneous proofs), each proof step will be folded into a single instance which can be easily checked in one setting and dynamically modified using the power of IVC. Now, adding new changes to such a formal specification will not require validating the entire proof but instead only the newly added steps. As output, the designers will get a succinct SNARK showing that the inputted operation's formal specification

works as expected. This succinct SNARK for each operation can be efficiently verified by any client before purchasing a processor to ensure all operations on the processor work as expected.

A more advanced use case for Lova could be for validating inferencing/training for machine learning models. Often times, we want to verify that a model's inference was computed correctly without any malicious or biased input. To check this, we can examine the outputs of each layer of the model and, based on the architecture, apply the necessary transformations. The resulting output can then be cross-checked with the inference. The Lova framework can be directly applied to this situation. The sequence of transformations or multiplications showing how the model computed its inference is essentially a mathematical proof. These steps can be translated into prepositions using interactive proof assistants like Coq. The result is a first-order logic proof. As a result, these proofs can be fed into the Lova framework to be validated and compressed into a succinct SNARK indicating their correctness. A similar workflow could be utilized to verify that transactions on a Blockchain were conducted correctly (checking hashes computed correctly etc.) or that new blocks contain valid transactions etc.

As part of next steps, developers could build on top of the existing Lova framework and use the multiplexing circuit design alongside the linear logic checking algorithms to check their own logic rules (instead of just first order logic). For example, the developer could define an operation over a group (addition, multiplication, etc) and create logic checking algorithms to verify the validity of that operation on elements in that group. Thus, any proof with logic rules programmed into the Lova framework can be easily and efficiently validated.

6.2 B - Proof of Modus Ponens Verification

We can show that the system demonstrated in 4 correctly validates Modus Ponens. Recall the Modus Ponens rule is as follows:

$$\frac{\begin{array}{c} p \\ p \rightarrow q \end{array}}{q} \tag{9}$$

Notice p is the same preposition in the 1st line and 2nd line of reasoning. In addition, q is the same preposition in the 2nd line of reasoning and the result. We simply need to check that the prepositions in these locations are equal to validate Modus Ponens. Now, we can assume that the input pre-processing step of Lova ensures that the inputted sequence is of the form:

$$\frac{\begin{array}{c} a \\ d \rightarrow f \end{array}}{g} \tag{10}$$

Thus, the input sequence contains (a,d,f,g) which can be assumed to be some valid prepositions. All the other possible elements of the input sequence are confirmed to be initialized to 0 (input pre-processing step). To show that this input sequence follows Modus Ponens logic, we need to check f=g and a=d. Now the first check ensures that the outputted statement (resulting statement) corresponds to the 2nd preposition of the Modus Tollens reasoning. This is given by f. To do this we can simply rely on the 4th line of Equation 4. Now we only need to check that the same preposition (p) appears in both lines of reasoning. We could simply check that a=d, however that would introduce conditional code segments into our circuit and cause it to be non-linear. Instead, we can simply check that the two reasoning lines of the proof sequence compute to a+d+f = 2a+f. Note that the first check ensures that the f value is valid. So as f is already valid and can be subtracted, this equation ensures that the same p value or a=d in this case. This is the same equation as the 5th line in Equation 4. Thus, this algorithm validates whether or not the input sequence follows the Modus Ponens rule.