Nebula: Efficient read-write memory and switchboard circuits for folding schemes

Arasu Arun[†] Srinath Setty[⋆]

[†]New York University [⋆]Microsoft Research

Abstract.

Folding schemes enable prover-efficient incrementally verifiable computation (IVC), where a proof is generated step-by-step, resulting in a space-efficient prover that naturally supports continuations. These attributes make them a promising choice for proving long-running machine executions (popularly, "zkVMs"). A major problem is designing an efficient read-write memory. Another challenge is overheads incurred by unused machine instructions when incrementally proving a program execution step.

Nebula addresses these with new techniques that can paired with modern folding schemes. First, we introduce *commitment-carrying IVC*, where a proof carries an incremental commitment to the prover's non-deterministic advice provided at different steps. Second, we show how this unlocks efficient read-write memory (which implies indexed lookups) with a cost-profile identical to that of non-recursive arguments. Third, we provide a new universal "switchboard" circuit construction that combines circuits of different instructions such that one can "turn off" uninvoked circuit elements and constraints, offering a new way to achieve pay-per-use prover costs.

We implement a prototype of a Nebula-based zkVM for the Ethereum Virtual Machine (EVM). We find that Nebula's techniques qualitatively provide a 30× smaller constraint system to represent the EVM over standard memory-checking techniques, and lead to over 260× faster proof generation for the standard ERC20 token transfer transaction.

1 Introduction

This paper studies the problem of producing succinct arguments of program executions on machines (e.g., RISC-V, EVM). In this setting, a *prover* and *verifier* are both given the specification of a (virtual) machine (e.g., the instruction set architecture and semantics), which can they can preprocess to obtain setup material. After preprocessing, given a program that is designed to run on such a machine (e.g., specified in the assembly language of the machine) along with some inputs, the prover produces a proof to convince the verifier that it correctly executed the specified program on the specified inputs to obtain some claimed outputs. The program can optionally take (possibly secret) non-deterministic advice inputs from the prover. Furthermore, the programs may maintain state during the execution in the form of a read-write memory and make lookup queries to read-only tables. A proof system satisfies "succinctness" if the length of a proof and the time to verify it are at most polylogarithmic in the time to run the program. Optionally, they may be "zero-knowledge" meaning that the proof should not reveal anything about the prover's private witness beyond what is implied by the statement itself.

These protocols, popularly referred to as "zero-knowledge VMs" (zkVMs),[1] have many applications in decentralized systems. For example, in blockchain rollups, an untrusted off-chain entity uses the proof system to produce a proof of correct execution of arbitrary programs on a specified machine (e.g., EVM) and verify the proof on-chain to get the effect of executing the said program on-chain [16,31,43]. Since verifying a proof is significantly cheaper than executing the original program on-chain, this approach provides scalability and lower transaction costs.

Folding schemes and zkVMs

We focus on the setting where the proof generation is required to be *incremental*: the prover produces a proof for each step (or a group of sequential steps) of the program execution independently and then merges that into a single proof. Incrementally verifiable computation (IVC) [41,10] formally captures this requirement. These proof systems have several notable advantages: (1) they natively provide "continuations", where a prover can produce a proof of a long-running computation step-by-step; (2) they consume less space than non-recursive ("monolithic") proof systems. In fact, prover space is a major bottleneck in scaling provers to long-running computations, so IVC is appealing in practice.

Folding schemes [29,25,27,15,20,26] have emerged as a prover-efficient way realize IVC. In IVC, at each step, the prover executes one or more program steps and folds the prior step into a running instance such that the the running instance and the associated witnesses serve as a proof of correct execution of the computation thus far. Folding-based IVC feature small recursion overheads, unlike the naive recursion approach that is used in existing zkVMs. Here, the overhead from recursion is the cost of folding a folding scheme verifier represented as a circuit: in Nova [29], this verifier computes a random linear combination of homomorphic commitments and performs some hashing, costing only 10,000 multiplication gates in R1CS [2]. HyperNova [27] requires slightly higher hashing in the circuit while unlocking high-degree constraint systems such as CCS [39].

Designing a folding-based zkVM poses many challenges. This is because techniques that apply to monolithic SNARKs [42,44,8,9,14] cannot be translated directly to the context of IVC. Below, we discuss two challenges.

Challenge 1: Efficient read-write memory

Program executions on machines such as RISC-V or EVM invoke operations that read values from and write values to memory. Braun et al. [13] describe a solution that employs Merkle trees [32,12] in the programming model of proof systems; this approach is also employed in prior zkVMs [10] and even recent folding-based zkVMs [5]. Unfortunately, handling memory via this method requires representing

1 Most zkVMs only offer succinctness, and not the zero-knowledge property. However, most constructions (including the ones in this work) can be made zero-knowledge without prover overheads.

expensive cryptographic operations as a constraint system. In particular, to perform a read or a write on a memory of size m, one must encode $O(\log m)$ hash operations. Even with SNARK-friendly hash functions (e.g., Poseidon) where a hash invocation costs about $\approx$250 constraints, each read/write memory operation costs at least 5,000 constraints on a modest-sized 1 MB memory.

Spice [35] and Spartan [34] bring offline memory checking [12,18] to proof systems. It is called "offline" because a batch of memory operations are checked at once. This approach provides an asymptotic improvement over Merkle trees: each memory operation only requires $O(1)$ multiset hash computations rather than $O(\log m)$-many cryptographic hashes, with the trade-off being that one needs $O(m)$ memory operations to amortize fixed costs. Lasso [40] and Jolt [8] adapt this approach to provide a low-cost memory in zkVMs. However, this adaptation is limited to non-recursive proof systems, where the prover requires space linear in the number of steps in the machine execution. This space requirement is prohibitive for long running computations. We now elaborate.

A core tool in offline memory checking is a multiset collision-resistant hash function. Spice [35] instantiates the multiset collision-resistant hash function with an elliptic-curve-based construction. This approach is compatible with recursive proof systems including folding-based IVC. However, it requires encoding an elliptic curve addition, a hash-to-curve function, and a cryptographic hash function in a constraint system. This requires about 1,000 constraints per memory operation.[2] Instead, Spartan [34], Lasso [40], and Jolt [8] instantiate the multiset hash function with a public-coin challenge based function. The resulting hashes are often referred to as "fingerprints" and the hashing process is referred to as "fingerprinting". This approach requires only a few field multiplications and a range check per memory operation in the constraint system. As a result, it at least an order of magnitude cheaper in the costs of memory operations as compared to the Spice instantiation. However, this requires public challenges, which in turn requires having the prover commit to all memory operations before sampling challenges. So, this instantiation does not directly lead to a solution to efficient read-write memory in the context of IVC. Furthermore, it is not clear how to ensure that the proofs are incrementally updateable.

A major problem is whether one can design a memory primitive for recursive arguments that is as efficient as the one in non-recursive arguments.

Challenge 2: "Pay-per-use" costs with multiple instructions per step

Since CPUs and VMs execute programs instruction by instruction, the circuit underlying a zkVM capturing a machine execution of n steps is generally constructed by replicating a smaller circuit capturing single step of machine execution. As the sequence of instructions is not known a priori, the latter must be general enough to capture all possible instructions, leading to a circuit size that is at least the sum of sizes of circuits of individual instructions. This in turn makes

2 This measurement is obtained when using the Poseidon hash function and the Elligator-2 hash-to-curve on the BN254 curve [35].

the prover incur enormous per-step proving costs. Many solutions exist in the literature to address this problem [42,44,8], but they focus on space-inefficient, non-recursive arguments.

In the recursive context, this is formalized as "non-uniform" IVC [25]: an IVC proof can be incremented using any one among a fixed set of step functions (this set is a singleton in standard "uniform" IVC) with the prover paying costs proportional to only the executed function ("a la carte" cost profile). SuperNova [25] meets this requirement, but it introduces a trade-off: SuperNova can profitably only execute one machine instruction per recursive step. This is because executing an instruction requires folding a circuit satisfiability instance into a running instance. So, if we execute β instructions per recursive step, it requires β invocations of the folding scheme (each invocation costs $\approx$10,000 gates). This is acceptable only if each instruction of the machine when represented as a circuit requires far more than 10,000 gates. On the other hand, generic IVC schemes with a universal circuit (which implements a multiplexer over all the supported instructions) can execute multiple instructions per step with a *single* invocation of the folding scheme. However, the prover's cost at each step is proportional to the sum of sizes of circuits of all instructions supported by the machine, which does not satisfy the "a la carte" requirement noted earlier. Protostar [15] *implicitly* addresses this problem, but inherently increases the degree of constraints, which in turn increases the prover work.

A key question is the following: can we get the best of both worlds where one can execute any number of instructions per step (like in Nova) while paying only for the instructions that were actually executed (like in SuperNova)?

1.1 Our solution: Nebula

Our central result is Nebula, an IVC scheme where the step function gets access to a global memory. Specifically, IVC proceeds by executing a non-deterministic function F repeatedly feeding the output of F at step i as input to F at step $i+1$. Our extended model allows F to issue read and write operations to a global memory, where if F performs a write to an address a at step i, and later F performs a read at step $j \geq i$ to address a, the read response from the global memory reflects the effects of the write performed at step i. This memory semantics is captured formally by sequential consistency [30].

(1) Commitment-carrying IVC. We first introduce a natural generalization of IVC, which we refer to as *commitment-carrying IVC*. In a nutshell, commitment-carrying IVC produces a proof along with a commitment to non-deterministic advice given by the prover at each step of IVC. We show that existing folding-based IVC schemes (e.g., Nova, HyperNova) generalize easily to provide commitment-carrying IVC. Furthermore, by taking advantage of commitments to witnesses already computed by the prover in existing folding schemes (e.g., Nova, HyperNova), we realize this IVC variant with minor overheads: the folding verifier performs a few extra hashes per step, leading to a negligible increase in the verifier circuit size and the prover costs.

(2) Read-write memory and lookups. Commitment-carrying IVC unlocks a powerful capability in recursive arguments. It enables the prover to cryptographically commit to its non-deterministic advice, generate a random challenge, and then use that challenge to perform randomized checks during IVC steps. We show how to use this capability to instantiate Spice-style memory checking with a public challenge in the context of a recursive argument. By designing an efficient read-write memory primitive for recursive arguments, we also immediately obtain an efficient read-only memory, which directly provides an indexed lookup argument [40]. This achieves the so-called "lookup singularity" in recursive proof systems, a recent advance that represents all instructions of a VM with lookups into a read-only memory [40,8]. For many (bitwise and non-arithmetic) operations, this is more compact than representing them with constraints.

With the public coin hash function, memory checking amounts to computing a grand product operation defined over a set of address-value-timestamp tuples. Computing this grand product inside a circuit (e.g., with R1CS or CCS) means that the intermediate values of the grand product can be arbitrary field elements, which the prover has to commit. We describe an optimization leveraging the hybrid grand product protocol of Setty and Lee [37] (which is often referred to as the "the Quarks method"). In a nutshell, this optimization offloads multiplications from within a circuit to an auxiliary protocol, and the circuit only needs to encode the verifier of the auxiliary protocol as a circuit. This optimization ensures that the prover only commits to "small" field elements. By "small" field elements, we mean that the committed values are from the set $\{0, \ldots, m-1\}$, where $\log m << \log p$ and the finite field used is $\{0, \ldots, p-1\}$. Concretely, $\log m = 8$ when the memory is byte-addressable or 32 when it is word-addressable. Lasso [40] and Jolt [8] achieve a similar property in the context of non-recursive arguments.

(3) Folding IVC proofs to retain incremental proofs. Since Nebula leverages a public-coin hash function, the challenge must be generated after the prover commits to its non-deterministic witnesses. We observe that in the context of folding-based IVC, an IVC proof of a machine execution consists of a set of instance-witness pairs for some relation (e.g., committed relaxed R1CS in the case of Nova and linearized CCS in the case of HyperNova). Furthermore, we can instantiate an auxiliary IVC scheme whose work is to fold instances with an IVC proof of a machine execution, after performing some compatibility checks (e.g., the starting memory contents of the second IVC proof of machine execution must match the ending memory contents of the first IVC proof of machine execution). This allows us to get the best of both worlds: leverage randomized checks for efficient read-write memory while retaining the incrementality features of IVC.

(4) Switchboard construction. We describe a new approach to realize non-uniform IVC. In a nutshell, we combine a set of circuits (e.g., circuits in R1CS form representing different instructions of a VM) into a new "switchboard" R1CS circuit that can behave as any one of the original circuits. We name it a switchboard circuit because our transformation employs "switch" variables that the prover uses to "power down" unused circuits not relevant to the current step.

For witness variables associated with the unused circuits, the prover can safely set the variables to 0 and this still satisfies all constraints. In folding schemes that only commit to the witness (e.g., HyperNova [27]), this already ensures that the prover's commitment cost for a VM step depends only the circuit size associated with instruction that is executed. This is because committing to 0s is free in MSM-based commitment schemes.

We also show that even folding schemes that commit to other data can benefit from our switchboard circuits (e.g., Nova [29] commits to a cross-term in addition to witness). In particular, Nova's cross-term entries are not guaranteed to contain zeros for entries associated with circuits corresponding to instructions that were not executed. We address this by modifying Nova's procedure to compute and commit to the cross-term.

1.2 Recent related work

Mangrove [33] describes an approach to prove Plonkish [22] constraint systems with IVC based on folding schemes such as Nova [29] and Protostar [15]. As part of this, it proves Plonkish's copy constraints, which checks if value assigned to a particular variable in the constraint system equals the value assigned to another designated variable. To prove copy constraints, Mangrove provides an incremental implementation of the permutation argument. This primitive can be viewed as achieving an efficient read-only memory primitive in IVC. However, Mangrove's proofs are not incrementally updateable, so it is not an IVC scheme. In contrast, Nebula provides a read-write memory with sequential consistency semantics within IVC. Furthermore, we provide a clean abstraction in the form of commitment-carrying IVC that unlocks other randomized checks in IVC steps beyond offline memory checking. Indeed, Mangrove can be built on top of Nebula to prove Plonkish in a scalable manner. This option unlocks a new efficiency result: Nebula shows how to avoid commiting to arbitrary field elements as part of memory checking, which benefits Mangrove if Plonkish's witness elements are "small". Currently, Mangrove computes grand products for permutation checks, which makes the prover commit to arbitrary field elements.

Very recently, Eagen et al. [21] define a relaxation of IVC where the step function gets access to a global memory. Their interface is that of a write-once, read-many memory, which is less general than the read-write memory with sequential consistency supported by Nebula. Furthermore, their relaxed version of IVC only leads to a SNARK rather than an IVC scheme as achieved by Nebula. As a result, Nebula's memory contents can persist forever allowing one to perform operations on the global memory for any number of IVC steps—without the size of the proof growing with the number of steps.

Bunz and Chen [17] propose a way to provide a read-write memory primitive in IVC. However, there are several differences between their proposal and Nebula. Their approach requires static bounds on the size of the memory, whereas Nebula's memory is elastic. Furthermore, the verifier circuit must encode $(c+1) \cdot \log T$ scalar multiplications, where T is the size of memory, and c is an arbitrary constant such that $|T|^{1/c}$ is minimal. Even for tiny memories, this requires hundreds of scalar multiplications (so 100,000 constraints in the verifier circuit). Whereas Nebula's verifier circuit only performs a handful of scalar multiplications. Additionally, their verifier circuit is of high degree (at least 4, and up to 7 in their most efficient constructions), while Nebula's is just degree 2. Thus Nebula can efficiently work with R1CS. Beyond these, their construction requires multiple sub-protocols: permutation argument, memory-update protocol, logupGKR, bivariate sum-check, etc. making the overall construction far more complex than ours.

In a companion work, NeutronNova [28] provides a new folding scheme for the zero-check relation, which improves upon prior folding schemes including HyperNova [27], Protostar [15], and Protogalaxy [20]. They show simple, constant-round reductions from complex relations including CCS [39], grand products [37], and indexed lookups [40]. NeutronNova aims to provide a new foundation for building folding-based zkVMs. Nebula's read-write memory primitive can be rephrased in their framework to benefit from their improved folding scheme. Additionally, this rephrasing would provide an efficient read-write memory primitive in NeutronNova, which is necessary to build a NeutronNova-based zkVM.

2 Preliminaries

We use λ to denote the security parameter and $\mathbb{F}$ to denote a finite field (e.g., the prime field $\mathbb{F}_p$ for a large prime p). We use $\mathsf{negl}(\lambda)$ to denote a negligible function in λ. We write $\Pr[X] \approx \epsilon$ to mean that $|\Pr[X] - \epsilon| = \mathsf{negl}(\lambda)$. Throughout the paper, the depicted asymptotics depend on λ, but we elide this for brevity. We write PPT to refer to probabilistic polynomial time algorithms. For relations $\mathcal{R}_1$ and $\mathcal{R}_2$ we let $\mathcal{R}_1 \times \mathcal{R}_2$ denote a new relation such that $((u_1, u_2), (w_1, w_2)) \in \mathcal{R}_1 \times \mathcal{R}_2$ if and only if $(u_1, w_1) \in \mathcal{R}_1$ and $(u_2, w_2) \in \mathcal{R}_2$. We write $\mathbb{F}^d[X_1, \ldots, X_n]$ to denote multivariate polynomials over field $\mathbb{F}$ in the variables $X_1, \ldots, X_n$ with degree bound d for each variable. We omit the superscript if there is no bound.

We defer formal definitions of arguments of knowledge (§A.2), and commitment schemes (§A.1) to Appendix A.

2.1 Committed relations

Instead of directly working with R1CS or CCS (which generalizes R1CS, Plonkish, and AIR), Nova and HyperNova work with a variant type of relation where a commitment to the witness is additionally presented in the instance. They generically refer to them as *committed relations.*

Definition 1 (Committed relation). *Consider a relation $\mathcal{R}$ over structure, instance, witness tuples where witnesses are in some space W. Consider a commitment scheme* $\mathsf{com} = (\mathit{Gen}, \mathsf{Commit})$ *over message space W. We define the corresponding committed relation over public parameter, structure, instance, witness tuples characterized by* com *as follows.*

$$\mathcal{R}(\mathsf{com}) = \left\{ (\mathsf{pp}_{\mathsf{com}}, \mathsf{s}, (C, u), (w, r)) \,\middle|\, \begin{array}{l} (\mathsf{s}, u, w) \in \mathcal{R}, \\ C = \mathsf{Commit}(\mathsf{pp}_{\mathsf{com}}, w, r) \end{array} \right\}$$

We say relation $\mathcal{R}$ is the underlying relation for committed relation $\mathcal{R}(\mathsf{com})$.

2.2 IVC and Folding Schemes

In incrementally verifiable computation (IVC) [41], for some polynomial-time function F, the prover takes as input a statement (i, z_0, z) and a proof Π_i proving that $z_i \leftarrow F^i(z_0)$ and then increments it to produce a new statement $(i+1, z_0, z_{i+1}$ and a proof Π_{i+1} proving that $z_{i+1} \leftarrow F^{i+1}(z_0)$. Notably, the prover's work to update the proof does not depend on the number of steps executed thus far, and the verifier's work to verify a proof does not grow with the number of steps.

NIVC is a generalization of IVC in which the function executed at a particular step can be any one among a fixed set of functions $(F_1, \ldots, F_\ell)$. Which function is active at a given step is determined by the current input and witnesses and is captured by a polynomial-time function φ. Formally, an NIVC proof establishes that there exists $(\omega_0, \ldots, \omega_{i-1})$ such that on initial input z_0 and claimed output z, by computing $z_{j+1} \leftarrow F_{\varphi(z_j,\omega_j)}(z_j, \omega_j)$ for all $j \in \{0, \ldots, i-1\}$, we have that $z = z_i$. IVC can be seen as a special case of NIVC when $\ell = 1$. A formal definition of NIVC is in Appendix A.4.

Folding schemes [29] are protocols used to construct IVC (and NIVC) schemes. In a folding scheme, a prover and verifier reduce the task of checking two instances of a relation $\mathcal{R}$ with some structure s to checking a single instance in $\mathcal{R}$ with the same structure. HyperNova generalizes this with *multi-folding* schemes which take two relations $\mathcal{R}_1$ and $\mathcal{R}_2$ that together satisfy certain simple compatibility predicates. For some parameters μ and ν, checking the correctness of μ instances of a relation $\mathcal{R}_1$ with structure s_1 and ν instances of a relation $\mathcal{R}_2$ with structure s_2 can be reduced to checking a single instance in $\mathcal{R}_1$.

Definition 2 (Multi-folding schemes [27]). *Consider relations $\mathcal{R}_1$ and $\mathcal{R}_2$ over public parameters, structure, instance, and witness tuples, a predicate compat that structures for instances in $\mathcal{R}_1$ and $\mathcal{R}_2$ must satisfy, and size parameters $\mu, \nu \in \mathbb{N}$. A multi-folding scheme for $(\mathcal{R}_1, \mathcal{R}_2, \mathsf{compat}, \mu, \nu)$ is defined by PPT algorithms $(g, \mathcal{P}, \mathcal{V})$ and deterministic Enc denoting the generator, prover, verifier and encoder respectively with the following interface:*

- $g(1^\lambda, N) \rightarrow \mathsf{pp}$*: on input security parameter λ and size bounds N, samples public parameters pp*
- $\mathsf{Enc}(\mathsf{pp}, (\mathsf{s}_1, \mathsf{s}_2)) \rightarrow (\mathsf{pk}, \mathsf{vk})$*: on input pp, and structures s_1 and s_2, among the instances the instances to be folded, output prover and verifier keys*
- $\mathcal{P}(\mathsf{pk}, (\vec{\mathsf{u}}_1, \vec{\mathsf{w}}_1), (\vec{\mathsf{u}}_2, \vec{\mathsf{w}}_2)) \rightarrow (\mathsf{u}, \mathsf{w})$*: on input a vector of instances $\vec{\mathsf{u}}_1$ in $\mathcal{R}_1$ of length μ with structure s_1 and a vector of instances $\vec{\mathsf{u}}_2$ in $\mathcal{R}_2$ of length ν with structure s_2, and corresponding witness vectors $\vec{\mathsf{w}}_1$ and $\vec{\mathsf{w}}_2$ outputs a folded instance-witness pair (u, w) in $\mathcal{R}_1$ with structure s_1.*
- $\mathcal{V}(\mathsf{vk}, (\vec{\mathsf{u}}_1, \vec{\mathsf{u}}_2)) \rightarrow \mathsf{u}$*: on input a vector of instances $\vec{\mathsf{u}}_1$ and a vector of instances $\vec{\mathsf{u}}_2$ outputs a new instance u.*

HyperNova provides a compiler to go from multi-folding schemes, a generalization of folding schemes, to NIVC, given that the folding schemes satisfy four certain properties that make them "NIVC-compatible".

Definition 3 (NIVC-compatible multi-folding schemes [27]). *Consider a relation $\mathcal{R}_1$, and a committed relation $\mathcal{R}_2$ over an underlying relation $\mathcal{R}_2'$. A succinct, non-interactive multi-folding scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathcal{P}, \mathcal{V})$ with deterministic $\mathcal{V}$ for $(\mathcal{R}_1, \mathcal{R}_2, \mathsf{compat}, 1, 1)$ is NIVC-compatible if it satisfies the following properties.*

1. *NP-completeness: There exists a deterministic polynomial-time efficiently invertible function* enc *such that for any arithmetic circuit F, input x, non-deterministic input w, and output y, for structure-instance-witness tuple $(\mathsf{s}_2, \mathsf{u}, \mathsf{w}) \leftarrow \mathsf{enc}(F, (x, y), w)$ we have that $(\mathsf{s}_2, \mathsf{u}, \mathsf{w}) \in \mathcal{R}_2'$ iff $F(x, w) = y$.*
2. *Partial functions: There exists deterministic, efficiently-invertible polynomial-time functions $\mathsf{enc}_{\mathsf{str}}$ and $\mathsf{enc}_{\mathsf{inst}}$ such that for any arithmetic circuit F, input x, non-deterministic input w, and output y, for $\mathcal{R}_1'$ and $\mathcal{R}_2'$ structures $(\mathsf{s}_1, \mathsf{s}_2) \leftarrow \mathsf{enc}_{\mathsf{str}}(F)$ and $\mathcal{R}_2'$ instance $u \leftarrow \mathsf{enc}_{\mathsf{inst}}((x, y))$ we have that $(\mathsf{s}_2, \mathsf{u}, \mathsf{w}) = \mathsf{enc}(F, (x, y), w)$ for some $\mathcal{R}_2'$ witness w and that $\mathsf{compat}(\mathsf{s}_1, \mathsf{s}_2) = 1$.*
3. *Monotonicity: For arithmetic circuits F and G, given $|F| \leq |G|$ we have that $|\mathsf{enc}_{\mathsf{str}}(F)| \leq |\mathsf{enc}|_{\mathsf{str}}(G)$. The term $|F|$ denotes the total number of gates in F and $|\mathsf{enc}_{\mathsf{str}}(F)|$ denotes the number of constraints in $\mathsf{enc}_{\mathsf{str}}(F)$.*
4. *Default instances: There exists $(\mathsf{u}_\perp, \mathsf{w}_\perp)$ such that for any public parameters pp and structure s, we have that $(\mathsf{pp}, \mathsf{s}, \mathsf{u}_\perp, \mathsf{w}_\perp) \in \mathcal{R}_1$.*

3 Commitment-carrying NIVC from multi-folding schemes

This section introduces "commitment-carrying NIVC", a generalization of NIVC where an NIVC proof contains a commitment to the non-deterministic witness given thus far by the prover. Section 4 shows how this unlocks an efficient read-write memory in folding schemes.

3.1 Formalizing commitment-carrying NIVC

For commitment-carrying NIVC, we require a commitment scheme for committing to a sequence of vectors such that it meets two properties (Definition 6). First, it satisfies the standard notion of *binding*: it is computationally infeasible to find two different sequences of vectors that commit to the same commitment. Second, the commitment scheme is *incremental*: given a commitment $\mathcal{C}_i$ for a sequence of vectors $(\omega_0, \ldots, \omega_{i-1})$, one can efficiently compute a commitment $\mathcal{C}_{i+1}$ for the sequence $(\omega_0, \ldots, \omega_{i-1}, \omega_i)$—while only accessing ω_i and $\mathcal{C}_i$. An example of such a commitment scheme is a hash chain where nodes store vectors and the tail of the hash chain is a commitment to a particular sequence of vectors: $\mathcal{C}_{i+1} \leftarrow H(\mathcal{C}_i, \omega_i)$. In Nebula, for efficiency, the incremental commitment scheme (§3.2) hashes *commitments* to witness vectors (e.g., Pedersen or KZG [24]) rather than the vectors themselves. Below, we focus on defining its interface.

Definition 4. *An incremental commitment scheme is a tuple of PPT algorithms* $IC = (\mathsf{Gen}, \mathsf{Commit}, \mathsf{Open})$.

- $\mathsf{Gen}(\lambda) \to \mathsf{pp}$*: On input security parameter* λ*, produces public parameters* pp.
- $\mathsf{Commit}(\mathsf{pp}, C_i, \omega_i) \to C_{i+1}$*: For some* $i \geq 0$*, on input a message* ω_i*, and a prior commitment* C_i*, with* $C_0 = \bot$*, outputs an updated commitment* C_{i+1}.
- $\mathsf{Open}(\mathsf{pp}, C_{i+1}, [\omega_0, \dots, \omega_i]) \to \{0, 1\}$*: For some* $i \geq 0$*, on input a commitment* C_{i+1} *a purported sequence of messages* $[\omega_0, \dots, \omega_i]$*, outputs a Boolean.*

Below, we define commit-carrying NIVC, generalizing Definition 16.

Definition 5 (Commitment-carrying NIVC). *A commitment-carrying non-uniform incrementally verifiable computation (NIVC) scheme is defined by PPT algorithms* $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ *and a deterministic* $\mathcal{K}$ *denoting the generator, the prover, the verifier, and the encoder respectively, and an* $\boxed{\textit{incremental commitment scheme } \mathsf{IC}}$, *with the following interface:*

- $\mathcal{G}(1^\lambda, N) \to \mathsf{pp}$*: on input security parameter* λ *and size bounds* N*, samples public parameters* pp *including parameters for* IC *that we denote with* $\boxed{\mathsf{pp}_{\mathsf{IC}}}$.
- $\mathcal{K}(\mathsf{pp}, ((F_1, \dots, F_\ell), \varphi)) \to (\mathsf{pk}, \mathsf{vk})$*: on input public parameters* pp*, a control function* φ*, and functions* $F_1, \dots, F_\ell$ *deterministically produces a prover key* pk *and a verifier key* vk.
- $\mathcal{P}(\mathsf{pk}, (i, z_0, z_i, \boxed{C_i}), \omega_i, \Pi_i) \to \Pi_{i+1}$*: on input a prover key* pk*, a counter* i*, initial input* z_0*, initial commitment* $C_0 = \bot$*, claimed output after* i *increments* z_i*, claimed carried commitment after* i *increments* C_i*, a non-deterministic advice* ω_i*, and an NIVC proof* Π_i *attesting to* z_i *and* C_i*, produces a new proof* Π_{i+1} *attesting to* $z_{i+1} = F_{\varphi(z_i,\omega_i)}(z_i, \omega_i)$ *and* $C_{i+1} = \mathsf{IC}.\mathsf{Commit}(\mathsf{pp}_{\mathsf{IC}}, C_i, \omega_i)$.
- $\mathcal{V}(\mathsf{vk}, (i, z_0, z_i, \boxed{C_i}), \Pi_i) \to \{0, 1\}$*: on input a verifier key* vk*, a counter* i*, an initial input* z_0*, a claimed output after* i *increments* z_i*, a claimed carried commitment after* i *increments* C_i*, and an NIVC proof* Π_i *attesting to* z_i *and* C_i*, outputs* 1 *if* Π_i *is accepting,* 0 *otherwise.*

An NIVC scheme $(\mathcal{G}, \mathcal{K}, \mathcal{P}, \mathcal{V})$ *satisfies following requirements.*

(i) Completeness: For any PPT adversary $\mathcal{A}$ *we have that*

$$\Pr\left[b = 1 \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda, N), \\ (((F_1, \dots, F_\ell), \varphi), (i, z_0, z_i, \boxed{C_i}), (\omega_i, \Pi_i)) \leftarrow \mathcal{A}(\mathsf{pp}), \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, ((F_1, \dots, F_\ell), \varphi)), \\ \mathcal{V}(\mathsf{vk}, (i, z_0, z_i, \boxed{C_i}), \Pi_i) = 1, \\ z_{i+1} \leftarrow F_{\varphi(z_i,\omega_i)}(z_i, \omega_i), \\ \boxed{C_{i+1} \leftarrow \mathsf{IC}.\mathsf{Commit}(\mathsf{pp}_{\mathsf{IC}}, C_i, \omega_i)}, \\ \Pi_{i+1} \leftarrow \mathcal{P}(\mathsf{pk}, (i, z_0, z_i, \boxed{C_i}), \omega_i, \Pi_i), \\ b \leftarrow \mathcal{V}(\mathsf{vk}, (i+1, z_0, z_{i+1}, \boxed{C_{i+1}}), \Pi_{i+1}) \end{array} \right] = 1$$

where $\ell \geq 1$, φ produces an element in $\mathbb{Z}^*_{\ell+1}$, Moreover, φ and each F_j for $j \in \{1, \dots, \ell\}$ are a polynomial-time computable function represented as arithmetic circuits.

(ii) Knowledge Soundness: Consider constant $n \in \mathbb{N}$. For all expected polynomial-time adversaries $\mathcal{P}^*$ there exists an expected polynomial-time extractor $\mathcal{E}$ such that

$$\Pr_{\mathsf{r}}\left[\begin{array}{l|l} z_n = z \wedge \boxed{C_n = C} \text{ where} & \\ z_{i+1} \leftarrow F_{\varphi(z_i,\omega_i)}(z_i, \omega_i) & \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda, N), \\ \boxed{C_{i+1} \leftarrow \mathsf{IC.Commit}(\mathsf{pp_{IC}}, C_i, \omega_i)} & (((F_1, \dots, F_\ell), \varphi), (z_0, z, \boxed{C}), \mathit{\Pi}) \leftarrow \mathcal{P}^*(\mathsf{pp}, \mathsf{r}), \\ \forall i \in \{0, \dots, n-1\} & (\omega_0, \dots, \omega_{n-1}) \leftarrow \mathcal{E}(\mathsf{pp}, \mathsf{r}) \end{array}\right] \approx$$

$$\Pr_{\mathsf{r}}\left[\mathcal{V}(\mathsf{vk}, (n, z_0, z, \boxed{C}), \mathit{\Pi}) = 1 \,\middle|\, \begin{array}{l} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda, N), \\ (((F_1, \dots, F_\ell), \varphi), (z_0, z, \boxed{C}), \mathit{\Pi}) \leftarrow \mathcal{P}^*(\mathsf{pp}, \mathsf{r}), \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, ((F_1, \dots, F_\ell), \varphi)) \end{array}\right]$$

where r denotes an arbitrarily long random tape.

(iii) Succinctness: The NIVC proof size is independent of the iteration count.

(iv) Efficiency: The prover's time complexity at any step i is linear in the size of the function applied at step i and the total number of functions ℓ.

3.2 A compiler from folding schemes to commitment-carrying NIVC

A straightforward way to construct CC-NIVC is building it atop an existing NIVC scheme: augment the step function to maintain and increment the carried commitment. Concretely, consider NIVC with a single step function F. The augmented step function F' takes at step i input tuple $z'_i = (z_i, C_i)$ and outputs $(F(z, \omega), \mathsf{IC.Commit}(C_i, \omega_i))$. It can be shown that any generic IVC scheme instantiated with F' as the step function and initial state $z'_0 = (z_0, \perp)$ realizes commitment-carrying IVC. The drawback, however, is that this transformation requires $\mathsf{IC.Commit}$ to be performed as part of the step function, which means capturing it as an arithmetic circuit. This is expensive even with SNARK-friendly primitives as the size of this circuit is linear in the size of the witness.

To avoid this overhead, we leverage the fact that most folding schemes already require the prover to compute commitments to witnesses at each step. We now describe a general compiler that takes a multi-folding scheme for an NP-complete relation with mild requirements and produces a commitment-carrying NIVC scheme. This compiler is a direct augmentation of the HyperNova [27] compiler.

We first design an incremental commitment scheme using by any commitment scheme any hash function. This will be instantiated with the commitment scheme underlying the committed relations in the folding scheme.

Construction 1. Let hash be a hash function and $\mathsf{Com} = (\mathsf{Gen}, \mathsf{Commit}, \mathsf{Open})$ be a commitment scheme. Let n denote an upper bound on the size of the witness that will be committed. Define incremental commitment scheme $\mathsf{IC}_{H,\mathsf{Com}}$ as:

- $\mathsf{IC.Gen}(1^\lambda, n) \to \mathsf{pp}$: Output $\mathsf{Com.Gen}(\lambda, n)$
- $\mathsf{IC.Commit}(\mathsf{pp}, C_{i-1}, \omega_{i-1}) \to C_i$:

  If $i = 1$, set $C_0 = \perp$;

  Output $H(C_{i-1}, \mathsf{Com.Commit}(\mathsf{pp}, \omega_{i-1}))$
- $\mathsf{IC.Open}(\mathsf{pp}, C_i, [\omega_0, \dots, \omega_{i-1}]) \to \{0, 1\}$:

  Set $C'_0 \leftarrow \perp$;

  For $j = 1$ to i, $C'_i \leftarrow \mathsf{Com.Commit}(\mathsf{pp}, C'_{i-1}, \omega_{i-1})$;

  Output 1 if $C_i = C'_0$ and 0 otherwise

Construction 2 (CC-NIVC from multi-folding schemes). Consider a relation $\mathcal{R}_1$ and a committed relation $\mathcal{R}_2$ for a commitment scheme $\mathsf{Com} = (\mathsf{Gen}, \mathsf{Commit}, \mathsf{Open})$. Let NIFS be an NIVC-compatible non-interactive multi-folding scheme for a single instance of $\mathcal{R}_1$ and $\mathcal{R}_2$. Let $(\mathsf{u}_\perp, \mathsf{w}_\perp)$ be a default instance-witness pair for $\mathcal{R}_1$ that satisfies any structure and public parameters. We construct an NIVC scheme as follows.

Consider deterministic polynomial-time functions φ, ℓ polynomial-time functions $(F_1, \dots, F_\ell)$ that take non-deterministic input, and a cryptographic hash function hash. We now define augmented functions F'_j for $j \in [\ell]$, where all input arguments are taken as non-deterministic advice, as follows.

$\underline{F'_j(\mathsf{vk}_{\mathsf{fs}}, \mathsf{U}_i, \mathsf{u}_i, \mathsf{pc}_i, (i, z_0, z_i, \boxed{C_{i-1}}), \omega_i, \pi) \to \mathsf{x}}$:

1. Compute the next program counter $\mathsf{pc}_{i+1} \in [\ell] \leftarrow \varphi(z_i, \omega_i)$.
2. Compute the next output $z_{i+1} \leftarrow F_j(z_i, \omega_i)$.
3. If $i = 0$:
   (a) Check that $z_0 = z_i$ to ensure that the statement holds in the base case.
   (b) Set $\mathsf{U}_{i+1} \leftarrow (\mathsf{u}_\perp, \dots, \mathsf{u}_\perp)$.
4. Otherwise:
   (a) Parse u_i as (C_W, u'_i), a commitment to the witness and the remainder. $\boxed{\text{Further parse } C_W \text{ as } (C_{\omega_{i-1}}, C_{aux_{i-1}})}$.
   (b) Check that u'_i references U_i in the output of the prior iteration:

   $$\mathsf{u}'_i \overset{?}{=} \mathsf{enc}_{\mathsf{inst}}(\mathsf{hash}(\mathsf{vk}_{\mathsf{fs}}, i, z_0, z_i, \mathsf{U}_i, \mathsf{pc}_i, \boxed{C_{i-1}})).$$

   (c) Check that $1 \le \mathsf{pc}_i \le \ell$.
   (d) Copy $\mathsf{U}_{i+1} \leftarrow \mathsf{U}_i$ and update $\mathsf{U}_{i+1}[\mathsf{pc}_i] \leftarrow \mathsf{NIFS}.\mathcal{V}(\mathsf{vk}_{\mathsf{fs}}[\mathsf{pc}_i], \mathsf{U}_i[\mathsf{pc}_i], \mathsf{u}_i, \pi)$.

5. $\boxed{\text{If } i = 0 \text{ then } C_i \leftarrow \perp, \text{ else } C_i \leftarrow \mathsf{hash}(\mathsf{pp.pp_{IC}}, C_{i-1}, C_{\omega_{i-1}})}$

6. Output $\mathsf{x} \leftarrow \mathsf{hash}(\mathsf{vk_{fs}}, i+1, z_0, z_{i+1}, \mathsf{U}_{i+1}, \mathsf{pc}_{i+1}, \boxed{C_i})$.

Next, we define the CC-NIVC scheme $(\mathcal{G}, \mathcal{K}, \mathcal{P}, \mathcal{V})$ as follows.

$\underline{\mathcal{G}(1^\lambda, N) \rightarrow \mathsf{pp}}$: Output $(\mathsf{IC.Gen}(1^\lambda, N), \mathsf{NIFS}.\mathcal{G}(1^\lambda, N))$.

$\underline{\mathcal{K}(\mathsf{pp}, ((F_1, \ldots, F_\ell), \varphi)) \rightarrow (\mathsf{pk}, \mathsf{vk})}$:

1. Compute $(\mathsf{s}_{1,j}, \mathsf{s}_{2,j}) \leftarrow \mathsf{enc_{str}}(F'_j)$ for all $j \in [\ell]$.
2. Compute $(\mathsf{pk}_{\mathsf{fs},j}, \mathsf{vk}_{\mathsf{fs},j}) \leftarrow \mathsf{NIFS}.\mathcal{K}(\mathsf{pp}, \mathsf{s}_{1,j}, \mathsf{s}_{2,j})$ for all $j \in [\ell]$.
3. Compute and output the prover and verifier keys.

$$\begin{aligned} \mathsf{vk} &\leftarrow (\mathsf{pp}, (\mathsf{vk}_{\mathsf{fs},1}, \ldots, \mathsf{vk}_{\mathsf{fs},\ell}), (\mathsf{s}_{1,1}, \ldots, \mathsf{s}_{1,\ell}), (\mathsf{s}_{2,1}, \ldots, \mathsf{s}_{2,\ell})) \\ \mathsf{pk} &\leftarrow ((\varphi, (F_1, \ldots, F_\ell)), (\mathsf{pk}_{\mathsf{fs},1}, \ldots, \mathsf{pk}_{\mathsf{fs},\ell}), \mathsf{vk}) \end{aligned}$$

$\underline{\mathcal{P}(\mathsf{pk}, (i, z_0, z_i, \boxed{C_i}), \omega_i, \Pi_i) \rightarrow \Pi_{i+1}}$:

1. Parse Π_i as $((\mathsf{U}_i, \mathsf{W}_i), (\mathsf{u}_i, \mathsf{w}_i), \mathsf{pc}_i, \boxed{C_{i-1}})$.
2. $\boxed{\text{Parse } \mathsf{u}_i.C_W \text{ as } (C_{\omega_{i-1}}, C_{aux_{i-1}}). \text{ Abort if } C_i \neq \mathsf{hash}(C_{i-1}, C_{\omega_{i-1}})}$
3. Compute the next program counter $\mathsf{pc}_{i+1} \in [\ell] \leftarrow \varphi(z_i, \omega_i)$.
4. If $i = 0$: Let $(\mathsf{U}_{i+1}, \mathsf{W}_{i+1}, \pi) \leftarrow ((\mathsf{u}_\perp, \ldots, \mathsf{u}_\perp), (\mathsf{w}_\perp, \ldots, \mathsf{w}_\perp), \perp)$.
   Otherwise: Copy $\mathsf{U}_{i+1} \leftarrow \mathsf{U}_i$ and $\mathsf{W}_{i+1} \leftarrow \mathsf{W}_i$, and update
   $$(\mathsf{U}_{i+1}[\mathsf{pc}_i], \mathsf{W}_{i+1}[\mathsf{pc}_i], \pi) \leftarrow \mathsf{NIFS}.\mathcal{P}(\mathsf{pk}[\mathsf{pc}_i], (\mathsf{U}_i[\mathsf{pc}_i], \mathsf{W}_i[\mathsf{pc}_i]), (\mathsf{u}_i, \mathsf{w}_i)).$$
5. Compute the output $y \leftarrow F'_{\mathsf{pc}_{i+1}}(\mathsf{vk_{fs}}, \mathsf{U}_i, \mathsf{u}_i, \mathsf{pc}_i, (i, z_0, z_i, \boxed{C_{i-1}}), \omega_i, \pi)$.
6. Compute an instance-witness pair encoding the valid execution of $F'_{\mathsf{pc}_{i+1}}$:
   $$(\_, \mathsf{u}'_{i+1}, \mathsf{w}_{i+1}) \leftarrow \mathsf{enc}(F'_{\mathsf{pc}_{i+1}}, (\perp, y), (\mathsf{vk_{fs}}, \mathsf{U}_i, \mathsf{u}_i, \mathsf{pc}_i, (i, z_0, z_i, \boxed{C_{i-1}}), \omega_i, \pi)).$$
7. Compute the committed instance: $\mathsf{u}_{i+1} \leftarrow (\mathsf{Commit}(\mathsf{pp}, \mathsf{w}_{i+1}), \mathsf{u}'_{i+1})$.
8. Output $\Pi_{i+1} \leftarrow ((\mathsf{U}_{i+1}, \mathsf{W}_{i+1}), (\mathsf{u}_{i+1}, \mathsf{w}_{i+1}), \mathsf{pc}_{i+1}, \boxed{C_i})$

$\underline{\mathcal{V}(\mathsf{vk}, (i, z_0, z_i, \boxed{C_i}), \Pi_i) \rightarrow \{0,1\}}$:

1. If $i = 0$, output 1 if $z_i = z_0$ and 0 otherwise.
2. Parse Π_i as $((\mathsf{U}_i, \mathsf{W}_i), (\mathsf{u}_i, \mathsf{w}_i), \mathsf{pc}_i, \boxed{C_{i-1}})$.

3. Parse u_i as (C, u'_i). Check that $\mathsf{u}'_i = \mathsf{enc_{inst}}(\mathsf{hash}(\mathsf{vk_{fs}}, i, z_0, z_i, \mathsf{U}_i, \mathsf{pc}_i, \boxed{C_{i-1}}))$.

4. Check that $1 \leq \mathsf{pc}_i \leq \ell$.

5. Check $(\mathsf{pp}, \mathsf{s}_{1,j}, \mathsf{U}_i[j], \mathsf{W}_i[j]) \in \mathcal{R}_1$ for $j \in [\ell]$ and $(\mathsf{pp}, \mathsf{s}_{2,\mathsf{pc}_i}, \mathsf{u}_i, \mathsf{w}_i) \in \mathcal{R}_2$.

6. $\boxed{\text{Parse } \mathsf{u}_i.C_W \text{ as } (C_{\omega_{i-1}}, C_{aux_{i-1}}). \text{ Then check that } C_i = \mathsf{hash}(C_{i-1}, C_{\omega_{i-1}})}$

We formally prove the following lemma in Appendix B.1.

Theorem 1 (CC-NIVC from multi-folding schemes). *Construction 2 takes a NIVC-compatible multi-folding scheme and produces a CC-NIVC scheme.*

Proof (Intuition). Given a malicious prover $\mathcal{P}_i$ that produces a convincing CC-NIVC proof $(\mathit{\Pi}_i, C_i)$ of i iterations with non-negligible probability, we can construct an extractor $\mathcal{E}_{i-1}$ that extracts a proof $\mathit{\Pi}_{i-1}$ along with witness ω_{i-1} by the knowledge soundness of the underlying folding scheme. Note that $\mathit{\Pi}_i$ contains C_{i-1} and u_i from which $C_{\omega_{i-1}}$ can be parsed. $(\mathit{\Pi}_i, C_i)$ being convincing implies that $C_i = \mathsf{hash}(C_{i-1}, C_{\omega_{i-1}})$. As the commitments are binding, $C_{\omega_{i-1}} = \mathsf{Com.Commit}(\omega_{i-1})$. Thus, $C_i = \mathsf{IC.Commit}(\mathsf{pp_{IC}}, C_{i-1}, \omega_{i-1})$ as desired. This lets us construct a new prover $\mathcal{P}^*_{i-1}$ that uses $\mathcal{E}_{i-1}$ to output convincing proof $(\mathit{\Pi}_{i-1}, C_{i-1})$. We can now recursively repeat this process with $\mathcal{P}^*_{i-1}$ to construct an extractor that extracts the witness at each step of CC-NIVC. $\square$

Lemma 1 (Overheads of CC-NIVC over NIVC). *In Construction 2, the overhead to support CC-NIVC on top of HyperNova's NIVC compiler is as follows. The committed instances contain an additional commitment. The size of an CC-NIVC proof is larger by two commitments and a hash value. Each F'_j now performs an additional hash computation and a group scalar multiplication operation.*

Proof. The increase in sizes of committed instances is due to them holding a commitment to the non-deterministic witness ω provided at each step. The extra work performed by each F'_j is incrementing the carried commitment at each step i as $C_i \leftarrow \mathsf{hash}(C_{i-1}, C_{\omega_{i-1}})$. Note that this cost is independent of $|\omega_{i-1}|$. Furthermore, since the non-deterministic advice is committed separately, the verifier circuit performs an additional group scalar multiplication. $\square$

4 Efficient read-write memory in IVC

This section describes how to devise an efficient read-write memory primitive for recursive proof systems using commitment-carrying NIVC. For ease of exposition, we focus on IVC, but our description generalizes easily to NIVC.

4.1 Problem statement: Augmenting IVC with memory operations

Suppose that F is a potentially non-deterministic function of the form $z' \leftarrow F(z, \omega)$, where z is an input and ω is a non-deterministic input, and z' is the output. An IVC scheme proves the knowledge of witnesses for instances of the form (i, z_0, z_i), where i is the number of iterations executed thus far, z_0 is the initial input, and z_i is the output after i applications of F.

When F represents the transition function of a CPU or virtual machine (e.g., RISC-V, EVM), F must be able to read from and write to a *global* memory. By global, we mean that if at step i, a particular value is written to memory, a subsequent read operation performed at steps $j \geq i$ must reflect the effects of that write. This is formally captured by sequential consistency [30]. In a bit more detail, suppose that the prover maintains a memory M, where M is a vector of certain size. During invocations of F, it may perform one or more memory operations, where an operation can be either a read and a write. For read(a), where a is some address (i.e., index) in M, an honest prover provides the value currently stored at index a in M. For write(a, v), where a is some address and v is some value, an honest prover stores v at index a in M.

Research question. We wish to design an IVC scheme where the prover proves the correct execution of invocations of F including the correctness of all memory operations performed during those invocations of F. In addition, the prover's space requirement should be proportional to $O(|M| + |F|)$.

4.2 An overview of Nebula's approach.

To solve the question posed above, Nebula employs offline memory checking [12,18,7,35,34,40,8], notably Spice [35] that provided the first integration of offline memory checking with zkSNARKs. In a nutshell, offline memory checking provides a streaming procedure that allows checking if a sequence of memory operations respect sequential consistency i.e., a read(a) returns the value that last written to address a. Specifically, this approach reduces the task of checking memory operations to the task of multiset equality checks and a collection of range checks on timestamp entries in the multisets. The multiset equality check is performed on collision-resistant hashes of multisets.

There are two known choices for a multiset hash function: (1) a deterministic hash function [35]; and (2) a randomized hash function [34,8]. The first option has the advantage that it is straightforward to integrate offline memory checking with IVC: the approach in Spice in the context of SNARKs effectively transfers over to the setting of IVC. The main downside of this approach is that each memory operation requires up to thousands of constraints per memory operation. The second approach computes a randomized fingerprint of multisets by computing a grand product of entries in multisets. Compared to the first option, it only requires a handful of constraints per entry in the multiset, but the randomness must be chosen *after* the prover commits to the multisets. In monolithic proof systems (e.g., Jolt [8]), the latter is not a problem: the prover executes the program until completion, commits to the required multisets, and then proves the correct computation of randomized fingerprints. Whereas, in IVC, the proof must be incrementally updateable. For example, the prover can pause the execution of a VM, externalize a proof of the execution thus far, and then resume the execution of the VM from the paused state. When the prover resumes a paused VM, it should be able to continue updating the proof it has already produced.

Nebula solves this with a two-layered IVC. In the first layer, the prover produces IVC proofs that can access a global memory. These IVC proofs carry claims about the global memory, where verifying them requires time proportional to the number of memory operations made and the size of the global memory. These claims also leak information about the execution. To avoid these, the prover *finalizes* these IVC proofs by producing an *auxiliary* IVC proof that verifies claims about the global memory. In a nutshell, the auxiliary IVC proof proves the correct computation of multiset hashes of committed vectors.

The finalized IVC proofs are no longer incrementally updateable. To restore incrementality, we observe that an IVC proof produced in folding-based IVC schemes is a collection of instance-witness pairs in some relation (e.g., CCS), so the prover uses the second layer to fold finalized IVC proofs from the first layer. So, in the scenario discussed earlier, when the VM must be paused, the prover finalizes its IVC proof, feeds it to the second layer to obtain a new proof that can be externalized. To resume the VM, the prover starts a new IVC proof in the second layer and at any point it can can finalize it and feed it to the first layer.

Since an auxiliary IVC proof computes multiset hashes of committed multisets, it involves computing products of values that will not be "small" even if the memory contents and addresses lie in a small set (e.g., the set $\{0, \ldots, 2^{32} - 1\}$). We discuss how to devise a special folding scheme for computing these multiset hashes such that the prover only commits to "small" field elements, a property achieved by recent monolithic proof systems [40,8,38].

4.3 Detour: Reducing memory checks to grand product checks

We adapt this subsection from prior work [40,34,23]. We generalize their description to reduce read-write memory checking to grand product checks rather than reducing read-only memory checking to grand product checks.

Recall that in offline memory checking [12], a *trusted checker* issues operations to an untrusted memory. To enable efficient checking using multiset-fingerprinting techniques, the memory is modified so that in addition to storing a value at each address, the memory also stores a timestamp with each address. Moreover, each read operation is followed by a write operation that updates the timestamp associated with that address (but not the value stored there). Similarly, each write operation is preceded by a read operation that reads the current timestamp associated with that address and writes back a new value and a new timestamp. The memory-checking procedure is captured in the codebox below.

Local state of the checker. Three sets: IS, RS, and WS, which are initialized as follows.[3] $\mathsf{RS} = \mathsf{WS} = \{\}$, and for an M-sized memory, IS is initialized to the following set of tuples: for all $i \in [\mathsf{M}]$, the tuple (i, v_i, it_i) is included in IS, where v_i is the initial value stored at address i and it_i is the initial timestamp associated with the value (e.g., 0). Here, $[\mathsf{M}]$ denotes the set $\{0, 1, \ldots, \mathsf{M} - 1\}$. Additionally, the checker maintains a timestamp counter ts that is initialized to the highest timestamp value in IS.

Memory operations and an invariant. For a read operation at address a, suppose that the untrusted memory responds with a value-timestamp pair (v, t). Then the checker updates its local state as follows:

```
1. $ts \leftarrow ts + 1$
2. assert $t < ts$
3. $\mathsf{RS} \leftarrow \mathsf{RS} \cup \{(a, v, t)\}$;
4. store $(v, ts)$ at address $a$ in the untrusted memory; and
5. $\mathsf{WS} \leftarrow \mathsf{WS} \cup \{(a, v, ts)\}$.
```

For a write operation at address a that wishes to write a value of v', suppose that the untrusted memory responds with a value-timestamp pair (v, t). Then the checker updates its local state as follows:

```
1. $ts \leftarrow ts + 1$
2. assert $t < ts$
3. $\mathsf{RS} \leftarrow \mathsf{RS} \cup \{(a, v, t)\}$;
4. store $(v', ts)$ at address $a$ in the untrusted memory; and
5. $\mathsf{WS} \leftarrow \mathsf{WS} \cup \{(a, v', ts)\}$.
```

The following lemma captures the invariant maintained on the sets of the checker.

Lemma 2. *Let* IS*,* WS*, and* RS *denote the multisets maintained by the checker in the above algorithm at the conclusion of memory operations. If for every read operation, the untrusted memory returns the tuple last written to that location, then there exists a set* FS *with cardinality* M *consisting of tuples of the form* (i, v_i', t_i) *for all* $k \in [\mathsf{M}]$ *such that* $\mathsf{IS} \cup \mathsf{WS} = \mathsf{RS} \cup \mathsf{FS}$*. Moreover,* FS *is computable in time linear in* M *(i.e.,* FS *is the final state of memory viewed as a set of address-value-timestamp tuples). Conversely, if the untrusted memory ever returns a value* v *for a memory call* $k \in [\mathsf{M}]$ *such* v *does not equal the value last written to cell* k*, then there does not exist any set* FS *such that* $\mathsf{IS} \cup \mathsf{WS} = \mathsf{RS} \cup \mathsf{FS}$*.*

Proof. A proof of a more general form of the result is given in [36, Lemma C.1]. □

Corollary 1. *Let* $\mathbb{F}$ *be a prime order field. Assuming that the domain of timestamps is* $\mathbb{F}$ *and that the number of memory operations issued* N *is smaller than*

3 The checker in [12] maintains a fingerprint of these sets, but for our exposition, we let the checker maintain full sets.

the field characteristic $|\mathbb{F}|$. Let $\mathsf{IS} = \{(i, v_i, t_i)\}_{i=0}^{\mathsf{M}-1}$, $\mathsf{WS} = \{(wa_i, wv_i, wt_i)\}_{i=0}^{\mathsf{N}-1}$, and $\mathsf{RS} = \{(ra_i, rv_i, rt_i)\}_{i=0}^{\mathsf{N}-1}$ denote the multisets maintained by the checker in the above algorithm at the conclusion of memory operations. If for every read operation, the untrusted memory returns the tuple last written to that location, then there exists a set FS with cardinality M consisting of tuples of the form (i, v_i', t_i') for all $i \in [\mathsf{M}]$ such that, the following holds with probability 1 over the choice of $\gamma_1, \gamma_2 \in \mathbb{F}$:

$$\prod_{i=0}^{\mathsf{M}-1} (i + \gamma_1 \cdot v_i + \gamma_2^2 \cdot t_i - \gamma_2) \cdot \prod_{i=0}^{\mathsf{N}-1} (wa_i + \gamma_1 \cdot wv_i + \gamma_1^2 \cdot wt_i - \gamma_2)$$
$$= \prod_{i=0}^{\mathsf{N}-1} (ra_i + \gamma_1 \cdot rv_i + \gamma_1^2 \cdot rt_i - \gamma_2) \cdot \prod_{i=0}^{\mathsf{M}-1} (i + \gamma_1 \cdot v_i' + \gamma_1^2 \cdot t_i' - \gamma_2)$$

Moreover, FS is computable in time linear in M (i.e., FS is the final state of memory viewed as a set of address-value-timestamp tuples).

Conversely, if the untrusted memory ever returns a value v for a memory call $k \in [\mathsf{M}]$ such v does not equal the value last written to cell k, then for any set FS, the above equality checks holds with probability at most $O(\mathsf{M} + \mathsf{N})/|\mathbb{F}|$, where the probability is over the choice of $\gamma_1, \gamma_2 \in \mathbb{F}$.

Proof. The desired result follows from using public-coin multiset hash functions ([34, §7.2.1]) in the multiset equality check provided by Lemma 2. □

4.4 Details of Nebula's approach

As discussed earlier, Nebula employs a two-layered IVC scheme to solve the problem introduced earlier (§4.1). We discuss each component in Nebula's solution.

(1) Use commitment-carrying IVC to carry untrusted advice regarding memory operations. In our setting (§4.1), the step function F is allowed to perform read and write memory operations on a global memory. To enable this, F receives additional advice for each memory operation that it invokes. This advice is in the form of *two* (address, value, timestamp) tuples per memory operation. Specifically, for a read operation, the advice is (a, v, rt) and (a, v, wt); F checks that the address a in the advice matches the address it requested and then uses the provided value v (e.g., in the rest of its computation). For a write operation, the advice is (a, v, rt) and (a, v', wt); F checks that the address a and the value v' match the address and value it wishes to write. Otherwise, F ignores the remaining components in the provided advice.

Finally, instead of IVC to prove the repeated execution of F, Nebula uses commitment-carrying IVC (§3) to carry an incremental commitment to the untrusted advice provided. Let Π_{F} denote this commitment-carrying IVC proof for an instance (n, z_0, z_n, C_n) after executing n iterations of F with a carried commitment of C_n. Observe that C_n is an incremental commitment to multisets RS and WS in the description provided earlier (§4.3).

(2) Finalize Π_{F} by computing multiset hashes underneath C_n. The proof Π_{F} is incrementally updateable (i.e., it is a commitment-carrying IVC proof). The prover can continue to update it by applying additional iterations of F. Thus, a verifier verifying Π_{F} must also validate that multisets underneath C_n respect sequential consistency. The latter requires the prover to send a preimage of C_n and for the verifier to compute multiset hashes, which we avoid as follows.

We devise two functions F_{ops} and F_{scan} to compute multiset hashes of $(\mathsf{RS}, \mathsf{WS})$ and $(\mathsf{IS}, \mathsf{FS})$ respectively. They also perform the necessary timestamp checks on entries in RS and WS (§4.3). We depict these functions in the following codeboxes.

```
$\mathsf{Hash}(\gamma_1, \gamma_2, a, v, t) \rightarrow h$

  1. return $(a + v \cdot \gamma_1 + t \cdot \gamma_1^2 - \gamma_2)$
```

```
$F_{\mathsf{ops}}(z_i = (\gamma_1, \gamma_2, ts, \mathsf{h_{RS}}, \mathsf{h_{WS}}), \omega_i = (\mathsf{RS}, \mathsf{WS})) \rightarrow z_{i+1}$

  1. assert $|\mathsf{RS}| = |\mathsf{WS}|$
  2. for $i$ in $0..|\mathsf{RS}|$,
      (a) $(a, v, rt) \leftarrow \mathsf{RS}[i]$
      (b) $(a', v', wt) \leftarrow \mathsf{WS}[i]$
      (c) $ts \leftarrow ts + 1$
      (d) assert $rt < ts$
      (e) assert $wt = ts$
      (f) $\mathsf{h_{RS}} \leftarrow \mathsf{h_{RS}} \cdot \mathsf{Hash}(\gamma_1, \gamma_2, a, v, rt)$
      (g) $\mathsf{h_{WS}} \leftarrow \mathsf{h_{WS}} \cdot \mathsf{Hash}(\gamma_1, \gamma_2, a', v', wt)$
  3. return $(\gamma_1, \gamma_2, ts, \mathsf{h_{RS}}, \mathsf{h_{WS}})$
```

```
$F_{\mathsf{scan}}(z_i = (\gamma_1, \gamma_2, \mathsf{h_{IS}}, \mathsf{h_{FS}}), \omega_i = (\mathsf{IS}, \mathsf{FS})) \rightarrow z_{i+1}$

  1. assert $|\mathsf{IS}| = |\mathsf{FS}|$
  2. for $i$ in $0..|\mathsf{IS}|$,
      (a) $(a, v, it) \leftarrow \mathsf{IS}[i]$
      (b) $(a', v', ft) \leftarrow \mathsf{FS}[i]$
      (c) assert $a = a' = i$
      (d) $\mathsf{h_{IS}} \leftarrow \mathsf{h_{IS}} \cdot \mathsf{Hash}(\gamma_1, \gamma_2, a, v, it)$
      (e) $\mathsf{h_{FS}} \leftarrow \mathsf{h_{FS}} \cdot \mathsf{Hash}(\gamma_1, \gamma_2, a', v', ft)$
  3. return $(\gamma_1, \gamma_2, \mathsf{h_{IS}}, \mathsf{h_{FS}})$
```

F_{scan} and F_{ops} require public challenges $\gamma_1, \gamma_2 \in_R \mathbb{F}$ as initial input. For the soundness guarantees of Lemma 1, these challenges must be sampled after the prover has committed to its multisets. For this, the prover first computes an incremental commitment to multisets IS and FS (wlog, assume that these multisets viewed as vector of tuples are chunked into n pieces, where n is the number of steps executed in step (1)). It then derives challenges by hashing those commitments along with C_n. Finally, Nebula use commitment-carrying IVC to iteratively prove the correct execution of F_{ops} and F_{scan}. Let Π_{ops} denote the commitment-carrying

IVC proof proving the correct execution of n invocations of F_{ops} with the instance $(n, (\gamma_1, \gamma_2, ts, \mathsf{h_{RS}}, \mathsf{h_{WS}}), (\gamma_1, \gamma_2, ts', \mathsf{h_{RS}}', \mathsf{h_{WS}}'), C'_n)$. Similarly, let Π_{scan} denote the commitment-carrying IVC proofs proving the correct execution of n invocations of F_{scan} with the instance $(n, (\gamma_1, \gamma_2, \mathsf{h_{IS}}, \mathsf{h_{FS}}), (\gamma_1, \gamma_2, \mathsf{h_{IS}}', \mathsf{h_{FS}}'), (C_{\mathsf{IS}}, C_{\mathsf{FS}}))$.[4] The verifier checks the following in addition to verifying $\Pi_{\mathsf{F}}, \Pi_{\mathsf{ops}}, \Pi_{\mathsf{scan}}$:

1. check $\mathsf{h_{IS}} = \mathsf{h_{RS}} = \mathsf{h_{WS}} = \mathsf{h_{FS}} = 1$ // initial values are correct
2. check $C'_n = C_n$ // commitments carried in both Π_{ops} and Π_{F} are the same
3. check γ_1 and γ_2 are derived by hashing C_n and C''_n.
4. check $\mathsf{h_{IS}} \cdot \mathsf{h_{WS}} = \mathsf{h_{RS}} \cdot \mathsf{h_{FS}}$.

(3) Fold IVC proofs to restore incrementality ("layer 2"). Although Π_{F} is incrementally updateable to include additional iterations of F, Π_{ops} and Π_{scan} are *not* incrementally updateable. This is because the statement that they prove depend on commitments that they carry at the end of nth step. One option is to incrementally update Π_{F} for some number of steps $n' > n$, and then regenerate Π_{ops} and Π_{scan} using a new $C_{n'}$. Unfortunately, the cost to produce these latter proofs grows linearly with n' rather than $n' - n$.

Our core observation is that we can view $(\Pi_{\mathsf{F}}, \Pi_{\mathsf{ops}}, \Pi_{\mathsf{scan}})$ as a "finalized" IVC proof that proves a statement $(n, z_0, z_n, C_{\mathsf{FS}})$, where n is the number of iterations of F applied, z_0 is the initial input, z_n is the final output, and C_{FS} is the second commitment carried by Π_{scan} to the final contents of memory. Furthermore, we can build an IVC proof, which is incremental!, whose goal is to carry the statement that is proven and fold the underlying instances in the IVC proof into a "running" proof. How are IVC proofs themselves folded together? In folding-scheme-based IVC, an IVC proof is a set of instances and their associated witnesses. For example, with Nova [29], an IVC proof is a relaxed R1CS instance U and an R1CS instance u, and their associated witnesses W and w. So, there is a natural folding scheme for two IVC proofs $\Pi_1 = (\mathsf{U}_1, \mathsf{u}_1, \mathsf{W}_1, \mathsf{w}_1)$ and $\Pi_2 = (\mathsf{U}_2, \mathsf{u}_2, \mathsf{W}_2, \mathsf{w}_2)$. In particular, the folding scheme for IVC proofs folds the underlying instances into a single relaxed R1CS instance and the associated witness. Such a folding of IVC proofs can be realized with Nova (and other IVC schemes) by simply devising a step circuit that implements the IVC folding verifier, and that step circuit is tiny: ≈40,000 gates to fold four instances into one with Nova's folding scheme [2].

In more detail, we build the following step circuit whose goal is to fold the instances underlying finalized IVC proof, perform some consistency checks, and forward the statement proven by the finalized IVC proof in its public IO.

```
$\underline{F_{\mathsf{final}}((i, z_0, z_i, ts_i, \mathcal{C}_i, U_i), \omega_i)) \rightarrow (j, z_0, z_j, ts_j, \mathcal{C}_j, U_j)}$
  1. parse $\omega_i$ to obtain $(U_{\mathsf{F}}, U_{\mathsf{ops}}, U_{\mathsf{scan}})$ and public IO in those IVC proofs
     $(j, z_k, z_j, ts_k, ts_j, \gamma_1, \gamma_2, \mathcal{C}_{\mathsf{IS}}, \mathcal{C}_{\mathsf{FS}}, \mathsf{h_{RS}}, \mathsf{h_{FS}}, \mathsf{h_{WS}}, \mathsf{h_{IS}})$.
```

4 Here, we are carrying commitments to IS and FS separately, to facilitate carrying the commitment to final state as the commitment to the initial state in the next iteration in the second layer of IVC.

```
2. check that $j > i$ and $z_k = z_i$ // finalized proof continues prior execution
3. check that $\mathcal{C}_i \overset{?}{=} \mathcal{C}_{\mathsf{IS}}$ // finalized proof starts with previous memory
4. check $ts_i = ts_k$ // finalized proof starts with previous $ts$
5. $\gamma_1 \overset{?}{=} H(\mathcal{C}_{\mathsf{RS}}, \mathcal{C}_{\mathsf{WS}}, \mathcal{C}_{\mathsf{IS}}, \mathcal{C}_{\mathsf{FS}}, 0)$.
6. $\gamma_2 \overset{?}{=} H(\mathcal{C}_{\mathsf{RS}}, \mathcal{C}_{\mathsf{WS}}, \mathcal{C}_{\mathsf{IS}}, \mathcal{C}_{\mathsf{FS}}, 1)$.
7. $\mathsf{h}_{\mathsf{RS}} \cdot \mathsf{h}_{\mathsf{FS}} \overset{?}{=} \mathsf{h}_{\mathsf{WS}} \cdot \mathsf{h}_{\mathsf{IS}}$.
8. fold the instances $(U_{\mathsf{F}}, U_{\mathsf{ops}}, U_{\mathsf{scan}})$ into $U_i$ to get $U_j$
9. output $(j, z_0, z_j, ts_j, \mathcal{C}_j, U_j)$
```

Theorem 2. *Nebula is an IVC scheme where the step function F can read from and write to a global memory with sequential consistency semantics. The prover's space requirements are $O(|F| + \mathsf{M})$, where M is the size of the global memory.*

Proof (sketch). Completeness is easy to check. By invoking the knowledge soundness of the underlying CC-NIVC schemes, we can extract witnesses supplied to producing Nebula's IVC proof including the multisets whose fingerprints were computed using CC-NIVC. By the soundness of Lemma 1, we have that unless the prover respects sequential consistency, the probability that the multiset hash equality checks pass on the extracted vectors is bounded by $O(n + \mathsf{M})/|\mathbb{F}|$, which is negligible in the security parameter since $|\mathbb{F}| \approx 2^{\lambda}$. $\square$

Optimization: computing multiset hash function outside the circuit. In the solution described thus far, a step circuit computes running products, to compute a multiset hash of read sets and write sets. This means that the witness values of the circuit's satisfying assignment contains intermediate product values, which the prover commits to as part of the witness. We now sketch a method to reduce this commitment work substantially. The core idea is to compute this grand product using the protocol of Setty and Lee [37, §6] *outside* the circuit. The circuit then simply folds the verifier of this protocol using techniques from HyperNova [27]. As a result, the in-circuit work reduces to roughly logarithmic in the number of items multiplied together. Outside the circuit, when using the hybrid protocol of Setty and Lee, the prover only needs to commit to a small fraction of the intermediate product values.

5 Non-uniform IVC using a universal "switchboard" circuit

This section describes a new approach for achieving non-uniform IVC, where in an IVC scheme one can execute any number of functions per step (like in Nova [29]) while paying only for the functions that were actually executed (like in SuperNova [25]). Furthermore, this approach works with R1CS without increasing the degree of constraints.

Our core idea is to build a universal switchboard circuit that combines multiple circuits into a single circuit such that the prover can "turn on" one of them and the prover's cost is proportional only to the size of the active circuit.

Note that for commonly used used functionalities (e.g., hashing, digital signature verification), custom circuits are much more efficient than proving those functionalities via the primitive instruction set of a machine (e.g., with bitwise operations, integer arithmetic). Non-uniform IVC allows invoking custom circuits as necessary without paying for them when uninvoked.

5.1 Overview of switchboard circuits with "pay-per-use" costs.

Let ℓ denote the number of functions supported by non-uniform IVC. Suppose that $\mathcal{C}_1, \ldots, \mathcal{C}_\ell$ denote their R1CS representations. We construct a universal "switchboard" circuit, specified by another R1CS $\mathcal{C}$, with "switch" variables that let the prover "turn on" one of $\{\mathcal{C}_1, \ldots, \mathcal{C}_\ell\}$ while "powering down" the others. One can then use β copies of $\mathcal{C}$ to execute β functions per recursive step of an *uniform* IVC scheme (e.g., Nova [29]). Since there is a single circuit to execute β functions, this solution only requires a single invocation of the folding scheme for β functions, which amortizes recursion overheads over β functions. The latter property is unlike SuperNova [25] where each function requires an invocation of the folding scheme verifier.

There are two questions to answer: (1) how do we design switch variables to "turn off" certain constraints without increasing the degree of constraints? (2) how do we make sure the prover's work in the folding scheme is proportional only to size of the "active" circuit rather than the size of the universal circuit? The rest of this section answers these questions. Although we focus on R1CS for simplicity and efficiency, our approach applies to CCS [39], a generalization of R1CS, Plonkish, and AIR. We answer these questions in turn.

5.2 Details of Nebula's switchboard circuit

Recall that in R1CS, a circuit $\mathcal{C}$ is represented with three matrices (A, B, C). Given an input x and a purported witness w, the circuit is satisfying if $Az \circ Bz = Cz$, where $z = (1, x, w)$.

Construction 3 (Switchboard Circuit). Let $\mathcal{C}_i = (A_i, B_i, C_i)$ with dimensions $m_i \times n_i$ for $i \in \{1, \ldots, \ell\}$ denote R1CS matrices of the ℓ different functions. Nebula derives R1CS matrices of the universal circuit $\mathcal{C}^\star = (A^\star, B^\star, C^\star)$. For ease of exposition, we assume that ℓ R1CS circuits having the same public input and output lengths, but possibly different matrix dimensions. We now describe how the matrices of $\mathcal{C}^\star$ are constructed and the structure of its witness vector.

Step 1. Stitching submatrices. Let $|x|$ be the size of the public input. Matrices $A^\star, B^\star$, and $C^\star$ have $m = \sum_{i=1}^{\ell} m_i + 1 + \ell \cdot (1 + |x|)$ rows and $n = 1 + |x| + \sum_{i=1}^{\ell} n_i$ columns. We depict how $A^\star$ is constructed pictorially in Figure 1 ($B^\star$ and $C^\star$ are constructed in a similar manner).

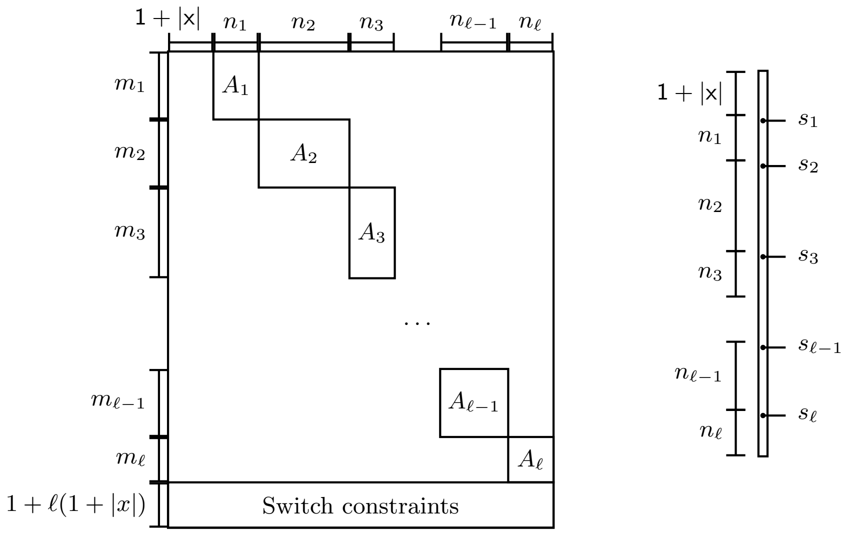

Fig. 1: A depiction of Construction 3, to stitch ℓ R1CS matrices circuits into a single matrix. On the left is matrix A (matrices B, C are constructed similarly) and the right is vector z.

Step 2. Switch constraints. Let z denote the witness vector of the universal circuit, with length $1+|x|+\sum_{i=1}^{\ell} n_i$. Let $s_1,\dots,s_\ell$ be the indices of the first columns of each matrix: $s_i=\sum_{j=1}^{i} n_j$ for $i=1\dots,\ell$. We introduce additional constraints, and each constraint can be realized with a single row to each of $A^\star, B^\star, C^\star$.

1. *Single switch constraint:* Enforce $\sum_{i=1}^{n} z[s_i]=1$.
2. *Binary switch constraints:* For $i\in[1,\ell]$, enforce $z[s_i]\cdot(1-z[s_i])=0$.
3. *Input consistency constraints:* For $i\in[1,\ell], j\in[0,|x|)$, enforce $z[j]=z[s_i]\cdot z[s_i+j]$

As a simple example, if circuit 2 is active at a given step with input x and witness w, then the variables are set as follows. First, $s_2=1$ and $s_i=0$ for all $i\neq 2$. The witness vector z for the switchboard will be of the form $[\mathbf{1},\mathbf{x},0,\mathbf{s_2},\mathbf{x},\mathbf{w},0\dots0]$. Observe that rows corresponding to matrices A_i (similarly B_i and C_i) for $i\neq 2$ result in 0 when multiplied with z so inactive constraints end up checking trivially satisfying constraints $0\cdot 0=0$.

We prove the following lemma in Appendix B.2.

Lemma 3. *Let $C^\star$ be the switch-board circuit generated from circuits $C_1,\dots,C_\ell$. For some input x, if $C^\star$ is satisfied by a witness $\omega^\star$ then we can parse a witness ω from $\omega^\star$ such that, for some $k\in[\ell]$, subcircuit $\mathcal{C}_k$ is satisfied by ω for input x. Moreover, the variables in $\omega^\star$ associated with turned-off subcircuits C_i for all $i\neq k$ are zeros.*

5.3 Making the prover only pay for the size of the active circuit

Lemma 3 shows that witnesses associated with inactive constraints can be safely set to zeros. This already allows the prover to not pay for any cryptographic costs in some folding-based IVC schemes (e.g., HyperNova [27], Mova [19]), where the prover only commits to a witness and committing to zeros is free. Whereas, some folding schemes (e.g., Nova [29], Ova [4]) require the prover to commit to a cross-term, which can be a vector of arbitrary field elements even if the witness contains zeros. We now show that Nova's cross-term commitment cost can be made proportional to the size of the active circuit rather than the size of the total switch-board circuit, which leads to a non-uniform IVC scheme.

Theorem 3. *Let $F_1, \ldots, F_\ell$ and φ be the step functions and selector function in non-uniform IVC. Let $C_1, \ldots, C_\ell$ be the R1CS circuits representing each of those ℓ functions (let these circuits internally embed the computation of φ and check if it selects the circuit it is embedded in). Let each C_i have dimensions $m_i \times n_i$ for $i \in [1, \ell]$. Let $C^\star$ denote the switchboard circuit constructed with $(C_1, \ldots, C_\ell)$. By using $C^\star$ with Nova's uniform IVC scheme, we obtain a non-uniform IVC scheme that can execute any number of functions per recursive step.*

Proof. Suppose prover $\mathcal{P}^\star$ produces an n-step Nova proof of the statement (n, x_0, x_n) with $C^\star$ as the step circuit. From the knowledge soundness of Nova, we can extract $x_1, \ldots, x_{n-1}$ and corresponding witnesses satisfying circuit $C^\star$ for each of the n steps. From Lemma 3, we know that extracting a witness satisfying $\mathcal{C}^\star$ is equivalent to extracting a witness satisfying a particular subcircuit. This implies that we can further parse these witnesses as $\omega_0, \ldots, \omega_n$ such that $F_{\mathsf{pc}_i}(x_i, \omega_i) = x_{i+1}$ for $i \in 0, \ldots, n-1$, where $\mathsf{pc}_i = \varphi(x_i, \omega_i)$ is implied by φ being embedded into each subcircuit.

We can extend this to perform a sequence of $\beta \geq 1$ functions per step of Nova using standard circuit repetition techniques: the step circuit will be β copies of $\mathcal{C}^\star$ with $(\beta - 1) \cdot |x|$ extra constraints to enforce sequential input-output consistency between them.

What remains to be shown is that using $C^\star$ with Nova satisfies the "pay-per-use" efficiency criteria of non-uniform IVC. Recall that Nova folds two relaxed R1CS instances satisfying the following equations: $Az_1 \circ Bz_1 = u_1 Cz_1 + E_1$, and $Az_2 \circ Bz_2 = u_2 Cz_2 + E_2$. Recall that in Nova's setting of IVC, $E_2 = 0$ and $u_2 = 1$, so the cross-term $T \in \mathbb{F}^m$ that the prover commits is given by:

$$T = A \cdot (z_1 + z_2) \circ B \cdot (z_1 + z_2) - u_1 \cdot C \cdot (z_1 + z_2) - E_1$$

As a quick sanity check, suppose that $z_2 = 0$, then $T = Az_1 \circ Bz_1 - u_1 Cz_1 - E_1 = 0$. So if the incoming instance's witness is all zeros, the error term commitment is commitment to a vector of zeros, which takes constant time to compute.

Now, consider the case where z_2 has $m > 0$ non-zero entries and the rest are zeros. We can rewrite the computation of $\overline{T}$ as follows:

$$\overline{T} = \overline{(Az_1 \circ Bz_2)} + \overline{(Az_2 \circ Bz_1)} - u_1 \cdot \overline{Cz_2} - \overline{Cz_1}$$

Without loss of generality assume that the matrix dimensions of ℓ different functions are equal and that the number of rows equals the number of columns. In our context, if z_2 is m-sparse (i.e., there are m non-zero entries and in our context $m = n_{active} + 1 + \ell \cdot (1 + |x|)$), Az_2, Bz_2, and Cz_2 vectors will also be m-sparse. This is because in the switchboard circuit construction, each sub-circuit only accesses witnesses associated with the sub-circuit. The only exception is constraints associated with switch variables. However, there are at most $1 + \ell \cdot (1 + |x|)$ of these constraints, so the sparsity of Az_2, Bz_2, Cz_2 is still guaranteed.

Since $Bz2$ is m-sparse, $Az1 \circ Bz2$ is m-sparse, so the prover can compute $\overline{(Az1 \circ Bz2)}$ with $O(m)$ work. Since Az_2 is m-sparse, $Az_2 \circ Bz_1$ is m-sparse, so the prover can compute $\overline{(Az_2 \circ Bz_1)}$ with $O(m)$ work. We are given Cz_2 is m-sparse, so the prover can compute $u_1\overline{Cz_2}$ with $O(m)$ work.

We need to show that $\overline{Cz_1}$ can be computed with $O(m)$ work. The key idea here is to have the prover cache $\overline{Cz_1}$ and efficiently update it in each iteration of IVC. In more detail, at step zero, $z_1 = 0$ (the default trivial witness in Nova), so $Cz_1 = 0$ and $\overline{(}Cz_1)$ can be computed with constant work. Then at each iteration, we update $z_1 \leftarrow z_1 + r \cdot z_2$, so the prover can do the following:

$$\overline{Cz_1} \leftarrow \overline{Cz_1} + r \cdot \overline{Cz_2}$$

At iteration 0, $\overline{Cz_1}$ is commitment to the zero vector, and the prover knows $\overline{Cz_2}$, so the prover can update $\overline{Cz_1}$ with constant work.

Finally, we observe that we retain pay-per-use efficiency when extending to work with β copies of $C^\star$ per step of Nova, as each step-wise segment of z_2 remains sparse. (the full vector has at most $\beta \cdot m + (\beta - 1) \cdot |x|$ non-zero elements).

$\square$

6 Evaluation

We implement Nebula's techniques, memory checking and switchboard circuits, atop the public Nova implementation [2], and use it to build a subset of the Ethereum virtual machine (EVM) [1]. In this section, we briefly describe the Nebula-based EVM implementation, and measure the impact of Nebula's techniques on proving the validity of Ethereum transactions.

6.1 EVM implementation

When compared to other popular zkVM targets like RISC-V, designing a proof system for the the EVM poses several challenges. It involves a larger number of primitive operations (about 150 in total). The operands involved are also large (256 bits). The latter makes it hard to employ existing techniques such as lookups [40,8] to arithmetize instructions. Moreover, EVM involves five different bodies of memory, some storing 256-bit values which are larger than the elements in most of the commonly used fields in proof systems.

Our proto-EVM implementation supports 100 operations with their circuit sizes from under 100 constraints (e.g., for the `POP` operation, which reads and removes the topmost element of the stack), to several thousands (e.g., for the `SHL` shift operation). The total size of the switch-board circuit is under 120,000 R1CS constraints with the largest opcode being a custom hash function we use in substitute of SHA3 account for 20% of the total size.[5] There are five bodies of memory in the EVM: (1) the program code, (2) stack, (3) RAM, (4) calldata, and (5) storage. We assume that the calldata and memory can be up to 2^{16} in length, which are reasonable estimates given the nature of the program and the "gas" (cost per operation in Ethereum) involved. The stack can reach a maximum height of 1024 cells, each of which is 256 bits in length. See Appendix C for more details on the EVM and how we capture various EVM opcodes and memory operations as R1CS constraints.

6.2 Experimental evaluation of proto-EVM

We evaluate Nebula-based EVM prover on the widely used ERC20 token transfer program. According to analysis [6] done in 2020, over 70% of Ethereum transactions are ERC20 token transfers, and they account for over a third of the gas costs on the Ethereum blockchain. The function takes 635 EVM steps to transfer tokens from one account to another.

Baselines. Our baseline here is Nova [2] using a universal multiplexer circuit with Spice memory checking [35] (which is more efficient than Merkle trees and is compatible with IVC). We instantiate Spice's multiset CRHF, which is a traditional hash function composed with a map-to-curve function, as follows: we use Poseidon for the hash function and Elligator-2 [11] for the map-to-curve function. Assuming that an address-value-timestamp variable can be packed inside two field elements, Spice costs a little over 500 R1CS constraints per multiset hash. Spice computes two multiset hashes, and a timestamp range check per memory operation. We perform range checks with simple bit decomposition, which costs $1 + \log m$ constraints where m is the number of memory operations in the program. (We note that there are more efficient methods, such as lookup arguments, to do this.) Nebula's memory is a Spice-style memory checking with a

5 We replace the SHA3 opcode with a cheaper hash function. This does not affect Nebula's relative performance. This was necessary to let our baselines run without running out of memory.

public-coin based hash function, we perform range checks with the same technique, so we find this to be an acceptable baseline.

On top of the baseline, we add each of Nebula's techniques separately and in tandem, and measure the prover time. We run our experiments on a commodity laptop with a 2.4GHz 8-core Intel Core i9 processor. Table 1 depicts our results. Without Nebula's techniques, the baseline setup ran out of memory when running on the laptop.

Methods			**Proving time**	
Memory	**Switchboard?**	**Constraints**	**per txn**	**Relative**
Spice	No	3640 K	OOM	OOM
Spice	✓	3640 K	2400 s	480×
Nebula	No	115 K	1300 s	260×
Nebula	✓	116 K	5 s	1×

Table 1: Circuit sizes and proving times obtained when proving the correctness of ERC-20 transactions (involving 635 steps) with a combination of Nebula's techniques and existing baselines. Note that the implementations batch multiple EVM steps per IVC step and the numbers shown are for the best-performing batch size in each setting.

Our benchmarks show a 260× improvement from using a switchboard circuit instead of a multiplexer-based universal circuit. This roughly correlates with the sizes of the operations involved relative to the total circuit size. Most of the commonly used opcodes (such as `PUSH, POP, DUP` and their variants) are under 250 constraints, which is around 500× smaller than the full circuit. About a third of the opcodes are between that and 1000 constraints (about 100× smaller than the size of the full circuit). The most expensive operations involved in an ERC20 transfer are three hash calls (5× smaller, with our custom hash implementation) and left shift (25× smaller). When moving from Spice's memory checking to Nebula's, we observe a 36× reduction in the circuit size and an outsized 480× improvement in proving time. We believe that this might be due to memory pressure in the baseline. For instance, we find that the baseline's prover runs out of memory when batching more than one EVM step per Nova step, whereas Nebula's prover can batch 20 EVM steps per Nova step.

References

1. ETHEREUM: A secure decentralised generalised transaction ledger. `https://ethereum.github.io/yellowpaper/paper.pdf`
2. Nova: Recursive SNARKs without trusted setup. `https://github.com/Microsoft/Nova`
3. OpenZeppelin: Contracts for secure blockchain applications. `https://docs.openzeppelin.com/contracts/2.x/erc20`
4. Ova: A slightly better Nova. `https://hackmd.io/V4838nnlRKal9ZiTHiGYzw`
5. The Nexus zkVM. `https://github.com/nexus-xyz/nexus-zkvm` (2024)
6. Alchemy: Ethereum statistics and data — alchemy (2023), `https://www.alchemy.com/overviews/ethereum-statistics`, accessed: 2023-10-02
7. Arasu, A., Eguro, K., Kaushik, R., Kossmann, D., Meng, P., Pandey, V., Ramamurthy, R.: Concerto: A high concurrency key-value store with integrity. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD) (2017)
8. Arun, A., Setty, S., Thaler, J.: Jolt: SNARKs for virtual machines via lookups. In: Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT) (2024)
9. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: Verifying program executions succinctly and in zero knowledge. In: Proceedings of the International Cryptology Conference (CRYPTO) (Aug 2013)
10. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Scalable zero knowledge via cycles of elliptic curves. In: Proceedings of the International Cryptology Conference (CRYPTO) (2014)
11. Bernstein, D.J., Hamburg, M., Krasnova, A., Lange, T.: Elligator: Elliptic-curve points indistinguishable from uniform random strings. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2013)
12. Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. In: Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS) (1991)
13. Braun, B., Feldman, A.J., Ren, Z., Setty, S., Blumberg, A.J., Walfish, M.: Verifying computations with state. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP) (2013)
14. Bruestle, J., Gafni, P., the RISC Zero Team: RISC Zero zkVM: Scalable, transparent arguments of RISC-V integrity (2023)
15. Bünz, B., Chen, B.: Protostar: Generic efficient accumulation/folding for special sound protocols. In: Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT) (2023)
16. Buterin, V.: The dawn of hybrid layer 2 protocols. `https://vitalik.ca/general/2019/08/28/hybrid_layer_2.html` (Aug 2019)
17. Bünz, B., Chen, J.: Proofs for deep thought: Accumulation for large memories and deterministic computations. Cryptology ePrint Archive, Report 2024/325 (2024)
18. Clarke, D., Devadas, S., Dijk, M.V., Gassend, B., Edward, G., Mit, S.: Incremental multiset hash functions and their application to memory integrity checking. In: Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT) (2003)
19. Dimitriou, N., Garreta, A., Manzur, I., Vlasov, I.: Mova: Nova folding without committing to error terms. Cryptology ePrint Archive, Paper 2024/1220 (2024)
20. Eagen, L., Gabizon, A.: ProtoGalaxy: Efficient ProtoStar-style folding of multiple instances. Cryptology ePrint Archive, Paper 2023/1106 (2023)

21. Eagen, L., Gabizon, A., Sefranek, M., Towa, P., Williamson, Z.J.: Stackproofs: Private proofs of stack and contract execution using Protogalaxy. Cryptology ePrint Archive, Paper 2024/1281 (2024)
22. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. ePrint Report 2019/953 (2019)
23. Golovnev, A., Lee, J., Setty, S., Thaler, J., Wahby, R.: Brakedown: Linear-time and field-agnostic SNARKs for R1CS. In: Proceedings of the International Cryptology Conference (CRYPTO) (2023)
24. Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT). pp. 177–194 (2010)
25. Kothapalli, A., Setty, S.: SuperNova: Proving universal machine executions without universal circuits. Cryptology ePrint Archive, Report 2022/1758 (2022)
26. Kothapalli, A., Setty, S.: CycleFold: Folding-scheme-based recursive arguments over a cycle of elliptic curves. Cryptology ePrint Archive, Report 2023/1192 (2023)
27. Kothapalli, A., Setty, S.: HyperNova: Recursive arguments for customizable constraint systems (2024)
28. Kothapalli, A., Setty, S.: NeutronNova: Folding everything that reduces to zero-check. Cryptology ePrint Archive (2024)
29. Kothapalli, A., Setty, S., Tzialla, I.: Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In: Proceedings of the International Cryptology Conference (CRYPTO) (2022)
30. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers **C-28**(9) (Sep 1979)
31. Lee, J., Nikitin, K., Setty, S.: Replicated state machines without replicated execution. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P) (2020)
32. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Proceedings of the International Cryptology Conference (CRYPTO) (1988)
33. Nguyen, W., Datta, T., Chen, B., Tyagi, N., Boneh, D.: Mangrove: A scalable framework for folding-based snarks. Cryptology ePrint Archive, Report 2024/416 (2024)
34. Setty, S.: Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In: Proceedings of the International Cryptology Conference (CRYPTO) (2020)
35. Setty, S., Angel, S., Gupta, T., Lee, J.: Proving the correct execution of concurrent services in zero-knowledge. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Oct 2018)
36. Setty, S., Angel, S., Gupta, T., Lee, J.: Proving the correct execution of concurrent services in zero-knowledge (extended version). ePrint Report 2018/907 (Sep 2018)
37. Setty, S., Lee, J.: Quarks: Quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive, Report 2020/1275 (2020)
38. Setty, S., Thaler, J.: BabySpartan: Lasso-based SNARK for non-uniform computation. Cryptology ePrint Archive, Report 2023/1799 (2023)
39. Setty, S., Thaler, J., Wahby, R.: Customizable constraint systems for succinct arguments. Cryptology ePrint Archive (2023)
40. Setty, S., Thaler, J., Wahby, R.S.: Unlocking the lookup singularity with Lasso. In: Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT) (2024)

41. Valiant, P.: Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In: Theory of Cryptography Conference (TCC). pp. 552–576 (2008)
42. Wahby, R.S., Setty, S., Ren, Z., Blumberg, A.J., Walfish, M.: Efficient RAM and control flow in verifiable outsourced computation. In: Proceedings of the Network and Distributed System Security Symposium (NDSS) (2015)
43. WhiteHat, B., Gluchowski, A., HarryR, Fu, Y., Castonguay, P.: Roll_up / roll_back snark side chain ~17000 tps. `https://ethresear.ch/t/roll-up-roll-back-snark-side-chain-17000-tps/3675` (Oct 2018)
44. Zhang, Y., Genkin, D., Katz, J., Papadopoulos, D., Papamanthou, C.: vRAM: Faster verifiable RAM with program-independent preprocessing. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P) (2018)

A Deferred Definitions

A.1 Commitment Schemes

Definition 6 (Commitment Scheme). *A commitment scheme is defined by polynomial-time algorithm* $\mathsf{Gen} : \mathbb{N}^2 \to P$ *that produces public parameters given the security parameter and size parameter, a deterministic polynomial-time algorithm* $\mathsf{Commit} : P \times M \times R \to C$ *that produces a commitment in* C *given a public parameters, message, and randomness tuple such that binding holds. That is, for any* PPT *adversary* $\mathcal{A}$*, given* $\mathsf{pp} \leftarrow \mathsf{Gen}(\lambda, n)$*, and given* $((m_1, r_1), (m_2, r_2)) \leftarrow \mathcal{A}(\mathsf{pp})$ *we have that*

$$\Pr[(m_1, r_1) \neq (m_2, r_2) \wedge \mathsf{Commit}(\mathsf{pp}, m_1, r_1) = \mathsf{Commit}(\mathsf{pp}, m_2, r_2)] \approx 0.$$

The commitment scheme is deterministic if Commit *does not use its randomness.*

Definition 7 (Hiding). *The commitment scheme* $(\mathsf{Gen}, \mathsf{Commit})$ *is hiding if for any* PPT *adversary* $\mathcal{A}$*, given* $\mathsf{pp} \leftarrow \mathsf{Gen}(\lambda, n)$*,* $((m_1, r_1), (m_2, r_2)) \leftarrow \mathcal{A}(\mathsf{pp})$*, and* $C_i \leftarrow \mathsf{Commit}(\mathsf{pp}, m_i, r_i)$ *for* $i \in \{1, 2\}$ *we have that*

$$\Pr[\mathcal{A}(\mathsf{pp}, C_1) = 1] \approx \Pr[\mathcal{A}(\mathsf{pp}, C_2) = 1].$$

Definition 8 (Homomorphic). *The commitment scheme* $(\mathsf{Gen}, \mathsf{Commit})$ *is homomorphic if the message space* M*, randomness space* R*, and commitment space* C *are groups and for all* $n \in \mathbb{N}$*, and* $\mathsf{pp} \leftarrow \mathsf{Gen}(\lambda, n)$*, we have that for any* $m_1, m_2 \in M$ *and* $r_1, r_2 \in R$

$$\mathsf{Commit}(\mathsf{pp}, m_1, r_1) + \mathsf{Commit}(\mathsf{pp}, m_2, r_2) = \mathsf{Commit}(\mathsf{pp}, m_1 + m_2, r_1 + r_2).$$

Definition 9 (Succinct Commitments). *A commitment scheme* $(\mathsf{Gen}, \mathsf{Commit})$*, over message space* M *and commitment space* R*, provides succinct commitments if for all* $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda)$*, and any* $m \in M$ *and* $r \in R$*, we have that* $|\mathsf{Commit}(\mathsf{pp}, m, r)| = O_\lambda(\mathsf{polylog}(|m|))$*.*

Definition 10 (Multilinear Polynomial Commitment Scheme). *A multilinear polynomial commitment scheme over polynomial ring* $\mathbb{F}^1[X_1, \ldots, X_n]$ *is a* *commitment scheme (Gen, Commit) over message space $\mathbb{F}^1[X_1, \ldots, X_n]$, equipped with an argument of knowledge (Definition 11) for relation $\mathcal{R}_{\mathsf{polyeval}}$ defined as follows*

$$\mathcal{R}_{\mathsf{polyeval}} = \left\{ (\mathsf{pp}, (C, x, y), (P, r)) \middle| \begin{array}{l} P \in \mathbb{F}^1[X_1, \ldots, X_n], \\ P(x) = y, \\ C = \mathsf{Commit}(\mathsf{pp}, P, r) \end{array} \right\}.$$

A.2 Arguments of Knowledge

Definition 11 (Argument of Knowledge). *Consider relation $\mathcal{R}$ over public parameters, structure, instance, and witness tuples. A reduction of knowledge for $\mathcal{R}$ is defined by PPT algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ and deterministic algorithm $\mathcal{K}$, denoting the generator, the prover, the verifier and the encoder respectively with the following interface.*

- $\mathcal{G}(\lambda, N) \to \mathsf{pp}$*: Takes as input security parameter λ and size parameters N. Outputs public parameters* pp*.*
- $\mathcal{K}(\mathsf{pp}, \mathsf{s}) \to (\mathsf{pk}, \mathsf{vk})$*: Takes as input public parameters* pp *and structure* s*. Outputs prover key* pk *and verifier key* vk
- $\mathcal{P}(\mathsf{pk}, \mathsf{u}, \mathsf{w}) \to \perp$*: Takes as input public parameters* pp*, and an instance-witness pair* (u, w)*. Interactively proves that* $(\mathsf{pp}, \mathsf{s}, \mathsf{u}, \mathsf{w}) \in \mathcal{R}$*.*
- $\mathcal{V}(\mathsf{pk}, \mathsf{u}) \to \{0, 1\}$*: Takes as input public parameters* pp*, and an instance* u*. Interactively checks* u*.*

Let $\langle \mathcal{P}, \mathcal{V} \rangle$ denote the interaction between $\mathcal{P}$ and $\mathcal{V}$. We treat $\langle \mathcal{P}, \mathcal{V} \rangle$ as a function that takes as input $((\mathsf{pk}, \mathsf{vk}), \mathsf{u}, \mathsf{w})$ *and runs the interaction on prover input* $(\mathsf{pk}, \mathsf{u}, \mathsf{w})$ *and verifier input* $(\mathsf{pp}, \mathsf{u})$*. At the end of the interaction, $\langle \mathcal{P}, \mathcal{V} \rangle$ outputs the verifier's decision. An argument of knowledge $(\mathcal{G}, \mathcal{K}, \mathcal{P}, \mathcal{V})$ satisfies the following conditions.*

(i) Completeness: For any PPT *adversary $\mathcal{A}$, given* $\mathsf{pp} \leftarrow \mathcal{G}(\lambda, N)$*,* $(\mathsf{s}, \mathsf{u}, \mathsf{w}) \leftarrow \mathcal{A}(\mathsf{pp})$ *such that* $(\mathsf{pp}, \mathsf{s}, \mathsf{u}, \mathsf{w}) \in \mathcal{R}$ *and* $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, \mathsf{s})$ *we have that*

$$\langle \mathcal{P}, \mathcal{V} \rangle((\mathsf{pk}, \mathsf{vk}), \mathsf{u}, \mathsf{w}) = 1$$

(ii) Knowledge Soundness: For any expected polynomial-time adversaries $\mathcal{A}$ and $\mathcal{P}^*$, there exists an expected polynomial-time extractor $\mathcal{E}$ such that given $\mathsf{pp} \leftarrow \mathcal{G}(\lambda, N)$*,* $(\mathsf{s}, \mathsf{u}, \mathsf{st}) \leftarrow \mathcal{A}(\mathsf{pp})$*, and* $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, \mathsf{s})$*, we have that*

$$\Pr[(\mathsf{pp}, \mathsf{s}, \mathsf{u}, \mathcal{E}(\mathsf{pp}, \mathsf{u}, \mathsf{st})) \in \mathcal{R}_1] \approx \Pr[\langle \mathcal{P}^*, \mathcal{V} \rangle((\mathsf{pk}, \mathsf{vk}), \mathsf{u}, \mathsf{st}) = 1].$$

Definition 12 (Succinctness). *An argument of knowledge is succinct if the communication complexity and the verifier time complexity is at most polylogarithmic in the size of the structure and witness.*

Definition 13 (Non-Interactivity). *An argument of knowledge is non-interactive if the interaction consists of a single message from the prover to the verifier. In this case, we denote this single message as the output of the prover, and as an input to the verifier.*

Definition 14 (Zero-knowledge). *An argument of knowledge* $(\mathcal{G}, \mathcal{K}, \mathcal{P}, \mathcal{V})$ *for relation* $\mathcal{R}$ *satisfies zero-knowledge if for any PPT adversary* $\mathcal{V}^*$ *there exists an EPT simulator* $\mathcal{S}$ *such that for any PPT adversary* $\mathcal{A}$ *for* $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda, N)$, $(\mathsf{s}, (u, w), \mathsf{st}_1) \leftarrow \mathcal{A}(\mathsf{pp})$ *such that* $(\mathsf{pp}, \mathsf{s}, u, w) \in \mathcal{R}$, *and* $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, \mathsf{s})$

$$\left\{ \mathsf{st}_2 \,\middle|\, \mathsf{st}_2 \leftarrow \langle \mathcal{P}, \mathcal{V}^*(\mathsf{st}_1) \rangle ((\mathsf{pk}, \mathsf{vk}), u, w) \right\} \cong \left\{ \mathsf{st}_2 \,\middle|\, \mathsf{st}_2 \leftarrow \mathcal{S}(\mathsf{pp}, \mathsf{s}, u, \mathsf{st}_1) \right\}$$

where st_2 *denotes the output of* $\mathcal{V}^*$ *after interaction. An argument of knowledge satisfies honest-verifier zero-knowledge (HVZK) if it satisifes zero-knowledge under an honest (but curious) verifier that behaves according to the interactive protocol but produces arbitrary output on the side.*

A.3 Customizable constraint systems (CCS)

CCS simultaneously generalizes R1CS, Plonkish, and AIR without overheads. We first provide an arithmetized variant of the original formulation. The definitions below are characterized by a finite field $\mathbb{F}$, but we leave this implicit.

Definition 15 (CCS [39]). *Consider size bounds* $m, n, N, \ell, t, q, d \in \mathbb{N}$ *where* $n > \ell$. *Let* $s = \log m$ *and* $s' = \log n$. *We define the customizable constraint system (CCS) relation,* $\mathcal{R}_{\mathsf{CCS}}$, *over structure, instance, witness tuples as follows.*

An $\mathcal{R}_{\mathsf{CCS}}$ *structure* s *consists of*

- *a sequence of sparse multilinear polynomials in* $s + s'$ *variables* $\widetilde{M}_1, \ldots, \widetilde{M}_t$ *such that they evaluate to a non-zero value in at most* $N = \Omega(m)$ *locations over the Boolean hypercube* $\{0,1\}^s \times \{0,1\}^{s'}$;
- *a sequence of* q *multisets* $[S_1, \ldots, S_q]$, *where an element in each multiset is from the domain* $\{1, \ldots, t\}$ *and the cardinality of each multiset is at most* d.
- *a sequence of* q *constants* $[c_1, \ldots, c_q]$, *where each constant is from* $\mathbb{F}$.

An $\mathcal{R}_{\mathsf{CCS}}$ *instance consists of public input and output vector* $\mathsf{x} \in \mathbb{F}^\ell$. *An* $\mathcal{R}_{\mathsf{CCS}}$ *witness consists of a multilinear polynomial* $\widetilde{w}$ *in* $s' - 1$ *variables. We have that* $(\mathsf{s}, \mathsf{x}, \widetilde{w}) \in \mathcal{R}_{\mathsf{CCS}}$ *if and only if for all* $x \in \{0,1\}^s$,

$$\sum_{i=1}^{q} c_i \cdot \left(\prod\nolimits_{j \in S_i} \left(\sum_{y \in \{0,1\}^{\log m}} \widetilde{M}_j(x, y) \cdot \widetilde{z}(y) \right) \right) = 0,$$

where $\widetilde{z}$ *is an* s'*-variate multilinear polynomial such that* $\widetilde{z}(x) = \widetilde{(w, 1, \mathsf{x})}(x)$ *for all* $x \in \{0,1\}^{s'}$.

A.4 Non-uniform IVC

Definition 16 (Non-uniform IVC). *A non-uniform incrementally verifiable computation (NIVC) scheme is defined by PPT algorithms* $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ *and a deterministic* $\mathcal{K}$ *denoting the generator, the prover, the verifier, and the encoder respectively, with the following interface:*

- $\mathcal{G}(1^\lambda, N) \to \mathsf{pp}$*: on input security parameter* λ *and size bounds* N*, samples public parameters* pp.
- $\mathcal{K}(\mathsf{pp}, ((F_1, \ldots, F_\ell), \varphi)) \to (\mathsf{pk}, \mathsf{vk})$*: on input public parameters* pp*, a control function* φ*, and functions* $F_1, \ldots, F_\ell$ *deterministically produces a prover key* pk *and a verifier key* vk.
- $\mathcal{P}(\mathsf{pk}, (i, z_0, z_i), \omega_i, \Pi_i) \to \Pi_{i+1}$*: on input a prover key* pk*, a counter* i*, initial input* z_0*, claimed output after* i *applications* z_i*, a non-deterministic advice* ω_i*, and an NIVC proof* Π_i *attesting to* z_i*, produces a new proof* Π_{i+1} *attesting to* $z_{i+1} = F_{\varphi(z_i,\omega_i)}(z_i, \omega_i)$.
- $\mathcal{V}(\mathsf{vk}, (i, z_0, z_i), \Pi_i) \to \{0, 1\}$*: on input a verifier key* vk*, a counter* i*, an initial input* z_0*, a claimed output after* i *applications* z_i*, and an NIVC proof* Π_i *attesting to* z_i*, outputs* 1 *if* Π_i *is accepting,* 0 *otherwise.*

An NIVC scheme $(\mathcal{G}, \mathcal{K}, \mathcal{P}, \mathcal{V})$ *satisfies following requirements.*

(i) Completeness: For any PPT adversary $\mathcal{A}$ *we have that*

$$\Pr\left[b = 1 \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda, N), \\ (((F_1, \ldots, F_\ell), \varphi), (i, z_0, z_i), (\omega_i, \Pi_i)) \leftarrow \mathcal{A}(\mathsf{pp}), \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, ((F_1, \ldots, F_\ell), \varphi)), \\ \mathcal{V}(\mathsf{vk}, (i, z_0, z_i), \Pi_i) = 1, \\ z_{i+1} \leftarrow F_{\varphi(z_i,\omega_i)}(z_i, \omega_i), \\ \Pi_{i+1} \leftarrow \mathcal{P}(\mathsf{pk}, (i, z_0, z_i), \omega_i, \Pi_i), \\ b \leftarrow \mathcal{V}(\mathsf{vk}, (i+1, z_0, z_{i+1}), \Pi_{i+1}) \end{array} \right] = 1$$

where $\ell \geq 1$ *and* φ *produces an element in* $\mathbb{Z}^*_{\ell+1}$*. Moreover,* φ *and each* F_j *for* $j \in \{1, \ldots, \ell\}$ *are a polynomial-time computable function represented as arithmetic circuits.*

(ii) Knowledge Soundness: Consider constant $n \in \mathbb{N}$*. For all expected polynomial-time adversaries* $\mathcal{P}^*$ *there exists an expected polynomial-time extractor* $\mathcal{E}$ *such that*

$$\Pr_{\mathsf{r}}\left[\begin{array}{l} z_n = z \text{ where} \\ z_{i+1} \leftarrow F_{\varphi(z_i,\omega_i)}(z_i, \omega_i) \\ \forall i \in \{0, \ldots, n-1\} \end{array} \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda, N), \\ (((F_1, \ldots, F_\ell), \varphi), (z_0, z), \Pi) \leftarrow \mathcal{P}^*(\mathsf{pp}, \mathsf{r}), \\ (\omega_0, \ldots, \omega_{n-1}) \leftarrow \mathcal{E}(\mathsf{pp}, \mathsf{r}) \end{array} \right] \approx$$

$$\Pr_{\mathsf{r}}\left[\mathcal{V}(\mathsf{vk}, (n, z_0, z), \Pi) = 1 \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda, N), \\ (((F_1, \ldots, F_\ell), \varphi), (z_0, z), \Pi) \leftarrow \mathcal{P}^*(\mathsf{pp}, \mathsf{r}), \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, ((F_1, \ldots, F_\ell), \varphi)) \end{array} \right]$$

where r *denotes an arbitrarily long random tape.*

(iii) Succinctness: The NIVC proof size is independent of the iteration count.

(iv) Efficiency: The prover's time complexity at any step i is linear in the size of the function applied at step i and the total number of functions ℓ.

B Deferred Proofs

B.1 Proof of Theorem 1 (Nebula)

Lemma 4 (Completeness). *Construction 2 is a CC-NIVC scheme that satisfies completeness.*

Proof. We adapt the proof of completeness of Hypernova's compiler [27, Lemma 16], highlighting the changes to support commitment-carrying capability in $\boxed{\text{boxes}}$. Consider an arbitrary PPT adversary $\mathcal{A}$. Suppose $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda, N)$. Suppose, on input pp, $\mathcal{A}$ produces polynomial-time functions $(\varphi, (F_1, \ldots, \mathbb{F}_\ell))$, instance $(i, z_0, z_i, \boxed{C_i})$, private input ω_i, and a CC-NIVC proof Π_i. Suppose that for

$$(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, (\varphi, (F_1, \ldots, F_\ell)))$$

we have that

$$\mathcal{V}(\mathsf{vk}, (i, z_0, z_i, \boxed{C_i}), \Pi_i) = 1.$$

Then, for $\mathsf{pc}_{i+1} \in [\ell] \leftarrow \varphi(z_i, \omega_i)$, given

$$z_{i+1} \leftarrow F_{\mathsf{pc}_{i+1}}(z_i, \omega_i),$$
$$\boxed{C_{i+1} \leftarrow \mathsf{IC.Commit}(\mathsf{pp}_{\mathsf{IC}}, C_i, \omega_i)}$$

and

$$\Pi_{i+1} \leftarrow \mathcal{P}(\mathsf{pk}, (i, z_0, z_i, \boxed{C_i}), \omega_i, \Pi_i)$$

we must show that

$$\mathcal{V}(\mathsf{vk}, i+1, z_0, z_{i+1}, \boxed{C_{i+1}}, \Pi_{i+1}) = 1$$

with probability 1. We show this by considering the case when $i = 0$ and when $i \geq 1$.

Indeed, suppose $i = 0$. By the base case of $\mathcal{P}$ and F'_{pc_1}, we have

$$\Pi_1 = (((\mathsf{u}_\perp, \ldots, \mathsf{u}_\perp), (\mathsf{w}_\perp, \ldots, \mathsf{w}_\perp)), (\mathsf{u}_1, \mathsf{w}_1), \mathsf{pc}_1, \boxed{C_0 = \perp})$$

for some $(\mathsf{u}_1, \mathsf{w}_1)$. By definition, the instance-witness pair $(\mathsf{u}_\perp, \mathsf{w}_\perp)$ is satisfying. Moreover, by construction, $(\mathsf{u}_1, \mathsf{w}_1)$ must also be satisfying. Additionally, by the construction of F'_{pc_1}, we have

$$\mathsf{u}'_1 = \mathsf{enc}_{\mathsf{inst}}(\mathsf{hash}(\mathsf{vk}, 1, z_0, F_{\mathsf{pc}_1}(z_0, w_0), \mathsf{u}_\perp, \mathsf{pc}_1), \boxed{C_0 = \perp}).$$

where u'_1 is the portion of u_1 that excludes the commitment to the w_1. Therefore, we have

$$\mathcal{V}(\mathsf{pp}, 1, z_0, z_1, \boxed{C_1}, \Pi_1) = 1.$$

Suppose instead that $i \geq 1$. Let Π_i be parsed as $((\mathsf{U}_i, \mathsf{W}_i), (\mathsf{u}_i, \mathsf{w}_i), \mathsf{pc}_i, \boxed{C_{i-1}})$ and let Π_{i+1} be parsed as $((\mathsf{U}_{i+1}, \mathsf{W}_{i+1}), (\mathsf{u}_{i+1}, \mathsf{w}_{i+1}), \mathsf{pc}_{i+1}, \boxed{C_i})$. By the construction of $\mathcal{P}$, we have that

$$(\mathsf{U}_{i+1}[\mathsf{pc}_i], \mathsf{W}_{i+1}[\mathsf{pc}_i], \pi) = \mathsf{NIFS.P}(\mathsf{pk}[\mathsf{pc}_i], (\mathsf{U}_i[\mathsf{pc}_i], \mathsf{W}_i[\mathsf{pc}_i]), (\mathsf{u}_i, \mathsf{w}_i)).$$

Thus, because Π_i is satisfying, we have that $(\mathsf{u}_i, \mathsf{w}_i)$ and $(\mathsf{U}_i[\mathsf{pc}_i], \mathsf{W}_i[\mathsf{pc}_i])$ are satisfying instance-witness pairs (with respect to compatible structures). Then, by the completeness of the underlying folding scheme, we have that $(\mathsf{U}_{i+1}[\mathsf{pc}_i], \mathsf{W}_{i+1}[\mathsf{pc}_i])$ is a satisfying instance-witness pair. Therefore, because $(\mathsf{U}_{i+1}, \mathsf{W}_{i+1})$ copies the remaining elements from $(\mathsf{U}_i, \mathsf{W}_i)$, we have that $(\mathsf{U}_{i+1}, \mathsf{W}_{i+1})$ contains satisfying instance-witness pairs. Additionally, by the premise, we have that $\mathsf{u}'_i = \mathsf{enc}_{\mathsf{inst}}(\mathsf{hash}(\mathsf{vk}, i, z_0, z_i, \mathsf{U}_i, \mathsf{pc}_i), \boxed{C_{i-1}})$ where u'_i represents the portion of u_i that excludes the commitment to the witness. Therefore, $\mathcal{P}$ can construct a satisfying instance-witness pair $(\mathsf{u}_{i+1}, \mathsf{w}_{i+1})$ that represents the correct execution of $F'_{\mathsf{pc}_{i+1}}$ on input $(\mathsf{vk}_{\mathsf{fs}}, \mathsf{U}, \mathsf{u}, \mathsf{pc}_i, (i, z_0, z_i, \boxed{C_{i-1}}), \omega_i, \pi)$. By construction, this particular input implies that

$$\mathsf{u}'_{i+1} = \mathsf{enc}_{\mathsf{inst}}(\mathsf{hash}(\mathsf{vk}, i+1, z_0, z_{i+1}, \mathsf{U}_{i+1}, \mathsf{pc}_{i+1}, \boxed{C_i})) \tag{1}$$

by the correctness of the underlying folding scheme (again u'_{i+1} represents the portion of u_{i+1} that excludes the commitment to the witness). Moreover, because $\mathsf{pc}_{i+1} = \varphi(z_i, \omega_i)$, by construction, we have that $1 \leq \mathsf{pc}_{i+1} \leq \ell$. Thus, by Equation (1) we have

$$\mathcal{V}(\mathsf{vk}, i+1, z_0, z_{i+1}, \boxed{C_{i+1}}, \Pi_{i+1}) = 1.$$

□

Lemma 5 (Knowledge Soundness). *Construction 2 is a CC-NIVC scheme that satisfies knowledge soundness.*

Proof. We adapt the proof of knowledge soundness of Hypernova's compiler [27, Lemma 17], highlighting the changes to support commitment-carrying capability in $\boxed{\text{boxes}}$.

Let n be a global constant. Consider a deterministic expected polynomial-time adversary $\mathcal{P}^*$. Let $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda, N)$. Suppose on input pp and randomness r, $\mathcal{P}^*$ outputs deterministic polynomial-time function φ, ℓ polynomial-time functions

$(F_1, \ldots, F_\ell)$, instance (z_0, z), $\boxed{\text{a carried commitment } C}$, and a CC-NIVC proof Π. Let $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, (\varphi, (F_1, \ldots, F_\ell)))$. Suppose that

$$\mathcal{V}(\mathsf{vk}, (n, z_0, z, \boxed{C}), \Pi) = 1$$

with probability ϵ. We must construct an expected polynomial-time extractor $\mathcal{E}$ that, with input $(\mathsf{pp}, \mathsf{r})$, outputs $(\omega_0, \ldots, \omega_{n-1})$ such that by computing

$$z_{i+1} \leftarrow F_{\varphi(z_i, \omega_i)}(z_i, \omega_i)$$
$$C_{i+1} \leftarrow \mathsf{IC.Commit}(\mathsf{pp}_{\mathsf{IC}}, C_i, \omega_i)$$

with $C_0 = \perp$, we have that $z_n = z$ and $C_n = C$ and with probability $\epsilon - \mathsf{negl}(\lambda)$. We show inductively that $\mathcal{E}$ can construct an expected polynomial-time extractor $\mathcal{E}_i(\mathsf{pp})$ that outputs $((z_i, \ldots, z_{n-1}), \boxed{(C_i, \ldots, C_{n-1})}, (\omega_i, \ldots, \omega_{n-1}), \Pi_i)$ such that for all $j \in \{i+1, \ldots, n\}$,

$$z_j = F_{\varphi(z_{j-1}, \omega_{j-1})}(z_{j-1}, \omega_{j-1})$$
$$\boxed{C_j = \mathsf{IC.Commit}(\mathsf{pp}_{\mathsf{IC}}, C_{j-1}, \omega_i)}$$

with $C_0 = \perp$ and

$$\mathcal{V}(\mathsf{vk}, i, z_0, z_i, \boxed{C_i}, \Pi_i) = 1 \tag{2}$$

for $z_n = z$ and $\boxed{C_n = C}$ with probability $\epsilon - \mathsf{negl}(\lambda)$. Then, because in the base case when $i = 0$, $\mathcal{V}$ checks that $z_0 = z_i$, the values $(\omega_0, \ldots, \omega_{n-1})$ retrieved by $\mathcal{E}_0(\mathsf{pp})$ are such that computing $z_{i+1} = F(z_i, \omega_i)$ and computing $\boxed{C_{i+1} = \mathsf{IC.Commit}(\mathsf{pp}_{\mathsf{IC}}, C_i, \omega_i)}$ (with $C_0 = \perp$) for all $i \geq 1$ gives $z_n = z$ and $C_n = C$.

At a high level, to construct an extractor $\mathcal{E}_{i-1}$, we first assume the existence of $\mathcal{E}_i$ that satisfies the inductive hypothesis. We then use $\mathcal{E}_i(\mathsf{pp})$ to construct an adversary for the non-interactive folding scheme (which we denote as $\widetilde{\mathcal{P}}_{i-1}$). This in turn guarantees an extractor for the non-interactive folding scheme, which we denote as $\widetilde{\mathcal{E}}_{i-1}$. We then use $\widetilde{\mathcal{E}}_{i-1}$ to construct $\mathcal{E}_{i-1}$ that satisfies the inductive hypothesis.

In the base case, for $i = n$, let $\mathcal{E}_n(\mathsf{pp}, \mathsf{r})$ output $(\perp, \perp, \Pi_n, \boxed{C_n})$ where Π_n, C_n is the output of $\mathcal{P}^*(\mathsf{pp}, \mathsf{r})$. By the premise, $\mathcal{E}_n$ succeeds with probability ϵ in expected polynomial-time.

For $i \geq 1$, suppose $\mathcal{E}$ can construct an expected polynomial-time extractor $\mathcal{E}_i$ that outputs $((z_i, \ldots, z_{n-1}), \boxed{(C_i, \ldots, C_{n-1})}, (\omega_i, \ldots, \omega_{n-1}))$, and Π_i that satisfies the inductive hypothesis. To construct an extractor $\mathcal{E}_{i-1}$, $\mathcal{E}$ first constructs an adversary $\widetilde{\mathcal{P}}_{i-1}$ for the non-interactive folding scheme as follows:

$\underline{\widetilde{\mathcal{P}}_{i-1}(\mathsf{pp}, \mathsf{r})}$:

1. Let $((z_i, \ldots, z_{n-1}), \boxed{(C_i, \ldots, C_{n-1})}, (\omega_i, \ldots, \omega_{n-1}), \Pi_i) \leftarrow \mathcal{E}_i(\mathsf{pp}, \mathsf{r})$.
2. Parse Π_i as $((\mathsf{U}_i, \mathsf{W}_i), (\mathsf{u}_i, \mathsf{w}_i), \mathsf{pc}_i, \boxed{C_{i-1}})$.
3. Compute compatible structures $(\mathsf{s}_{1,\mathsf{pc}_i}, \mathsf{s}_{2,\mathsf{pc}_i}) \leftarrow \mathsf{enc}_{\mathsf{str}}(F'_{\mathsf{pc}_i})$.
4. Parse non-deterministic inputs $(\mathsf{U}_{i-1}, \mathsf{u}_{i-1}, \pi_{i-1}, \mathsf{pc}_{i-1})$ to F'_{pc_i} from $\mathsf{enc}^{-1}(\mathsf{s}_{2,\mathsf{pc}_i}, \mathsf{u}_i, \mathsf{w}_i)$.
5. Output structures $(\mathsf{s}_{1,\mathsf{pc}_{i-1}}, \mathsf{s}_{2,\mathsf{pc}_{i-1}})$, unfolded instances $(\mathsf{U}_{i-1}[\mathsf{pc}_{i-1}], \mathsf{u}_{i-1})$, folded instance-witness pair $(\mathsf{U}_i[\mathsf{pc}_{i-1}], \mathsf{W}_i[\mathsf{pc}_{i-1}])$, and folding proof π_{i-1}.

We now analyze the success probability of $\widetilde{\mathcal{P}}_{i-1}$. By the inductive hypothesis, we have that $\mathcal{V}(\mathsf{vk}, i, z_0, z_i, \boxed{C_i}, \Pi_i) = 1$, where $\Pi_i \leftarrow \mathcal{E}_i(\mathsf{pp}, \mathsf{r})$ with probability $\epsilon - \mathsf{negl}(\lambda)$. Therefore, by the the verifier's checks we have that $(\mathsf{u}_i, \mathsf{w}_i)$ is satisfying, $(\mathsf{U}_i, \mathsf{W}_i)$ consists of satisfying instance-witness pairs, and that

$$\mathsf{u}'_i = \mathsf{enc}_{\mathsf{inst}}(\mathsf{hash}(\mathsf{vk}, i, z_0, z_i, \mathsf{U}_i, \mathsf{pc}_i, \boxed{C_{i-1}}))$$

where u'_i represents the portion of u_i that excludes the commitment to the witness. Then, by the construction of F'_{pc_i} and the binding property of the hash function, we have that $1 \leq \mathsf{pc}_{i-1} \leq \ell$ and

$$\mathsf{U}_i[\mathsf{pc}_{i-1}] = \mathsf{NIFS}.\mathcal{V}(\mathsf{vk}, \mathsf{U}_{i-1}[\mathsf{pc}_{i-1}], \mathsf{u}_{i-1}, \pi_{i-1})$$

with probability $\epsilon - \mathsf{negl}(\lambda)$. Thus, $\widetilde{\mathcal{P}}_{i-1}$ succeeds in producing an accepting folded instance-witness pair $(\mathsf{U}_i[\mathsf{pc}_{i-1}], \mathsf{W}_i[\mathsf{pc}_{i-1}])$, for instances $\mathsf{U}_{i-1}[\mathsf{pc}_{i-1}]$ and u_{i-1}, with probability $\epsilon - \mathsf{negl}(\lambda)$ in expected polynomial-time.

Then, by the knowledge soundness of the underlying non-interactive multi-folding scheme there exists an extractor $\widetilde{\mathcal{E}}_{i-1}$ that outputs $(\mathsf{W}_{i-1}[\mathsf{pc}_{i-1}], \mathsf{w}_{i-1})$ such that $(\mathsf{U}_{i-1}[\mathsf{pc}_{i-1}], \mathsf{W}_{i-1}[\mathsf{pc}_{i-1}])$ and $(\mathsf{u}_{i-1}, \mathsf{w}_{i-1})$ are satisfying with respect to structures $\mathsf{s}_{1,\mathsf{pc}_{i-1}}$ and $\mathsf{s}_{2,\mathsf{pc}_{i-1}}$ respectively with probability $\epsilon - \mathsf{negl}(\lambda)$ in expected polynomial-time.

Given an expected polynomial-time $\widetilde{\mathcal{P}}_{i-1}$ and an expected polynomial-time $\widetilde{\mathcal{E}}_{i-1}$, $\mathcal{E}$ constructs an expected polynomial time $\mathcal{E}_{i-1}$ as follows

$\underline{\mathcal{E}_{i-1}(\mathsf{pp}, \mathsf{r})}$:

1. Run $\widetilde{\mathcal{P}}_{i-1}(\mathsf{pp}, \mathsf{r})$ to retrieve unfolded instances $(\mathsf{u}'_{i-1}, \mathsf{u}_{i-1})$ and parse

$$((z_i, \ldots, z_{n-1}), \boxed{(C_i, \ldots, C_{n-1})}, (\omega_i, \ldots, \omega_{n-1}), \Pi_i)$$

   from its internal state.
2. Parse Π_i as $((\mathsf{U}_i, \mathsf{W}_i), (\mathsf{u}_i, \mathsf{w}_i), \mathsf{pc}_i, \boxed{C_{i-1}})$.
3. Compute $(\mathsf{s}_{1,\mathsf{pc}_i}, \mathsf{s}_{2,\mathsf{pc}_i}) \leftarrow \mathsf{enc}_{\mathsf{str}}(F'_{\mathsf{pc}_i})$
4. Parse private inputs , z_{i-1}, $\boxed{C_{i-2}}$, ω_{i-1}, and pc_{i-1} to F'_{pc_i} from $\mathsf{enc}^{-1}(\mathsf{s}_2, \mathsf{u}_i, \mathsf{w}_i)$.

5. Let $(\mathsf{w}'_{i-1}, \mathsf{w}_{i-1}) \leftarrow \widetilde{\mathcal{E}}_{i-1}(\mathsf{pp})$.
6. Set $(\mathsf{U}_{i-1}, \mathsf{W}_{i-1}) \leftarrow (\mathsf{U}_i, \mathsf{W}_i)$ and update
$$(\mathsf{U}_{i-1}[\mathsf{pc}_{i-1}], \mathsf{W}_{i-1}[\mathsf{pc}_{i-1}]) \leftarrow (\mathsf{u}'_{i-1}, \mathsf{w}'_{i-1})$$
7. Let $\Pi_{i-1} \leftarrow ((\mathsf{U}_{i-1}, \mathsf{W}_{i-1}), (\mathsf{u}_{i-1}, \mathsf{w}_{i-1}), \mathsf{pc}_{i-1}, \boxed{C_{i-2}})$.
8. Output $((z_{i-1}, \ldots, z_{n-1}), \boxed{(C_{i-1}, \ldots, C_{n-1})}, (\omega_{i-1}, \ldots, \omega_{n-1}), \Pi_{i-1})$.

We first reason that the output $(z_{i-1}, \ldots, z_{n-1})$, $\boxed{(C_{i-1}, \ldots, C_{n-1})}$ and $(\omega_{i-1}, \ldots, \omega_{n-1})$ are valid. By the inductive hypothesis, we already have that for all $j \in \{i+1, \ldots, n\}$,

$$z_j = F_{\mathsf{pc}_j}(z_{j-1}, \omega_{j-1}),$$
$$\boxed{C_j = \mathsf{IC.Commit}(\mathsf{pp}_{\mathsf{IC}}, C_{j-1}, \omega_{j-1})}$$

and that $\mathcal{V}(\mathsf{vk}, i, z_0, z_i, \boxed{C_i}, \Pi_i) = 1$ with probability $\epsilon - \mathsf{negl}(\lambda)$. Because $\mathcal{V}$ additionally checks that

$$\mathsf{u}'_i = \mathsf{enc}_{\mathsf{inst}}(\mathsf{hash}(\mathsf{vk}, i, z_0, z_i, \mathsf{U}_i, \mathsf{pc}_i, \boxed{C_{i-1}})), \tag{3}$$

where u'_i represents the portion of u_i that excludes the commitment to the witness, by the construction of F'_{pc_i} and the binding property of the hash function, we have

$$F_{\mathsf{pc}_i}(z_{i-1}, \omega_{i-1}) = z_i$$

with probability $\epsilon - \mathsf{negl}(\lambda)$.

Similarly, because $\mathcal{V}$ checks that

$$C_i = \mathsf{IC.Commit}(\mathsf{pp}_{\mathsf{IC}}, C_{i-1}, C_{\omega_i}),$$

where C_{ω_i} is parsed from the commitment to the witness in u_i, by the construction of F' and the binding property of the incremental commitment scheme, we have

$$C_i = \mathsf{IC.Commit}(\mathsf{pp}_{\mathsf{IC}}, C_{i-1}, \omega_i))$$

Next, we argue that Π_{i-1} is valid. Because $(\mathsf{u}_i, \mathsf{w}_i)$ satisfies F', and $(\mathsf{U}_{i-1}, \mathsf{u}_{i-1})$ were retrieved from w_i, by the binding property of the hash function, and by Equation (3), we have that

$$\mathsf{u}'_{i-1} = \mathsf{enc}_{\mathsf{inst}}(\mathsf{hash}(\mathsf{vk}, i-1, z_0, z_{i-1}, \mathsf{U}_{i-1}, \mathsf{pc}_{i-1}), \boxed{C_{i-2}})$$

where u'_{i-1} represents the portion of u_{i-1} that excludes the commitment the witness. Additionally, in the case where $i = 1$, by the base case check of $F'_{\varphi(z_0,\omega_0)}$, we have that $z_{i-1} = z_0$ and $\boxed{C_0 = \bot}$. Because $\widetilde{\mathcal{E}}_{i-1}$ succeeds with probability $\epsilon - \mathsf{negl}(\lambda)$, and the remainder of the elements of $(\mathsf{U}_{i-1}, \mathsf{W}_{i-1})$ are directly copied from $(\mathsf{U}_i, \mathsf{W}_i)$ we have that all the elements of $(\mathsf{U}_{i-1}, \mathsf{W}_{i-1})$ are satisfying. Moreover, by construction of F'_{pc_i} we have that $1 \leq \mathsf{pc}_{i-1} \leq \ell$. Thus, we have that

$$\mathcal{V}(\mathsf{vk}, i-1, z_0, z_{i-1}, \boxed{C_{i-1}}, \Pi_{i-1}) = 1$$

with probability $\epsilon - \mathsf{negl}(\lambda)$. □

B.2 Proof of Lemma 3

Proof. Let $z = (1, x, w^\star)$ denote the variables vector satisfying circuit $C^\star$. By the switchboard circuit construction, we know that exactly one switch variable of z must be 1. Let k be the subcircuit with this active switch: that is, $z[s_k] = 1$. Let the variables corresponding to $\mathcal{C}_k$ be $(1, x_k, \omega)$. By the input consistency constraints, we have that $x_k = x$. Thus, ω is the desired witness for $\mathcal{C}_k$.

The switches of all subcircuits $j \neq k$ is set to 0. By the input consistency constraints, this in turns implies that the input variables corresponding to it must be set to 0. Thus, the witness variables can be set to 0 as well, leading all constraints in the subcircuit to be trivially satisfied.

Sparsity: the number of non-zero elements in $w^\star$ is at most $1 + |x| + |\omega|$ and they all correspond to the one active subcircuit. □

C EVM implementation

Memory in the EVM. There are five bodies of memory in an EVM program: (1) the program code, (2) stack, (3) RAM, (4) calldata and (5) storage. Each step of the EVM reads the current operation from the program code. Then, most instructions pop operands from the stack, perform the operation, and push results back into the stack. Memory and storage accesses are restricted to a few dedicated instructions each. Program code, RAM, and calldata can be represented as standard byte-addressable linear arrays. While calldata and RAM do not have a fixed upper bound on size, using them costs "gas" (Ethereum's way of measuring the computational work done by each operation), which limits the total computational work done by an Ethereum block. For the purposes of the ERC-20 program, we assume that the calldata and memory can be upto 2^{16} in length, which are reasonable estimates given the nature of the program and the gas costs involved. The stack can reach a maximum height of 1024 elements, each of 256-bit length. As these values are bigger than the field elements used in our implementation, they are represented with two field elements, slightly increasing the cost of the multiset hashes performed for memory-checking (see Section 4.4).

Storage is a persistent, long-term, key-value store with 256-bit keys and values. (Persistent here refers to there being only one storage associated with a given smart contract that different transactions can perform memory operations on.) Reading the value of an uninitialized key returns 0. This peculiarity, along with its addresses being sampled from a 2^{256}-sized address space, sets storage apart from the other memory types in the EVM. Fortunately, it can still be conveniently captured by the offline memory-checking techniques we employ. In fact, the original Spice protocol [35] is described for general key-value stores, of which an addressable array of memory cells is a specific case. To support this, when accessing an unitialized key k, the prover performs a special "insert" operation, adding $(k, 0)$ to the key-value store. This is effectively a write without a preceding read. As proven in Spice, the offline memory checking protocol used in Nebula provides security against "double insertions" where a prover attempts to insert a new value to an already-initialized key. Also, instead of a memory scan over the entire address space (which is prohibitively large), it suffices for the prover to only scan the initialized keys.

Table 2 shows the amortized number of constraints (that is, including the linear scans of memory performed in Spice and Nebula) required to encode a write operation (except for program code, which is read-only). For the sizes of the data structures seen in the EVM, Nebula's cost per transaction is much lower than that obtained by using Merkle trees or Spice when the number of transactions exceeds 2^{15} per data structure. This number is a easy target for most large EVM programs (especially those exceeding that many steps, as each step reads the next operation from the program code). However, for storage, the scan cost depends on the total number of keys that are initialized for the contract. As this is persistent, this could be anywhere from a few thousands to tens of millions.

	Memory Body / Total Size			
Method	Code/2^{16}	Stack/2^{10}	RAM/2^{16}	Storage/2^{20}
Merkle Tree	3750	5000	8000	-
Spice*	1100	1100	1100	1100
Nebula	4	4	4	4
+Range Checks	rc_m	rc_m	rc_m	rc_m
+Scans	$2^{19}/m$	$2^{13}/m$	$2^{19}/m$	$2^{23}/m$

Table 2: Microbenchmarks comparing different memory-checking techniques applied to different memories in the EVM. The cost depicted is that of a write, except for program code which is read-only. m denotes the number of memory operations performed. The sizes of program code, stack and RAM are $2^{16}, 2^{10}, 2^{16}$, respectively. For storage, we assume that about 2^{20} keys are initialized. We denote the cost of a range check involving values of up to m by rc_m. Each of the two scans performed in in Nebula for a body of memory M costs under $4 \cdot |M|$ constraints. (*For Spice, range check and scan costs are ignored for brevity.)

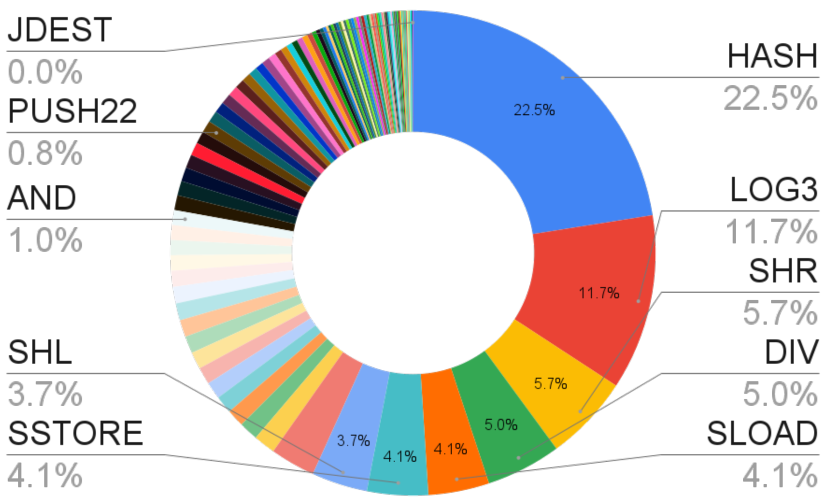

Fig. 2: A depiction of the sizes of the 100 opcode circuits in our prototype EVM implementation.

The EVM subcircuits Our proto-EVM implementation supports 100 out of 141 EVM opcodes. Each opcode is written as a circuit and the full EVM circuit is a switchboard of these individual opcode circuits. Figure 2 shows the range of subcircuit sizes involved in the 100 EVM opcodes we implemented. The total size of the circuit together is about 120,000 gates and the largest subcircuit is $\approx$20% of the total size. Note that the constraints that capture reading the step's instruction from the

As shown in Table 1, we find a 30$\times$ reduction in the total EVM circuit size when using Nebula's memory-checking techniques over Spice. Note that this includes the linear scan constraints for all memory types, amortized over the EVM steps.

The ERC-20 transaction. The EVM program that we prove using Nebula is the standard ERC-20 token transfer. According to analysis done in 2020[6], over 70% of Ethereum transactions are ERC-20 token transfers, and they account for over a third of the gas costs of on the blockchain. We obtain the bytecode and calldata for the ERC-20 using the Remix IDE.[7] The transfer function itself is implemented through the standardized OpenZeppelin [3] library. We use the Istanbul fork of Ethereum for our tests and measurements. ERC-20 program takes 635 EVM steps to transfer tokens from one account to another. In a nutshell, the costliest operations an ERC-20 transfer makes are 3 calls to `SHA3` (which

6 https://www.alchemy.com/overviews/ethereum-statistics
7 remix.ethereum.org

we model with a cheaper custom hash function), 2 calls to the shift opcodes, 19 memory accesses and 8 storage accesses.