Coral: Fast Succinct Non-Interactive Zero-Knowledge CFG Proofs

Sebastian Angel Sofía Celi† Elizabeth Margolin Pratyush Mishra Martin Sander Jess Woods
University of Pennsylvania †*Brave Software & University of Bristol*

***Abstract*—We introduce Coral, a system for proving in zero-knowledge that a committed byte stream corresponds to a structured object in accordance to a *Context Free Grammar*. Once a prover establishes the validity of the parsed object with Coral, they can selectively prove facts about the object—such as fields in Web API responses or in JSON Web Tokens—to third parties or blockchains. Coral reduces the problem of correct parsing to a few simple checks over a *left-child right-sibling* tree and introduces a novel *segmented memory* abstraction that unifies and extends prior constructions for RAM in zkSNARKs. Our implementation of Coral runs on a standard laptop, and non-interactively proves the parsing of real Web responses (JSON) and files (TOML and C) in seconds. The resulting proofs are small and cheap to verify.**

1. Introduction

Parsing a stream of bytes into a structured object—upon which further operations can be performed—is a fundamental task in systems like browsers, Web services, compilers, firewalls, and more. Developers typically rely on *Context Free Grammars* (CFGs) to formally specify the structure of these byte streams for a wide range of data formats (e.g., JSON and TOML), programming languages (e.g., C, JavaScript), and Internet protocols (e.g., HTTP). Given the recent advances in the efficiency of zero-knowledge proof systems, there is growing optimism that we can soon cryptographically commit to a stream of bytes and then prove that they parse into a valid structured object. This, in turn, enables one to subsequently prove complex statements about the parsed structured object, without revealing anything beyond the satisfiability of those statements. Applications for this technology include:

- *zk-TLS*: Given a commitment to the byte stream of a TLS session [20, 34, 54, 78, 80, 81], a user can prove statements about their interaction with an existing unmodified Web service to a third party (e.g., that they accessed their bank's website and the website reported a specific account balance for their account). This capability is useful for building *oracles* in which users can prove facts about Web data to a blockchain smart contract.
- *zk-Authorization*: Given a signed token (like a JSON Web Token or MDOC) from an access delegation service (e.g., OAuth provider, OpenID Connect authority, government), systems like zkLogin [27], FS [39] and zkCreds [67] let users prove facts about the token. The proof reveals nothing beyond the truth of the facts.
- *zk-Compilation*: Given a commitment to bytes representing a program's source code, a user can prove that an executable or bytecode file (e.g., x86 ELF, WASM) was the result of compiling the code with some compilation pipeline, without revealing the code. When combined with prior work on *zk static analyses* [37], a verifier without access to the code can confirm that the executable satisfies certain semantic or safety properties.
- *zk-Middleboxes*: Given a commitment to the byte stream of a network protocol (e.g., DNS), the sender can prove to a network intermediary (e.g., a firewall or a policy-enforcing middlebox [58]) that such bytes satisfy a policy without revealing the stream itself [44, 79]. For example, in DNS the sender could prove that it is not requesting a blocked domain or only transmitting permitted data.

While the above applications are promising and have been previously explored, *they all elide (or only partially address) how to bridge the gap between a commitment to a byte stream, and the in-memory representation of the structured objects they represent and on which their proofs operate*. For example, existing zk-TLS and zk-Authorization systems [27, 34, 54, 78, 81], which prove facts about Web API responses typically formatted as JSON, either defer parsing correctness to future work, reveal relevant portions of the API response to the verifier (who then checks them directly [81]), or assume that the byte stream is a valid JSON object and focus on proving the presence of certain substrings within the byte stream [27]. In this latter case, if this assumption is violated (e.g., if a Web server is misconfigured or outputs an error message), a malicious prover could exploit the invalid message to prove a false claim.

Concretely, suppose that a user authenticates to a server using the string `'email':'admin@domain.com'` as an email. The server recognizes this as invalid and returns:

```
HTTP 400 Bad Request.
The Email "'email':'admin@domain.com'" is invalid.
```

A malicious prover who has a commitment to this error message (e.g., as part of a zk-TLS session) can pretend that the commitment is actually to a JWT token. The prover can then prove in zero-knowledge that it contains the substring `'email':'admin@domain.com'` and could use it to authenticate to a third party as having the authority of 'admin@domain.com'. Hence, assuming that inputs are well-formed (e.g., JWT in this case) poses a security risk.

To address this crucial gap between a commitment to a byte stream and the corresponding in-memory data structure

that is of interest to these applications, we introduce Coral. Coral enables a prover to demonstrate, in zero-knowledge, that a private committed byte stream is correctly parsed——according to a public specification given as a context-free grammar (CFG)——into a structured object stored in a committed *random access memory* (RAM). This parsed object can then be used by the prover in subsequent ZK applications to prove arbitrary facts about its contents.

Our implementation of Coral can prove correct parsing of C programs, TOML configuration files from large code repositories, and complete JSON API responses from real-world services, including banks, currency exchanges, sports betting sites, leaked credentials from *haveibeenpwned* [45], and Google's OAuth JWT tokens. Coral's proofs for these concrete applications are fast to generate (1.2 to 8.2 seconds), non-interactive, and succinct: they are small (16.7 to 18.3 kB) and cheap to verify (31 to 70 ms). Moreover, Coral requires minimal memory (under 2.3 GB) and uses no hardware acceleration, so it can run on any machine. We tested on a Lenovo laptop running Linux and an M3 MacBook Pro.

1.1. Ideas that make Coral practical

Many of the applications discussed earlier acknowledge that, in principle, one could express the logic of a parser as a *rank-1 constraint satisfiability* (R1CS) instance (or as any other constraint system or circuit), and then prove its satisfiability using a suitable ZK proof system. However, as they note, doing so is both challenging and expensive, which is why this step is often avoided in practice. We concur with this assessment, particularly when the goal is to support a general-purpose parser for arbitrary CFGs, as opposed to a specialized parser tailored to a specific data format. This is precisely the motivation behind Coral: to explore how to most effectively *specify the desired grammar*, *decide on what statement to prove in ZK*, and *prove the chosen statement*.

Modern grammars. Coral supports grammars written in modern syntax that developers expect [38]. Coral supports the full expressivity of CFGs, in addition to two important extensions: *exclusion* rules and *non-atomic* rules [14]. These rules are present in all of the grammars that we studied.

Exclusion rules are of the form "allow any character *except for* {a,b,c}". Non-atomic rules apply when whitespace is not meaningful (e.g., compressed JSON and pretty-printed JSON are both valid but have different numbers of white spaces). With Coral, developers write grammars in `pest` [10] and can use exclusion, atomic, and non-atomic rules.

NP checkers and LCRS Parse Trees. Coral leverages the common observation that checking the answer of a computation is often cheaper than computing the answer. This is especially relevant for us: verifying that a byte stream has been correctly parsed takes linear time (in the length of the stream), while CFG parsing takes more than quadratic time in the worst case [55]. Consequently, instead of encoding the parser's logic in R1CS, we arithmetize a *complete*, *sound*, and efficient NP checker that takes as input the byte stream, the `pest` grammar, the parsed object, and auxiliary hints, and verifies that the parsed object results from parsing the byte stream according to the grammar.

Coral's representation of a parsed object, similar to prior work [57], is a *parse tree*, which is a type of concrete syntax tree. Coral's checker walks through each node in the tree once, verifying conditions that collectively ensure conformity to the grammar and correspondence with the byte stream. To bound the number of leaves in the parse tree (required by R1CS and other arithmetizations), Coral converts any arbitrary parse tree into a *left-child right-sibling* (LCRS) tree [49]. This allows Coral to use any CFG grammar and remain compatible with additional features that developers have come to expect. In contrast, prior work requires grammars to be written in *Chomsky Normal Form* (to bound the size of the parse tree), which blows up the number of rules and fails to efficiently support features like exclusion and non-atomic rules.

Recursion to limit statement size and resource usage. The core of Coral's checker is a DFS traversal over LCRS trees: an algorithm that is naturally recursive and uniform. Coral creates an R1CS instance that confirms the correctness of a tree node, and the prover recursively proves each instance using a *folding scheme* [51] while showing that it followed DFS order (without revealing anything about the tree beyond its size). Folding not only reduces proving time, but it also enables the prover to efficiently run on a laptop since one can prove one *step* (R1CS instance) at a time. For proving, Coral uses Nova [51] with an additional technique from recent work [50] to make Nova's protocol zero-knowledge.

Segmented memory. Coral proves the correctness of an in-memory representation of the parsed object, namely the private LCRS parse tree. This tree can then be *persisted* to be used in any other ZK proof via the use of a commitment. Coral also requires cheap private random access to the public grammar rules and a private stack to track progress in the parse tree. To support this mix of memory types (read-only, read/write, stack, public and private, volatile and persistent), we introduce *segmented memory*. Segmented memory is not a new theoretical contribution. Instead, it acts as an elegant and clean abstraction over a variety of highly optimized memory constructions that we implement. This abstraction significantly simplifies the development and management of memory objects within a proof system.

In particular, with segmented memory a proof developer specifies a set of memory segments, each annotated with a type (RAM, ROM, Stack), a visibility (public, private), and a persistence flag (yes, no). Then, the library automatically instantiates each segment with a suitable cryptographic primitive that provides the desired properties and manages all of the associated complexity. The developer can then interact with these segments in their R1CS instance using simple APIs, with the library taking care of the rest.

Behind the scenes, our implementation of segmented memory is based on Nebula [25] with several optimizations. For example, we remove some of the checks in Nebula if the segment is read-only, remove more checks for public memory, and even more checks if the segment is for a stack.

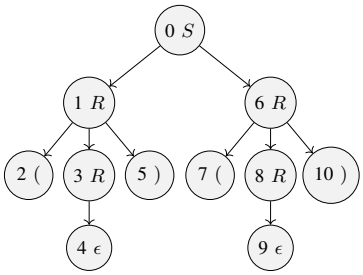


Figure 1: Parse tree for the string "()()" and the grammar $S \to (S) \mid SS \mid \epsilon$. We use ϵ to denote the empty string.

Finally, segmented memory allows any individual segment to be extracted into a portable commitment that can be used in other proofs.

Summary of contributions:

- The design, implementation, and evaluation of Coral, a practical system for proving that a committed byte stream corresponds to a specific LCRS parse tree, and that this tree is consistent with a given CFG.
- The segmented memory API and our automated memory builder that specializes the constraints needed for each access type and segment type.
- The integration of segmented memory, witness blinding, and hiding commitments into Nova.

Limitations. Many real-world formats are *context-sensitive*. For example, network protocols like DNS or TCP include fields that determine the length of subsequent ones. Similarly, popular markup languages like HTML and XML, file formats like PDF, and programming languages like Python are not context-free. Moreover, modern grammar frameworks such as `pest` also support two useful features: rule priority to resolve ambiguities, and *negative predicates* [63], which generalize exclusion rules. Supporting context-sensitive grammars and these extra features is important future work.

2. Background

This section reviews CFGs, rank-1 constraint satisfiability (R1CS), NP checkers, and zero-knowledge succinct non-interactive arguments of knowledge (zkSNARKs).

2.1. Context Free Grammars (CFG)

A CFG G is a tuple $(V, \Sigma, \mathcal{R}, S)$ where V is the set of non-terminals, Σ is the set of terminals, $\mathcal{R} \subseteq V \times (V \cup \Sigma)^*$ is the set of production rules and can contain any combination of terminals and non-terminals, and S is the start symbol.

The process of *derivation* consists of applying rules in $\mathcal{R}$ in order to generate strings in the language defined by the grammar G. Consider the language of strings with balanced parentheses, given by the grammar G:

$$S \to R$$
$$R \to (R) \mid RR \mid \epsilon$$

A derivation of the string "()()" according to G is:

(1) $S \to R$	(4) $R \to (R)(R)$
(2) $R \to RR$	(5) $R \to (\epsilon)(R)$
(3) $R \to (R)R$	(6) $R \to (\epsilon)(\epsilon)$

Derivations are also represented with a *parse tree*, where the leaves of the tree are the terminal symbols in the string and the internal nodes are non-terminals. Parse trees have 2 features: (1) each node and its children form a valid rule in the grammar; and (2) the leaves of the tree, when concatenated from left to right, are equivalent to the string being parsed. Figure 1 gives the parse tree for the above derivation.

2.2. zkSNARKs

A *zero-knowledge succinct non-interactive argument of knowledge* (zkSNARK) is a protocol between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$. $\mathcal{P}$ produces a proof π that convinces $\mathcal{V}$ that $\mathcal{P}$ knows a satisfying witness w to an NP statement without revealing w. zkSNARKs typically target the general NP complete problem of *circuit satisfiability* (e.g., R1CS [42, 70], Plonkish [41], CCS [71]). Informally, zkSNARKs are:

1) **Zero-knowledge:** The proof π reveals no information about w beyond the fact that it is a valid witness.
2) **Succinct:** The size of π and the time to verify it are sublinear in the size of the satisfiability instance.
3) **Non-interactive:** No interaction is required between $\mathcal{P}$ and $\mathcal{V}$ besides transferring public inputs/outputs and π.
4) **Argument of knowledge:** $\mathcal{P}$ must convince $\mathcal{V}$ that it knows a witness w that satisfies the instance. This argument is complete and computationally sound.

- *Perfect completeness:* If $\mathcal{P}$ knows a satisfying w, $\mathcal{P}$ can always generate a proof π that convinces $\mathcal{V}$.
- *Knowledge soundness:* If $\mathcal{P}$ does not know a satisfying w, it cannot produce a proof π that $\mathcal{V}$ will accept, except with negligible probability.

We defer a formal definition of knowledge soundness to definition E.1.

Rank-1 Constraint Satisfiability (R1CS). We focus on *rank-1 constraint satisfiability* (R1CS) as this is the arithmetization supported by the zkSNARK we use [51], but our ideas apply to other arithmetizations (e.g., CCS [71]). An R1CS instance is given by a tuple $(\mathbb{F}, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, x, \text{rows}, \text{cols})$, where $\mathbb{F}$ is a finite field, x is the public input and output of the instance, $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{F}^{\text{rows} \times \text{cols}}$ are matrices, and cols $\geq |x| + 1$. The instance is satisfiable if and only if there exists a witness $w \in \mathbb{F}^{\text{cols}-|x|-1}$ such that the solution vector $\boldsymbol{Z} = (w, x, 1)$ satisfies $(\boldsymbol{A} \cdot \boldsymbol{Z}) \circ (\boldsymbol{B} \cdot \boldsymbol{Z}) = (\boldsymbol{C} \cdot \boldsymbol{Z})$, where $\cdot$ is the matrix-vector product and $\circ$ is the Hadamard product. The value 1 in z enables encoding constants.

2.3. NP checkers

It is often significantly cheaper to *verify* a result than to *compute* it: this is especially true in the context of R1CS. For

example, over a finite field $\mathbb{F} = \mathbb{Z}_p$, computing the multiplicative inverse $1/x$ requires $\log(p)$ R1CS constraints using Fermat's Little Theorem (i.e., computing x^{p-2}). However, if $\mathcal{P}$ supplies a candidate inverse inv, verifying that it is correct requires only a single constraint: $\text{inv} \cdot x - 1 = 0$. This is an example of a *NP checker*: a relation that checks the correctness of a claimed result more efficiently than one can compute the result. Checkers of this kind have been developed in various settings [22, 23, 32, 46, 73, 77, 83].

In this work, we design a new NP checker for context-free grammar (CFG) matching, leveraging the structure of left-child right-sibling (LCRS) parse trees (§4).

3. Overview

As noted in Section 1, our aim is to build an efficient system for proving the correct parsing of a byte stream into a parse tree. We start by formalizing our setting, stating our goal, and examining alternative approaches. We then highlight what distinguishes Coral from prior efforts.

Participants. The system involves three parties: a *committer* $\mathcal{M}$, an untrusted *prover* $\mathcal{P}$, and a *verifier* $\mathcal{V}$. In some applications $\mathcal{M}$ and $\mathcal{P}$ may be the same party, and in others they may be separate parties.

The committer $\mathcal{M}$ generates a hiding and binding commitment C_{B} to a byte stream B. The mechanism for ensuring that C_{B} commits to a meaningful byte stream in the first place is orthogonal to this work, and is instead within the purview of the larger application that uses Coral. For instance, in the zk-TLS context, C_{B} corresponds to a commitment to a TLS transcript resulting from the interaction of a client with a Web server; the commitment is usually computed via a secure two-party computation between the browsing client and an auxiliary party, acting as $\mathcal{M}$.

The prover $\mathcal{P}$ is any party who knows B in the clear and who wishes to prove to others—using the commitment C_{B}—some statement about it without revealing anything beyond the truth of the statement.

Finally, the verifier $\mathcal{V}$ is any entity who, given the public C_{B} and a proof π attesting to some statement over B, seeks to verify the validity of π non-interactively and efficiently.

Goal. The objective is to establish that a hiding and binding commitment C_T correctly commits to a structured object T, which represents the parse tree resulting from parsing the byte stream B according to a public CFG. This correspondence between B and T is a crucial first step toward proving complex statements about the original data. Once this link is established, the structure of T can be leveraged to prove additional facts. For instance, if T is the parse tree of a JSON document, one can use its nodes to directly access specific fields (keys or values), and subsequently prove statements about those fields without revealing the full document.

We give the formal NP relation that Coral proves (which fully specifies the above informal goal) in Appendix B.

3.1. Related work and prior approaches

Before presenting Coral, we review related works and a few existing methods for achieving the goal presented above.

Related works that solve a different problem. The closest works to Coral are Reef [23], `zkreg` [64], zk Regex [21], Zombie [79], the work of Luo et al. [56], and DECO [81]. The first five target *regular expressions* rather than CFGs, and therefore their techniques cannot handle any of the applications we consider. DECO supports parsing specific fields in JSON documents, but it requires leaking to the verifier both the target field and the surrounding context in which it appears. This precludes its use as a general primitive for proving that a byte stream corresponds to a structured object, as it requires revealing portions of the object itself.

Existing solutions. The following proposals address our goal but do so at a high cost.

zkVMs. The simplest approach is to use a *zero-knowledge virtual machine* (zkVM): a proof system that takes as input a sequence of low-level operations and produces a succinct proof that all operations were executed correctly. A representative example is RISC Zero [65], which is a highly optimized zkVM for RISC-V instructions. To achieve our goal, one could execute the following program inside RISC Zero: (1) take as input the grammar G, byte stream B, and the blinding value used by $\mathcal{M}$ to generate C_{B}; (2) verify that C_{B} is a valid commitment to B; (3) run a general-purpose parser (e.g., `pest`) to obtain a parse tree T of B; and (4) output a new commitment C_T to the parse tree.

The advantage of this approach is its simplicity: one can reuse existing parsers. However, this comes at a significant cost. zkVMs must emulate a CPU's fetch-decode-execute logic for each instruction, resulting in high overhead. Moreover, executing a parser inside the zkVM forfeits the efficiency gains that can be achieved with NP checkers that verify the parsing result instead of doing the parsing.

Compile the parser into R1CS. A second approach is to compile the parsing logic into R1CS instead of RISC-V. This avoids the overhead associated with emulating a CPU and can result in more efficient proofs in principle. However, there is no existing way to automatically transform a complex parsing library like `pest` into R1CS. Existing circuit compilers like Circom [2], ZoKrates [19], and Noir [7], require developers to reimplement the application in a domain-specific language; CirC [60] supports a subset of C, but it cannot handle a full parser. Moreover, this approach still gives up on the efficiency gains offered by NP checkers.

Compile an NP checker. The most efficient approach is to arithmetize an NP checker. Coral, as well as the recent proposal by Malvai et al. [57], adopt this strategy. Their approach involves transforming the grammar into Chomsky Normal Form (which blows up the number of rules quadratically), and then constructing an NP checker over the resulting parse tree. This method does not support non-atomic or exclusion rules. Moreover, as we show in our evaluation (§6), their most efficient construction—which is interactive and not succinct, so verification is expensive and the proofs are large—is over an order of magnitude slower than Coral (which is both succinct and noninteractive).

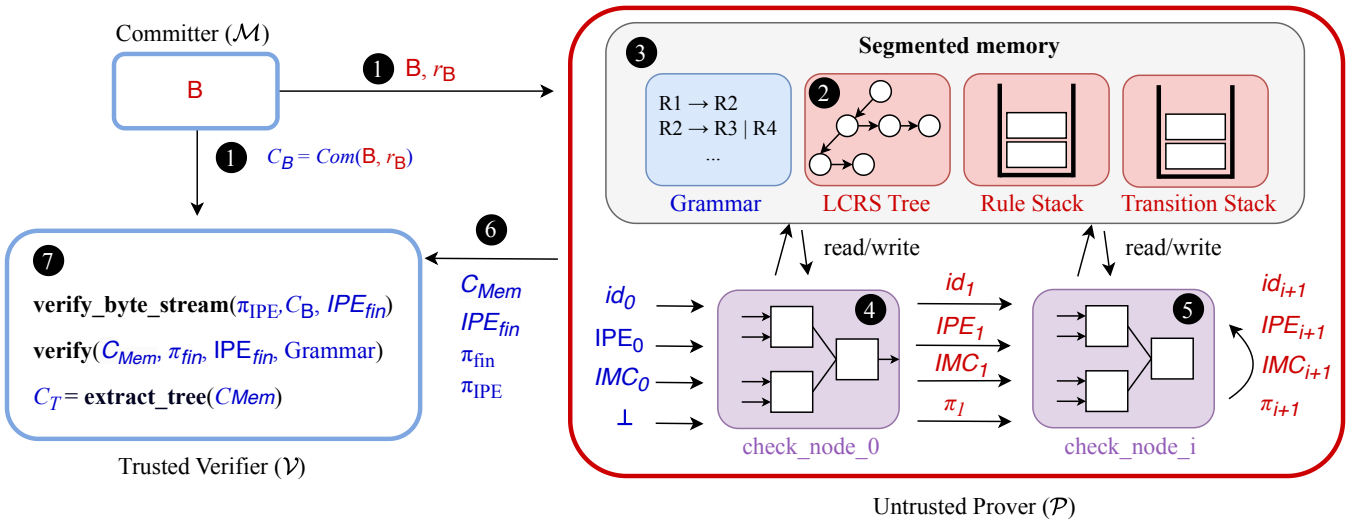


Figure 2: Overview of Coral. Blue text is public information. Red text is private information visible only to $\mathcal{M}$ and $\mathcal{P}$.

3.2. Coral's approach

Coral's design is the result of three observations.

First, building an NP checker eliminates the complexity of arithmetizing a parser or the overhead of a zkVM. Coral's checker works by traversing a *parse tree* (the output of parser libraries such as `pest`), and confirming some facts about its nodes. However, parse trees can have nodes with arbitrary out-degree, which is problematic when working over circuit-like computational models such as R1CS, where one must upper bound the number of children per node. To address this, Coral converts the parse tree into an *LCRS* tree via the Knuth transform [49]. LCRS trees are binary trees, and so each node has at most two children. In Section 4, we present a *complete* and *sound* NP checker for correct parsing that operates over LCRS trees.

Second, validating each node of the LCRS tree involves repeatedly applying the same function: `check_node`. This repetition is a great fit for *folding schemes* [51] in which a single "step" of computation is expressed as R1CS (or other arithmetization), and $\mathcal{P}$ *folds* many steps with different inputs into a single R1CS instance. $\mathcal{P}$ can then prove the satisfiability of the folded instance with a SNARK, which implies the satisfiability of all of the steps. This is beneficial because folding is cheaper than proving, and $\mathcal{P}$ only needs to maintain two steps in memory at a time, enabling proving on machines with limited resources.[1]

Third, Coral's checker requires multiple memory types: a private stack for tree traversal and tracking production rules, a public memory for the grammar, and a private persistent memory for the LCRS parse tree. Normally, one would implement each type of memory with a different specialized construction [23, 28–30, 33, 36, 40, 50, 62, 69, 72, 74, 76]. However, many of these constructions are either incompatible with folding, and or have fixed costs that are only amortized over many memory accesses (e.g., lookup arguments).

If memory types are implemented separately, amortization is limited to each type's individual access pattern. This is especially inefficient in the context of folding, where each step may perform only a few memory operations.

Coral, as we discuss in Section 5, uses one memory that is fast, foldable, *segmented*, and supports all of our use cases.

1. Coral can work with any system for *commitment-carrying incrementally verifiable computation* [25, 75], but for concreteness, we describe our construction in terms of Nova and Nebula [25, 51].

3.3. Coral's end-to-end operations

Coral combines the above ideas into the end-to-end system seen in Figure 2. We detail each action below.

1. The committer $\mathcal{M}$ generates a commitment to the byte stream $C_B = \mathsf{Commit}(B, r_B)$ with blind r_B using a commitment scheme that is hiding, binding, and allows for sequential or random access to entries in B (§4.3). $\mathcal{M}$ then makes the commitment public (the figure shows $\mathcal{M}$ sending it to $\mathcal{V}$ directly, but any reasonable publication method works). $\mathcal{M}$ also sends both the byte stream B and the blind r_B used to generate C_B to $\mathcal{P}$.
2. $\mathcal{P}$ generates a parse tree for the byte stream B using any parser and the grammar G (Section 4). $\mathcal{P}$ transforms this parse tree to an LCRS tree T. Each node in T is given a unique `id` representing when it is accessed in DFS pre-order traversal. The root's `id` is $\mathtt{id}_0 = 0$.
3. $\mathcal{P}$ initializes Coral's memory with 4 segments: the grammar G is placed in a ⟨public, read-only, volatile⟩ segment; the LCRS tree T is placed in a ⟨private, read-only, persistent⟩ segment; and two empty stacks are placed in ⟨private, stack, volatile⟩ segments. $\mathcal{P}$ then generates a commitment C_{Mem} that commits to the initial private memory segments (T and the empty stacks), as well to any the memory operations it will make against the public or private memories. For this, $\mathcal{P}$ uses an incremental commitment scheme that allows for random access (Section 5).

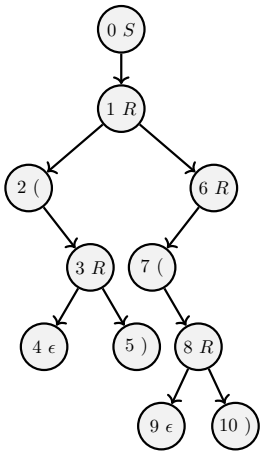


Figure 3: LCRS tree corresponding to the tree in Figure 1.

❹ $\mathcal{P}$ runs the zeroeth step of the `check_node` function with the constant $\mathtt{id}_0$ as input. It outputs the identifier of the next node to process ($\mathtt{id}_1$), an incremental evaluation of a polynomial that captures which leaves have been seen so far (IPE_1), an incremental commitment to any memory operations performed so far (IMC_1), and a proof (π_1) that step 0 was executed correctly.

❺ $\mathcal{P}$ continues to run `check_node` for every node in T, feeding the output of step i into step $i+1$.

❻ Finally, $\mathcal{P}$ sends the original commitment to the memory performed during action ❸ (C_{Mem}), the final incremental polynomial evaluation that summarizes the leaves seen ($\mathsf{IPE}_{\mathsf{fin}}$), a polynomial evaluation proof (π_{IPE}), and the proof (π_{fin}) that the final instance is correct.

❼ $\mathcal{V}$ checks that C_{B} and $\mathsf{IPE}_{\mathsf{fin}}$ encode the same byte stream with π_{IPE}. $\mathcal{V}$ then checks the final proof π_{fin} by using C_{Mem} as the value for the last incremental memory commitment ($\mathsf{IMC}_{\mathsf{fin}}$). If this proof validates, this implies that $\mathsf{IPE}_{\mathsf{fin}}$ and $\mathsf{IMC}_{\mathsf{fin}}$ were computed correctly and all prior proofs were verified. Lastly, $\mathcal{V}$ can obtain a commitment to T from C_{Mem} to use in other proofs.

In the following sections we provide details.

4. Coral's parse tree and NP checker

In this section, we address the following task: given a CFG G, a byte stream B, and a parse tree T, $\mathcal{P}$ must prove that T is a valid parse tree for B, and T conforms to G. The main source of complexity lies in the flexibility of CFG rules: a node in T can have an arbitrary number of children. However, expressing the `check_node` step function in R1CS requires a fixed upper bound. To minimize overhead, this bound must be as tight as possible to avoid excessive padding. For this, we convert parse trees to LCRS trees.

4.1. Left-Child Right-Sibling (LCRS) Trees

A classic way to handle parse trees where each node can have an (a priori) unbounded number of children is to rewrite G into a grammar G' where each non-terminal rule expands to either a single terminal or at most two other non-terminal rules. This form (adopted by prior work [57]) is known as *Chomsky Normal Form* (CNF), and results in binary parse trees. However, expressing a grammar in CNF is cumbersome and rarely done outside of pedagogical contexts. Further, converting a modern grammar (such as `pest`) to CNF can cause a quadratic blowup in the number of rules.

Instead of rewriting the grammar to eliminate rules of arbitrary length, Coral asks $\mathcal{P}$ to restructure the parse tree itself. Knuth [49] proposed a method to transform any tree into a binary tree using a single linear pass: the result is an LCRS tree as shown in Figure 3. Each node in an LCRS tree has at most two edges: a *left child* edge and a *right sibling* edge. The left child edge points to the node's leftmost child, and the right sibling edge points to the sibling immediately to its right. LCRS tree nodes in Coral look like:

```
Node {
  id: field,
  parent_id: field,
  l_child_id: field,
  r_sibling_id: field,
  is_leaf: bool,
  symbol: field,
}
```

$\mathcal{P}$ assigns to each node identifiers following a pre-order traversal: the root has $\mathtt{id}_0 = 0$ and the remaining nodes have monotonically increasing `id`s. Each node stores the identifiers of the node's parent (`parent_id`), left child (`l_child_id`), and right sibling (`r_sibling_id`). The boolean `is_leaf` indicates whether the node is a leaf in the *original* (non-LCRS) parse tree, and `symbol` is the symbol of the node. If the node corresponds to a non-terminal, `symbol` contains the rule name; otherwise, it holds the byte value of the terminal.

The construction of the LCRS tree (including the assignment of identifiers, leaf statuses, and symbols) is entirely untrusted: $\mathcal{P}$ may construct it arbitrarily. Our NP checker is guaranteed to detect any invalid LCRS structure, as well as any inconsistency between the tree, G, and the byte stream.

4.2. Representing grammar rules for G

As mentioned in Section 3.2, the public grammar G is placed in public, read-only, volatile memory. While we describe the details of our memory construction in Section 5, it is helpful here to explain how grammar rules are represented. For the matching parentheses example from Section 2.1, we encode the grammar as a table with four entries, where each alternation corresponds to a separate row.

$$S \to R$$
$$R \to (R)$$
$$R \to RR$$
$$R \to \epsilon$$

Each rule is then converted to a vector of the form $[\mathsf{LHS}, \mathsf{RHS}_1, \mathsf{RHS}_2, \ldots, \mathsf{RHS}_n]$, where LHS is the left-hand side and RHS_i is the i-th term on the right-hand side. For example, the rules above become: $[S, R], [R, (, R,)], [R, R, R]$,

and $[R, \epsilon]$. Since all table rows must be of equal length, each rule is padded to the length of the longest one: $[S, R]$ becomes $[S, R, \mathsf{null}, \mathsf{null}]$. Finally, we append two boolean flags to each rule. The first indicates whether the rule is atomic (marked in `pest` syntax with an `@` on the RHS). The second indicates whether the rule contains exclusions. We describe how Coral handles such rules in Appendix A.

4.3. Tying the parse tree T to the byte stream B

Recall from Section 2.1 and Figure 1, that the leaves of a standard parse tree when concatenated from left to right reconstruct the underlying byte stream B. $\mathcal{P}$ can establish this correspondence between the original parse tree and B by proving that a pre-order DFS traversal of the tree, that concatenates encountered leaves, yields a stream equal to B.

However, as seen in Figure 3, this does not hold for an LCRS parse tree (see nodes 2 and 7). To resolve this, we introduce a small amount of bookkeeping along with a key structural observation. The bookkeeping consists of the additional `is_leaf` field in the node data structure. A node is considered a leaf if and only if `is_leaf = true` and `l_child_id = NULL`. We will henceforth use "leaf" to refer exclusively to nodes in T satisfying these conditions. The structural observation is that a DFS pre-order traversal of the original parse tree and its corresponding LCRS tree T visits the nodes in the same order. Thus, performing a pre-order traversal of T is equivalent to traversing the original tree.

Commitment options. As noted, $\mathcal{M}$ should commit to B to produce the commitment C_{B}. The key requirement for this is that during each execution of the `check_node` step function, when processing a leaf, we must compare the leaf's symbol to the *next* byte in B and ensure they match. There are two main approaches to enable this: an *individual* (or "online") method that checks each leaf symbol matches (e.g., Merkle trees), and a *batch* (or "offline") method that checks that all symbols match at once (e.g., hash chain). Coral is general and modular, and supports both verification strategies. Below we describe the approach we use in our implementation, which verifies all symbols as a batch.

Coral's commitment to B. First, $\mathcal{M}$ samples three uniformly random blinds: $r^1_{\mathsf{B}}, r^2_{\mathsf{B}}, r^3_{\mathsf{B}} \leftarrow \mathbb{F}$. Then, $\mathcal{M}$ encodes the byte stream $\mathsf{B} = b_1 || \cdots || b_{|\mathsf{B}|}$ as the degree $|\mathsf{B}| + 1$ univariate polynomial $B(x) = (x - r^1_{\mathsf{B}}) \cdot (x - a_1) \cdot (x - a_2) \cdots (x - a_{|\mathsf{B}|})$, where $a_i = 2^{32} \cdot b_i + i$. That is, each byte b_i is combined with its position $i \in [1, |\mathsf{B}|]$ to yield a unique field element. The factor 2^{32} is chosen because Coral treats Unicode characters as the smallest atom of the byte stream. $\mathcal{M}$ then commits to $B(x)$ using a hiding variant of the *KZG polynomial commitment* [47] using the blinds r^2_{B} and r^3_{B} to produce C_{B}. The full construction and formal security guarantees of this hiding variant are detailed in [35]: at a high level, the blind r^1_{B} ensures that a *single* evaluation of $B(x)$ reveals no information about B, while r^2_{B} and r^3_{B} ensure that the commitment C_{B} and a *single* evaluation proof do not leak information about B. Finally, $\mathcal{M}$ either publishes C_{B} or sends it directly to $\mathcal{V}$ (depending on the application), and sends $\mathsf{B}, r^1_{\mathsf{B}}, r^2_{\mathsf{B}}, r^3_{\mathsf{B}}$ to $\mathcal{P}$.

Enforcement. At this point, $\mathcal{P}$ constructs the LCRS parse tree T from B and G, and commits to the memory containing the grammar rule table (§4.2), the tree T, and two empty stacks (we defer details to Section 5.3). To verify that T is consistent with B, the `check_node` procedure incrementally evaluates a polynomial derived from the symbols of the leaves of T, evaluated at a specific point c. We denote this computation as the incremental polynomial evaluation IPE. The point $c \in \mathbb{F}$ is a random challenge obtained via the Fiat-Shamir transform over the transcript, after $\mathcal{P}$ has added all relevant commitments including C_{B} and the memory commitment (which includes T).

More concretely, in the first step of `check_node`, $\mathcal{P}$ supplies a public counter $i = 0$, an initial accumulator $\mathsf{IPE}_0 = 1$, and a secret witness r^1_{B} (among other values). The procedure then computes $\mathsf{IPE}_1 = \mathsf{IPE}_0 \cdot (c - r^1_{\mathsf{B}})$ and increments i. For each subsequent step, when `check_node` encounters a leaf node, it computes $\mathsf{IPE}_{i+1} = \mathsf{IPE}_i \cdot (c - (2^{32} \cdot \mathsf{leaf}_{\mathtt{symbol}} + i))$ and increments i. We denote the final output as $\mathsf{IPE}_{\mathsf{fin}}$.

Separately from the zkSNARK for `check_node`, $\mathcal{P}$ generates a KZG evaluation proof, π_{IPE}, showing that the committed polynomial $B(x)$ evaluates to a value val at the challenge point c. $\mathcal{V}$ then: (1) verifies that $\mathsf{IPE}_{\mathsf{fin}}$ was correctly computed using the zkSNARK proof π_{fin}; (2) checks the KZG evaluation proof π_{IPE} asserting $\mathsf{B}(c) = \mathsf{val}$ using the commitment C_{B}; and (3) confirms that $\mathsf{IPE}_{\mathsf{fin}} = \mathsf{val}$. If all checks succeed, it guarantees that the leaf symbols in T are consistent with the committed byte stream B. We provide a proof of this in Appendix F.

4.4. Validating parse tree T for grammar G

Recall that in a standard parse tree, each node and its immediate children must match one of the rules in G: the parent should appear in the LHS and the children in the RHS of a production rule. By applying this check to every non-terminal node, one can verify that the tree is consistent with G. As noted, a challenge in Coral is that we use LCRS trees, in which nodes no longer store direct edges to all of their children: for instance, as seen in Figure 3, there is no edge between nodes 0 and 6.

To address this, Coral requires $\mathcal{P}$ to explicitly provide the *symbols* of the children of the current node being processed, along with the secret memory address of the corresponding rule in the public grammar table (§4.2). These are given as inputs to `check_node`, which then verifies that the parent's symbol and the proposed child symbols appear together at the given memory address as a valid production rule. We defer the details of memory access to Section 5.

Note that a malicious $\mathcal{P}$ might supply a set of symbols for children that do not appear in T, or it could provide the correct symbols for the children but in a different order than they appear. To prevent this, `check_node` maintains a stack: `rule_stack`. Whenever $\mathcal{P}$ proposes child symbols for a node, `check_node` pushes them onto `rule_stack` in *reverse order* alongside a boolean indicating whether they are the last symbol in the rule. For example, if the

```
check_node(field id, field ipe,
    field[] rule_stack, field[] tran_stack,
    field r1_B, field[] proposed_sym) {

  Node cur = get_node(id); // ROM access

  ipe = if (id == 0) {
    accumulate(ipe, r1_B) // poly eval
  } else {
    // validate prior proposed rules
    field (symbol, rule_end) = rule_stack.pop();
    assert(cur.symbol == symbol);
    assert((cur.r_sibling_id==NULL) == rule_end);

    // accumulate poly eval if it's a leaf
    if cur.is_leaf {
      accumulate(ipe, symbol)
    } else { ipe }
  };

  if !cur.is_leaf {
    // confirm exists in grammar (ROM access)
    assert(lookup(cur.symbol, proposed_sym));

    // number of nonzero symbols in proposal
    field n_sym = proposed_sym.length();

    for i in 1..n_symbols {
      rule_stack.push(proposed_sym[n_sym-i],i==1);
    }
  }

  if cur.r_sibling_id != NULL {
    tran_stack.push(cur.r_sibling_id);
  }

  field next_id = if cur.is_leaf {
    tran_stack.pop()
  } else { cur.l_child_id };

  assert(id + 1 == next_id);

  return (next_id, ipe, rule_stack, tran_stack);
}
```

Figure 4: Coral's `check_node` step function (simplified).

proposed symbols are $[(, R,)]$, then `check_node` pushes onto `rule_stack` the tuples $[], \texttt{true}]$, $[R, \texttt{false}]$, and $[(, \texttt{false}]$. `check_node` then continues its pre-order traversal of T. As it transitions to a new node, it pops the current symbol and the `rule_end` flag from `rule_stack` and verifies that it matches the symbol at the current node. If the `rule_end` is `true`, it also checks that the current node has no right sibling. If $\mathcal{P}$ supplies a child that does not appear in T, or one out of order, then these checks will fail. Since the pre-order traversal of the LCRS tree preserves the order of nodes from the original parse tree, this approach encounters a node's children in the same relative order.

4.5. The `check_node` function

As explained in steps ❹ and ❺ of Section 3.3, `check_node` is executed for every node in T. Since the traversal is preorder, we need a way to keep track of the siblings of nodes as we encounter them and to allow `check_node` to know where to go after it reaches a leaf (which has no left child). We track this information in a transition stack: `tran_stack`. Figure 3 gives the pseudocode for a simplified version of `check_node`. The simplifications are: (1) only a subset of the inputs is shown (e.g., memory-related witnesses are omitted); (2) we omit some checks; and (3) abstract operations are used instead of detailed R1CS.

We now walk through a simple example illustrating how an honest $\mathcal{P}$ can use `check_node` to convince $\mathcal{V}$ that the matching-parenthesis grammar and the LCRS tree in Figure 3 are consistent. In the first iteration, $\mathcal{P}$ supplies $\texttt{id}_0 = 0$, prompting `check_node` to read the root of the tree from memory which has symbol S. $\mathcal{P}$ also proposes $[R, R]$ as the symbols of S's children. `check_node` then accesses memory to verify that $S \to RR$ is a valid rule of the grammar table, and pushes $[R, \texttt{true}]$ and $[R, \texttt{false}]$ onto the `rule_stack`. Since S has a child at address $\texttt{l\_child\_id} = 1$ and no siblings, `check_node` returns the tuple $(1, \text{IPE}, \texttt{rule\_stack}, \texttt{tran\_stack})$. In the next iteration, `check_node` reads the node at address 1 in T's memory segment, pops the top symbol and flag from `rule_stack` (which is R and `false`), and confirms that it matches the current node's symbol and that it has a sibling. It then verifies that $R \to (R)$ is a valid production rule, and pushes $[], \texttt{true}], [R, \texttt{false}], [(, \texttt{false}]$ onto the `rule_stack`. Because node 1 is not a leaf, `check_node` does not update the IPE, and, since node 1 has a sibling ($\texttt{r\_sibling\_id} = 6$), it pushes 6 onto `tran_stack`, and returns $(2, \text{IPE}, \texttt{rule\_stack}, \texttt{tran\_stack})$. This process continues until all nodes are processed. A formal completeness and soundness proof is provided in Appendix B.

5. Segmented and foldable memory

Over the past decade, numerous approaches have been proposed for incorporating state (i.e., storage and memory) into (zk)SNARKs. These include techniques based on routing networks, Merkle trees, lookup arguments, RSA accumulators, and offline memory-checking. The proposals differ significantly in their functionality and performance characteristics. For our system, we require that the approach satisfy the following:

1) Concrete efficiency when instantiated in R1CS.
2) Compatibility with folding schemes.
3) Support for both public and private RAM.
4) Ability to export to portable commitments.

Among the available techniques, we find that the *offline memory-checking* approach, introduced by Blum et al. [31], later adapted for SNARKs by Spice [69], and further refined by Nebula [25], is particularly well-suited to our purposes. We adopt Nebula as the foundation for our RAM representation for the following reasons. First, Nebula achieves concrete efficiency in R1CS, with experimental results showing state-of-the-art performance. Second, Nebula is explicitly designed to integrate with folding frameworks, enabling amortization of the large one-time costs of offline memory-checking across all memory accesses from *every* computation step,

```
check_mem_op(field read_hash, field write_hash,
  bool op, field ts, field segment,
  // below are untrusted inputs from Prover
  field addr, field[] new_vals,
  field time_last_op, field[] vals_last_op) {

  ts += 1;
  assert(time_last_op < ts); // not needed for ROM

  read_hash *= hash(time_last_op, addr,
                vals_last_op, segment);

  field[] vals = if op == READ { vals_last_op }
                 else { new_vals };

  write_hash *= hash(ts, addr, vals, segment);
  return (ts, read_hash, write_hash);
}
```

Figure 5: Coral's `check_mem_op` function (simplified).

```
check_scan(field read_hash, field write_hash,
  // below are untrusted inputs from Prover
  field[][] in_vals, field[][] fin_vals,
  field[] fin_ts, field[] segments) {

  field in_hash = 1;
  field fin_hash = 1;

  for i in 0..size_of_mem {
    in_hash *= hash(0, i, in_vals[i], segments[i]);
    fin_hash *= hash(fin_ts[i], i, fin_vals[i],
                     segments[i]);
  }
  assert(in_hash*write_hash==read_hash*fin_hash);
}
```

Figure 6: Coral's `check_scan` function. Note that instead of computing this function at once, we split the loop into fixed-sized chunks and place each chunk at the end of the step function (e.g., `check_node`). It is computed incrementally and the assert is only performed during the last step.

thereby minimizing overhead. Third, it supports both public and private RAM in a natural way. Finally, its construction yields an explicit, canonical final state, which can be directly committed to and exported for other proofs.

Coral provides an independent implementation of Nebula since, at the time of writing, we could not find a public implementation of Nebula. This effort required us to specify numerous low-level details that were left implicit in the original work. In the following sections, we detail our implementation of Nebula, describe the optimizations made, and show the integration into the segmented memory abstraction.

5.1. Starting point: RAM and ROM from Nebula

Each memory operation is a tuple (t, a, v), where $t \in \mathbb{F}$ is a timestamp, $a \in \mathbb{F}$ is the address read-from/written-to, and $v \in \mathbb{F}^\ell$ is the vector of actual values stored in memory. Following Nebula, we define four multisets of memory operations: the initial set (**IS**), the read set (**RS**), the write set (**WS**), and the final set (**FS**). For an M-sized memory, **IS** is defined as $\textbf{IS} = \{(0, i, v_i) \mid i \in \{0, \cdots, M-1\}\}$, where $v_i \in \mathbb{F}^\ell$ is the initial vector of values stored at address i. **RS** and **WS** are initialized as empty sets, and will track memory accesses to particular addresses. **FS** contains the final values of the memory after the entire computation (i.e., all of the steps of `check_node`) runs.

Since maintaining multisets within R1CS becomes unwieldy as they grow, Spice [69] introduced the idea of using multiset collision-resistant hash functions to compress sets into one field element while allowing incremental accumulation of new entries. However, Spice's hash function is expensive; more recent works like Hekaton [66] and Nebula instead use a *public-coin* hash function or fingerprint. Coral uses a slightly different public coin hash function in order to support values that are vectors of arbitrary length ℓ:

$$\mathsf{hash}(t, a, v) = c_0 - (t + c_1 a + \sum_{j=1}^{\ell} (c_1^{j+1} \cdot v[j]))$$

Here, (c_0, c_1) are random challenges sampled after $\mathcal{P}$ commits to all of the multisets (we discuss memory commitments in Section 5.3). This formulation packs a 32-bit timestamp and the address into a single field element. The hash of an empty multiset is 1, and the hash of multiset with n entries is $\prod_{i=1}^{n} \mathsf{hash}(t_i, a_i, v_i) \in \mathbb{F}$. This hash can be computed incrementally by multiplying one entry at a time.

NP checker for memory. We use an NP checker for memory operations. It consists of two parts: `check_mem_op` (Figure 5), which runs on every memory operation, and `check_scan` (Figure 6), which runs after all memory accesses have been done. `check_mem_op` and `check_scan` are identical to the checkers proposed in Spice [69] with three small changes: (1) they use the above public coin hash; (2) they include a segment field (described later); and (3) as is done in Nebula [25] and Lasso [72] they increment the global timestamp by 1 rather than the involved calculation of Spice. We refer the reader to Spice [69] and Nebula [25] for the correctness argument of these memory checkers. The crucial property, however, is this: if `check_mem_op` is run after every memory operation and `check_scan` is computed at the end, then, provided all checks pass, the memory accesses are guaranteed to be *sequentially consistent* [53]. That is, every read operation for a given address returns the value most recently written to that address.

A drawback of `check_scan`, however, is that it requires scanning all initialized memory at once. This breaks uniformity: one must first compute all steps of `check_node` and then perform `check_scan` separately. Nebula takes this approach: it proves the application's step function (the analogue of Coral's `check_node`), which itself includes many instances of `check_mem_op` with one proof system (e.g., Nova), and then proves `check_scan` with another proof system. In effect, $\mathcal{V}$ receives and verifies two proofs.

In contrast, Coral computes `check_scan` incrementally. Concretely, $\mathcal{P}$ first runs the `check_node` logic for all of its steps (without proving anything) to determine what memory operations, addresses, timestamps, and values will be used.

```
check_push(field write_hash, field ts,
  field sp, field segment,
  // untrusted inputs from Prover
  field[] vals) {

  ts += 1;
  write_hash *= hash(ts, sp, vals, segment);
  sp += 1;
  return (ts, write_hash, sp);
}


check_pop(field read_hash, field ts,
  field sp, field segment,
  // untrusted inputs from Prover
  field[] vals, field push_time) {

  ts += 1;
  sp -= 1;
  read_hash *= hash(push_time, sp, vals, segment);
  assert(push_time < ts);
  return (ts, read_hash, sp);
}
```

Figure 7: Coral's check_push and check_pop functions.

It then commits to this memory trace (§5.3), and samples the random challenges c_0 and c_1 via the Fiat-Shamir transform. Next, Coral partitions the check_scan function into chunks, each responsible for a subset of the for-loop iterations that scan memory (e.g., the first chunk processes the first 20 addresses, the second chunk the next 20, and so on). We append the verification of one such chunk to the end of each check_node step function. In this way, every step function performs (i) some number of check_mem_op operations, and (ii) the verification of a portion of the initial and final multiset hash computation (check_scan). This makes the computation uniform allowing us to use a single proof system. The final invocation of the step function can then check the invariant $\mathbf{IS} \cup \mathbf{WS} = \mathbf{RS} \cup \mathbf{FS}$ by multiplying hashes (assuming multiset-collision resistance, multiplying multiset hashes is equivalent to hashing the union of the multisets).

5.2. Segmented memory

The construction in Section 5.1 is for a general private RAM. However, memory accesses in Coral are quite varied: we use private ROM, private access to public ROM, and multiple private stacks. In principle, we could use k independent memories, one for each purpose. However, it is advantageous to maintain a *single* memory. This is because each Nebula memory requires tracking 4 sets ($\mathbf{IS}, \mathbf{RS}, \mathbf{WS}, \mathbf{FS}$) together with auxiliary state such as the global timestamp. These sets must be passed from one Nova step to the next, and inter-step I/O requires collision-resistant hashing (Nova uses Poseidon [43]). Using a single memory reduces the number of objects that must be hashed per step from $5k$ to just 5.

However, mixing all the memories together introduces a key challenge: how do we ensure that $\mathcal{P}$ performs memory accesses only within the correct memory region? We address this by *segmenting* the memory into disjoint namespaces, each with a distinct *segment descriptor*. Memory operations are now represented by a 4-tuple (t, a, v, s), where $s \in \mathbb{F}$ is the segment descriptor that indicates the memory segment being accessed. This descriptor allows us to cheaply enforce that each memory access is in the correct segment without costly range checks. The multiset hash now includes s:

$$\mathsf{hash}(t, a, v, s) = c_0 - (s + c_1 t + c_1^2 a + \sum_{j=1}^{\ell} (c_1^{j+2} \cdot v[j]))$$

We prove the soundness of this memory construction in Appendix E. Below we describe how Coral reduces costs even further by specializing memory accesses by type:

RAM. In Coral's use cases, we notice that the segment descriptor, the timestamps, and the addresses are all much smaller than a field element. For example, only 2 bits are needed to differentiate between Coral's 4 memory segments. We can optimize our multiset hash function by packing these values into one field element, which results in fewer multiplications by c_1:

$$\mathsf{hash}(t, a, v, s) = c_0 - (s + 2^2 \cdot t + 2^{34} \cdot a + \sum_{j=1}^{\ell} (c_1^j \cdot v[j]))$$

Private ROM. The private ROM is just like the generalized private RAM described in Section 5.1, except that the check_mem_op disallows write operations and we do not need to add a range check for ts. Note that removing this range check disallows the above packing optimization (see Appendix D for details). In Coral, the private ROM is used to store the LCRS tree.

Public ROM. Coral uses the public ROM to store read-only information that must be known to both $\mathcal{P}$ and $\mathcal{V}$—most notably, the grammar table. While the contents of the public ROM are not private, *reads* to it do need to be kept private; otherwise, $\mathcal{V}$ would learn which grammar rule is being accessed by $\mathcal{P}$ in any given step.

Hence, we must prove these accesses in zero-knowledge. However, doing so requires ensuring that the contents of public ROM are correctly initialized in a manner that can be verified by $\mathcal{V}$. To achieve this, we simply have $\mathcal{V}$ compute the multiset hash of **IS** on their own since they have all of the required information (e.g., the grammar table). We can thus remove constraints for the computation of **IS** for public ROM segments from check_scan. Concretely, let $\mathbf{IS}_{\mathsf{pub}}$ denote the public ROM's initial set, and $\mathbf{IS}_{\mathsf{priv}}$ denote the initial set of the remaining segments. The final consistency check in check_scan becomes: $\mathbf{IS}_{\mathsf{pub}} \cup \mathbf{IS}_{\mathsf{priv}} \cup \mathbf{WS} = \mathbf{RS} \cup \mathbf{FS}$.

Stack. Coral requires two stacks. Blum et al. [31] described a way to implement stack accesses using offline memory-checking. Their design implements a stack pointer abstraction on top of check_mem_op as CPUs normally do. In our segmented memory, this would work as follows: we assign each stack a segment descriptor and a stack pointer, sp, publicly initialized to start at the bottom of a stack. Pushes become writes at address sp (checked by check_mem_op), followed by an increment of sp. Pops are handled by

decrementing sp and then performing a read at address sp (checked by check_mem_op).

This approach prevents $\mathcal{P}$ from performing out-of-segment accesses or violating stack discipline. However, it also incurs inefficiencies: we must initialize a number of addresses equal to the maximum possible stack height, and must process the same number of tuples in check_scan. To address this, Blum et al. introduced some clever optimizations that are possible when the stacks start empty. We port those optimizations to our segmented memory as well.

Concretely, only **WS** needs to be updated during a push, since the address where it will push the value is guaranteed to be empty (recall that the pop operation in a stack returns the top element but also empties the address). Following this same logic, only **RS** needs to be updated during a pop. This, together with the fact that the stack pointer enforces the stack address is correct, and that our stacks end empty allows us to skip operating on any stack segment during check_scan. Another benefit of this optimization is that we do not need to do a range check on the timestamp during push operations. In short, these optimizations provide significant speedups over a naive instantiation of a stack on top of a RAM segment. Figure 7 illustrates the final check_push and check_pop functions that we use in Coral. Our hash-input-packing optimization is also applied to stack operations.

Memory API. Our implementation of segmented memory allows developers to register a new memory segment by specifying its type (RAM, ROM, or Stack), visibility (public or private), and whether the segment should be persistent. Our library outputs a segment descriptor that can be used to access the memory segment inside the step function. Our memory infrastructure automatically inserts the appropriate checks based on the needs of the registered segments.

5.3. Incremental commitments to memory

The soundness of the memory checkers relies on $\mathcal{P}$ committing to a trace of all the memory accesses it plans to make *before* proving starts. This is to guarantee that $\mathcal{P}$ samples the random challenges c_0 and c_1 independent of the memory trace. However, to ensure that the memory operations in the commitment are the same as those provided by $\mathcal{P}$ during the actual proving steps, we need to somehow link the two (e.g., by opening the commitment inside R1CS).

Here two issues arise. First, we need to use a commitment scheme that supports *incremental* opening, so that the i-th step of proving only needs to open the commitment to the memory operations performed in that step. Second, existing commitment schemes (even those based on SNARK-friendly hashes like Poseidon [43]) incur high concrete costs in R1CS.

We describe how we address the first issue now, and defer to Appendix E the solution to the second issue.

Incremental commitments. Borrowing from prior works on commitment-carrying IVC [25, 59], $\mathcal{P}$ commits to its memory traces using an incremental commitment (IC) scheme, $\mathsf{IC}_{H,\mathsf{CM}}$ that is parameterized by a hash function H and a "chunk" commitment scheme CM. Roughly, the idea is to break up the trace of memory operations w into chunks $(w_1, \ldots, w_n)$, so that w_i consists of all the memory operations that were made in the i-th step of proving.

The memory operations for each step are the tuples in $\mathbf{IS}_{\mathsf{priv}}$ and **FS** of the form (t, a, v, s) processed during that step. These are followed, in order of application, by the tuples from **RS**, **WS**.

Then, each w_i is committed via CM, and then all these chunk-commitments are hashed via a hash-chain using H:

$$\mathsf{IC}_{H,\mathsf{CM}}.\mathsf{Commit}(\mathsf{pp}, w) = H(C_n, H(C_{n-1}, \ldots, H(C_1))),$$

where $C_i = \mathsf{CM}.\mathsf{Commit}(\mathsf{pp}, w_i)$ is the chunk-commitment to w_i. Coral instantiates H with Poseidon [43], and CM with a commitment scheme that is native to the underlying proof system that we use. Since we use Nova, which uses multilinear KZG commitments with a particular evaluation proof (the protocol is called "HyperKZG" [82]), CM is a variant of HyperKZG that we make hiding by adding a blind. Note that unlike the byte stream commitment C_{B} in Section 4.3, where we need the commitment, the polynomial evaluation, and the polynomial evaluation proofs to all be hiding and zero-knowledge, here we only need the commitment to be hiding; the evaluation and its proof need not be zero-knowledge because of the witness blinding technique [50] that we implement in Nova (Appendix C).

Incremental opening. To enforce commitment opening consistency, Coral recomputes the commitment inside R1CS as follows. The R1CS instance for the i-th step takes as input the running hash H_{i-1} as well as a commitment, C_i, to the i-th chunk w_{i-1} of memory operations. For each operation in the tuple, it enforces the checks in Section 5.1. Finally, it outputs the updated running hash $H_i = H(C_i, H_{i-1})$. At the end of proving all steps, $\mathcal{P}$ outputs the final running hash H_n, which should be equal to the existing commitment $\mathsf{IC}_{H,\mathsf{CM}}.\mathsf{Commit}(\mathsf{pp}, w)$. In the notation of Figure 2, H_i is IMC_i and $\mathsf{IC}_{H,\mathsf{CM}}$ is C_{Mem}.

Persistent memory. In Appendix E.1 we discuss the construction of *Split Witness Relaxed R1CS*, which makes it straightforward to share segmented/incremental memory between different proofs. Essentially, one can commit to any segment of **IS** or **FS** (or their entirety) separately, and then incrementally and verifiably access that segment across proofs. For example, one might use Coral to derive a parse tree of a JSON document, and export the segment of **FS** (that stores the parse tree) to another proof that proves facts about a particular field in the document. One caveat is that follow up proofs need to use the same chunk size, and the timestamp checks in check_mem_op and check_scan would change since it would no longer start at 0.

If the incremental memory commitments above are not appropriate for a given use case, Coral can export any segment of memory to a portable external commitment, similarly to how Coral handles the byte stream in Section 4.3.

6. Evaluation

In evaluating Coral, we answer three questions:

Q1. Is Coral fast enough to prove the parsing of real byte streams with real-world grammars on modest machines?
Q2. How does Coral compare to other approaches?
Q3. How does Coral scale with larger applications?

We answer these questions in the context of the following implementation, baselines, and experimental testbed.

Implementation. Coral is open source and consists of ≈12K lines of Rust: 4K for the core system [4], 5.1K for segmented memory [12], 1K to make Nova zero-knowledge [17], and 1.5K to implement commitment-carrying IVC and to merge SW-R1CS to R1CS in Nova [18]. We describe the details of commitment-carrying IVC and merging SW-R1CS to R1CS in Appendix E. Our modified version of Nova builds on Nova v0.41.0 [8]. We support both HyperKZG commitments [82] and Pedersen commitments; our experimental results presented are for HyperKZG. The generators for KZG come from the *Perpetual Powers of Tau* [9]. We write the `check_node` function in arkworks [24], which we then convert to Nova's R1CS format. To improve performance, instead of proving one `check_node` invocation per Nova step, we prove a *batch* of invocations. The batch size depends on the byte stream and grammar size and can be computed empirically or with a simple cost model.

Baselines. We compare against 3 baselines.

- *R0+Custom*. We run a specialized single-use parser (JSON, C, TOML) inside the RISC Zero zkVM [65]. The program returns a hash of the byte stream along with a boolean indicating whether it was parsed correctly.
- *R0+Pest*. We run `pest` inside the RISC Zero zkVM. This allows us to pass in a CFG grammar and a byte stream at runtime. Unlike the R0+Custom baseline, this is an apples-to-apples comparison with Coral because it can handle any CFG grammar (not specialized). This program also returns a hash of the document and a boolean indicating whether the document parsed correctly.
- *MHNM*. The work of Malvai et al. [57], which also uses an NP checker over a parse tree. Their code is unfortunately proprietary so we report the performance numbers in their paper. Their paper includes two implemented variants: MHNM-VOLE, which is the interactive and non-succinct version of their system (proofs are large and expensive to verify), and MHNM-zkSNARK, which is the non-interactive version with small proofs.

We note that there are many zkVMs other than RISC Zero, such as Jolt [26] and SP1 [52]. However, most of these systems are not suitable baselines for our evaluation: either they do not support zero-knowledge (e.g., Jolt only achieves succinctness), or only support it with high costs. For example, to achieve zero-knowledge, SP1 wraps its hash-based proofs in Groth16 proofs. This approach requires hundreds of seconds to prove the parsing of even our smallest JSON examples with the custom JSON parser, and is orders of magnitude slower than both RISC Zero and Coral.

Experimental setup. We tested Coral on a Lenovo Linux laptop, a large server, and a MacBook Pro. However, we discovered that RISC Zero has been extensively optimized to leverage hardware acceleration available on Apple silicon. To showcase each baseline under its most favorable conditions, we present the results on an MacBook Pro with Apple M3 Pro silicon (12 cores) and 18 GB of RAM. Coral does not have hardware acceleration; despite this, as we will show, Coral outperforms the R0+Pest and MHNM baselines by orders of magnitude, and also exceeds the performance of the R0+Custom baseline.

6.1. Applications

We evaluate Coral, R0+Custom, and R0+Pest on applications inspired by zk-TLS, zk-Authorization, and zk-Compilation systems. We use real grammars written in `pest` and real byte streams obtained from online sources. Note that we only perform the *parsing* portion of the applications (which is the crux of Coral); we do not implement the full application which may require proving additional facts.

JSON . We group *zk-TLS* and *zk-Authorization* together since they typically operate on JSON byte streams.

1) **Proof of Age or Identity.** A client proves in ZK to a 3rd party that they are over 18 or the resident of a certain state. The byte streams come from the Veratad API [16].
2) **Banking.** A client proves in ZK that they have some amount of money in their bank account or hold certain types of financial assets. The byte streams come from CitiBanks's API [3] and Plaid's API [11].
3) **Sports betting.** A client proves the score of a game to a smart contract (ZK is not necessary but succinctness is). The byte stream is from DraftKings' API [5].
4) **Account compromise.** A client proves that accounts associated with their email are not on a list of data breaches. We use the Have I Been Pwned API [45].
5) **JSON Web Token (JWT).** Existing systems [27, 67] prove facts in ZK about signed JWTs, but do not check that the JWT is well-formed. The byte streams are Google OAuth JWTs from Sui's zkLogin implementation [13].

The second set of applications relate to *zk-Compilation*: parsing C and TOML.

C. There is no existing `pest` grammar for C, so we wrote one for a subset of the language (the full grammar would take months to write). We evaluate four programs: three from The C Programming Language [48] and one from the LLVM test suite [6]. The programs contain data structures, loops, and function calls.

TOML. TOML is formally specified in Augmented Backus Naur Form [15] so we were able to translate it into `pest`. We parse the Cargo.toml file from the R0+Custom baseline (T1), Coral's own Cargo.toml file (T2), and the Cargo.toml of the arkworks `r1cs-std` library (T3) [24].

CFG	app	byte stream size	LCRS tree size	#R1CS	nodes per step	setup time (s)	solving time (s)	proving time (s)	verifying time (s)	proof size (kB)	memory (GB)
JSON	Age	748	2,594	212,606	305	3.117	0.181	2.458	0.040	17.194	0.970
	CitiBank	2,336	9,503	413,129	611	4.426	0.397	5.323	0.055	17.738	1.947
	Plaid	312	1,348	105,668	141	2.464	0.150	1.876	0.034	16.650	0.525
	Sports	1,547	7,263	431,620	641	4.497	0.321	4.400	0.049	17.738	1.985
	HIBP	191	869	102,183	136	2.374	0.135	1.470	0.032	16.650	0.505
	JWT	384	1,777	145,281	201	2.689	0.156	1.953	0.037	17.194	0.792
C Code	C1	56	241	89,382	87	2.370	0.119	1.208	0.031	16.650	0.471
	C2	237	1,173	141,229	155	2.872	0.151	2.154	0.046	17.194	0.764
	C3	399	2,027	211,320	242	3.152	0.172	2.583	0.041	17.194	0.949
	LLVM	1,431	6,652	455,274	543	5.194	0.309	5.920	0.070	18.282	2.386
TOML	T1	379	2,050	200,766	267	3.371	0.189	2.686	0.048	17.194	0.945
	T2	2,413	12,868	399,318	554	4.707	0.644	8.132	0.058	17.738	1.813
	T3	2,799	14,484	418,724	580	4.714	0.691	8.259	0.057	17.738	1.864

Figure 8: Summary of all costs for all applications evaluated in Coral. R1CS Constraints are for the step function in Nova. Times show the mean over 10 runs; standard deviation was under 5% of the mean. Proving time includes witness synthesis.

6.2. Can Coral support real applications?

To assess whether Coral can support real applications, we use it to parse byte streams for each application in Section 6.1 using their respective grammars. Figure 8 gives the full results, which we discuss below.

Step function size (# of R1CS constraints). Coral's step function requires $\mathcal{O}(n_s \cdot G_{\max})$ constraints where n_s is the number of nodes we process per step (i.e., the number of times we call `check_node` inside a step function), and $G_{\max}$ is the length of the longest rule in the grammar. On average, it takes $\approx$ 750 R1CS constraints to represent `check_node` for a single node.

Setup. This includes the generation of the public parameters in Nova for both the verifier and the prover, and the generation of the R1CS instance (§2.2). This is the most time-consuming step, but it only needs to be done once.

Solving. This includes parsing (with `pest`) to obtain a parse tree, the transformation to obtain the LCRS tree (§4.1), the generation of the memory trace (§5.1), and the commitment to the memory and the grammar.

Proving. This includes witness synthesis, folding, proving the satisfiability of the final (blinded) folded instance with a SNARK, and generating the KZG opening proof (π_{IPE}). Coral pipelines witness synthesis (i.e., finding a satisfying assignment to the R1CS instance in a given step) and folding: while the prover is folding step i, they are in parallel assigning the witness values for step $i+1$.

Verification. This includes verifying π_{fin} and π_{IPE}. The dominant cost is verifying π_{fin}, which requires computing 2 pairings, $\mathcal{O}(\#\text{R1CS})$ field operations, and a $\mathcal{O}(\log(\#\text{R1CS}))$-sized MSM.

Proof size. This represents all of the materials given to $\mathcal{V}$ in order to verify $\mathcal{P}$'s claim. This includes π_{fin}, π_{IPE}, the final incremental polynomial evaluation of the leaves $\mathsf{IPE}_{\mathsf{fin}}$, and the commitment to the memory C_{Mem}. All of these are single group elements, except for π_{IPE} which is a group element and a field element, and π_{fin} which contains $\mathcal{O}(\log(\#\text{R1CS}))$ group elements.

Persistent segment extraction (not shown in the table). In the above applications, Coral only proves that the LCRS parse tree T is valid for the CFG $\mathcal{G}$ and byte stream B. However, Coral is designed to be composed with proofs of other statements, as part of a larger application. This is where persisting a segment comes into play: it allows other proofs and proof systems that use our segmented memory construction to continue accessing the persistent segment's data. We discuss this in Step 7 of Figure 2 and in Section 5.3.

With this feature enabled, Coral can persist the segment containing T so that other proofs can use T with the assurance that it is valid. One key benefit of our approach is that the cost to persist a segment is independent of the size (or amount of data) in that segment—it is constant overhead. In particular, persisting a segment requires 1 extra Poseidon hash and 1 extra group scalar multiplication, which results in $\approx$5K constraints that need to be added to Nova's *secondary circuit*. For context, Nova works over a cycle of curves, each of which operates on a different circuit: a primary circuit which contains the step function's R1CS, and a secondary circuit that implements a verifier for the folding operation.

The concrete cost to persist a segment is an additional $\approx$0.5 sec of proving time. The main source of this cost is that it makes Nova's secondary circuit go from 11.5K constraints to 16.5K constraints, which causes Nova to pad this circuit to the next power of 2 (32K).

Takeaway. We find that Coral is able to prove all of the applications we tested in a few seconds on a standard laptop using under 4 GB of memory. The proof sizes are small (under 18.3 KB) and take between 31–70 ms to verify. We therefore conclude that Coral can efficiently support applications of practical interest.

6.3. How does Coral compare to alternatives?

Having established that Coral can viably support the aforementioned applications, we now investigate whether existing systems can do the same. We compare each application against the R0+Custom and R0+Pest baselines, but not against the MHNM baselines, as their code is not open source and their paper reports only scalability results on

synthetic circuits of varying sizes; we provide comparisons for the latter in Section 6.4.

Results. Figure 9 shows Coral's *total prover time* (i.e. solving + proving) in relation to both R0+Custom and R0+Pest.

Compared to R0+Pest, which has the same features as Coral, the results are overwhelming: Coral's total proving time is orders of magnitude faster (we even had to truncate the figure). Compared to R0+Custom, Coral is faster across all applications, and significantly faster for TOML. This is despite the fact that RISC Zero uses hardware acceleration (whereas Coral does not) and R0+Custom is optimized for specific grammars (whereas Coral is for any CFG).

Takeaway. Of the existing approaches, R0+Pest is not viable whereas R0+Custom is, but its performance is strictly worse than Coral's across all applications. Beyond performance, there is a qualitative benefit to using Coral. Trusting a proof from Coral requires: (1) that Coral's NP checker is sound (which we prove in Appendix B); (2) that its implementation in R1CS is sound; and (3) that the commitment scheme and underlying zkSNARKs that we use are sound. In contrast, trusting a proof from RISC Zero requires: (1) that the zkVM is sound; (2) that the parser running on the zkVM (considerably more complex than our simple linear-time NP checker) is bug-free; and (3) that the underlying zkSNARK and commitments are sound. As such, Coral's use of NP checkers not only improves performance but also reduces the trusted computing base because the NP checker is fundamentally simpler than a full parser. Coral's checker is also general and applies to all CFGs.

6.4. How does Coral scale?

The prior sections show that Coral works well on existing applications, which all happened to have relatively small byte streams. We now ask: what happens to Coral (and the baselines) when the byte stream gets larger? To study this question we run a scalability test using the JSON grammar on byte streams ranging from 64 B to 64 kB.

We compare with R0+Custom, R0+Pest, and both variants of the MHNM baseline by using the numbers they report in Figures 10 and 12 of their paper [57]. MHNM evaluate their system using a grammar that has fewer rules than a real JSON grammar and on a server with 32 vCPUs and 64 GB of RAM. We therefore feel that this is a very generous comparison, since Coral runs on a full JSON grammar and on a laptop with 12 vCPUs and 18 GB of RAM.

Results. Figure 10 shows the results of our scalability experiment. In comparison to the RISC Zero baselines, the results echo the experiments in our applications: Coral is significantly more efficient than R0+Pest. Coral is also between 1.15× and 2.5× faster than R0+Custom (it might not seem this way but notice that the y-axis of the graph is log-scale). In terms of memory usage, Coral requires slightly more memory than R0+Custom, particularly for smaller sized documents. However, its memory usage is still low enough for Coral to run on a standard iPhone or laptop. Additionally, in our experiments we optimized for prover time. If a user were willing to accept a longer proving time, they could reduce the memory usage by using a smaller batch size.

Compared to both MHNM baselines, Coral is an order of magnitude faster. This is quite significant since MHNM's VOLE baseline is interactive and has large proofs that are expensive to verify. This demonstrates that despite VOLE approaches being widely believed to offer lower proving time compared to SNARKs, Coral's checker, use of a folding scheme, and the introduction of segmented memory lead to a very efficient and high performing system.

Takeaway. Coral and the baselines all exhibit a similar scaling behavior as the size of the byte stream increases, although Coral's concrete costs are generally the lowest. For the applications that we tested (§6.1), Coral's performance is acceptable. But if one wishes to use Coral for very large byte streams (>32 kB), then additional ideas will be needed since Coral's proving times start to approach two minutes. For example, Coral could use hardware acceleration, which it currently lacks, and which has been shown in some early tests to improve Nova's performance by ≈ 5× [1]. Separately, we could specialize Coral's checker to a particular grammar, in much the same way that R0-Custom works. This would obviously mean that we would have to remove claims of grammar generality, but it might yield considerable speedups.

7. Conclusion

Coral is a practical system for proving in zero-knowledge that the parsing of a secret byte stream is consistent with a public grammar. Coral's proofs are small and very cheap to verify, and Coral's prover can run in seconds on a laptop. Compared to recent alternatives, Coral is significantly more efficient. Coral allows us to no longer rely on the assumption that byte streams are correctly formatted in the first place, which is crucial to the future of a variety of ZK applications.

Acknowledgments

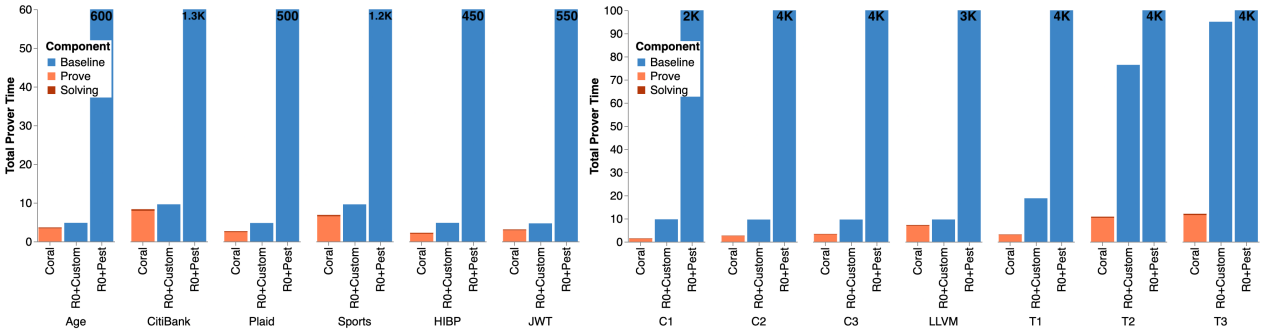
We thank the S&P 2026 reviewers and shepherd for their thorough comments on our work. We also thank Srinath Setty for help with Nebula and the Nova codebase, and Steve Zdancewic for some pointers on CFGs. This work was funded in part by NSF Award CNS-2045861, and gifts from Sony, Sui, Ethereum Foundation, Ingonyama, and the Arcological Swiss Association.

Code Availability

https://github.com/eniac/coral

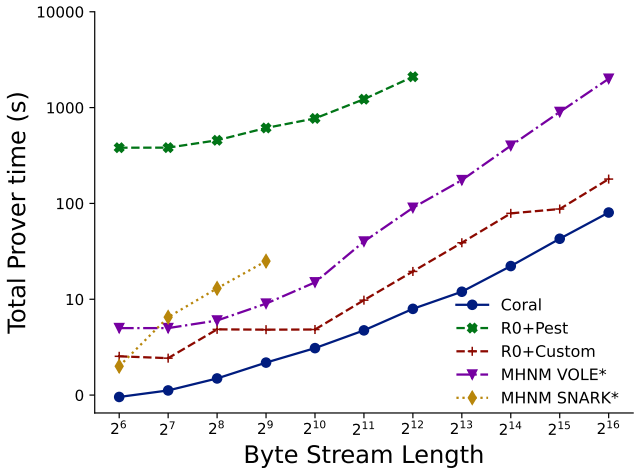
References

[1] Add GPU acceleration to bn256_grumpkin. https://github.com/microsoft/Nova/pull/374.
[2] Circom. https://github.com/iden3/circom.
[3] Citibank consumer apis. https://developer.citi.com/.
[4] Coral: Fast succinct non-interactive zero-knowledge CFG proofs. https://github.com/eniac/coral.

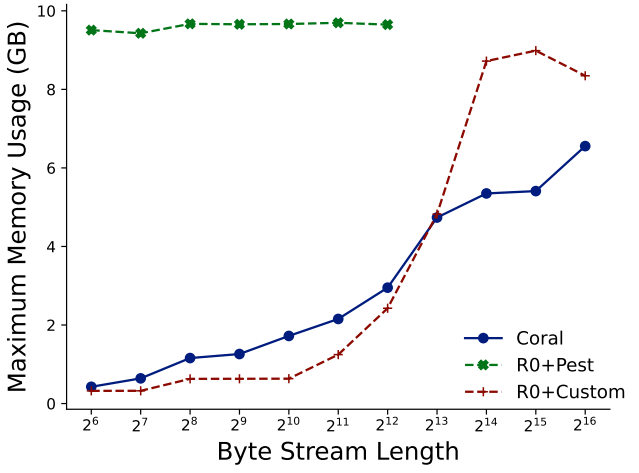


(a) JSON (b) C and TOML

Figure 9: Comparison of Proving Time for Coral and baselines for proving the parsing of a variety of applications and grammars. R0+Pest time has been truncated since it is orders of magnitude slower than the other approaches.



(a) Proving Time



(b) Memory

Figure 10: Time and resource use of Coral and baselines for proving the parsing of JSON byte streams. *Results are for a system running on 32 CPUs and are extracted from their paper [57]; Coral and RISC Zero run on 12 CPUs.

[5] DraftKings. https://www.draftkings.com/.

[6] Llvm test suite. https://github.com/llvm/llvm-test-suite.

[7] The Noir programming language. https://github.com/noir-lang/noir.

[8] Nova: Recursive SNARKs without trusted setup. https://github.com/microsoft/Nova.

[9] Perpetual Powers of Tau. https://github.com/privacy-scaling-explorations/perpetualpowersoftau.

[10] Pest: The elegant parser. https://pest.rs/.

[11] Plaid. https://plaid.com/.

[12] Segmented memory for R1CS. https://github.com/eniac/segmented-memory.

[13] Sui zkLogin Demo. https://sui-zklogin.vercel.app/.

[14] A thoughtful introduction to the pest parser: non-atomic rules. https://pest.rs/book/grammars/syntax.html#non-atomic.

[15] Toml: Tom's obvious minimal language. https://toml.io/en/.

[16] Veratad. https://veratad.com/.

[17] Zero knowledge fold for nova ivc from hypernova.

[18] zkNova with CC-IVC. https://github.com/jkwoods/Nova/.

[19] ZoKrates: A toolbox for zkSNARKs on ethereum. https://github.com/Zokrates/ZoKrates.

[20] TLS Notary, 2023. https://tlsnotary.org/.

[21] Zk email verify. https://github.com/zkemail/zk-email-verify, 2023.

[22] S. Angel, A. J. Blumberg, E. Ioannidis, and J. Woods. Efficient representation of numerical optimization problems for SNARKs. In *Proceedings of the USENIX Security Symposium*, 2022.

[23] S. Angel, E. Ioannidis, E. Margolin, S. Setty, and J. Woods. Reef: Fast succinct non-interactive zero-knowledge regex proofs. In *Proceedings of the USENIX Security Symposium*, 2024.

[24] arkworks contributors. `arkworks` zksnark ecosystem. https://arkworks.rs, 2022.

[25] A. Arun and S. Setty. Nebula: Efficient read-write memory and switchboard circuits for folding schemes. Cryptology ePrint Archive, Paper 2024/1605, 2024.

[26] A. Arun, S. T. V. Setty, and J. Thaler. Jolt: Snarks for virtual machines via lookups. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2024.

[27] F. Baldimtsi, K. K. Chalkias, Y. Ji, J. Lindstrøm, D. Maram, B. Riva, A. Roy, M. Sedaghat, and J. Wang. zklogin: Privacy-preserving blockchain authentication with existing credentials. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[28] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In *Proceedings of the conference on Innovations in Theoretical Computer Science*, 2013.

[29] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2013.

[30] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the USENIX Security Symposium*, 2014.

[31] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991.

[32] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[33] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[34] S. Celi, A. Davidson, H. Haddadi, G. Pestana, and J. Rowell. Distefano: Decentralized infrastructure for sharing trusted encrypted facts and nothing more. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2025.

[35] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2020.

[36] L. Eagen, D. Fiore, and A. Gabizon. cq: Cached quotients for fast lookups. Cryptology ePrint Archive, Paper 2022/1763, 2022.

[37] Z. Fang, D. Darais, J. P. Near, and Y. Zhang. Zero knowledge static program analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2021.

[38] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2004.

[39] M. Frigo and abhi shelat. Anonymous credentials from ECDSA. Cryptology ePrint Archive, Paper 2024/2010, 2024.

[40] A. Gabizon and Z. J. Williamson. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Paper 2020/315, 2020.

[41] A. Gabizon and Z. J. Williamson. Proposal: The turbo-PLONK program syntax for specifying SNARK programs. https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-turbo_plonk.pdf, 2020.

[42] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2013.

[43] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In *Proceedings of the USENIX Security Symposium*, 2021.

[44] P. Grubbs, A. Arun, Y. Zhang, J. Bonneau, and M. Walfish. Zero-knowledge middleboxes. In *Proceedings of the USENIX Security Symposium*, 2022.

[45] T. Hunt. Have i been pwned? https://haveibeenpwned.com/.

[46] K. Jiang, D. Chait-Roth, Z. DeStefano, M. Walfish, and T. Wies. Less is more: refinement proofs for probabilistic proofs. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2023.

[47] A. Kate, G. M. Zaverucha, and I. Goldberg. 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)*, 2010.

[48] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice-Hall, Inc., USA, 1978.

[49] D. Knuth. *The Art of Computer Programming, Vol. I: Fundamental Algorithms*. Addison-Wesley, Reading, MA, 1968.

[50] A. Kothapalli and S. Setty. HyperNova: recursive arguments for customizable constraint systems. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2024.

[51] A. Kothapalli, S. Setty, and I. Tzialla. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2022.

[52] S. Labs. SP1. https://github.com/succinctlabs/sp1.

[53] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers*, 100(9):690–691, 1979.

[54] J. Lauinger, J. Ernstberger, A. Finkenzeller, and S. Steinhorst. Janus: Fast privacy-preserving data provenance for TLS. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, 2025.

[55] L. Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *Journal of the ACM (JACM)*, 49(1), 2002.

[56] N. Luo, C. Weng, J. Singh, G. Tan, R. Piskac, and M. Raykova. Privacy-preserving regular expression matching using nondeterministic finite automata. Cryptology ePrint Archive, Paper 2023/643, 2023.

[57] H. Malvai, S. Hussain, G. Neven, and A. Miller. Practical proofs of parsing for context-free grammars. Cryptology ePrint Archive, Paper 2024/562, 2024.

[58] D. Naylor, R. Li, C. Gkantsidis, T. Karagiannis, and P. Steenkiste. And then there were more: Secure communication for more than two parties. In *Proceedings of the International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2017.

[59] W. D. Nguyen, T. Datta, B. Chen, N. Tyagi, and D. Boneh. Mangrove: A scalable framework for folding-based snarks. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2024.

[60] A. Ozdemir, F. Brown, and R. S. Wahby. Circ: Compiler infrastructure for proof systems, software verification, and more. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2022.

[61] A. Ozdemir, E. Laufer, and D. Boneh. Volatile and persistent memory for zksnarks via algebraic interactive proofs. In *Proceedings of the USENIX Security Symposium*, pages 3309–3327, 2025.

[62] A. Ozdemir, R. Wahby, B. Whitehat, and D. Boneh. Scaling verifiable computation using efficient set accumulators. In *Proceedings of the USENIX Security Symposium*, 2020.

[63] T. J. Parr and R. W. Quong. Adding semantic and syntactic predicates to LL(k): pred-LL(k). In *Proceedings of the International Conference on Compiler Construction*, 1994.

[64] M. Raymond, G. Evers, J. Ponti, D. Krishnan, and X. Fu. Efficient zero knowledge for regular language. In *Proceedings of the EAI Conference on Security and Privacy in Communication Networks*, 2023.

[65] RISC ZERO. https://www.risczero.com/.

[66] M. Rosenberg, T. Mopuri, H. Hafezi, I. Miers, and P. Mishra. Hekaton: Horizontally-scalable zksnarks via proof aggregation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2024.

[67] M. Rosenberg, J. White, C. Garman, and I. Miers. zk-creds: Flexible anonymous credentials from zkSNARKs and existing identity infrastructure. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2023.

[68] S. Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2020.

[69] S. Setty, S. Angel, T. Gupta, and J. Lee. 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.

[70] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2013.

[71] S. Setty, J. Thaler, and R. Wahby. Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, Paper 2023/552, 2023.

[72] S. Setty, J. Thaler, and R. Wahby. Unlocking the lookup singularity with lasso. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2024.

[73] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *Proceedings of the USENIX Security Symposium*, 2012.

[74] T. Solberg. A brief history of lookup arguments. https://github.com/ingonyama-zk/papers/blob/main/lookups.pdf, 2023.

[75] P. Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Proceedings of the Theory of Cryptography Conference (TCC)*, pages 552–576, 2008.

[76] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[77] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine

learning. In *Proceedings of the USENIX Security Symposium*, 2021.
[78] X. Xie, K. Yang, X. Wang, and Y. Yu. Lightweight authentication of web data via garble-then-prove. In *Proceedings of the USENIX Security Symposium*, 2024.
[79] C. Zhang, Z. DeStefano, A. Arun, J. Bonneau, P. Grubbs, and M. Walfish. Zombie: Middleboxes that don't snoop. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024.
[80] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.
[81] F. Zhang, D. Maram, H. Malvai, S. Goldfeder, and A. Juels. DECO: Liberating web data using decentralized oracles for TLS. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2020.
[82] J. Zhao, S. Setty, W. Cui, and G. Zaverucha. MicroNova: Folding-based arguments with efficient (on-chain) verification. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2025.
[83] L. Zhao, Q. Wang, C. Wang, Q. Li, C. Shen, and B. Feng. VeriML: Enabling integrity assurances and fair payments for machine learning as a service. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

Appendix A. Atomic rules and negative predicates

An important design goal of Coral is to support modern syntactic constructs. A key aspect of this is enabling support for *non-atomic rules* and *exclusion rules*.

Atomic rules control whether implicit whitespace is inserted during parsing. In Pest syntax, a reserved rule named `WHITESPACE` defines a pattern that, if present, is implicitly allowed between every sequence element and repetition [10]. Marking a rule as *atomic* disables this behavior, ensuring that no implicit whitespace is inserted.

Exclusion rules provide a convenient shorthand for disallowing specific characters or strings within a rule. For example, if we want to define a rule stating that r_1 matches any alphanumeric character *except* `'a'`, we could explicitly enumerate all acceptable characters: $r_1 \rightarrow \mathsf{b}|\mathsf{c}|\ldots$, or we could leverage the exclusion rule and write $r_1 \rightarrow$ `(!'a')ALPHANUMERIC`.

Beyond supporting expressive syntax, both atomic and exclusion rules introduce non-trivial technical challenges. Both of these rules are recursive: any rules called by an atomic or exclusive rule are themselves treated as atomic or exclusive, even if they are not defined as such. At any point in our parse tree, we need to keep track of whether an ancestor node was atomic or exclusive. For this, we keep track of both the node where the exclusion or atomic flag was invoked, as well as of its parent. This allows us to detect when we have exited the subtree rooted at the atomic or exclusion node.

Furthermore, while an exclusion rule can theoretically be rewritten as an explicit enumeration of all permitted characters, this expansion would significantly increase memory usage and proof size. To avoid this, we augment the grammar rule table with a *exclusion table*, where each row has the form: $[\texttt{ruleID}, \texttt{exclusion}_1, \texttt{exclusion}_2, \ldots]$ When `check_node` encounters a leaf within an exclusion subtree, it checks the symbol of that leaf against the corresponding exclusion table. Since these lists are typically short, we perform this check efficiently by constructing a *vanishing polynomial* over the set of excluded characters.

Appendix B. Correctness of `check_node`

This section provides the soundness and completeness proofs for Coral's NP checker (Figure 11), which iteratively invokes the `check_node` step function (Figure 4). Readers may note that the arguments passed to `check_node` in Figure 11 differ from those in Section 4.5. We make the inclusion of T and $\mathcal{G}$ explicit (since they are used in `check_node` by `get_node()` and `lookup()`, respectively), but condense the non-deterministic hints provided to the NP checker into the parameter aux.

Here we only reason about Coral's NP checker, not the end-to-end Coral system. As a result, we will not reason about commitments, so we will ignore the blind r_{B}^1 that is part of aux. We prove completeness, soundness, and zero-knowledge for all of Coral (which includes reasoning about various commitments, zkSNARKs, and the memory accesses) in Appendix F.

```
np_checker(T, B, G, aux) {
  id = 0;
  rule_stack = empty_stack();
  tran_stack = empty_stack();
  ipe = 0;
  while id < T.size() {
    id, ipe, rule_stack, tran_stack =
      check_node(id, ipe, rule_stack, tran_stack,
                 T, G, aux);
    }
  assert!(rule_stack.is_empty());
  assert!(tran_stack.is_empty());
  assert!(id == T.size());
  assert!(ipe == B);
  return true;
}
```

Figure 11: Coral's NP Checker, which iteratively calls the `check_node` step function from Figure 4.

Definition B.1 (LCRS Tree)**.** An LCRS tree T is a binary tree with the following properties:

- All nodes of T have either a single left child, a single right sibling, both a left child and right sibling, or neither.
- Nodes that do not have a left child are *logical leaves*.
- Given a node $n \in T$, n's *logical children* consist of its left child ℓ, in addition to all nodes that can be reached by taking only right sibling edges starting from ℓ.

Definition B.2 (Valid LCRS Parse Tree)**.** Let $\mathcal{G} = (V, \Sigma, R, S)$ be a CFG. An LCRS Tree T is a valid parse tree for $\mathcal{G}$ and byte stream B, if it satisfies:

- All logical leaves are terminal symbols $\in \Sigma$.
- All logical non-leaves are non-terminal symbols $\in V$.

- The logical leaves, when concatenated in the order they are encountered under a pre-order DFS traversal of T form B.
- Every logical non-leaf node and its logical children form valid production rules in R.

Definition B.3 (Coral NP Relation)**.** Let $\mathcal{G}$ be a public CFG. The NP relation for Coral, denoted $\mathcal{R}$, is the set of instance-witness pairs (x, w) where the instance is $x = (\mathcal{G})$ and the witness is $w = (T, \mathsf{B})$ such that B is a byte stream and T is a valid LCRS parse tree with respect to $\mathcal{G}$ and B.

Theorem B.1 (Soundness)**.** *Given a CFG $\mathcal{G}$, if* `np_checker` *accepts on inputs $T, \mathsf{B}, \mathcal{G}$ and non-deterministic hint aux, then $x = (\mathcal{G})$ and $w = (\mathsf{B}, T)$ is a valid instance-witness pair for* Coral*'s NP relation (Definition B.3).*

Proof. We show that `np_checker` is sound for Coral's NP Relation via the following lemmas.

1) Lemma B.1 proves that the input to `np_checker` is a valid LCRS tree T (Definition B.1).
2) Lemma B.2 proves that `np_checker` performs a pre-order DFS traversal of the nodes in T.
3) Lemma B.3 proves that T is a valid LCRS parse tree for B and $\mathcal{G}$ (Definition B.2).

Lemma B.1 (LCRS tree is valid)**.** *Given purported tree T, if* `np_checker` *accepts, then T is a a valid LCRS tree.*

Proof. Assume that the size of T, denoted $|T|$, is known. `np_checker` maintains a counter for each node it is run on. This counter enforces that each node it visits has a unique ID, by requiring that the nodes are assigned IDs sequentially based on a pre-order DFS traversal. When `np_checker` traverses T, for a node n with id i, `check_node` enforces that either n's left child or right sibiling are non-null. If the left child is non-null, then `np_checker` traverses to that child. If the right sibiling is non-null, then `check_node` pushes it onto `tran_stack`. If both are null, then `np_checker` asserts that its counter is equal to $|T|$ and terminates.

`check_node` only allows for one left child and one right sibling per node. No other potential children or siblings are accessible, so they are never considered part of T.

□

Lemma B.2 (Node traversal is DFS)**.** *Assume that* `np_checker` *accepts and by Lemma B.1 let T be the resulting valid LCRS tree. Then,* `np_checker` *visits the nodes in T following a DFS traversal order O.*

Proof. Consider a particular node k in T. First, `check_node` enforces that left children are always prioritized (i.e., visited before siblings). Specifically, `check_node` selects the next node in the traversal order to be the left child of k, if one exists. Second, all siblings are visited in left-to-right order. If node k has a right sibling, it is pushed onto `tran_stack`. Since left children are prioritized, k's right sibling will be popped from `tran_stack` and processed only *after* k's left child (and its entire subtree) has been processed (i.e., a logical leaf is reached).

Now we consider the edge cases. In the first step, `check_node` takes as public input the root `node_id`, and an empty transition stack. Since `check_node` always selects the next node in the order to be either the current node's child or a node popped from the stack, all successive nodes after the first node must be descendants of the first node. `check_node` enforces the uniqueness of each `id`, so there are no cycles. Even if ancestors of the first node exist, they are never seen and this node is functionally the root.

After the last step, the transition stack is confirmed to be empty, so no unprocessed nodes can remain in the stack. This means that during the last `check_node` step, the final logical leaf (which should have no right sibling) is processed. □

Lemma B.3 (Valid parse tree)**.** *Assume that* `np_checker` *accepts, and by Lemmas B.1 and B.2 T is a valid LCRS tree with a corresponding pre-order DFS traversal O, then T is a valid LCRS parse tree for $\mathcal{G}, \mathsf{B}$.*

Proof. Let the *logical parent* of a node k refer to the node of which k is a logical child (Definition B.1). Given node k, `check_node` enforces that the symbol of k matches the symbol popped from `rule_stack`. This symbol was pushed onto the stack by k's logical parent.

If k is a left child, the correct corresponding symbol will be at the top of the `rule_stack`, as its logical parent, node $k-1$, pushed that symbol immediately before transitioning to k. Lemma B.2 assures us left children are prioritized.

Now consider the case where k is a right sibling. Assume, inductively, that all rules corresponding to its logical parent's previous children (and their descendants) have been fully processed before k is reached. That is, the symbols pushed for them have already been matched and popped from the `rule_stack`. Since the logical children of a node are visited in left-to-right order (Lemma B.2), the symbol for each child will rise to the top of the `rule_stack` in the correct order. Thus, by the time k is processed, its corresponding symbol will be at the top of the `rule_stack`. This argument holds recursively: for each child of the logical parent, their descendants are processed in DFS order, and their symbols are pushed and popped from the `rule_stack` before the next child is processed. Hence, the invariant that the top of the `rule_stack` matches the current node's symbol is preserved throughout the traversal.

Each symbol is pushed with a boolean flag indicating if it is the last symbol in its production rule. `check_node` enforces that if a symbol is not the last in its rule, the corresponding node k must have a right sibling, which will be processed next in the traversal order (guaranteed by Lemma B.2).

This mechanism is critical for security and it prevents a traversal in which a rule is started from one node and finished with the logical children of another, earlier, unrelated node. This would require rewinding the `rule_stack` to a prior state. However, because the traversal is a strict Depth-First Search, the Last-In-First-Out nature of the `rule_stack` ensures that all symbols pushed by a node's descendants must be popped before the stack returns to the state it was in when the node's rule was first pushed. Therefore, a rule pushed by k's logical parent can only be fully popped and completed

after all of its logical children (and their descendants) have been successfully verified and popped, culminating with the last sibling at k's level.

If k itself is not a logical leaf (as leaves do not generate new rules to check), then the correct grammar rule to check against k's logical children is provided as a non-deterministic hint `proposed_sym`, which is part of aux. `check_node` enforces that this grammar rule does indeed come from the publicly available grammar $\mathcal{G}$. The symbols (and boolean flags) for this grammar rule are pushed onto the `rule_stack` in reverse order (later to be popped and confirmed to match T's symbols).

Now we consider the edge cases. In the first step, `check_node` does not pop from the `rule_stack` for its symbol check. It instead just enforces that the symbol for the first node is a special rule *root*. After the last step, the `rule_stack` can be confirmed to be empty by checking that the stack pointer (which is part of the public output) is equal to the lower bound of the stack segment. This implies that there can be no unverified rules.

`check_node` also accumulates symbols seen in logical leaves as it encounters them. `check_node` enforces a counter that starts at 0 and increments after every accumulation, so leaves are tied to the order in which they are seen. The equality check at the end of `np_checker` establishes that the byte stream B is exactly the concatenation of the symbols of the logical leaves of T, in the order they are visited. □

This completes the proof of Theorem B.1.

□

Theorem B.2 (Completeness)**.** *If $x = (\mathcal{G})$ and $w = (\mathsf{B}, T)$ is a valid instance-witness pair for* Coral*'s NP relation (Definition B.3), then there exist a non-deterministic hint aux such that* `np_checker` *accepts on inputs T, B, $\mathcal{G}$, aux. Moreover, the non-deterministic hint $proposed\_sym$, which is a component of aux, can be computed efficiently from T.*

Proof. It is straightforward to do a DFS traversal of T to assign incrementally increasing node ids to all nodes, as well as initialize the empty `rule_stack` and `tran_stack`.

Since T respects $\mathcal{G}$, all logical non-leaves form a production rule with their logical children. When running `np_checker`, these nodes will each push a valid rule from $\mathcal{G}$ onto the `rule_stack`, that will be correctly verified (in reverse order) by its children. See Lemma B.3. All logical leaves in T are terminal symbols, which will be correctly accumulated in a pre-order DFS traversal during `np_checker` to form B.

For each node n in T, the non-deterministic hint $proposed\_sym$ is the list of symbols which form the right hand side of the production rule where n's symbol is the left hand side.

This can be computed in polynomial time as follows: For each node n in T, if n has a left child add it to n's proposed symbols. Then, traverse n's left child's right siblings and add each of them to n's proposed symbols until a node with no right sibling is reached. □

Appendix C. Witness blinding in Nova

The starting point for Coral's proof system is the open source implementation of Nova [8]. This implementation contains an (non-ZK) IVC that is based on the folding scheme for relaxed R1CS that is described in the Nova paper [51]. The IVC proof from the folding scheme is then compressed using a non-ZK SNARK that is based on Spartan [68] (but that is different from the open source implementation of Spartan which is zero-knowledge but is not used by Nova).

In this section, we explain how we make Nova's code base zero-knowledge. Our changes have been merged to the open source implementation of Nova and are available for all to use. The key mechanism that we implement is the *witness randomization technique* described in Appendix D of HyperNova [50]. At a high level, this requires: (1) making the commitments supported by Nova (non-hiding Pedersen and HyperKZG) hiding; (2) adding a folding of a random relaxed R1CS instance into the existing folding instance; and (3) "deblinding" the commitments after the randomized folding so that we can continue to use the non-ZK version of Spartan that is implemented in Nova.

The key idea behind these changes is that adding a random relaxed R1CS instance effectively hides the witness. This ensures that the prover could, if desired, reveal the folded randomized witness to the verifier. Of course, to reduce the communication costs and the computation that the verifier needs to do in order to verify this witness, we instead compress the (non-private) randomized witness with a SNARK, which yields succinct verification for the verifier.

In this section we describe the details of these 3 changes.

C.1. Committed relaxed R1CS

We first describe the committed relaxed R1CS relation from Nova [51] (Definition 12). We denote Δ as a commitment scheme over finite field $\mathbb{F}$. A committed relaxed R1CS structure is defined by sparse matrices $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{F}^{m \times m}$. A committed relaxed R1CS witness is defined by a witness vector $W \in \mathbb{F}^{m-\ell-1}$, an error vector $E \in \mathbb{F}^m$, and commitment blinds $(r_W \in \mathbb{F}, r_E \in \mathbb{F})$. A committed relaxed R1CS instance is defined by commitments CM_W to the witness vector and CM_E to the error vector, a scalar $u \in \mathbb{F}$, and a public input/output vector $x \in \mathbb{F}^\ell$, where $\ell < m$.

A witness *satisfies* an instance (i.e. the structure, instance, and witness are in the R1CS relation) if and only if:

$$\begin{aligned}
&\mathsf{CM}_W = \Delta.\mathsf{Commit}(\mathsf{pp}_W, W, r_W)\\
&\mathsf{CM}_E = \Delta.\mathsf{Commit}(\mathsf{pp}_E, E, r_E)\\
&(\boldsymbol{A} \cdot \boldsymbol{Z}) \circ (\boldsymbol{B} \cdot \boldsymbol{Z}) = u \cdot (\boldsymbol{C} \cdot \boldsymbol{Z}) + E,\\
&\text{where } \boldsymbol{Z} = (W, x, u)
\end{aligned}$$

Here, pp_E and pp_W are the commitment parameters for vectors of size m and $m - \ell - 1$. Notice that an R1CS instance can be expressed as a relaxed R1CS instance by setting $u = 1, E = 0$.

C.2. Folding Scheme for committed relaxed R1CS

The folding scheme requires that Δ be a succinct and homomorphic commitment scheme over the finite field $\mathbb{F}$. For a witness W, the verifier takes two committed relaxed R1CS instances:

$$(\mathsf{CM}_E, u, \mathsf{CM}_W, x)$$
$$(\mathsf{CM}_{E'}, u', \mathsf{CM}_{W'}, x')$$

The prover takes the two instances and (additionally) witnesses that satisfy both instances:

$$(E, r_E, W, r_W)$$
$$(E', r_{E'}, W', r_{W'})$$

The prover and verifier take the following steps in order to produce a new folded instance and witness:

1) $\mathcal{P}$: Send $\mathsf{CM}_T = \Delta.\mathsf{Commit}(\mathsf{pp}_E, T, r_T)$, where:

$$r_T \leftarrow_R \mathbb{F}$$
$$T = \boldsymbol{A}Z \circ \boldsymbol{B}Z' + \boldsymbol{A}Z' \circ \boldsymbol{B}Z - u \cdot \boldsymbol{C}Z' - u' \cdot \boldsymbol{C}Z$$
$$Z = (W, x, u), Z' = (W', x', u')$$

2) $\mathcal{V}$: Sample and send challenge $r \leftarrow_R \mathbb{F}$
3) $\mathcal{P}, \mathcal{V}$: Output folded instance:

$$\mathsf{CM}_{E_{new}} \leftarrow \mathsf{CM}_E + r \cdot \mathsf{CM}_T + r^2 \cdot \mathsf{CM}'_E$$
$$u_{new} \leftarrow u + r \cdot u'$$
$$\mathsf{CM}_{W_{new}} \leftarrow \mathsf{CM}_W + r \cdot \mathsf{CM}_{W'}$$
$$x_{new} \leftarrow x + r \cdot x'$$

4) $\mathcal{P}$: Output folded witness:

$$E_{new} \leftarrow E + r \cdot T + r^2 \cdot E'$$
$$r_{E_{new}} \leftarrow r_E + r \cdot r_T + r^2 \cdot r_{E'}$$
$$W_{new} \leftarrow W + r \cdot W'$$
$$r_{W_{new}} \leftarrow r_W + r \cdot r_{W'}$$

C.3. Producing a zero-knowledge SNARK.

During compression, Spartan interprets the commitments to vectors as commitments to multilinear polynomials and uses an evaluation argument to prove evaluations of the committed polynomials. A straightforward way to make the Nova implementation zero-knowledge would be to make the commitment schemes hiding, and then modify the current Spartan implementation in Nova (which is not ZK) to produce a zkSNARK. However, this requires the Spartan verifier to perform group scalar multiplications. There is a cheaper to achieve this, as detailed in HyperNova: the zero-knowledge folding layer for IVC (see HyperNova Appendix D.4), which is significantly simpler. Its description is reproduced (with some clarifications) here.

Zero-knowledge folding layer, then compression We imagine the prover first produces (or otherwise receives) a Nova IVC proof $\Pi = (\mathsf{U}_i, \mathsf{u}_i, \mathsf{W}_i, \mathsf{w}_i, r_i)$, that proves a statement (i, z_0, z_i), with a verifier key vk. The instance-witness pair $(\mathsf{U}_i, \mathsf{W}_i)$ is the running pair (that has had some arbitrary number of instance-witness pairs folded into it), and the pair $(\mathsf{u}_i, \mathsf{w}_i)$ is the latest pair (that deals with a "real" witness—it should be a true non-relaxed R1CS instance). The randomness r_i is included here for consistency with the original Nova construction. Instead of taking whole instances/statements as input to IVC proofs, Nova uses a hash to compress them. r_i serves to make these hashes hiding commitments. Each committed relaxed R1CS instance (U_i and u_i) corresponds to the tuple $(\mathsf{CM}_E, u, \mathsf{CM}_W, x)$ above, and each committed relaxed R1CS witness (W_i and w_i) corresponds to the tuple (E, r_E, W, r_W). The prover produces the randomized IVC proof Π' with the following steps.

1) Fold $(\mathsf{U}_i, \mathsf{W}_i)$ with $(\mathsf{u}_i, \mathsf{w}_i)$ to produce the new folded instance-witness pair $(\mathsf{U}_f, \mathsf{W}_f)$ and the folding proof (a commitment to the cross-term CM_T).
2) Randomly sample a satisfying relaxed R1CS instance-witness pair $(\mathsf{U}_r, \mathsf{W}_r)$. This is done by randomly sampling W, x, u, r_E, r_W, and then computing the correct $E, \mathsf{CM}_E, \mathsf{CM}_W$.
3) Fold $(\mathsf{U}_f, \mathsf{W}_f)$ with $(\mathsf{U}_r, \mathsf{W}_r)$ to produce $(\mathsf{U}_n, \mathsf{W}_n), \mathsf{CM}_{T_n}$.
4) Prove that W_n is a satisfying witness to U_n with a SNARK π.
5) Output the randomized and compressed $\Pi' = (\mathsf{U}_i, \mathsf{u}_i, r_i, \mathsf{U}_r, \mathsf{CM}_T, \mathsf{CM}_{T_n}, \pi)$ for the statement (i, z_0, z_i) and the verifier key vk. Note that the prover could output W_n instead of π and leak nothing. π is only for compression.

The verifier is given $\Pi' = (\mathsf{U}_i, \mathsf{u}_i, r_i, \mathsf{U}_r, \mathsf{CM}_T, \mathsf{CM}_{T_n}, \pi)$ for the statement (i, z_0, z_i) and the verifier key vk. If $i = 0$, they check $z_0 = z_i$. Otherwise:

1) Check that $\mathsf{u}_i.x = \mathsf{hash}(\mathsf{vk}, i, z_0, z_i, \mathsf{U}_i, r_i)$ (Remember, $\mathsf{u}_i.x$ is the public IO of u_i.)
2) Check that u_i is a regular (non-relaxed) R1CS instance.
3) Fold U_i and u_i using CM_T to get U_f.
4) Fold U_f and U_r using CM_{T_n} to get U_n.
5) Verify π using U_n.

Deblinding SNARK commitments Notice that the verifier is required to check the folding of the relaxed R1CS instances $\mathsf{U}_i, \mathsf{u}_i$, which contain commitments to witness and instance vectors. The commitments must be hiding to preserve zero-knowledge. We implement hiding commitments (for both non-hiding Pedersen and HyperKZG) by adding a blind r to the existing commitment scheme Δ using the public blinding generator h like so: $C \cdot h^r$.

However, now we have a problem. After the randomizing layer $\mathsf{U}_r, \mathsf{W}_r$ has been folded into the IVC proof to produce $\mathsf{U}_n, \mathsf{W}_n$, there are still blinds attached to the commitments in U_n. We want to use non-ZK Spartan to produce a proof that W_n satisfies U_n, without using a ZK evaluation argument. There is a simple solution to "deblind" the U_n commitments. (This should occur after step 3 during proving and after step 4 during verifying.) The prover simply provides the blinds r_w, r_e to the witness and error commitments, C_w, C_e, and the verifier removes them: $C'_w = C_w / h^{r_w}$ and $C'_e = C_e / h^{r_e}$. Since the vectors underlying C'_w, C'_e have been randomized, nothing is leaked. The prover/verifier can now prove/check W_n satisfies U_n with a non-ZK SNARK.

Appendix D. Soundness of Coral's memory checker

This section describes the details of Coral's memory checker, including the concrete multiset hash function that we use and our segment-specific optimizations.

Public coin multiset hash functions. To fully formalize the soundness error of our public coin multiset hash functions, we define a function h_c which maps tuples to a single element in $\mathbb{F}$ using a challenge $c \in \mathbb{F}$, a map H_c, which takes n sets of timestamps, addresses, value vectors, and segment descriptors, and produces n hashed tuples, and a function $\mathcal{H}_c$ which maps multisets of elements from $\mathbb{F}$ to a single element in $\mathbb{F}$ using a challenge $c \in \mathbb{F}$, following the public-coin hash idea in Spartan [68].

$$h_c(t, a, v, s) = s + ct + c^2 a + \sum_{j=1}^{\ell} (c^{j+2} \cdot v[j])$$

$$H_c(T, A, V, S) = [h_c(T[0], A[0], V[0], S[0]), ..., h_c(T[n-1], A[n-1], V[n-1], S[n-1])]$$

$$\mathcal{H}_c(M) = \prod_{e \in M} (c - e)$$

We can see in Coral's hash, each tuple (t, a, v, s) is hashed with $h_{c_1}(t, a, v, s)$, and a multiset M of hash tuples is accumulated with $\mathcal{H}_{c_0}(M)$.

Lemma D.1. *For any two tuples* (t_1, a_1, v_1, s_1), $(t_2, a_2, v_2, s_2) \in (\mathbb{F}, \mathbb{F}, \mathbb{F}^\ell, \mathbb{F})$*, the probability (over* c*) that* $h_c(t_1, a_1, v_1, s_1) = h_c(t_2, a_2, v_2, s_2)$*, given* $(t_1, a_1, v_1, s_1) \neq (t_2, a_2, v_2, s_2)$ *is* $\leq (3 + \ell)/|\mathbb{F}|$.

Proof. Follows from the Schwartz-Zippel lemma. □

Lemma D.2. *For any* $n > 0$*,* (T_1, A_1, V_1, S_1), $(T_2, A_2, V_2, S_2) \in (\mathbb{F}^n, \mathbb{F}^n, \mathbb{F}^{n \times \ell}, \mathbb{F}^n)$*, the probability (over* c*) that there exists an* i *where* $H_c(T_1, A_1, V_1, S_1)[i] = H_c(T_2, A_2, V_2, S_2)[i]$ *given* $(T_1, A_1, V_1, S_1) \neq (T_2, A_2, V_2, S_2)$ *is* $\leq (3 + \ell) \cdot n/|\mathbb{F}|$.

Proof. Follows from Lemma D.1 and a standard union bound. □

Lemma D.3. *For any two multisets* M_1, M_2 *of size* n *over* $\mathbb{F}$*,* $Pr_c[\mathcal{H}_c(M_1) = \mathcal{H}_c(M_2)$ *given* $M_1 \neq M_2] \leq n/|\mathbb{F}|$.

Proof. Follows from the Schwartz-Zippel lemma. □

Segment overlap. The segment that any particular memory operation takes place in is enforced by the R1CS instance (see `check_mem_op`, `check_scan`, `check_push`, `check_pop`). The prover cannot pretend to perform an operation in one segment of memory, while actually doing it in another. If a prover wanted to hide which segment they accessed at which time, this would be possible with a more complex multiplexer in the R1CS instance.

RAM. Sequential consistency and the existence of an efficiently computable final set **FS** is proven in Nebula [25]. That is, if $\mathcal{P}$ ever tries to provide a value v at address a in the memory, such that v does not equal the value last written to a, then $\mathcal{P}$ will be unable to come up with a set **FS** such that $\mathbf{IS} \cup \mathbf{WS} = \mathbf{RS} \cup \mathbf{FS}$. But if $\mathcal{P}$ respects sequential consistency, then such a set **FS** can be computed.

We modify the public coin multiset hash function slightly. But the randomness c_0, c_1 for this function is sampled in the same way - that is, after the memory has been committed to via Fiat–Shamir.

ROM. In our ROM construction, we do away with the $t < ts$ check, so we no longer have sequential consistency. That is, the prover now has the ability to read address a before they write to address a. But this is not a problem, since writes are disallowed. The prover may swap reads around—that is, read address a at the (nondeterministically supplied) "time" t_2, and then later read address a at the "time" t_1, where $t_1 < t_2$. But since the value at address a never changes, this is not a problem.

Stacks. The sets of tuples $\mathbf{IS}, \mathbf{WS}, \mathbf{RS}, \mathbf{FS}$ can be divided into two subsets: RAM/ROM and stacks. For stacks, $\mathbf{IS}, \mathbf{FS}$ are empty. This is enforced by the R1CS instance - no stack segments are allowed during `check_scan`. Therefore, the final memory check, $\mathbf{IS} \cup \mathbf{WS} = \mathbf{RS} \cup \mathbf{FS}$ implies two things. First, that for the RAM/ROM subsets of $\mathbf{IS}, \mathbf{WS}, \mathbf{RS}, \mathbf{FS}$, this statement ($\mathbf{IS} \cup \mathbf{WS} = \mathbf{RS} \cup \mathbf{FS}$) is true. Second, for the stack subsets, $\mathbf{WS} = \mathbf{RS}$ is true. Sequential consistency then follows from Blum et al's Lemma 1 and Theorem 1 [31].

Packing. In several places in our memory implementation (e.g., the public coin hash), we pack timestamps, addresses, and segments into single field elements for efficiency. (Multiplication by constants and additions are free in R1CS.) This is only sound if $\mathcal{V}$ can be sure that there will not be overflows, which is the case for Coral: the range of timestamps and addresses are publicly known. Timestamps are between 0 and the final value of the global timestamp `ts` (public output). Tuples in **WS** just use the current value of `ts`. Tuples in **RS** use a nondeterministic value that is confirmed to be less than `ts`. (Notice in the case of ROM, we cannot pack timestamps, as there is no check that the value is within the 0-`ts` range.) Addresses are enforced to be between 0 and the total size of the memory in `check_scan`. The segment values are hardcoded in the R1CS instance.

Appendix E. Constructing a SNARK with memory

To construct a SNARK with support for memory accesses, we proceed as follows. First, we construct a folding scheme that supports reasoning about committed witnesses, and then invoke existing results to obtain an incrementally-verifiable computation scheme that supports similar reasoning. Then, we construct a new zkSNARK for the verifier of this IVC scheme, and show that composing the IVC with this SNARK results in a commit-carrying zkSNARK for the original relation.

Prior work [61, 66] shows how to use such a commit-carrying zkSNARK to compile an offline-memory checker that uses public-coin hashes/randomness to a zkSNARK with support for memory accesses. We can invoke their results with our CC-SNARK and memory checker as the latter satisfies the foregoing requirements.

E.1. Split-witness Committed Relaxed R1CS

We extend the definition of committed relaxed R1CS from Nova to allow additional witness commitments. Let Δ be a commitment scheme to vectors over a finite field $\mathbb{F}$. Then the *Split-Witness Relaxed R1CS* (SW-R1CS) relation for matrices with m rows/columns and p split witnesses consists of triples $(\mathbb{i}, \mathbb{x}, \mathbb{w})$ satisfying the following properties.

- The structure $\mathbb{i}$ is a tuple $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$ where each of $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ is a constraint matrix in $\mathbb{F}^{m \times m}$.
- The instance $\mathbb{x}$ is a tuple $([\mathsf{CM}_{w_i}], \mathsf{CM}_E, u, x)$ where for each $i \in [0, p-1]$, CM_{w_i} is a commitment to the i-th witness vector, CM_E is a commitment to the error vector, u is a scalar in $\mathbb{F}$, and $x \in \mathbb{F}^\ell$ is a public input/output vector.
- The witness $\mathbb{w}$ is a tuple $([w_i], E, r_E, [r_{w_i}])$, where $i \in [0, p-1]$ is the number of separate witness vectors, $w_i \in \mathbb{F}^{n_i}$, $E \in \mathbb{F}^m$ is an error vector, and $r_E \in \mathbb{F}$ and $r_{w_i} \in \mathbb{F}$ are commitment blinds.

A triple $(\mathbb{i}, \mathbb{x}, \mathbb{w})$ is in the SW-R1CS relation if and only if:

$$\begin{aligned}
&\mathsf{CM}_E = \Delta.\mathsf{Commit}(\mathsf{pp}_E, E, r_E),\\
&\forall i, \mathsf{CM}_{w_i} = \Delta.\mathsf{Commit}(\mathsf{pp}_W, w_i^{\text{ext}}, r_{w_i}), \text{ and}\\
&(\boldsymbol{A} \cdot \boldsymbol{Z}) \circ (\boldsymbol{B} \cdot \boldsymbol{Z}) = u \cdot (\boldsymbol{C} \cdot \boldsymbol{Z}) + \boldsymbol{E},\\
&\text{where } \boldsymbol{Z} = (w_0, \ldots, w_{p-1}, x, u)
\end{aligned}$$

Here, pp_E and pp_W are commitment parameters for vectors of size m and $n = \sum_{i=0}^{p-1} n_i = m - \ell - 1$, and for each i, the vector $w_i^{\text{ext}} \in \mathbb{F}^n$ is obtained from w_i as follows. Let $k_i = \sum_{i'<i} n_{i'}$. Then,

$$\forall j \in [0, n-1], w_i^{\text{ext}}[j] = \begin{cases} w_i[j - k_i], & \text{if } k_i < j < k_i + n_i \\ 0, & \text{otherwise} \end{cases}$$

This padding in the definition allows us to align the witnesses in different vectors with the generators in pp_W so that no two witnesses share a single generator. This will be useful when the time comes to produce a SNARK from a folded SW-R1CS instance. Note that this padding does not increase commitment time, as committing to 0s is free with the KZG polynomial commitment scheme we use.

Clearly, when the number p of witness vectors (and their commitments) is 1, SW-R1CS is identical to relaxed R1CS. In Coral, by default $p = 2$. We set w_0 to be the vector of witnesses representing the memory tuples (t, a, v, s) for that step of Coral's folding, and set w_1 to be the remaining variables in the SW-R1CS witness.

The fact that KZG can be used natively with Nova now makes it easy to accumulate our incremental commitment to memory. The natural commitment CM_{w_0} to w_0 in SW-R1CS is exactly the KZG commitment we want to hash for our incremental commitment. Nova uses a cycle of two elliptic curves. In the first curve, we prove `check_node` and manipulate w_0 and w_1. We add a Poseidon hash to the R1CS instance of the second curve, where the KZG group elements can be represented with coordinates natively, in the second curve's scalar field. This hash enforces that the accumulation of CM_{w_0} is done correctly. If a developer wishes to accumulate x memory segments separately for import/export (see Section 5.3) this means that $p = x + 2$, and results in $x + 1$ hashes.

Construction E.1 (Folding scheme for SW-R1CS)**.** We now describe our folding scheme for SW-R1CS. The folding scheme requires that Δ be a succinct, hiding, and homomorphic commitment scheme over the finite field $\mathbb{F}$. The verifier takes two committed SW-R1CS instances:

$$\begin{aligned}
&(\mathsf{CM}_E, u, \mathsf{CM}_{w_0}, ..., \mathsf{CM}_{w_{p-1}}, x)\\
&(\mathsf{CM}_{E'}, u', \mathsf{CM}_{w'_0}, ..., \mathsf{CM}_{w'_{p-1}}, x')
\end{aligned}$$

The prover takes the two instances and (additionally) witnesses that satisfy both instances:

$$\begin{aligned}
&(E, r_E, w_0, ..., w_{p-1}, r_{w_0}, ..., r_{w_{p-1}})\\
&(E', r_{E'}, w'_0, ..., w'_{p-1}, r_{w'_0}, ..., r_{w'_{p-1}})
\end{aligned}$$

The prover and verifier take the following steps in order to produce a new folded instance and witness:

1) $\mathcal{P}$: Send $\mathsf{CM}_T = \Delta.\mathsf{Commit}(pp_E, T, r_T)$, where:

$$\begin{aligned}
&r_T \leftarrow_R \mathbb{F}\\
&T = \boldsymbol{A}Z \circ \boldsymbol{B}Z' + \boldsymbol{A}Z' \circ \boldsymbol{B}Z - u \cdot \boldsymbol{C}Z' - u' \cdot \boldsymbol{C}Z\\
&Z = (w_0, ..., w_{p-1}, x, u), Z' = (w'_0, ..., w'_{p-1}, x', u')
\end{aligned}$$

2) $\mathcal{V}$: Sample and send challenge $r \leftarrow_R \mathbb{F}$
3) $\mathcal{P}, \mathcal{V}$: Output folded instance:

$$\begin{aligned}
&\mathsf{CM}_{E_{new}} \leftarrow \mathsf{CM}_E + r \cdot \mathsf{CM}_T + r^2 \cdot \mathsf{CM}'_E\\
&u_{new} \leftarrow u + r \cdot u'\\
&\forall i, \mathsf{CM}_{w_{i_new}} \leftarrow \mathsf{CM}_{w_i} + r \cdot \mathsf{CM}_{w'_i}\\
&x_{new} \leftarrow x + r \cdot x'
\end{aligned}$$

4) $\mathcal{P}$: Output folded witness:

$$\begin{aligned}
&E_{new} \leftarrow E + r \cdot T + r^2 \cdot E'\\
&r_{E_{new}} \leftarrow r_E + r \cdot r_T + r^2 \cdot r_{E'}\\
&\forall i, w_{new_i} \leftarrow w_i + r \cdot w'_i\\
&\forall i, r_{w_{new}} \leftarrow r_{w_i} + r \cdot r_{w'_i}
\end{aligned}$$

We extend the Nova implementation to fold SW-R1CS. The key change is that the instance in Nova must enforce the correct folding of SW-R1CS instances, which requires encoding in R1CS an extra scalar multiplication and group addition per additional witness commitment (see step 3 above). If a developer has separate persistent segments of memory, this is an extra multiplication and addition per segment.

Construction E.2 (SNARK for Committed SW-R1CS)**.** Nova uses (non-ZK) Spartan [68] to produce a SNARK

proving the satisfiability of the final folded relaxed R1CS instance. Coral uses Spartan to prove (and compress) its final folded SW-R1CS instance as well. It does so by converting this instance to a standard (non-split) relaxed R1CS instance as follows.[2]

$\mathcal{P}$ first produces a fresh relaxed R1CS instance $(\mathsf{CM}_W, \mathsf{CM}_E, u, x)$, and claims that this instance is equivalent to the SW-R1CS instance $([\mathsf{CM}_{w_0}, \ldots, \mathsf{CM}_{w_{p-1}}], \mathsf{CM}_E, u, x)$. That is, $\mathcal{P}$ claims that the commitment CM_W is a commitment to the concatenation W of all of the witness vectors $w_0, \ldots, w_{p-1}$ committed in $\mathsf{CM}_{w_0}, \ldots, \mathsf{CM}_{w_{p-1}}$. If $\mathcal{P}$ can convince $\mathcal{V}$ of this latter claim, then it can invoke Spartan on the relaxed R1CS instance and complete the proof. We now describe how $\mathcal{P}$ convinces $\mathcal{V}$ of this claim.

$\mathcal{V}$ first checks that:

$$\mathsf{CM}_W = \sum_{i=0}^{p} \mathsf{CM}_{w_i} \ .$$

If the prover were honest, this would suffice: each commitment CM_{w_i} would be a commitment to the correct padded vector w_i^{ext}, and hence would contain non-zero values only in the positions corresponding to w_i in the concatenated vector W. That is, there would be no overlaps. However, we cannot assume honesty, and in particular the simple check above is vulnerable to the following attack.

$$\begin{aligned}
\mathsf{CM}_{w_0} &= \Delta.\mathsf{Commit}(\mathsf{pp}_W, [a_0, a_1, a_2, 0, 0], r_{w_0}) \\
\mathsf{CM}_{w_1} &= \Delta.\mathsf{Commit}(\mathsf{pp}_W, [x, 0, 0, a_3, a_4], r_{w_1}) \\
\mathsf{CM}_W &= \Delta.\mathsf{Commit}(\mathsf{pp}_W, [a_0 + x, a_1, a_2, a_3, a_4], r_W)
\end{aligned}$$

The cheating prover here is using x to modify the value of the first witness in the Spartan SNARK.

To prevent this attack, we require $\mathcal{P}$ to additionally prove that none of the CM_{w_i} commitments have "overlapping" witnesses. We do this via sumcheck. $\mathcal{V}$ generates a random challenge $\tau \in \mathbb{F}^q$, and then $\mathcal{P}$ proves the following sumcheck equation:

$$0 =? \sum_{x \in \{0,1\}^q} \widetilde{\mathrm{eq}}(\tau, x) \cdot \big(\prod_{i=0}^{p} \rho^i \cdot \widetilde{\mathrm{sel}}_i(x) \cdot \widetilde{w}_i^{\mathrm{ext}}(x) \big)$$

Here $\widetilde{w}_i^{\mathrm{ext}}$ is the multilinear extension of w_i^{ext} vector when treated as evaluations of a univariate polynomial. $\mathrm{sel}_i(x)$ is a function that outputs 1 when $w_i^{\mathrm{ext}}[x]$ is padding and 0 when $w_i^{\mathrm{ext}}[x]$ is an actual value of w_i, and $\widetilde{\mathrm{sel}}_i(x)$ is its multilinear extension. Finally, $\widetilde{\mathrm{eq}}$ is the multilinear polynomial defined as $\widetilde{\mathrm{eq}}(\tau, x) = \prod_{i=1}^{q}(\tau_i x_i + (1-\tau_i)(1-x_i))$. All multilinear polynomials are over q variables. The random challenge ρ combines claims about various $\widetilde{w}_i^{\mathrm{ext}}(x)$ polynomials into one claim.

The sumcheck reduces the foregoing summation claim to evaluation claims about $\widetilde{\mathrm{eq}}(\tau, x)$ and each $\widetilde{\mathrm{sel}}_i(x)$ and $\widetilde{w}_i^{\mathrm{ext}}(x)$ at a random point $r \in \mathbb{F}^q$. The first two types of claims can be checked by $\mathcal{V}$ in the clear, and the last type of claim can be checked using KZG evaluation proofs. In Coral's implementation, these are batched with the evaluation proofs in Spartan.

2. We could extend Spartan to directly prove SW-R1CS, but we avoided this route to reuse the existing optimized Spartan implementation for relaxed R1CS in Nova.

E.2. Commit-carrying SNARK

Definition E.1. A *commit-carrying SNARK* (CC-SNARK) for an NP relation $\mathcal{R}$ is a tuple of efficient algorithms $(\mathcal{S}, \mathcal{C}, \mathcal{P}, \mathcal{V})$ satisfying the following properties.

Completeness: For all $(x, w_1, w_2) \in \mathcal{R}$, the following probability is 1:

$$\Pr\left[\mathcal{V}(\mathsf{vk}, x, c, \pi) = 1 \;\middle|\; \begin{array}{r} (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{S}(1^\lambda, \mathcal{R}) \\ c \leftarrow \mathcal{C}(\mathsf{pk}, x, w_1) \\ \pi \leftarrow \mathcal{P}(\mathsf{pk}, x, c, w_1, w_2) \end{array} \right] .$$

Knowledge soundness: For every efficient malicious prover $\mathcal{P}^*$, there exists an efficient extractor $\mathcal{E}$ such that, for all x, the following probability is negligible in the security parameter λ:

$$\Pr\left[\begin{array}{c} \mathcal{V}(\mathsf{vk}, x, c, \pi) = 1 \\ \wedge \\ (x, w_1, w_2) \notin \mathcal{R} \end{array} \;\middle|\; \begin{array}{r} (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{S}(1^\lambda, \mathcal{R}) \\ (c, \pi) \leftarrow \mathcal{P}^*(\mathsf{pk}, x) \\ (w_1, w_2) \leftarrow \mathcal{E}(\mathsf{pk}, x, c, \pi) \end{array} \right] .$$

Zero-knowledge: There exists an efficient simulator Sim such that for every efficient verifier $\mathcal{V}^*$ and all $(x, w_1, w_2) \in \mathcal{R}$, the following distributions are equal:

$$\left\{ \begin{array}{r} (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{S}(1^\lambda, \mathcal{R}) \\ c \leftarrow \mathcal{C}(\mathsf{pk}, x, w_1) \\ \pi \leftarrow \mathcal{P}(\mathsf{pk}, x, c, w_1, w_2) \\ b \leftarrow \mathcal{V}^*(\mathsf{vk}, x, c, \pi) \end{array} \right\} = \left\{ \begin{array}{r} (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{S}(1^\lambda, \mathcal{R}) \\ (c, \pi) \leftarrow \mathsf{Sim}(\mathsf{pk}, x) \\ b \leftarrow \mathcal{V}^*(\mathsf{vk}, x, c, \pi) \end{array} \right\} .$$

Lemma E.1. *The folding scheme for SW-R1CS in Construction E.1 implies a commit-carrying IVC.*

Proof. At a high level, Construction E.1 does a few things. First, it simplifies Construction 2 of Nebula [25] (let us call it NebulaC2 to avoid ambiguity with our own Construction E.2) to be specific to uniform IVC, whereas NebulaC2 works for the more general case of non-uniform IVC. That is Construction E.1 has no notion of different functions or program counters, unlike NebulaC2. Second, Construction E.1 fills in all of the details for our specific SW-R1CS folding scheme, whereas NebulaC2 is generic over a multi-folding scheme.

Nevertheless, since Construction E.1 can be seen as a concrete instantiation of NebulaC2 to SW-R1CS, we can apply Theorem 1 in Nebula which states that: "*[NebulaC2] takes a NIVC-compatible multi-folding scheme and produces a CC-NIVC scheme*". It thus follows that Construction E.1 is a commit-carrying (uniform) IVC. □

Theorem E.1. *Let $\mathcal{R}$ be an NP relation described by the repeated application of a step function F. Denote by IVC the instantiation of the CC-IVC from Lemma E.1 with the step function F, and by Π the SNARK from Construction E.2. Then the composition of IVC with Π is a CC-SNARK for $\mathcal{R}$.*

Proof. Completeness follows by inspection, so we focus on knowledge soundness and zero-knowledge.

Knowledge-soundness follows from the underlying components. In particular, the extractor for the composed SNARK first invokes the extractor for Π to obtain a valid committed relaxed SW-R1CS instance, and then it invokes the extractor for IVC to obtain a valid NP witness for repeated applications of F, which is precisely a valid witness for the relation $\mathcal{R}$.

The simulator for the new CC-SNARK simply invokes the simulator for Π. By Appendix C, the result hides all information about the witness. □

Appendix F. Proofs of security for Coral

We show that Coral (as described in Section 3.3) is a commit-carrying SNARK (construction E.2) for the relation defined in Definition B.3.

For simplicity, in the following, we take an alternate viewpoint on Coral: instead of a direct construction in terms of folding schemes, we describe Coral's construction in terms of the memory-supporting zkSNARK from Appendix E (denoted Π henceforth). This allows us to avoid handling low-level details pertaining to incremental execution and extraction.

We proceed in two parts: Appendix F.1 proves knowledge soundness, and Appendix F.2 proves zero-knowledge.

F.1. Knowledge soundness

We describe the extractor $\mathcal{E}_{\mathsf{Coral}}$ for Coral below, and then analyze its success probability.

$\mathcal{E}_{\mathsf{Coral}}(\mathsf{pk}, x, c, \pi)$:

1) Parse the proof π as $(C_{\mathsf{Mem}}, \mathsf{IPE}_{\mathsf{fin}}, \pi_{\mathsf{IPE}}, \pi_{\mathsf{fin}})$.
2) Set up inputs for the extractor of Π:
   set $x' := (x, \mathsf{IPE}_{\mathsf{fin}})$;
   set $c' := C_{\mathsf{Mem}}$; and
   set $\pi' := \pi_{\mathsf{fin}}$.
3) Invoke the extractor for Π: $w \leftarrow \mathcal{E}_{\Pi}(\mathsf{pk}, x', c', \pi')$.
4) Parse w to obtain a set of nodes T, a byte stream B, randomness r_{B} for the commitment to B, and auxiliary information aux.
5) Assert that the extracted byte stream matches the commitment c: $c = \mathsf{Commit}(\mathsf{B}; r_{\mathsf{B}})$.
6) Assert that the NP checker in Figure 11 accepts when given as input T, B, and aux.
7) Output the extracted witness $\mathtt{w} := (T, \mathsf{B})$.

We now show that the extractor $\mathcal{E}_{\mathsf{Coral}}$ outputs a valid witness with non-negligible probability. The analysis proceeds in two parts. First, we will show that if the extractor does not abort, then the extracted witness is valid. Then, we will argue that the probability that the extractor aborts is negligible.

Validity of extracted witness. If the extractor $\mathcal{E}_{\mathsf{Coral}}$ does not abort at any of the assertions, then we are guaranteed that the parse tree T and the byte stream B were accepted by Coral's NP checker. Then, by Theorem B.1, we have that the parse tree T is valid with respect to the grammar $\mathcal{G}$, and that the byte stream B corresponds to the leaves of T. Furthermore, because the assertion in Step 5 passed as well, we know that B matches the committed byte stream.

Probability of aborts is negligible. Because Π is a SNARK, the probability that its extractor $\mathcal{E}_{\Pi}$ aborts is negligible. Similarly, the probability that the extracted witness does not satisfy the NP checker is also negligible.

Finally, the probability that the assertion in step 5 fails is negligible: since Coral's verifier accepts, we know by the evaluation binding property of KZG [47] that π_{IPE} is a valid proof that the polynomial committed in $c = C_{\mathsf{B}}$ evaluates to $\mathsf{IPE}_{\mathsf{fin}}$ at the challenge point. Because Coral's NP checker accepted as well, we know that $\mathsf{IPE}_{\mathsf{fin}}$ is indeed the evaluation of the polynomial corresponding to the byte stream B at the challenge point. Finally, by Schwartz-Zippel, because these evaluations agree, we can conclude that the committed polynomial equals the polynomial corresponding to B with all but negligible probability.

F.2. Zero-knowledge

The simulator for Coral invokes the KZG simulator to first produce a commitment C_{Mem} to a random polynomial. Then, it uses this commitment to obtain the challenge point, and again invokes the KZG simulator to obtain a single evaluation $\mathsf{IPE}_{\mathsf{fin}}$ and corresponding evaluation proof π_{IPE}. It feeds the challenge point and this evaluation into the simulator for Π to obtain a simulated proof π_{fin}. It assembles these into a Coral proof and outputs it.

Clearly, each component reveals no information about the witness. Furthermore, this simulated proof is indistinguishable from a real proof:

1. The KZG commitment and evaluation proof are hiding, and so are identically distributed in the two worlds.
2. In the real world, the committed polynomial contains a randomizing blind, and hence one evaluation of it is uniformly random. In the ideal world, the committed polynomial is uniformly random, and hence so are its evaluations. Thus the evaluations are identically distributed in the two worlds.
3. Finally, Π is zero-knowledge, and so the simulated π_{fin} is identically distributed in the two worlds.