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 [17, 32, 51, 73, 75, 76], 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 [24], FS [37] and zkCreds [63] 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* [35], 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 particular name resolution protocol (e.g, DNS), the sender can prove to a network intermediary (e.g, a firewall or a policy-enforcing middlebox [55]) that such bytes satisfy a policy—such as not requesting a blocked domain or only transmitting permitted data—without revealing the stream itself [42, 74].

While the above applications are promising and have been previously explored, they lack a core missing component: *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*. As one example, existing zk-TLS and zk-Authorization systems [24, 32, 51, 73, 76], 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 [76]), or assume the input is already a valid JSON object [24]. In the 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. As another example, zk-Middlebox systems lack the ability to reason about data formats like JSON altogether.

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* [43],

and Google's OAuth JWT tokens. Coral's proofs for these concrete applications are fast to generate (a few seconds), non-interactive, and succinct: they are small (under 20 kB) and cheap to verify (under 150 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 [36]. Coral supports the full expressivity of CFGs, in addition to two important extensions: *exclusion* rules and *non-atomic* rules [13]. 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 number 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 [52]. 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 [54], 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 [47]. 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 rules and non-atomics.

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* [49] 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 [49] with an additional technique we implement from recent work [48] to make Nova's code 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 uses Nebula [23] to implement RAM segments. We remove some of the checks in Nebula if the segment is read-only and remove even more checks if the segment is for public memory. Additionally, we implement stack segments using a public-coin hash chain, which is more efficient than anything described in prior work. Segmented memory also 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.

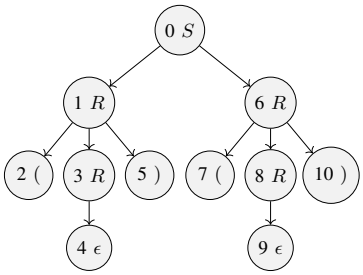


Figure 1: Parse tree for the string "()()" and the grammar $S \to (S) \mid SS \mid \epsilon$. We use ϵ to denote the empty string.

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. The HTML used in the Web today is not context-free, and neither are formats like XML, PDF, or Python. Moreover, modern grammar frameworks such as `pest` also support two useful features: rule priority to resolve ambiguities and *negative predicates* [59], which generalize exclusion rules. Extending support for context-sensitive grammars and these extra features is an important direction for 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, $\mathcal{R} \subseteq V \times (V \times \Sigma)^*$ 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:

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

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 where a prover $\mathcal{P}$ produces a proof π that convinces a verifier $\mathcal{V}$ that it 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 [40, 66], Plonkish [39], AIR [25], CCS [67]). Informally, zkSNARKs are:

1) **Zero-knowledge:** The proof π reveals no information about w beyond the fact that $\mathcal{P}$ knows w.
2) **Succinct:** The size of π and its verification is 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.

Rank-1 Constraint Satisfiability (R1CS). We focus on *rank-1 constraint satisfiability* (R1CS) as this is the arithmetization supported by the particular implementation of the zkSNARK we use [49], but our ideas apply to other arithmetizations (e.g., CCS [67]). An R1CS instance is given by a tuple $(\mathbb{F}, A, B, C, x, \text{rows}, \text{cols})$, where $\mathbb{F}$ is a finite field, x is the public input and output of the instance, $A, B, C \in \mathbb{F}^{rows \times cols}$ are matrices, and $cols \geq |x| + 1$. The instance is satisfiable if and only if there exists a witness $w \in \mathbb{F}^{cols-|x|-1}$ that makes up a solution vector $z = (w, x, 1)$ such that $(A \cdot z) \circ (B \cdot z) = (C \cdot z)$, where $\cdot$ is the matrix-vector product and $\circ$ is the Hadamard product. The value 1 in z allows constants to be encoded.

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: $inv \cdot x - 1 = 0$. This is an example of a *NP checker*: a constraint system that verifies the correctness of a claimed result far more efficiently than re-computing it. Checkers of this kind have been developed in various settings [20, 21, 30, 44, 69, 72, 77].

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.

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 trusted *committer* $\mathcal{M}$, an untrusted *prover* $\mathcal{P}$, and a *verifier* $\mathcal{V}$.

The committer $\mathcal{M}$ generates a hiding and binding commitment C_{B} to a byte stream B in a trustworthy manner, typically as part of a larger application context. The mechanism for ensuring that C_{B} commits to a meaningful byte stream in the first place is orthogonal to this work. 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 honestly generated 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_{PTree} correctly commits to a structured object PTree, which represents the parse tree resulting from parsing the byte stream B according to a public CFG. This correspondence between B and PTree is a crucial first step toward proving complex statements about the original data. Once this link is established, the structured nature of PTree can be leveraged to prove additional facts. For instance, if PTree 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.

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 [21], `zkreg` [60], zk Regex [18], Zombie [74], the work of Luo et al. [53], and DECO [76]. 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 [61], 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 PTree of B; and (4) output a new commitment C_{PTree} 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 [16], and Noir [7], require developers to reimplement the application in a domain-specific language; CirC [57] 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. [54], 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).

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 models like 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 [47]. LCRS trees are binary trees so they have 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* [49] 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

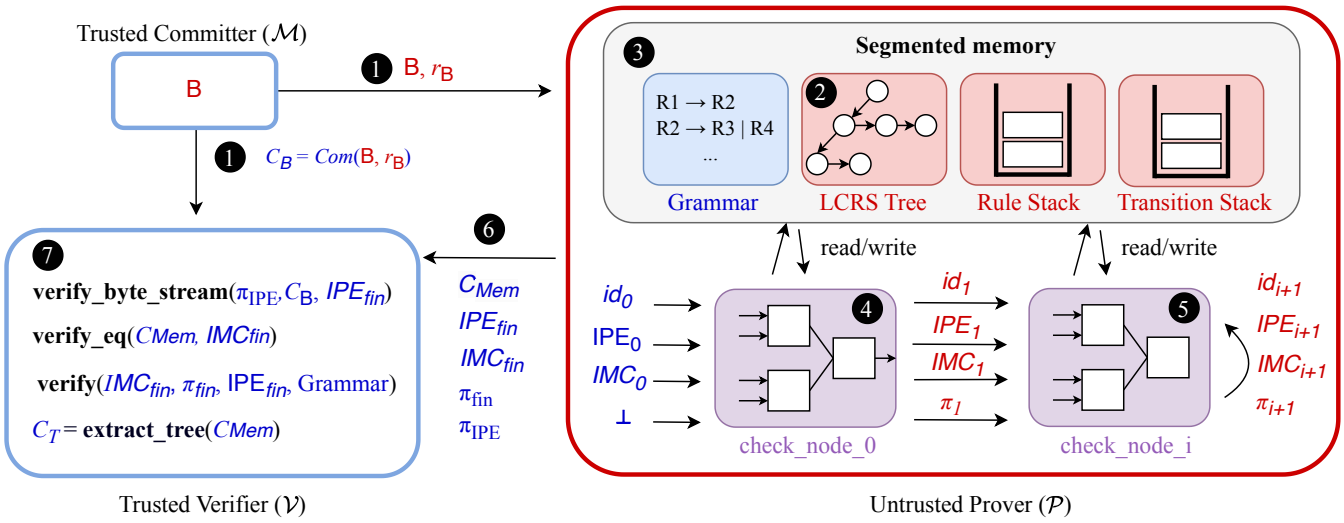


Figure 2: Overview of Coral. Blue text is public information. Red text is private information visible only to $\mathcal{M}$ and $\mathcal{P}$.

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.

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 suitable primitive [19, 21, 26–28, 31, 34, 38, 48, 58, 65, 68, 70, 71], but many of these constructions (e.g., routing networks, lookup arguments, offline memory checking) have fixed costs that are amortized over many memory accesses. If memory types are implemented separately, amortization is limited to each type's individual access pattern. This is especially inefficient in recursive and folding systems, where each step may perform only a few memory operations. Furthermore, many of these memory primitives are not foldable at all.

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

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).
4. $\mathcal{P}$ then runs the 0^{th} step of the `check_node` function, which takes in the constant $\mathtt{id}_0$. It outputs the `id` of the next node to process ($\mathtt{id}_1$), an incremental evaluation of a polynomial that succinctly 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.
5. $\mathcal{P}$ continues to run `check_node` for every node in T, feeding the output of step i into step $i+1$.
6. Finally, $\mathcal{P}$ sends the original commitment to the memory performed during action 3 (C_{Mem}), the final incremental polynomial evaluation that summarizes the leaves seen ($\mathsf{IPE}_{\mathsf{fin}}$) and a polynomial evaluation proof π_{IPE}, the final incremental commitment to the memory operations performed ($\mathsf{IMC}_{\mathsf{fin}}$), and the proof ($\pi_{\mathsf{fin}}$) that the final instance was performed correctly.
7. $\mathcal{V}$ checks that C_B and $\mathsf{IPE}_{\mathsf{fin}}$ encode the same byte stream with π_{IPE}, and that C_{Mem} and $\mathsf{IMC}_{\mathsf{fin}}$ are equal, which implies that the memory accesses were consistent with the committed memory and the operations. $\mathcal{V}$ then checks the final proof π_{fin}, which implies that both $\mathsf{IPE}_{\mathsf{fin}}$ and $\mathsf{IMC}_{\mathsf{fin}}$ were computed correctly and all prior proofs were verified. Lastly, $\mathcal{V}$ can extract a commitment to the LCRS Tree T from C_{Mem} to use in other proofs.

In the following section we provide the details.

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

Figure 4: Structure of a node in Coral's LCRS Parse Tree.

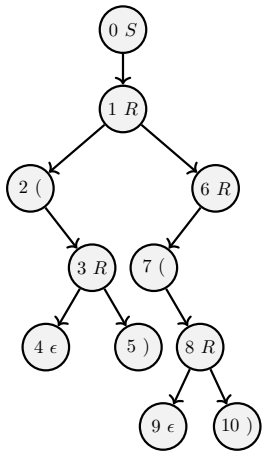


Figure 3: LCRS tree corresponding to the tree in Figure 1.

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: (1) T is a valid parse tree for B; and (2) 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 unbounded numbers of children in a parse tree is to rewrite G into a grammar G', such that each non-terminal rule expands to either a single terminal or at most two other non-terminal rules. This form (adopted by prior work [54]) is known as *Chomsky Normal Form* (CNF), and results in a tree where nodes have at most two children. 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 causes 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 [47] proposed a method to transform any tree into a binary tree using a single linear pass: the result is an LCRS tree. 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. Figure 3 shows this transformation.

In Figure 4, we show the layout of an LCRS tree node in Coral. $\mathcal{P}$ assigns to each node `id`s following a pre-order traversal: the root has $\texttt{id}_0 = 0$ and the remaining nodes have monotonically increasing `id`s. Each node stores the `id`s 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 `id`s, 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 into a vector of field elements 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 right-hand side. For example, the first four rules become: $[S, R], [R, (, R,)], [R, R, R]$, and $[R, \epsilon]$. Since all rows in the table must have the same length, we pad each rule to match 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, as marked in `pest` syntax using an `@` before the RHS. The second indicates whether the rule includes any exclusions. We describe how Coral handles atomic rules and exclusions 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 bytestream 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, $\mathcal{M}$ generates three uniformly random blinds: $r^1_{\mathsf{B}}, r^2_{\mathsf{B}}, r^3_{\mathsf{B}} \in \mathbb{F}$. Then, $\mathcal{M}$ encodes the byte stream $\mathsf{B} = b_1||\cdots||b_{|\mathsf{B}|}$ as the degree $|\mathsf{B}|$ 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* [45] 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 [33]: 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.

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`. `check_node` 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 proposed symbols are $[(, R,)]$, then `check_node` pushes onto `rule_stack` the tuples $[), \mathtt{true}]$, $[R, \mathtt{false}]$, and $[(, \mathtt{false}]$. `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 introduced 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`

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

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; (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 last decade, there have been dozens of proposals for incorporating state (storage and memory) into (zk)SNARKs, including routing networks, Merkle trees, lookup arguments, RSA accumulators, and offline memory checking. The proposals vary widely in their features and costs. For our system, we require:

1) Concrete efficiency when expressed in R1CS.
2) Compatible with folding schemes.
3) Handles public and private RAM.
4) Compatible with portable commitments.

Among the available techniques, we find that the offline memory-checking approach introduced by Blum et al. [29], later adapted to be compatible with SNARKs by Spice [65] and refined by Nebula [23], is particularly suited for our purposes. First, Nebula achieves concrete efficiency in R1CS (its experimental evaluation reports state-of-the-art performance). Second, the scheme is designed with folding schemes in mind, which allows amortizing the large one-time costs associated with offline memory-checking across all memory accesses from *all* computation steps: hence, the overhead is minimal. Third, it can handle public and private RAM naturally. Finally, its design exposes an explicit, canonical final state, which can be readily committed to and exported as a portable digest.

As such, we adopt Nebula as the foundation of our RAM representation, though we provide our own independent implementation (as of the time of this writing there is no public implementation of the original Nebula construction). This required us to flesh out a lot of low-level details that were not explicit in the original paper. In the next sections, we detail our implementation, describe the optimizations that we make, define our segment representation, and finally show how we support stacks.

5.1. Memory construction

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,

```
check_mem_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 7: 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.

```
check_mem_op(field read_hash, field write_hash,
  bool op, field ts, field addr, field segment,
  // below are untrusted inputs from Prover
  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 6: Coral's `check_mem_op` function (simplified).

IS is a set of tuples $(0, i, v_i)$ for $i \in \{0, ..., M-1\}$, where v_i is the initial vector of values of length ℓ stored at address i. **RS** and **WS** are initialized as empty sets and will track memory accesses to particular addresses. **FS** will contain 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 [65] 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 [62] 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:

$$\mathsf{hash}(t, a, v) = c_0 - (t + 2^{32} \cdot a + \sum_{j=1}^{\ell} (c_1^j \cdot v[j]))$$

In the public coin hash 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 $\prod_{i=1}^{n} \mathsf{hash}(t_i, a_i, v_i)$. A crucial part is that the hashes can be computed incrementally (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 6), which runs on every memory operation, and `check_scan` (Figure 7), which runs after all memory accesses have been done. `check_mem_op` and `check_scan` are identical to the checkers proposed in Spice [65] with three small changes: (1) they use the above public coin hash; (2) they include a segment field; and (3) as done in Nebula [23] and Lasso [68] they increment the global timestamp by 1 (instead of a more involved calculation in Spice). We refer the reader to Spice [65] and Nebula [23] to understand why these checkers are correct, but the key property they guarantee is this: if one runs `check_mem_op` after every memory operation, and then one computes `check_scan` at the end, if all checks pass then the memory accesses were *sequentially consistent* [50]: a read at a given address returns the value that was most recently written to that address.

A drawback of `check_scan`, however, is that it requires scanning all initialized memory at once. This disrupts uniformity: we would either need to first compute all steps of `check_node` and then perform `check_scan` separately, or fold both operations into a single large uniform step function with a selector to distinguish between `check_node` and `check_scan`. Nebula chooses the first approach: it proves the application's step function (the equivalent of `check_node` in our context) with one proof system (e.g., Nova), and then it proves `check_scan` another proof system. In essence, $\mathcal{V}$ receives and verifies two proofs.

Coral does something morally equivalent but with a single step function that produces a single proof. In detail, $\mathcal{P}$ runs the `check_node` logic for all of its steps first (without proving anything) to determine what memory operations, addresses, timestamps, and values will be used. Then, $\mathcal{P}$ commits to this memory trace (§5.3). After committing, $\mathcal{P}$ can sample the random challenges c_0 and c_1 using the Fiat-Shamir transform. Coral then splits the `check_scan` function into chunks that perform some of the iterations of the for loop that goes over all of memory (e.g., the first chunk would operate over the first 20 addresses, the second chunk over the next 20 addresses, and so on). We then append the verification of a chunk at the end of our `check_node` step function. In this way, each step function is computing some number of `check_mem_op` operations, in addition to verifying a portion of the initial and final multiset hash computation (`check_scan`). This makes the computation uniform. 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 equal to hashing the union of the multisets).

```
check_stack(field stack_hash, bool op,
  // below are untrusted inputs from Prover
  field[] vals, field preimage, field result) {

  if op == POP {
    assert(stack_hash == result);
  } else { // PUSH
    assert(stack_hash == preimage);
  };

  field result = hash(preimage, vals);

  field new_stack_hash = if op == POP {
    preimage
  } else { // PUSH
    result
  }

  return new_stack_hash;
}
```

Figure 8: Coral's `check_stack` function (simplified).

5.2. Segmented Memory

In Section 5.1, we described a general private RAM construction. However, memory accesses in Coral are quite varied: we use private ROM, private access to public ROM, and even a private stack. While specializing to these patterns enables optimizations, it introduces a key challenge: how do we ensure that $\mathcal{P}$ performs memory accesses only within the correct memory object?

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 type of memory being accessed. The segment descriptor allows us to cheaply enforce that each memory access is in the correct segment without costly range checks. The multiset hash then includes the segment descriptor s:

$$\mathsf{hash}(t, a, v, s) = c_0 - (s + 2 \cdot t + 2^{33} \cdot a + \sum_{j=1}^{\ell} (c_1^j \cdot v[j]))$$

The above assumes 2 (non-stack) segments which is all we need in Coral. We show this hash is sound in Appendix C.2; we also show how to trivially generalize it to more segments there. We now describe how Coral specializes each memory type to further reduce cost:

Private ROM. The private ROM functions 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. 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 of it are private, so $\mathcal{V}$ is unaware of which grammar rule is being accessed by $\mathcal{P}$ in any given step. A small issue that arises is that we must ensure that the memory storing the table is correctly initialized from the grammar in a manner that can be verified by $\mathcal{V}$. For this, we simply have $\mathcal{V}$ compute the multiset hash of **IS** on their own since they have all of the information needed. We can thus remove constraints related to 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: $\mathbf{IS}_{\mathsf{pub}} \cup \mathbf{IS}_{\mathsf{priv}} \cup \mathbf{WS} = \mathbf{RS} \cup \mathbf{FS}$.

Stacks. Coral maintains two stacks. This could be done by maintaining a stack pointer abstraction on top of `check_mem_op` as CPUs normally do, but there is a more efficient way. Figure 8 illustrates `check_stack`, which uses a public-coin hash to represent the current stack state as a hash chain. This is extremely efficient (more so than regular RAM or ROM), and prevents $\mathcal{P}$ from violating stack discipline:

$$\mathsf{hash}(stack, v) = (c_0 - stack) \cdot \prod_{j=1}^{\ell} (c_0 - v[j]))$$

A new stack is simply the value 0. To push a vector of values, $\mathcal{P}$ appends it to the hash chain. To pop, $\mathcal{P}$ supplies a preimage and checks that appending the popped values produces the current stack hash. As with the memory construction in Section 5.1, the challenge c_0 is generated by applying the Fiat-Shamir heuristic to the transcript *after* the prover has committed to the memory trace (which includes the stack). We discuss the details in Section 5.3.

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 that were made in that step. Second, existing commitment schemes (even those based on SNARK-friendly hashes like Poseidon [41]) incur high concrete costs when expressed in R1CS.

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

Incremental commitments. Borrowing from prior works on commitment-carrying IVC [23, 56], $\mathcal{P}$ 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 "block" 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**, and the stack values for that step. Stack operations do not need time or address stamps. Since `check_stack` is separate from `check_mem_op`, each stack has a unique circuit and does not need a domain separator either.

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 hash [41], 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 ("HyperKZG"), CM is a variant of multilinear KZG that we make hiding following the analysis of prior work [33].

Incremental opening. To enforce commitment opening consistency, Coral recomputes the commitment inside R1CS as follows. The R1CS instance for the the i-th step takes as input the running hash H_{i-1} as well as the the i-th chunk w_{i-1} of memory operations. For each operation in the tuple, it enforces the checks in Section 5.1, and then recomputes C_i from w_i. 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 consistent with the existing commitment $\mathsf{IC}_{H,\mathsf{CM}}.\mathsf{Commit}(\mathsf{pp}, w)$. In the notation of Figure 2, H_i is IMC and $\mathsf{IC}_{H,\mathsf{CM}}$ is C_{Mem}.

Persistent memory. In Appendix C.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 using an 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 a modest machine?
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 [4]: 8K for the core system, 2K for segmented memory, 1K to make Nova zero-knowledge, and 1K to implement commitment-carrying IVC and to merge SW-R1CS to R1CS in Nova (Appendix C). Our modified version of Nova builds on Nova v0.41.0 [8]. We support both the multilinear variant of KZG commitments ("HyperKZG") 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 [22], which we then to convert to R1CS. To improve performance, we do not run one instance of `check_node` per step in Nova. Instead, each step consists of a batch of `check_node` invocations: the batch size dependents 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. 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. [54], 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.

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 matches or 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 zkCompilation 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 [15].
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 [43].
5) **JSON Web Token (JWT).** Existing systems [24, 63] prove facts in ZK about signed JWTs, but none checks that the JWT is well-formed. The byte streams are Google OAuth JWTs from Sui's zkLogin implementation [12].

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: 3 from The C Programming Language [46] and 1 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 [14] 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) [22].

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 9 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 rules in the grammar. On average, it takes ≈750 R1CS constraints to represent `check_node` for a single node.

Setup. This includes the generation of the public parameters in Nova and the generation of the R1CS instance (§2.2).

Solving. This includes the parsing itself (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}, π_{IPE}, and the equality of the incremental commitment to memory (§5.3) to the original commitment (C_{Mem}). 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 $\mathsf{IPE}_{\mathsf{fin}}$, the original commitment to the memory C_{Mem}, and the final incremental commitment to memory $\mathsf{IMC}_{\mathsf{fin}}$. 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.

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 fairly small and take only 10s of ms to verify. We therefore think that Coral is indeed practical.

6.3. How does Coral compare to alternatives?

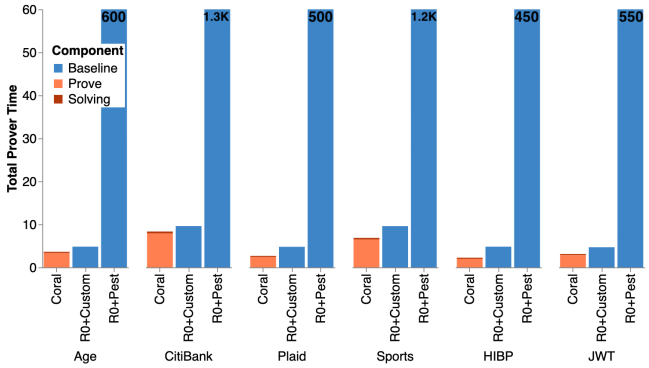
Having established that Coral is practical, we now consider whether existing systems could have served as viable alternatives. To put our results in context, we compare each application against the R0+Custom and R0+Pest baselines introduced earlier. Unfortunately, we are unable to include the MHNM baselines, as their code is not open source and their paper reports only scalability results on synthetic circuits of varying sizes, rather than concrete applications (we will compare with those in Section 6.4).

Results. Figure 10 shows Coral's *total prover time* (this includes solving and proving as defined in the prior section) in relation to both R0+Custom and R0+Pest.

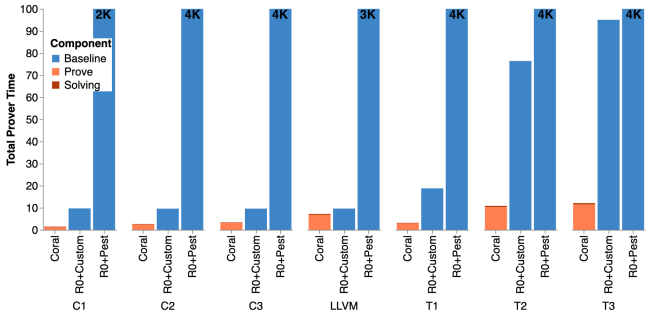
Compared to R0+Pest, which has the same features as Coral, the results are overwhelming: Coral's total proving time is orders of magnitude faster (we even had to truncate the figure). Compared to R0+Custom, Coral is faster across all applications, and significantly faster for TOML. This is despite the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

CFG	App	Byte stream size	LCRS tree size	#R1CS	Nodes per step	Setup time (s)	Solving time (s)	Proving time (s)	Verify time (s)	Proof size (kB)	Max memory (GB)
JSON	Age	747	2,594	162,280	305	2.124	0.269	3.357	0.149	17.194	1.210
	CitiBank	2,366	9,503	411,732	814	2.915	0.418	7.915	0.162	17.738	2.183
	Plaid	312	1,348	111,876	201	2.003	0.247	2.408	0.149	17.194	0.971
	Sports	1,547	7,263	388,297	769	2.855	0.364	6.487	0.162	17.738	2.142
	HIBP	191	869	106,133	190	1.947	0.244	2.001	0.145	16.650	0.759
	JWT	384	1,777	112,115	201	2.030	0.253	2.857	0.151	17.194	0.983
C Code	C1	56	241	71,546	87	1.798	0.229	1.226	0.140	16.650	0.604
	C2	237	1,173	123,119	177	2.026	0.247	2.401	0.149	17.194	1.003
	C3	399	2,027	204,011	311	2.237	0.264	3.116	0.151	17.194	1.302
	LLVM	1,460	6,652	371,929	589	2.856	0.363	6.851	0.166	17.738	2.052
TOML	T1	378	2,050	200,930	356	2.231	0.261	2.890	0.152	17.194	1.252
	T2	2,412	12,868	421,269	782	2.963	0.537	10.292	0.164	17.738	2.258
	T3	2,799	14,484	406,197	754	2.917	0.594	11.483	0.163	17.738	2.251

Figure 9: 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.



(a) JSON



(b) C and TOML

Figure 10: 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.

Takeaway. Of the existing approaches, R0+Pest is not viable whereas R0+Custom is, but its performance is, at best, the same as Coral and in many cases it is worse. 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); and (2) 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 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.

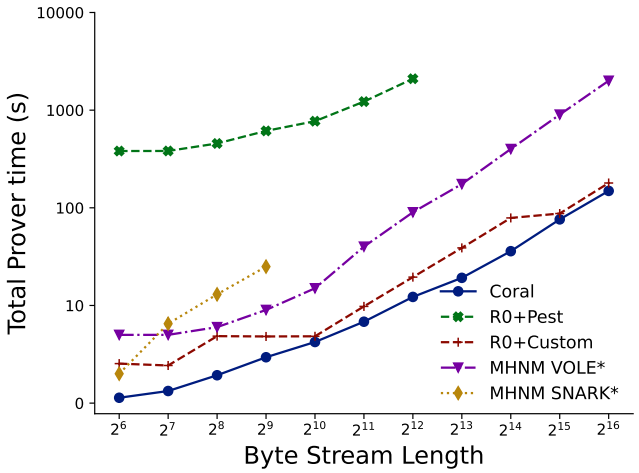
6.4. How does Coral scale?

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.

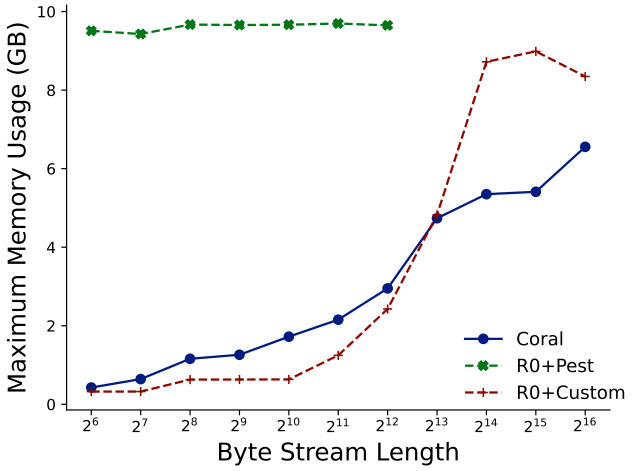
We compare with R0+Custom, R0+Pest, and both variants of the MHNM baseline by using the numbers they report in Figures 10 and 12 of their paper [54]. MHNM evaluate their system using a grammar that has fewer rules than a real JSON grammar and on a server with 32 vCPUs and 64 GB of RAM. We therefore feel 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 11 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



(a) Proving Time



(b) Memory

Figure 11: 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 [54]; Coral and RISC Zero run on 12 CPUs.

expensive to verify. This demonstrates that despite VOLE approaches being widely considered to be computationally more efficient than 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 $\approx 5\times$ [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.

Code

Our code is open source and available at:

https://github.com/eniac/coral

Acknowledgments

We 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 Sui, Ethereum Foundation, Ingonyama, and the Arcological Swiss Association.

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: Zero knowledge cfg. anonymized.
[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] Sui zkLogin Demo. https://sui-zklogin.vercel.app/.
[13] A thoughtful introduction to the pest parser: non-atomic rules. https://pest.rs/book/grammars/syntax.html#non-atomic.
[14] Toml: Tom's obvious minimal language. https://toml.io/en/.
[15] Veratad. https://veratad.com/.
[16] ZoKrates: A toolbox for zkSNARKs on ethereum. https://github.com/Zokrates/ZoKrates.
[17] TLS Notary, 2023. https://tlsnotary.org/.
[18] Zk email verify. https://github.com/zkemail/zk-email-verify, 2023.
[19] M. Abe, G. Fuchsbauer, J. Groth, K. Haralambiev, and M. Ohkubo. Structure-preserving signatures and commitments to group elements. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2010.
[20] 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.
[21] 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.
[22] arkworks contributors. `arkworks` zksnark ecosystem, 2022.

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

[24] 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.

[25] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046, 2018.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] S. Celi, A. Davidson, H. Haddadi, G. Pestana, and J. Rowell. DiStefano: Decentralized infrastructure for sharing trusted encrypted facts and nothing more. Cryptology ePrint Archive, Paper 2023/1063, 2023.

[33] 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.

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

[35] 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.

[36] 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.

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

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

[39] 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.

[40] 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.

[41] 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.

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

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

[44] 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.

[45] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2010.

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

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

[48] A. Kothapalli and S. Setty. HyperNova: recursive arguments for customizable constraint systems. *Cryptology ePrint Archive*, 2023.

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

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

[51] 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.

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

[53] 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.

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

[55] 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.

[56] 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.

[57] 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.

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

[59] 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.

[60] M. Raymond, G. Evers, J. Ponti, D. Krishnan, and X. Fu. Efficient zero knowledge for regular language. Cryptology ePrint Archive, Paper 2023/907, 2023.

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

[62] 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.

[63] M. Rosenberg, J. White, C. Garman, and I. Miers. zk-creds: Flexible anonymous credentials from zkSNARKs and existing identity infrastructure. Cryptology ePrint Archive, Paper 2022/878, 2022.

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

[65] 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.

[66] 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.

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

[68] 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.
[69] 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.
[70] T. Solberg. A brief history of lookup arguments. https://github.com/ingonyama-zk/papers/blob/main/lookups.pdf, 2023.
[71] 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.
[72] 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.
[73] 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.
[74] 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.
[75] 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.
[76] 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.
[77] 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: $r1 \rightarrow b|c|...$. Or we could leverage the exclusion rule and write $r1 \rightarrow (!"a")$`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`

Lemma B.1. *If a* Coral *proof passes verification then there exists an order of traversed nodes O that completes a depth-first traversal of a valid LCRS tree T.*

Proof. Consider a particular node k that $\mathcal{P}$ provides as a witness. 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, only after k's left child (and its entire subtree) has been processed (i.e., a leaf is reached) will k's right sibling be popped from `tran_stack` and processed.

Now we consider the edge cases. In the first step, the Coral proof 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, $\mathcal{V}$ confirms the transition stack is empty, so no unprocessed nodes can remain in the stack. This means that during the last `check_node` step, the final leaf is processed. $\mathcal{P}$ is allowed to non-deterministically specify that this is the last node to be processed, in which case the (empty) stack is not queried. □

Lemma B.2. *Given grammar $\mathcal{G}$, if a* Coral *proof passes verification then there exists an LCRS tree T, traversed in order O, that corresponds to a valid parse tree for $\mathcal{G}$.*

Proof. Let *rule parent* of a node k refer to the node that was k's parent before the parse tree was transformed into an LCRS tree 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 rule parent

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

Now consider the case where k is a right sibling. Assume, inductively, that all rules corresponding to its rule 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 children of a rule parent are visited in left-to-right order (Lemma B.1), 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 rule 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 that indicates whether this is the last symbol in a rule or not. `check_node` enforces that unless a symbol is the last one in a rule, k must have a sibling, that will be processed in the correct order (Lemma B.1). This prevents a cheating $\mathcal{P}$ from beginning a rule with one rule parent, and finishing it at a previous rule parent. In other words, the rule that defined the symbol for node k is pushed onto the `rule_stack` by k's rule parent, and popped completely off the `rule_stack` when we reach the last sibiling on k's level.

If k itself is not a leaf (as leaves do not generate new rules to check), then $\mathcal{P}$ nondeterministically proposes the correct grammar rule to check against k's children's symbols. `check_node` enforces that this grammar rule does indeed come from the publicly available grammar table that $\mathcal{V}$ checks. The symbols (and boolean flags) for this grammar rule are pushed onto the `rule_stack` in reverse order.

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, $\mathcal{V}$ confirms the `rule_stack` is 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. □

Lemma B.3. *Given a commitment to a byte stream C_{B}, if a* Coral *proof passes verification, then there exists a byte stream* B *that is an accumulation of the symbols of leaves of T as they appear in order O.*

Proof. `check_node` only accumulates symbols into the incremental polynomial evaluation when either the current node k is (1) a leaf node or (2) the root node. In the case where k is the root, the blind r^1_{B} is accumulated. The root can never be a leaf, a fact $\mathcal{V}$ can verify by looking at the public grammar, so we do not have to worry about accumulating the blind instead of a leaf symbol.

`check_node` enforces a counter that starts at 0 and increments after every accumulation of the incremental polynomial evaluation. This counter is accumulated with each symbol in the polynomial evaluation, so leaves are tied to the order in which they are seen.

If Coral's proof verifies, then the associated proof of opening for the commitment C_{B} also verifies. That is, C_{B} is a binding commitment to a polynomial that, when evaluated at a verifier-chosen challenge point c, yields the same value as the final accumulator output from the last `check_node` step. Since c is sampled after the commitment C_{B} is fixed, this evaluation is sound. Therefore, the polynomial committed to in C_{B} (which encodes the ordered, indexed byte stream B) must match the polynomial computed by `check_node` (which encodes the ordered, indexed characters of the leaves of T). This establishes that the committed byte stream B is exactly the concatenation of the symbols of the leaves of T, in the order they are visited. □

Theorem B.1. ***Soundness.*** *Given a grammar $\mathcal{G}$ and a commitment to a byte stream C_{B}, if a* Coral *proof passes verification then there exists a byte stream* B *and an order of traversed nodes O such that:*

1) *O is a depth-first traversal of an LCRS tree T,*
2) *T respects $\mathcal{G}$,*
3) *C_{B} is a binding commitment to* B*, which is an accumulation of the symbols of leaves of T as they appear in order O.*

Proof. Each of the three claims is necessary for the verifier to accept, and all are enforced jointly: (1) follows from Lemma B.1, (2) from Lemma B.2, and (3) from Lemma B.3. Since the verifier checks all three, soundness holds only if all three conditions are simultaneously satisfied. □

Theorem B.2. ***Completeness*** *Given a grammar $\mathcal{G}$ and a commitment to a byte stream C_{B}, a prover $\mathcal{P}$ who possesses the byte stream* B *underlying C_{B} can produce a valid* Coral *proof that C_{B} corresponds to a byte stream that is in the language of $\mathcal{G}$.*

Proof. First, we prove that $\mathcal{P}$ can produce a valid LCRS tree for B that respects $\mathcal{G}$. Since we assume $\mathsf{B} \in L(\mathcal{G})$, there must exist at least one derivation of B from the grammar's start symbol, which corresponds to a parse tree. Many algorithms exist for this purpose. Each non-leaf node of this parse tree will correspond to a non-terminal symbol and its children will correspond to the symbols on the right-hand side of a production rule applied to the non-terminal. The leaves of this parse tree, read from left to right, form B.

Now that $\mathcal{P}$ has this parse tree it is straightforward to obtain the LCRS tree T by applying the Knuth transform [47].

$\mathcal{P}$ performs a DFS traversal of T to assign incrementally increasing node ids to all nodes. During this linear scan, $\mathcal{P}$ stores all of the nodes in the memory construction. They also store $\mathcal{G}$ and initialize empty stacks. Remember, in our memory construction, $\mathcal{P}$ maintains an untrusted memory that they can query during solving. It is now straightforward for $\mathcal{P}$ to run `check_node` over every node in DFS order during its solving pass. If a node is a non-leaf node, then it will 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.2. $\mathcal{P}$ keeps track of which memory operations have been performed, so that they can produce the incremental commitments to memory, and sample randomness.

Now, during proving, $\mathcal{P}$ trivially has all of the witnesses necessary to satisfy check_node. $\mathcal{M}$ also gives $\mathcal{P}$ the blinds needed to create a proof of opening for C_{B}. Completeness for the actual proof follows from the underlying SNARK. □

Appendix C. Integrating memory into Nova

Incorporating memory into Nova requires: (1) generalizing Nova's Relaxed R1CS to *Split Witness Relaxed R1CS* (SW-R1CS) in order to utilize Nova's native commitments to efficiently accumulate the memory commitment; (2) constructing committed versions of SW-R1CS using KZG commitments; (3) showing how to fold committed SW-R1CS; (4) applying a transformation in Nebula [23] to produce a commitment-carrying uniform IVC for SW-R1CS; (5) showing how to transform a folded SW-R1CS instance into a Relaxed R1CS, enabling compatibility with Spartan [64], which is implemented for Relaxed R1CS (not SW-R1CS) and provides witness compression in Nova.

C.1. Split-witness Committed Relaxed R1CS

This definition extends the original definition of committed relaxed R1CS from Nova, with changes for our memory constructions. Here, $\mathbb{F}$ is a finite field and CM is a commitment scheme over $\mathbb{F}$. We define m, the size of the sparse matrices, and p, the number of split witnesses (and commitments to them).

- An SW-R1CS structure is defined by sparse matrices $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{F}^{m \times m}$.
- An SW-R1CS instance is defined by commitments C_{w_i} for $i \in [0..p-1]$ separate witness vectors, a commitment C_E to the error vector, a scalar $u \in \mathbb{F}$, and a public input/output vector $x \in \mathbb{F}^{\ell}$.
- A SW-R1CS witness is defined by $W = [w_0, ..., w_{p-1}]$ where each $w_i \in \mathbb{F}^{n_i}$ is a separate witness vector, the error vector $E \in \mathbb{F}^m$, and commitment blinds $r_E \in \mathbb{F}$ and $\forall i, r_{w_i} \in \mathbb{F}$. A witness *satisfies* a SW-R1CS instance if and only if:

$$
\begin{aligned}
&C_E = \mathsf{CM}.\mathit{Commit}(pp_E, E, r_E)\\
&\forall i, C_{w_i} = \mathsf{CM}.\mathit{Commit}(pp_W, w_i^{ext}, r_{w_i})\\
&(\boldsymbol{A} \cdot \boldsymbol{Z}) \circ (\boldsymbol{B} \cdot \boldsymbol{Z}) = u \cdot (\boldsymbol{C} \cdot \boldsymbol{Z}) + \boldsymbol{E},\\
&\text{where } \boldsymbol{Z} = (w_0, ..., w_{p-1}, x, u)
\end{aligned}
$$

Here, pp_E and pp_W are commitment parameters for vectors of size m and $\sum_{i=0}^{p} n_i$. We define the total number of witnesses across vectors as $n = \sum_{i=0}^{p} n_i = m - \ell - 1$. We define each vector $w_i^{ext} \in \mathbb{F}^n$ as a padded extension of $w_i \in \mathbb{F}^{n_i}$ below. First, let $k_i = \sum_{i'<i} n_{i'}$. Now we can say:

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

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. Notice this padding does not add to the proving time, as committing to 0s is free with the KZG polynomial commitment scheme we use.

Clearly, when the number p of w_i witness vectors (and commitments to them) is 1, SW-R1CS reduces to relaxed R1CS. In Coral, $p = 2$. 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. Then w_1 is all of 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 C_{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 C_{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.

Folding scheme for split-witness committed relaxed R1CS The folding scheme requires that CM be a succinct, hiding, and homomorphic commitment scheme over the finite field $\mathbb{F}$. The verifier takes two committed SW-R1CS instances:

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

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 $C_T = \mathsf{CM}.\mathit{Commit}(pp_E, T, r_T)$, 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:

$$
\begin{aligned}
&C_{E_{new}} \leftarrow C_E + r \cdot C_T + r^2 \cdot C'_E\\
&u_{new} \leftarrow u + r \cdot u'\\
&\forall i, C_{w_{i\_new}} \leftarrow C_{w_i} + r \cdot C_{w'_i}\\
&x_{new} \leftarrow x + r \cdot x'
\end{aligned}
$$

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 big change is that the instance in Nova must enforce the correct folding of SW-R1CS instances, which requires an extra scalar multiplication and group addition encoded in R1CS (see step 3 above). If a developer has separate persisting segments of memory, this is an extra multiplication and addition per segment.

Producing a SNARK. The original implementation of Nova uses (non ZK) Spartan [64] to produce a SNARK proving the satisfiability of the final folded relaxed R1CS instance. For implementation ease, Coral uses Spartan to prove (and compress!) it's SW-R1CS instance as well, by converting SW-R1CS to typical relaxed R1CS. (Though, of course, Spartan could be made to work over SW-R1CS as well.) $\mathcal{P}$ makes a claim that it knows a committed relaxed R1CS instance. This instance is comprised of a commitment C_W to the witness, a commitment C_E to the error vector, a public scalar $u \in F$ and public io $x \in F^\ell$ that corresponds to its SW-R1CS instance. This latter instance is itself comprised of a vector of commitments to its witnesses, $[C_{w_0}, \ldots, C_{w_{p-1}}]$, a commitment $C_{E'}$ to the error vector, a public scalar $u' \in F$ and public io $x' \in F^\ell$. To do this, $\mathcal{P}$ must prove:

1) $C_E = C_{E'}, u = u', x = x'$ (trivial)
2) The commitment C_W is a commitment to the concatenation W of all of the witness vectors $w_0, \ldots, w_{p-1}$ committed to by $C_{w_0}, \ldots, C_{w_{p-1}}$

If $\mathcal{V}$ is convinced of these things, then they are convinced that the Spartan proof of knowledge of a relaxed R1CS instance implies proof of knowledge of a split-witness relaxed R1CS instance. To verify (2), $\mathcal{V}$ checks: $C_W = \sum_{i=0}^{p} C_{w_i}$.

However, one more problem remains. A cheating prover might provide, for example, the following commitments:

$$C_{w_0} = \mathsf{CM}.Commit(pp_W, [a_0, a_1, a_2, 0, 0], r_{w_0})$$
$$C_{w_1} = \mathsf{CM}.Commit(pp_W, [x, 0, 0, a_3, a_4], r_{w_1})$$
$$C_W = \mathsf{CM}.Commit(pp_W, [a_0 + x, a_1, a_2, a_3, a_4], r_W)$$

The cheating prover here is using x to modify the value of the first witness in the Spartan SNARK. To solve this, $\mathcal{P}$ is required to additionally prove that none of the C_{w_i} commitments have "overlapping" witnesses. In other words, our definition of w_i^{ext} is obeyed. $\mathcal{P}$ uses sumcheck to do this, for a challenge $\tau \in F^q$:

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

Recall that w_i^{ext} is the padded extension of witness vector w_i. So $\widetilde{w_i^{ext}}$ is the multilinear extension of the w_i^{ext} vector when treated as evaluations of a univariate polynomial. $\widetilde{sel_i}(x)$ is a function that outputs 1 when $w_i^{ext}[x]$ is padding and 0 when $w_i^{ext}[x]$ is an actual value of w_i. The multilinear extension $\widetilde{eq}(\tau, x) = \prod_{i=1}^{q}(\tau_i x_i + (1-\tau_i)(1-x_i))$. All of these are multilinear polynomials in q variables. The random ρ combines claims about various $\widetilde{w_i^{ext}}(x)$ polynomials into one claim. At then end of this sumcheck, $\mathcal{V}$ must compute the evaluations of $eq(\tau, r)$ and $\forall i, sel_i(r)$ over a random point $r \in \mathbb{F}^q$, which can be done in the clear, and check evaluations proofs of $\forall i, \widetilde{w_i^{ext}}(r)$. These are batches with the evaluation proofs that Spartan produces in Coral's implementation.

C.2. Soundness of Coral's memory

The randomness c_0, c_1 for the public coin multiset hash function is sampled after the memory has been committed to via Fiat-Shamir; collision resistance follows from the Schwartz-Zippel lemma. Sequential consistency and the existence of an efficiently computable final set **FS** is proven in Nebula [23]. The segment that any particular memory operation takes place in is publicly enforced by `check_mem_op`, `check_scan`, and `check_stack`, so the verifier does not need to worry about a cheating prover manipulating the segment of any particular memory tuple.

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. 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); addresses are between 0 and the total size of the memory (the number of operations done in `check_scan`). $\mathcal{V}$ can check these ranges. The segment values are all publicly constrained.

If the field $\mathbb{F}$ is not large enough to pack the address, timestamp, and desired number of segments, one can use the same approach we use to compress the ℓ-bit value vectors: use linear combinations with powers of c_1. However, packing is a nice optimization, as multiplication by constants and additions are free in R1CS.