Practical Proofs of Parsing for Context-free Grammars

Harjasleen Malvai *UIUC/IC3*
Siam Hussain *Chainlink Labs*
Gregory Neven *Chainlink Labs*
Andrew Miller *UIUC/IC3*

Abstract

We present a scheme to prove, in zero-knowledge (ZK), the correct parsing of a string in context-free grammar (CFG). This is a crucial step towards applications such as proving statements about web API responses in ZK.

To the best of our knowledge, this is the first ZK scheme to prove the correctness of CFG parsing with complexity linear in the length of the string. Further, our algorithm flexibly accommodates different ZK proof systems. We demonstrate this flexibility with multiple implementations using both non-interactive and interactive proof paradigms.

Given general-purpose ZK programming frameworks, our implementations are not only compact (e.g., around 200 lines of code for the non-interactive version) but also deliver competitive performance. In the non-interactive setting, proving the correct parsing of a $\approx 1\text{KB}$ string takes 24 seconds, even for grammars with 2^{10} production rules. In the interactive setting the same proof takes just 1.6 seconds.

1 Introduction

Much of the world's digital information is structured according to standard formats that make them easier to process or parse by machines: static web pages are structured as HTML, dynamic web APIs usually communicate using JSON or XML, and email adheres to the SMTP standard.

Usually, a receiving machine will parse the entire document before locating and extracting the relevant information for its particular purposes. Imagine, however, a situation where the receiver does not have access to the full document. The document may be too large to download or store, for example, or the sender may not want to reveal its full contents. How can the receiver still be convinced that the document is correctly formatted, and that it contains some information that the sender chooses to reveal?

A zero-knowledge proof (ZKP) enables a prover to convince a verifier that they know a piece of information satisfying a certain condition, without having to reveal that information. Until recently, truly practical ZKPs only existed for simple mathematical or cryptographic statements, e.g., to prove knowledge of the discrete logarithm of a group element. Recent advances, however, have greatly improved the efficiency of ZKPs – it is now feasible to create a succinct proof of arbitrary statements that can be expressed as a circuit or as executable code.

This paper presents simple, efficient zero-knowledge proofs of correct parsing for the broad class of languages covered by context-free grammars (CFGs), which in particular includes the HTML, JSON, and XML languages that are popular in web services.

Applications. As a motivation for our work, consider the following practical applications:

- TLS-certified data: Most modern web servers protect their communication with clients using the Transport Layer Security (TLS) protocol. A client may want to convince an external verifier of some statement about the content of the communication with a server, but may not want to reveal the full transcript because it contains sensitive or irrelevant information.

  Some examples to motivate this requirement are as follows. A bank customer may want to prove financial liquidity by showing a partial account balance without revealing credentials or full financial details. An influencer may share engagement stats with sponsors while keeping her profile private. A blogger may include a verifiable quote from an article from a reputable source. A whistleblower may expose illegal content on a protected forum while preserving the privacy of innocent users.

  In general, exchanges with web APIs consist of transmitting structured documents, most commonly JSON or XML [23], transmitted over TLS. A recent line of work on TLS oracles [3, 27, 33, 38, 45] allows the TLS client to prove the authenticity of a TLS transcript with cryptographic guarantees. While the frameworks for proving authenticity (or provenence) of data from TLS oracles could be used off-the-shelf, simply proving authenticity of a TLS transcript is not sufficient for the fine-grained statements in the examples above. To this end, it is necessary to parse and filter content, using zero-knowledge. Unfortunately, the infrastructure for this purpose in existing work is rather brittle. For wider adoption of a TLS oracle solution, the first step is to provide simple and efficient algorithms for proofs of correct parsing, which may be easily implemented, even by non-expert programmers.

- Identity provision: Attribute authorities in an identity management (IDM) system certify users' attributes by issuing credentials, tokens or claims that can be verified by third parties. IDM systems often face a bootstrapping problem where existing authorities lack the knowledge, trust, or infrastructure to vet certain user attributes, while organizations with these capabilities do not issue tokens in a suitable format. Privacy-preserving IDM systems are particularly affected by this problem, because they require user attributes to be certified in a special cryptographic format, e.g., anonymous credentials [17, 32, 34, 36].

  Many vetting entities do give users access to their information in some form, whether by issuing signed OAuth, SAML, or JSON web tokens, or through a simple TLS interface. Proofs over structured data can break the deadlock by letting users prove a selection of their vetted attributes to a "re-issuing" authority, who verifies the proof and issues a credential of the appropriate form on the same attributes.

- Proving email authenticity: DomainKeys Identified Email (DKIM) [24] has email headers signed by the sender's domain, so that the receiver can verify that the message indeed originated from that domain. While originally intended to prevent email spoofing, it is also often used to prove the authenticity of leaked emails. Proofs over structured data make it possible to disclose only selected fields in the DKIM header, e.g., to reveal the sender but to hide the recipient address (who probably leaked the email).

- Light clients for smart contracts: Blockchains with support for smart contracts, like Ethereum and its many associated Layer-2 chains [15, 37], usually certify the root hash of the contracts' full state, either by including a Merkle root hash of the states in the block header, or by checkpointing the root hash at regular intervals.

  Suppose that a smart contract on a Layer-1 chain needs to be convinced about a statement about the state of second contract on a Layer-2 chain, also known as a "rollup". It is usually possible to reveal and authenticate the full state of the second contract using a simple Merkle proof. But if the full state of the second contract is too large to upload to the Layer-1 chain (which is rather likely, as the whole *raison d'être* of rollups is to be cheaper and more efficient than the Layer 1), or if the full state contains sensitive information that cannot be revealed on the Layer-1 chain (in particular, for zero-knowledge rollups that guarantee transaction privacy), it would be useful to upload just the required information, with a zero-knowledge proof of its authenticity with respect to the root hash of the Layer 2 chain. Our proofs of correct parsing can be used if the smart contract organizes its state in a context-free language such as JSON.

Related work. So far, work on proving privacy-preserving statements about strings in CFGs has relied either on (1) assumptions [45] about what is known about the string ahead-of-time, (2) partial redaction or revelation [35], or (3) executing [1, 2, 4] parsing algorithms in a ZKP system. Schemes in the first category assume that a JSON/XML only contains certain fields or each field has a unique key, resulting in a grammar that is not really a CFG due to such domain-specific assumptions. Schemes in the second category reveal the skeleton of the JSON (which reveals, for example, absence/presence of a field). For the third category, a naive implementation of input-oblivious parsing executes all possible branches for every character in the string, resulting in impractically slow proof generation. A more efficient implementation in [1] runs the parser through a ZK Virtual Machine (ZKVM) which hides the memory access pattern from the verifier. However, since even in plaintext the complexity of parsing in CFG is quadratic in input length, the complexity of the resulting proving scheme is at best quadratic. Moreover, currently there are no known ZKVMs for interactive proofs, which make those schemes impractical for proving correct parsing of strings obtained through interactive protocols like TLS.

On the other hand, parse trees for context-free grammars can be generated very quickly in a regular processor today, which leads to the crux of our solution.

Problem statement. Our main construction is a protocol for a prover to prove to a verifier, that it generated a valid parse tree for a string s, committed in a commitment com, based on a publicly known CFG $\mathcal{G}$.

Key insights. Observe that the applications that we are concerned with only require ensuring the correctness of the parse tree according to a CFG. Meaning, we only need to verify the correctness of a generated parse tree for a committed string, not actually emulate generating this tree in ZKP! The parse tree for a string of length n in a CFG has $O(n)$ nodes and edges, with only a small constant overhead [42]. In the proposed approach, the parse tree of the string is generated locally and input to the ZKP system as a private witness. The number of statements to be proven in ZKP is significantly smaller for proving the correctness of a parse tree compared to that for generating the parse tree.

The main idea of this construction exploits the fact that by definition, (1) a parse tree's leaves, concatenated from left to right must equal the original string, (2) every node in the tree must be a symbol in the grammar, and, (3) every non-leaf node branches out according to the rules of the grammar, thus reducing a parsing proof problem to a sequence of set-membership proof problems.

In addition to having a more efficient implementation in ZK, as prior works have done (e.g. [5]), we exploit programming language concepts to make CFG parsing more amenable to ZK. In ZK programming, all branches of input-dependent conditional statements must be executed. We observe that (1) if a grammar is in what is known as Chomsky Normal Form (CNF), then, the nodes in the parse tree have a much more limited branching factor, (2) every CFG can be translated into CNF, and (3) property (1) of CNF allows significantly simpler, efficient proof of a parse tree.

Practical realization. The proposed scheme is protocol agnostic and can be implemented in any proof system which provides proofs of NP statements. We demonstrate this with two realizations optimized for two different proving paradigms - a non-interactive proof based on arithmetic circuit and an interactive proof based on Boolean circuit. The complexities of these two versions are $O(|s|\,\log(|P|))$ for arithmetic circuit and $O((|s|+|P|)\,\log^3(|s|+|P|))$ for Boolean circuit, where $|s|$ and $|P|$ are string length and grammar size (number of rules), respectively. For a 1KB string and a grammar with 2^{10} production rules (on par with JSON), proof generation with a non-interactive ZK for the arithmetic circuit takes 24 seconds. In the same setting, proving with an interactive ZK protocol based on the Boolean circuit takes just 1.6 seconds.

Comparison with Reef [5]. In a recent work, Angel, et al. [5] propose an efficient proof of regular expression parsing. However, (1) parsing of regular expressions is significantly more straightforward than parsing context-free grammars, since they can be parsed by finite automatons, which require no backtracking, and (2) given that they work with finite automatons, their focus is on integrating finite automatons with a recursive proof system for practical gains. Moreover, they closely integrate with a particular ZKP system and program representation while our insight is more extensible and, can be used with any black-box proof system and commitment scheme. Besides, our scheme is amenable to recursive proof techniques and parallelization, and hence be combined with the techniques of Reef for further practical improvement.

2 Background and Definitions

In this section, we provide the requisite background relating to parsing and context-free grammars, as well as the cryptographic primitives which we will use in our constructions.

2.1 Strings, grammars and parsing

We use the notation $x\|y$ to denote the concatenation of two strings x and y and $|x|$ to denote the bit length of x.

Definition 1. *We denote a context-free grammar (CFG) as $\mathcal{G} = (V,\Sigma,S,P)$ where V is a set of non-terminal symbols, Σ a set of terminal symbols, $S \in V$ the starting symbol, and $P \subset V \times (V \cup \Sigma)^*$ a set of production rules.*

Given a context free grammar $\mathcal{G}$, let $\mathcal{L}(\mathcal{G})$ denote the language generated by $\mathcal{G}$, i.e., the set of all strings that can be generated by starting from the starting symbol S and following production rules in P until ending up with a string of terminal symbols.

For a string s a parser determines if $s \in \mathcal{L}(\mathcal{G})$, also written as $s \in \mathcal{G}$, by constructing a parse tree for s. The parse tree represents a sequence of production rules which can then be used to extract semantics.

Definition 2 (Chomsky Normal Form)**.** *A context-free grammar $\mathcal{G} = (V,\Sigma,S,P)$ is said to be in Chomsky normal form if $P \subset V \times (V^2 \cup \Sigma)$, i.e. any production yields either a pair of non-terminal symbols or a single terminal symbol.*

Remark 1. *Any context-free grammar $\mathcal{G}$ can be translated into another grammar $\mathcal{G}'$ in Chomsky normal form such that $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}')$ [42].*

Remark 2. *Given a string s of length n, if s is in a context-free grammar $\mathcal{G}$, written in Chomsky normal form, then the parse tree of s contains $3n-1$ nodes [42].*

2.2 Cryptography background

Zero-knowledge proofs. We will use the notation $R(x,w)$ to represent a relation, i.e. $R : (x,w) \to \{0,1\}$ and if $R(x,w) = 1$ we say that (x,w) satisfies R. A zero-knowledge proof scheme is a cryptographic protocol between two parties, a prover and a verifier. The prover and verifier agree upon a relation R and the prover would like to show that it knows some input (x,w) that satisfies R, without sharing w with the verifier. In particular, if the prover honestly generates a proof π such that the verifier, upon running the verification algorithm, is convinced that the prover knows an input (x,w) that satisfies R (a property called completeness). Further, the verifier learns nothing about the witness w, other than what it would learn by looking at R and x (this is the property of zero-knowledge). Finally, a prover who does not know a valid input (x,w) cannot generate a proof that verifies according to the verification algorithm (soundness[1]).

By convention, if R is the relation that is being proven using a zero-knowledge proof, then we say that x is the public input known to both the prover and the verifier and w is the secret input known only to the prover. We abstract out the use of a concrete zero-knowledge proof scheme and instead, define an ideal functionality $\mathcal{F}_{\text{ZK}}$ in fig. 15.

Vector commitments. A vector commitment scheme [16] is a cryptographic primitive that allows a party to commit to a vector $(v_1,\ldots,v_q)$, and later prove that a particular value v_i is committed at a particular position i. Merkle trees are a well-known instantiation of vector commitments, but there are others. We formally define the primitive in Def. 3. Note

[1]Note that the property formulated here is actually called knowledge soundness and is a technical requirement for UC proof formalization. For more details, see, e.g. [25].

that most works focusing on vector commitments (e.g. [40]) define vector commitments that can admit updates, of the form: *replace element v_i at position i with a new value v'_i*. In our case, since we do not need this capability, we define a simpler kind of vector commitment scheme, which we formally call a static vector commitment and defin in detail in Def. 3 (see App. B). Informally, a static vector commitment scheme VC provides algorithms VC.KeyGen to generate public parameters, VC.Commit to commit to a vector, VC.ProveCom to generate a proof that a particular position in the committed vector opens to a given value with respect to the commitment generated by VC.Commit and finally, VC.VerCom which verifies a proof of the value at a given position in the vector with respect to a commitment.

Another useful primitive, when using vector commitments is an aggregatable static vector commitment, which supplies the additional algorithms VC.ProveAgg to generate a sigle proof for the corresponding values for a batch of vector indices and VC.VerAgg, which verifies a batch of index-value pairs with respect to a commitment.

Note that any static vector commitment can be transformed into an aggregatable static vector commitment naively by instantiating the algorithm VC.ProveAgg by calling VC.ProveCom on each desired entry (i, v_i) and returning the vector of proofs π_i output by these invocations of VC.ProveCom. Correspondingly, VC.VerAgg would call VC.VerCom on each (i,v_i,π_i). In the rest of this work, when we refer to vector commitments, we will mean aggregatable static vector commitments, unless stated otherwise.

Accumulators. We define a slightly different primitive than a vector commitment scheme, called an accumulator. Note that the accumulators required in this work do not need to support dynamic operations, such as insertions or deletions (e.g. those defined by Camenisch and Lysyanskaya [12]). We simply want a tool to commit to unordered sets, so we provide definition and security properties more akin to the definitions in the work of Baric and Pfitzmann [7].

A static accumulator Acc comes with algorithms Acc.KeyGen (to generate some public parameters to commit to sets up to a specific size), Acc.Commit to commit to a set, Acc.ProveMem to prove the membership of a particular element in the set and Acc.VerMem to verify the proof of membership of a given element in a set with respect to its commitment. A more formal definition, Def. 4, is provided in App. B.

We only require sound accumulation schemes for our construction, i.e. an adversary should not be able to prove membership of an element not originally committed in the accumulator. We define this property more formally in Def. 5.

Note that some accumulation schemes also support non-membership proofs, but we omit that functionality for simplicity since it is not required for our applications. For more details on accumulators, see, for example [13, 28].

3 Components of proving correct CFG parsing

In this section, we provide intuition and then a more detailed algorithm for proving that a string s parses according to a context-free grammar $\mathcal{G}$. So far, we have made no distinction between proving that s parses according to a grammar $\mathcal{G}$ and proving that a particular parse tree is the parse-tree of the string s according to $\mathcal{G}$. For the rest of this paper, we actually provide proof of the stronger statement that a particular tree is the parse tree of a given string s with respect to the grammar $\mathcal{G}$.

3.1 A toy grammar

We first introduce a toy grammar to use as a running example. We define $\mathcal{G}_{\mathsf{toy}} = (V_{\mathsf{toy}}, \Sigma_{\mathsf{toy}}, S_{\mathsf{toy}}, P_{\mathsf{toy}})$ as follows: $S_{\mathsf{toy}} = \{S\}$, $V_{\mathsf{toy}} = \{$`S, A, B, C, AComma, BColon, Colon, Comma`$\}$, $\Sigma_{\mathsf{toy}} = \{$`b, c, ':', ','`$\}$ and define P_{toy} in fig. 1. Note that $\mathcal{G}_{\mathsf{toy}}$ is in Chomsky normal form. Some examples of strings in $\mathcal{G}_{\mathsf{toy}}$ are:

- `b:c`
- `bb:c,b:cc`

We consider the parse tree for the string `bb: c, b: cc`, given in fig. 2. Observe that the tree in fig. 2 has the following properties:

- The leaves, ordered left to right, concatenate to the string `bb: c, b: cc` and are actually terminals in $\mathcal{G}_{\mathsf{toy}}$.
- All non-leaf nodes are non-terminals in the grammar $\mathcal{G}_{\mathsf{toy}}$.
- Consider a non-leaf node and its children, for example, the node labelled `S` and its children `AComma` and `A`. When ordered left-to-right, the children of `S`, are the right hand side of a production rule, specifically: `S` $\rightarrow$ `AComma A`. This applies to all non-leaf nodes.

Also, since $\mathcal{G}_{\mathsf{toy}}$ is in Chomsky normal form, if a non-leaf node has two children, these children must be non-terminals. Otherwise, this non-leaf node must have exactly one child and it must be a terminal node in the grammar.

In Section 3.2, we will generalize these observations to create a protocol for a party to prove the correctness of a parse tree for a given string in a particular grammar.

3.2 Conditions for correctness of a parse tree

Recall that a parse tree for a grammar consists of applications of production rules from that grammar to form a tree whose root is a starting symbol, whose leaves are terminal symbols in the grammar and whose non-leaf nodes are non-terminals. In our construction, we assume for simplicity that the parse

```
S                   BColon        C
    AComma S            B Colon       c
    AComma A                          CC
                    B
AComma                  b         Comma
    A Comma             BB            ','

A                   Colon
    BColon C            ':'
```

Figure 1: Production rules for the toy grammar $\mathcal{G}_{\mathsf{toy}}$.

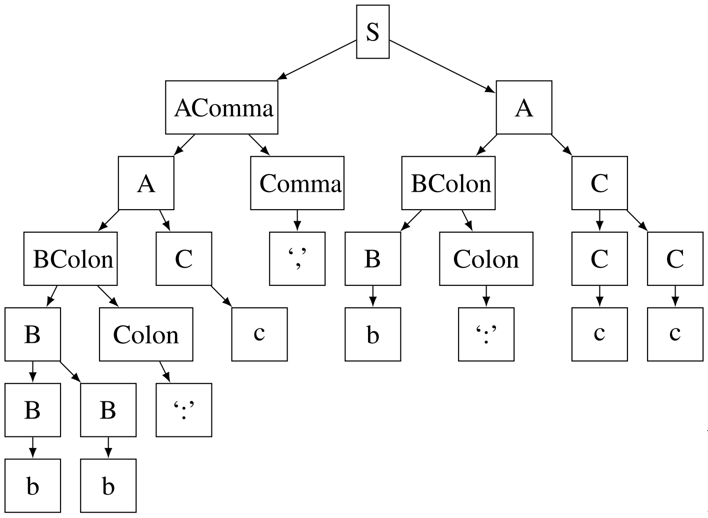

Figure 2: Parse tree for the string `bb:c,b:cc` in $\mathcal{G}_{\mathsf{toy}}$.

tree we would like to prove the correctness of, is generated according to the grammar $G=(V,\Sigma,S,P)$ in Chomsky normal form, where V is the set of non-terminals, Σ is the set of terminals, S is the set of start symbols and P is the set of production rules. While our construction can be generalized to a context-free grammar in any form, it is simpler to explain and implement it obliviously if we assume the grammar is in Chomsky normal form.

Correctness conditions. A binary tree $T = (\mathsf{Verts}, \mathsf{root}, \mathsf{Edges})$ is a directed graph with vertices Verts, root node $\mathsf{root} \in \mathsf{Verts}$, and edges $\mathsf{Edges} \subseteq \mathsf{Verts} \times \mathsf{Verts} \times \{\mathtt{left},\mathtt{right}\}$ that is

- binary, i.e., all $v \in \mathsf{Verts}$ have at most one left child l and at most one right child r such that $(v,l,\mathtt{left}),(v,r,\mathtt{right}) \in \mathsf{Edges}$;
- connected, i.e., for all $v \in \mathsf{Verts}$ there exists a path from root to v, meaning that there exists $(v_0,\ldots,v_m) \in \mathsf{Verts}^*$ with $v_0 = \mathsf{root}$, $v_m = v$, and $(v_i,v_{i+1},\cdot) \in \mathsf{Edges}$ for all $0 \leq i < m$;
- acyclic, i.e., there exists no $v \in \mathsf{Verts}$ with a path from v to v.

A parse tree (T, label) conforming to the grammar G consists of a binary tree $T = (\mathsf{Verts}, \mathsf{root}, \mathsf{Edges})$ and a labeling function $\mathsf{label}: \mathsf{Verts} \to V$ such that:

- the root of the tree is labelled with a valid start symbol, i.e. $\mathsf{label}(\mathsf{root}) \in S$.
- each node $v \in \mathsf{Verts}$ forms a valid production rule of G with its children, meaning that either
  - v has two children $(v,l,\mathtt{left}),(v,r,\mathtt{right}) \in \mathsf{Edges}$ and $(\mathsf{label}(v),\mathsf{label}(l),\mathsf{label}(r)) \in P$, or
  - v only has a left child $(v, l, \mathtt{left}) \in \mathsf{Edges}$ and $(\mathsf{label}(v),\mathsf{label}(l)) \in P$, or
  - v has no children and $\mathsf{label}(v) \in \Sigma$.

The tuple (T,label) is a parse tree of a string s according to grammar G if and only if it is a valid parse tree conforming to G and the concatenation of the labels of the leaf nodes, ordered depth-first from left to right, is the string s. See Figure 2 for an example parse tree in our toy grammar.

3.3 Algorithm for proving correctness of a parse tree

We first propose an appropriate data structure to represent the parse tree in zero-knowledge proof programming frameworks. A first approach could be to model the binary tree T and the labeling function label in a simple structure: and refer to the

```
struct Vertex:
    # Strawman for structs to represent a tree
    label: Terminal | Non−Terminal
    children: Array<Vertex>
```

Figure 3: A simple representation for the parse tree node.

tree with a pointer to the root node.

Recall that in zero-knowledge proofs, recursive functions and data structures most often need to be fully unrolled at compile time. This points to two shortcomings of our struct. First, when traversing the children array, the size of the circuit must be fixed at compile time to use its worst-case length. Using the Chomsky Normal form of the grammar easily resolves this issue, however, because a node in a CNF parse tree can have at most two children. Thus, we can update our vertex struct to:

```
struct Vertex:
    # Strawman for structs to represent a tree
    label: Terminal | Non−Terminal
    childL: Vertex | None
    childR: Vertex | None
```

Figure 4: A simple representation for the parse tree node of a string in a Chomsky Grammar Form.

The second shortcoming is that a recursive struct is not ideal for a 'proof-system and grammar agnostic' algorithm to proof parsing, because the depth of a parse tree varies considerably between grammars.

Our third attempt therefore avoids recursive structs by observing that for a grammar $\mathcal{G}$ in Chomsky Normal Form and a string s of length n, the parse tree consists of $3n-1$ vertices and $3n-2$ edges. Thus, we could update the vertex representation to consist of its label and a unique index, such as its index in a depth-first traversal of the tree, and let an edge consist of a pair of vertices (parent and child). While this approach is obviously input-oblivious, in that we only use non-recursive structs, it is not obvious how one might ensure that the graph represented by these vertices and directed edges is acyclic (i.e., a tree), without running a cycle detecting algorithm. Even this issue has a simple workaround: if the array of edges input is ordered as follows: $(v_1,v_2) < (v_3,v_4)$ if $(v_1.index < v_3.index)$ OR $((v_1 = v_3)$ AND $(v_2.index < v_4.index))$, thus, we can impose this condition as well.

Upon implementing, however, we found that such a tree representation consisting of nodes and edges, without consolidated productions, led to a slowdown in verifying a tree, since one of the properties to be verified for the tree is that each production is in the set of valid productions. Thus, we finally arrive at the following, two simple structs:

```
struct Vertex:
    label: Terminal | Non−Terminal
    id: int # The DFS index of this node
```

Figure 5: A vertex in the parse tree of a string in a grammar in Chomsky Normal Form.

```
struct Prod:
    # Struct that demarcates each production in the tree
    parent: Vertex
    left: Vertex
    right: Vertex | None
    # None case happens if childL represents a terminal
```

Figure 6: The production struct.

Then, we define a parse tree Π = Prods, where Prods is the array of all productions sorted by the parent index. The length of Prods is $2n-1$, because that's the number of non-leaf nodes in the parse tree.

To prove that the tree is a correct parse tree conforming to the grammar for string s, we can use the following algorithm that traverses the tree in depth-first order using a stack:

Claim 1. *If the check_parse_tree algorithm from Figure 7 returns true, then* Prods *represents a valid parse tree of string s according to grammar* $G = (V,\Sigma,S,P)$.

We provide the full proof for this claim in App. A.

```
1  function check_parse_tree(Prods,$s$,$P$):
2      n = |s|
3      stack = empty
4      expected = new Vertex(S, 0)
5      leaf_counter = 0
6      assert(|Prods| == 2n−1)
7      for i in 0..2n−2:
8          (parent, left, right) = Prods[i]
9          assert(parent == expected)
10         assert(left != None)
11         assert(left.id == parent.id + 1)
12         if right == None:
13             assert((parent.label, left.label) $\in P$)
14             assert(s[leaf_counter] == left.label)
15             leaf_counter += 1
16             if i < 2n−2:
17                 expected = stack.pop()
18                 assert(expected.id == left.id + 1)
19         else:
20             assert((parent.label, left.label, right.label) $\in P$)
21             expected = left
22             stack.push(right)
23     assert(leaf_counter == $|s|$)
24     assert(stack == empty)
```

Figure 7: The parse tree verification algorithm.

3.4 Instantiating assertions in Figure 7

Here, we briefly discuss the tools that can be used to instantiate the various assert statements.

Checking membership in production rules. The checks for production rules in the committed grammar (lines 7 and 20) are simple set membership proofs. This can be instantiated by committing to the set P of production rules using a set accumulator as defined in Def. 4.

Checking string consistency. We discuss various instantiations for checking string consistency in Sec. 4. Here, we note that if the string is committed using any vector commitment, as defined in Def. 3, the check on line 14 can be instantiated using the constraints for the VC.VerMem operation, with the index being the `leaf_counter`. Note that if the vector commitment to the string supports a VC.VerAgg operation which is more efficient that individually verifying string characters, the proving algorithm could be optimized by batch verifying the items of the string.

4 Efficient Proof Generation

We implement the `check_parse_tree algorithm` in Figure 7 in two proof systems: (i) a non-interactive proof based on an arithmetic circuit, and (ii) an interactive proof based on a Boolean circuit. The optimization strategies in these two systems differ from each other due to the nature of the circuits and efficiencies of different components in different proof systems. In this section we present the optimizations in both systems. Finally, we discuss recursion supporting proof systems and how they can be used to prove correct parsing for significantly larger strings in practice.

Note that these strategies can be carried over to any proof system which has similar properties to the stated proof system. In particular, the optimizations in Sec. 4.1 can be applied to any arithmetic circuit-based implmentation and those in Sec. 4.2 can be applied to any boolean circuit-based solution. Finally, in Sec. 4.3, we provide trade-offs and considerations when using a recursion-supporting proof system.

4.1 Non-interactive Proof: Arithmetic Circuit

4.1.1 Building Blocks

Accumulator. An accumulator allows committing to a set of unique elements and then proving that a particular value is a member of that set. In particular, we use a simple accumulator, without additional properties such as insertion or deletion, defined in Def. 4. We use the well-known Merkle tree as the accumulator in our implementation. It requires $O(\log(n))$ hash functions per membership proof for a set of size n.
Oblivious stack. The primary challenge in oblivious stack is hiding the push/pop patterns. At every push/pop operation, a condition c is provided as the prover's private input and the operation only happens if c is `true`. In a naive implementation, the stack is instantiated with d locations where d is the maximum possible depth. At every conditional operation, all d locations needs to be updated through a multiplexer circuit. The complexity of each operation would be $O(d)$. In our design, the circuit maintains a running hash st of the stack state after every operation. For every push, the circuit computes $st' = H(x \| st)$ and set the new value of st to st' only if c is `true`. For every pop, the prover commits to the hash preimage $x \| st$ and the circuits verifies that the new state st' satisfies $st' = H(x \| st)$ if c is `true`, $st' = st$ otherwise. Thus the complexity of every operation is $O(1)$, and is independent of the stack depth. Note that this technique is akin to the basic stack presented in Reef [5], for recursive proving but we found that it was beneficial even in the absence of a recursive prover setting to implement a DFS traversal of the parse tree.

4.1.2 Circuit Design

Grammar correctness. We need to prove that every member of the production array Prods is a member of the set P (lines 13 and 20 of the `check_parse_tree algorithm` in Figure 7). In our implementation, we make a slight modification in the representation of each Prod . We specify a special symbol for `None` and represent each terminal Prod as (`parent`,`left`,`None`). Following this convention for both Prods and P allows us to merge the two operations in lines 13 and 20 into one and take it out of the `if` condition. In our implementation, the Merkle root of P is a public input. For each element in Prods, the prover provides a Merkle proof of membership in P. Each proof requires $\log(|P|)$ hash operations which makes the complexity of this step $O(|s| \log(|P|))$.
String consistency. At line 14 of the `check_parse_tree algorithm`, we assert that all the terminal left labels concatenate to form the input string s. This operation requires private index access which in general have a complexity of $O(|s|)$ per access resulting in a total complexity of $O(|s|^2)$. In our design, we employ similar ideas to the oblivious stack. We initialize a hash chain $h_s = hash('\ ')$. For every element Prods$[i]$ in Prods, we compute the hash $h'_s = hash(h_s \| \text{Prods}[i].\texttt{left.label})$. Then the hash h_s is updated to h'_s if Prods$[i]$ is a terminal, if not h_s is unmodified. The final value of h_s is compared against the hash chain computed on the characters in s. The complexity of this step is $O(|s|)$.
Tree consistency. This check is performed by direct application of the oblivious stack. Since the `check_parse_tree algorithm` requires $2(2|s|-1)$ push/pop operations, the complexity of this step is $O(|s|)$.

4.2 Interactive Proof: Boolean Circuit

4.2.1 Building Blocks

Unlike arithmetic circuit in non-interactive proofs, hash functions are one of the most expensive operations in Boolean circuit in interactive proofs. However, comparison operation is pretty cheap in Boolean logic which leads to efficient data-oblivious sorting. Comparison is also a crucial component of the oblivious stack design used in our work. In this section, we first present three building blocks of the Boolean circuit implementation of the parser - oblivious sorting, counting the number of unique elements in an array and oblivious stack,. Then we present an efficient implementation of the `check_parse_tree algorithm` of Figure 7 in Boolean circuit.
Oblivious sorting. The sorting circuit is based on bitonic sort [8] which requires $O(n\log^2(n))$ comparisons and has a depth of $O(\log^2(n))$, where n is the number of elements in the array. In this work, we use a constant round ZK protocol, therefore the number of rounds is independent of the circuit depth. Each comparison requires b `AND` gates where b is the number of bits in each element. Thus the complexity of the oblivious sort in terms of the number of `AND` gates is $O(bn\log^2(n))$
Counting the unique elements in an array. Counting the number of unique elements in a sorted array is straight forward. Perform equality check between each pair of adjacent elements, and increment the counter if they are not equal. It has a complexity of $O(bn)$.
Oblivious stack. As mentioned in the previous section, the complexity of each push/pop in a naive implementation would be $O(d)$. We use the oblivious stack design presented in [44]. It proposes a hierarchical structure with $\log(d)$ levels. Each level i holds 5 slots of depth 2^i. The i-th slot is updated

at each 2^i-th push/pop. The amortized cost of each push/pop operation is $O(b\log(d))$.

We are now ready to present the optimize Boolean circuit design for Algorithm 7. We use the sorting and stack operations as black boxes in the rest of the section. They can be replaced with better designs, if available.

4.2.2 Circuit Design

Grammar correctness. We need to prove that every member of the production array Prods is a member of the set P. In Boolean circuit, the most practical way of checking the membership of a single value in a set is linear scan, which requires $O(|P|)$ equality checks. Thus checking the membership of each element in Prods individually would require $O(|s||P|)$ equality checks. We present a more efficient batch membership check which reduces the amortized cost. In our design, we concatenate P and Prods to form the combined array $P' = P \parallel prods$. If each element of Prods is a member of P, the number of unique elements in P' should be equal to $|P|$. The most expensive operation in counting the unique elements is sorting which has a complexity of $O(b(|s|+|P|)\log^2(|s|+|P|))$.

String consistency. In this step (line 14 of the `check_parse_tree algorithm`), we assert that all the terminal left labels concatenate to form the input string s. This operation requires private index access which in general have a complexity of $O(b|s|)$ per access resulting in a total complexity of $O(b|s|^2)$. We present an optimized circuit based on the observation that the private indices appear in an increasing order (see line 15 of the algorithm). In the proposed circuit, all the left child vertices are collected in an array s' where the *id* is modified as follows. If the production is non-terminal, *id* is set to $2|s|-1$ (higher than the maximum number of productions). Otherwise, *id* is left unmodified. Then, the array s' is sorted. If the parse tree indeed correspond to the committed string s, the first $|s|$ elements of s' after sorting should be equal to s. The complexity of this step is the same as sorting which is $O(b|s|\log^2(|s|))$.

Tree consistency. This check is performed by direct application of the oblivious stack circuit with a depth of $|s|$. The amortized cost of each push/pop is $O(b\log(|s|))$. Since the `check_parse_tree algorithm` requires $2(2|s|-1)$ push/pop operations, the complexity of this step is $O(b|s|\log(|s|))$.

Overall complexity. The complexity of the end-to-end implementation is $O(b(|s|+|P|)\log^2(|s|+|P|))$. We can set $b = O(\log(|s|))$. Moreover, in most practical settings, $|s| \gg |P|$. Therefore, the end-to-end complexity is $O(|s|\log^3(|s|))$.

Switching to the hash based design. In Sec. 4.1, we presented an arithmetic circuit based on hash chains with a complexity of $O(|s|)$. It requires $20|s|$ instances of the hash function. If the number of `AND` gates in the Boolean circuit of the hash function is h, the total number of `AND` gates is $20h|s|$. The number of `AND` gates in the proposed Boolean circuit is $5|s|\log^3(|s|)$. Therefore, we could switch to the hash based design if $\log^3(|s|) > 4h$. The string length $|s|$ that satisfies this condition would be impractically large for any known hash circuit.

4.3 Recursion-supporting proof systems

While non-interactive proof systems such as PLONK [30] or Groth'16 [31] provide efficient verification and small proof size, they rely on one-time trusted setup, sometimes followed by setup for a given program and a higher prover complexity to speed up proof and verification time. The setup for program often incurs high RAM overhead (see e.g. the discussion in [10]). As a concrete example, in the case of our Noir implementation, the setup for a proof of correctness for the parse tree of a string of size 2^9 with respect to a grammar of size 2^7, had maximum occupied RAM[2] over 10GB, while a proof for the same required over 11.5GB.

As the authors of Reef [5] pointed out, one workaround for the RAM limitations for a proof system is to use a recursive proof system.

4.3.1 Recursive proofs to overcome RAM limitations

IVC-based proving. The algorithm in fig. 7 has repeated applications of the same function in a `for` loop, which is particularly amenable incrementally verifiable computation (IVC) [39]. In our instantiation of an IVC-based implementation, each IVC iteration after the first one consists of (1) verifying the proof generated by the previous IVC iteration and, (2) generating a proof for the next batch of K iterations of the `for` loop in lines 7-22 of fig. 7, for a pre-selected parameter K. The proof generated by the last such iteration is then verified by the verifier.

Passing state between IVC provers. Since we use a hash-chain to implement the stack, iteration i checks that the end-state of its stack hashes to some public input h_i, and h_i is used as the starting hash of the stack state in iteration $i+1$. Similarly, the most recent state of the requisite variables (e.g. the leaf counter that tracks the string traversal) can be passed from on iteration of the IVC prover to the next.

4.3.2 Parallelized proving for faster wall-clock time

A chain of proofs, as is used in IVC is more efficient on the RAM front, however, requires each proving iteration to wait for its predecessor's proof to be generated. Inspired by [18], we propose distributing the proof generation using recursive proofs.

Simple PCD tree. One solution for leveraging parallelism in proof generation is proposed in Chiesa et al.'s work. Figure 9

[2]Maximum resident set size measured using the `gtime` [29] command on a Mac.

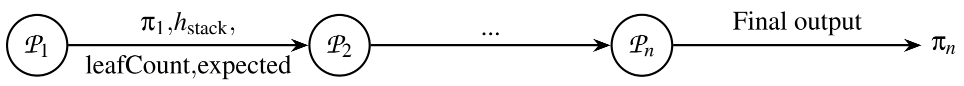

Figure 8: An illustration of an IVC-based solution to implementing fig. 7, here each node represents a proof of K iterations of the `for` loop and the requisite state is passed on to the next prover, along with the proof π generated by prover $\mathcal{P}_i$.

shows an example of such a technique for generating proof of correct parsing. The iterations in fig. 7 are proven in batches similar to the IVC case. However, instead of verifying the proof for one batch in the prover for the next (e.g. $\mathcal{P}_1$ and $\mathcal{P}_2$ in fig. 8) proofs for two consecutive sets of iterations can be generated independently e.g. by $\mathcal{P}_1$ and $\mathcal{P}_2$. A third prover $\mathcal{P}_3$ then verifies (1) the proofs generated by $\mathcal{P}_1$ and $\mathcal{P}_2$, and (2) the provenance of the relationship between the iterations proven by $\mathcal{P}_1$ and $\mathcal{P}_2$ by ensuring that the output state of the needed variables by the end of the iterations proven by $\mathcal{P}_1$ is the same as the input state of those variables used by the iterations proven by $\mathcal{P}_2$. We end up with a dependency graph between provers, where $\mathcal{P}_i$ points to $\mathcal{P}_j$ if $\mathcal{P}_j$ will verify the proof output by $\mathcal{P}_i$ and in the simplest form, this graph is a binary tree, as in fig. 9. Although, naively such a scheme results in twice as many provers as in the IVC case, depending on the proof system, the non-leaf (recursive) provers such as $\mathcal{P}_5$ and $\mathcal{P}_6$, may be more efficient than large leaf provers. Most importantly, provers on the same level in the tree can be run in parallel providing benefits in wall-clock time on multi-core devices, as we will see in Sec. 5.5.

PCD tree trade-offs. In the simple PCD tree described above, observe the following: (1) the tree is binary, and (2) certain provers only verify proofs, while others actually prove iterations of the algorithm in fig. 7. This can be changed in various ways, for instance by increasing the arity of the PCD tree i.e. verifying more proofs at a time in a recursive prover, and additionally by combining PCD and IVC. In Sec. 5.5 we discuss this in a bit more detail.

5 Implementation and Evaluation

In this section, we describe our implementations of the parsing proof protocol across two proof systems and in the context of recursion (see Sec. 5.1), followed by an evaluation across relevant experimental setups (which are described in Sec. 5.2). Finally, we provide detailed evaluation for our non-interactive (see Sec. 5.3) and interactive (see Sec. 5.4) implementations, followed by a simulated evaluation for our recursive solution, based on microbenchmarks of its components in Noir, in Sec. 5.5.

5.1 Proof Systems

We implement the presented parsing algorithm in two different proof systems. For the non-interactive proof, we use Noir [22] with the Barretenberg [19] backend. For interactive

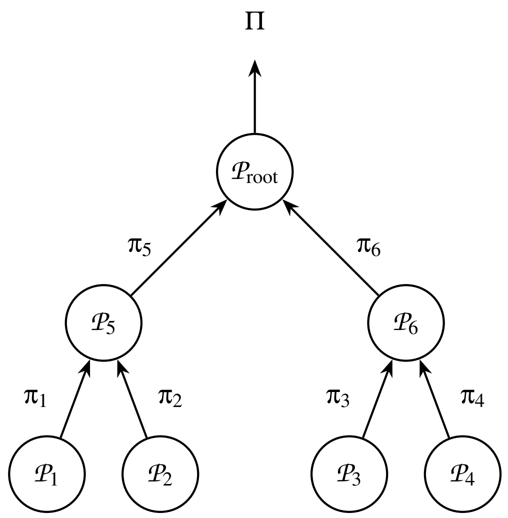

Figure 9: An illustration of a parallelized prover using a recursive proof-system. Each of $\mathcal{P}_1$ through $\mathcal{P}_4$ verifies $1/4$ of the iterations in fig. 7 and receives as public input the hash of the stack state, the expected parent for the next production and leaf count at the start and end of its iterations and outputs their proofs π_1 through π_4. $\mathcal{P}_5$ verifies π_1 and pi_2 with their public inputs (and similarly with $\mathcal{P}_6$). $\mathcal{P}_5$ takes as input the starting state of the variables for $\mathcal{P}_1$ and the end state for $\mathcal{P}_2$, similarly for $\mathcal{P}_6$. Finally $\mathcal{P}_{\text{root}}$ has as public input the expected final leaf count at the end of fig. 7 and verifies that the stack depth at the start and end of $\mathcal{P}_1$ and $\mathcal{P}_4$ is 0. The final proof π is verified by the verifier.

proof we use Quicksilver [43] from emp-toolkit [41]. Note that the techniques described in Section 4 are fairly generic and cam be implemented in most of the available proof systems of the respective classes with minimal modification.

Noir's non-interactive proof system. Noir [22] is a zero-knowledge proof-programming framework, with Rust-like syntax as well as a package manager and build-system called Nargo [21], similar to Rust's Cargo [26]. Noir compiles to an intermediate language called ACIR, designed with the intention of allowing different proof-systems, which they refer to as backends, to translate the ACIR into their respective constraint systems and correspondingly generate proofs. As of this writing, the latest version of Noir's toolchain (specifically Nargo) is 0.32.0 [20], which provides language support for recursion. However, as of this writing, the only backend that we found to be compatible with Nargo 0.32.0 is the default backend prover, provided by Aztec Labs, called Barretenberg which, by default uses the PLONK [30] proof system with the BN254 elliptic curve [6, 11].

Recursion in Noir. Noir also provides support for recursive proving through a standard library `verify_proof` function in the Noir language. The `verify_proof` takes as input a proof and a verification in field-form, the public inputs for this proof and a key hash, which can be part of a public input to ensure the correct program is being recursively verified. We found, however, that the output proof sizes for a program that includes a call to `verify_proof` and one that doesn't are actually different, necessitating the need for 3 different implementations to get even an IVC proof. These three implementations are: (1) a base implementation to prove that some K iterations of fig. 7's loop run correctly, (2) a second implementation that includes the base implementation as well as a called to `verify_proof` with the input `proof` an array of 93 field elements, and (3) a third imeplementation which is exactly the same as (2) except the type of the input `proof` from the previous iteration is an array of 109 field elements. Note that (2) can only verify the proof generated by (1), while the prover implemented in (3) supports verification for both itself and the proof generated by (2).

Quicksilver. The most efficient non-interactive proofs at present are based on Vector Oblivious Linear Evaluation (VOLE) [9]. Primary benefits of this proving paradigm are (i) constant round, which makes the proving time independent of circuit depth, and (ii) small communication per gate, especially for large circuits. We chose Quicksilver in this work because it reports the fastest proving/verification time. Quicksilver is part of the emp-toolkit library which provides an integrated circuit design and proof system. We used v0.2.1 of the library.

5.2 Experimental Setup

In our evaluation, we used two different platforms for non-interactive and interactive proofs since the requirements of the two proof systems are widely different - the non-interactive proofs are compute-intensive while the interactive proofs depends on the network bandwidth.

- For the non-interactive proof, we use AWS EC2 instance `g3.8xlarge` which has 32 CPUs and 244 GB memory.
- For the interactive proof, we use two (one as prover, one as verifier) AWS EC2 instances `c6gn.8xlarge` which has 32 CPUs, 64 GB memory and a bandwidth of 50 Gbps.

5.3 Non-interactive proof without recursion

Let us first provide an idea of the practicality of the proposed design. For a grammar with 1024 rules (on par with the size of JSON grammar), proving the correct parsing of strings with 2^9 characters (1KB in UTF-16) take 24.5 seconds. We now provide evaluations in a wide range of the parameter sizes.

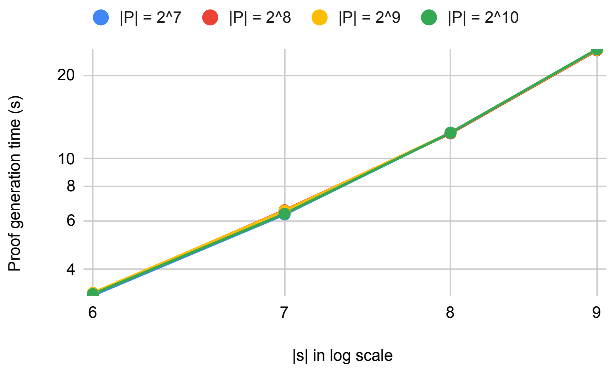

Figure 10: Run-time in the non-interactive proof as a function of the string length $|s|$ and grammar size $|P|$. As this graph shows, the string size is the dominant factor in determining the cost for the prover, even if we use a simple MT-based accumulator to commit to P.

Proving time. We show the proof generation time with Noir as a function of the string length $|s|$ and grammar size $|P|$ in Figure 10. As explained in Section 4.1, the number of constraints is linear in $|s|$ and logarithmic in $|P|$. Therefore, the proof generation time mostly depends on $|s|$.

Note that the proving time does not include the time to generate the proving keys and witnesses.

Verification time. As expected, the verification time for the PLONK prover underlying Noir's implementation consistently takes about the same time on a laptop, independently of the instance size (i.e. $|s|$ and $|P|$). Our verifier on a commidity machine with an Apple M1 Max processor and 32GB of RAM, the verifier consistently took under 0.14ms.

Proof Size. The proof size is 2.37KB irrespective of the string length or grammar size.

5.4 Interactive Proof

For this evaluation, we set the number of bits in each label to 16 (enough for UTF-16 encoding) and the number of bits in the each id to 25 (enough for a string with 10^7 characters or 1MB size).

First, we provide an idea of the practicality of the proposed design. For a grammar with 1024 rules (on par with the size of JSON grammar), proving the correct parsing of strings with 2^{12} and 2^{16} characters (8KB and 128KB, respectively) take 16.8 seconds and 6.7 minutes, respectively. In the interactive setting, we were even able to prove the correct parsing of a 2^{19} character (1GB) string in 1 hour. We now provide evaluations in a wide range of the parameter sizes.

Number of AND gates. As a protocol-agnostic evaluation of our implementation in the non-interactive proof system, we present the number of AND gates as a function of the string length $|s|$ and grammar size $|P|$ in Figure 11. For shorter strings, the number of AND gates depends both on the string length and grammar size. However, for longer strings, the performance is dictated by the string length. Moreover, the

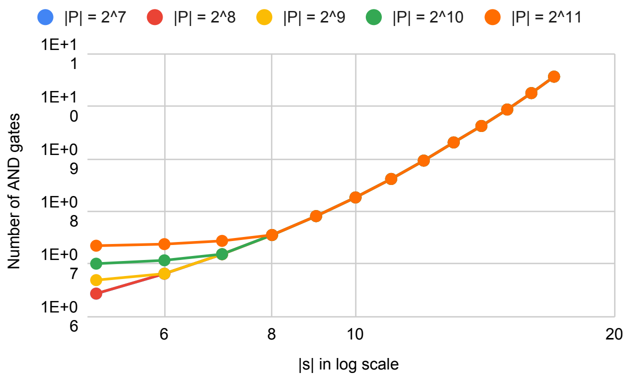

Figure 11: Number of AND gates in the interactive proof as a function of the string length $|s|$ and grammar size $|P|$

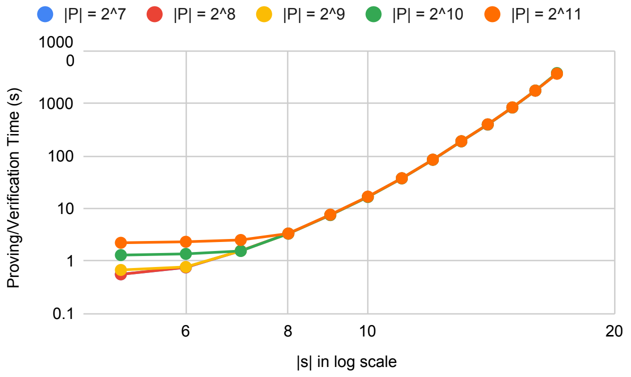

Figure 12: Run-time in the interactive proof as a function of the string length $|s|$ and grammar size $|P|$.

number of comparisons in bitonic sort is the "next power of 2" of the length of the input array. Therefore, often the grammar size does not have any effect on the total number of AND gates.
Runtime. We now present the run-time with Quicksilver in Figure 12. In the interactive setting prover and verifier have the same run-time. The run-time follows the trend in the number of AND gates, as expected.
Cost breakdown in different steps. We show the cost break down in terms of the number of AND gates at different steps of the proof described in Section 4.2 in Figure 13. It shows that the grammar size dominates the total cost for shorter strings but is largely inconsequential for larger strings. Moreover, the share of the cost by string consistency check increases with increase in string length. This is because the tree consistency check has a complexity of $O(|s|\log^2(|s|))$ whereas the string consistency check has a complexity of $O(|s|\log^3(|s|))$.

5.5 Recursive proof-based solution

In this section, we discuss the prover time for a simulated PCD solution. We describe the formulae one can use for such an estimation, based on the average time for generating one recursive proof and the proofs for a subset of the productions

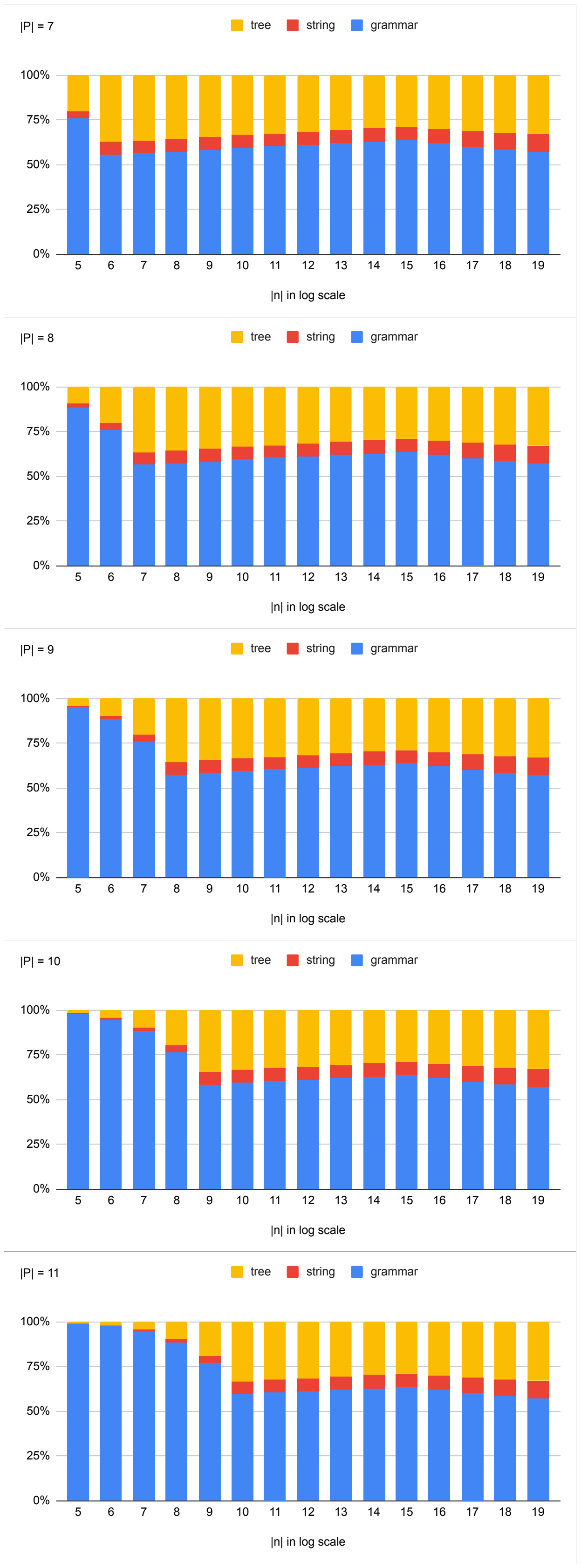

Figure 13: Cost breakdown in terms of number of AND gates as a function of the string length $|s|$ and grammar size $|P|$.

to be verified for a given string.

Note that the simplicity of our original algorithm still holds, the additional complexity introduced in the solution here is only for a recursive solution using PCD.

IVC recursion. While IVC recursion provides an improvement in concrete proving time on a machine due to reducing RAM consumption, we do not evaluate it here since (1) we believe the wall-clock time performance of the parallelizable proof will be better on most modern machines, and (2) this solution works very similarly to the implementation already discussed at length in [5].

Calculating wall-clock time for a recursive prover. To calculate this time, let us first define a few variables. Let t_{base} be the time it takes to generate the proof for K iterations of the for loop in fig. 7 and let t_{rec} be the amount of time for proving the verification of two proofs. To use a scheme as illustrated in fig. 9 to generate parallel proofs for a string with 2^{n-1} characters i.e. $2^n - 1$ productions. Let us assume that there are enough available cores such that each prover at the same depth of the tree can run in parallel. Then, the wall clock time for generating a fully parallelized proof, given pre-generated witnesses, would be

$$t_{\text{base}} \times \frac{2^n}{K} + t_{\text{rec}} \times (\log 2^n / K - 1).$$

Recall that in Noir, the proof size of the proof generated by a prover that contains a `verify_proof` call (i.e. implements a recursive verifier) and a prover that does not verify another proof are different. If the recursive verifier for a non-recursive proof runs in time t'_{rec} and the recursive verifier for a recursive proof runs in time t_{rec}. In fig. 9, the time t'_{rec} would correspond to the time for $\mathcal{P}_5$ and $\mathcal{P}_6$ to generate their proofs and t_{rec} would correspond to the time taken by $\mathcal{P}_{\text{root}}$ to generate its proof. Then the formula above can be updated to

$$t_{\text{base}} \times \frac{2^n}{K} + t'_{\text{rec}} + t_{\text{rec}} \times (\log 2^n / K - 2).$$

Working with limited cores. Now, suppose we would like to use a machine with C cores to generate the proof for a string with 2^{n-1} characters ($2^n - 1$ productions). Then, the wall clock time for generating a parallelized proof, given pre-generated witnesses in general, would be

$$t_{\text{base}} \times \frac{2^n}{CK} + t_{\text{rec}} \times \left(\sum_{i=1}^{\log(2^n/K)-1} \left\lceil \frac{2^n}{2^i CK} \right\rceil \right).$$

As before, in Noir, the prover that verify the outputs of non-recursive provers have a run-time t'_{rec} slightly lower than t_{rec} and this leads to the formula:

$$t_{\text{base}} \times \frac{2^n}{CK} + t'_{\text{rec}} \times \left\lceil \frac{2^n}{2CK} \right\rceil + t_{\text{rec}} \times \left(\sum_{i=2}^{\log(2^n/K)-1} \left\lceil \frac{2^n}{2^i CK} \right\rceil \right).$$

We measured the time for the recursive verifiers t'_{rec} and t_{rec} to generate proofs of 2 verifications.

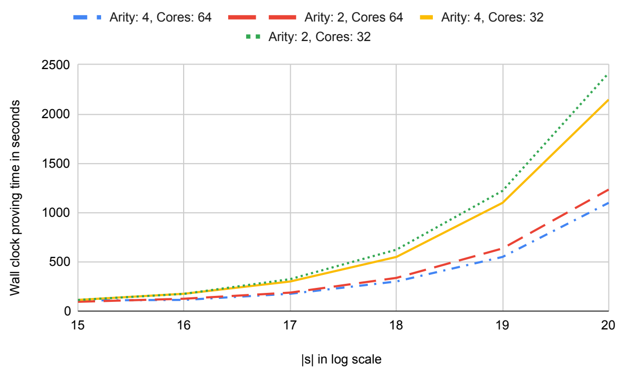

Figure 14: Simulated wall-clock time for using a PCD-based proving approach, as described in Section 5.5.

Evaluation of recursive solution wall-clock time. Based on these formulae, and the empirical cost for proving in Noir that (1) 2 `verify_proof` verifications output 1, (2) 4 `verify_proof` calls output 1, and (3) that 1024 iterations of the `for` loop in fig. 7 verify, we simulate the numbers in fig. 14. The arity denotes the number of proofs verified by a non-leaf node in the PCD tree, and we simulate the wall-clock proving time for 64 and 32 cores, assuming one core can only run one prover at a time. This time will be an over-estimate if more than one proof at a time can be generated on a particular machine.

Note that given a 64-core machine, the arity-4 PCD Noir implementation allows us to generate a proof even for a string of size 2^{20} in about 19 minutes.

The size of the final output proof should not be larger than a few kB, barring any public inputs.

Note that we only discuss in detail a parallelizable solution that relies on verifying two proofs in on prover, hence building a binary tree-like structure of provers as in fig. 9. We provide numbers for solutions where a PCD prover verifies either two or four proofs. However, even more proofs can be verified within the same prover in the Noir framework and more generally in PCD solutions. In fact, one can build fairly complex trees, even beyond n-ary trees whose nodes consist of provers and a node A points to a node B if B verifies the proof generated by A. We leave the exploration of such designs for future work.

References

[1] https://github.com/risc0/risc0/tree/main/examples/json. Accessed: Aug 31, 2024.

[2] Json parsing inside circuit. https://github.com/o1-labs/o1js/issues/91. Accessed: March 22, 2024.

[3] TLSNotary. https://tlsnotary.org/.

[4] zkjson. https://github.com/chokermaxx/zkjson/tree/b485c3aa03e928958b67bf977eacb749cb1d7185. Accessed: March 22, 2024.

[5] Sebastian Angel, Eleftherios Ioannidis, Elizabeth Margolin, Srinath Setty, and Jess Woods. Reef: Fast succinct non-interactive zero-knowledge regex proofs. Cryptology ePrint Archive, 2023.

[6] arkworks Contributors. ark-bn254 crate documentation. https://docs.rs/ark-bn254/latest/ark_bn254/, 2024. Accessed: 2024-09-04.

[7] Niko Barić and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In International conference on the theory and applications of cryptographic techniques, pages 480–494. Springer, 1997.

[8] Kenneth E Batcher. Sorting networks and their applications. In Proceedings of the April 30–May 2, 1968, spring joint computer conference, pages 307–314, 1968.

[9] Carsten Baum, Samuel Dittmer, Peter Scholl, and Xiao Wang. Sok: vector ole-based zero-knowledge protocols. Designs, Codes and Cryptography, 91(11):3527–3561, 2023.

[10] Wyatt Benno. Minimal space, maximum pace: How memory efficient zero-knowledge proofs work. https://blog.icme.io/minimal-space-maximum-pace-how-memory-efficient-zero-knowledge-proofs-work/, 2024. Accessed: 2024-09-04.

[11] Jean-Luc Beuchat, Jorge E González-Díaz, Shigeo Mitsunari, Eiji Okamoto, Francisco Rodríguez-Henríquez, and Tadanori Teruya. High-speed software implementation of the optimal ate pairing over barreto–naehrig curves. In Pairing-Based Cryptography-Pairing 2010: 4th International Conference, Yamanaka Hot Spring, Japan, December 2010. Proceedings 4, pages 21–39. Springer, 2010.

[12] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Advances in Cryptology—CRYPTO 2002: 22nd Annual International Cryptology Conference Santa Barbara, California, USA, August 18–22, 2002 Proceedings 22, pages 61–76. Springer, 2002.

[13] Matteo Campanelli, Dario Fiore, Semin Han, Jihye Kim, Dimitris Kolonelos, and Hyunok Oh. Succinct zero-knowledge batch proofs for set accumulators. Cryptology ePrint Archive, Paper 2021/1672, 2021. https://eprint.iacr.org/2021/1672.

[14] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In Proceedings of the thiry-fourth annual ACM symposium on Theory of computing, pages 494–503, 2002.

[15] Darko Čapko, Srđan Vukmirović, and Nemanja Nedić. State of the art of zero-knowledge proofs in blockchain. In 2022 30th Telecommunications Forum (TELFOR), pages 1–4. IEEE, 2022.

[16] Dario Catalano and Dario Fiore. Vector commitments and their applications. In Public-Key Cryptography–PKC 2013: 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26–March 1, 2013. Proceedings 16, pages 55–72. Springer, 2013.

[17] David Chaum. Blind signatures for untraceable payments. In Advances in Cryptology: Proceedings of Crypto 82, pages 199–203. Springer, 1983.

[18] Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In ICS, volume 10, pages 310–331, 2010.

[19] Aztec Protocol Contributors. Aztec protocol - packages repository. https://github.com/AztecProtocol/aztec-packages/tree/1ca48a4355370644dceb6680643680f7e8cd5228, 2023. Accessed: 2024-09-04.

[20] Noir Contributors. Noir - a rust-based zk-snarks language. https://github.com/noir-lang/noir/tree/5ef9daa8fb8d55b194d38d540a79dc29f0090351, 2023. Accessed: 2024-09-04.

[21] Noir Contributors. Noir lang - installation guide. https://noir-lang.org/docs/getting_started/installation/, 2024. Accessed: 2024-09-04.

[22] Noir Contributors. Noir lang documentation. https://noir-lang.org/docs, 2024. Accessed: 2024-09-04.

[23] Bryan Cooksey. Chapter 3: Api types and formats. https://zapier.com/resources/guides/apis/data-formats#. Accessed: March 22, 2024.

[24] Dave Crocker, Tony Hansen, and Murray Kucherawy. Rfc 6376: Domainkeys identified mail (dkim) signatures, 2011.

[25] Ivan Damgård. On σ-protocols. Lecture Notes, University of Aarhus, Department for Computer Science, 84, 2002.

[26] Rust Project Developers. Cargo - the rust package manager. https://doc.rust-lang.org/cargo/, 2024. Accessed: 2024-09-04.

[27] Jens Ernstberger, Jan Lauinger, Yinnan Wu, Arthur Gervais, and Sebastian Steinhorst. Origo: Proving provenance of sensitive data with constant communication. Cryptology ePrint Archive, 2024.

[28] Nelly Fazio and Antonio Nicolosi. Cryptographic accumulators: Definitions, constructions and applications. Paper written for course at New York University: www. cs. nyu. edu/nicolosi/papers/accumulators. pdf, 24, 2002.

[29] Free Software Foundation. Gnu time command. https://www.gnu.org/software/time/. Accessed: 2024-09-04.

[30] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, 2019.

[31] Jens Groth. On the size of pairing-based non-interactive arguments. In Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35, pages 305–326. Springer, 2016.

[32] Marios Isaakidis, Harry Halpin, and George Danezis. Unlimitid: Privacy-preserving federated identity management using algebraic macs. In Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society, pages 139–142, 2016.

[33] Jan Lauinger, Jens Ernstberger, Andreas Finkenzeller, and Sebastian Steinhorst. Janus: Fast privacy-preserving data provenance for tls 1.3. Cryptology ePrint Archive, 2023.

[34] Kai Rannenberg, Jan Camenisch, and Ahmad Sabouri. Attribute-based credentials for trust. Identity in the Information Society, Springer, 2015.

[35] Chainlink Labs Research. Deco research series #3: Parsing the response. https://blog.chain.link/deco-parsing-the-response/, 2023. Accessed: March 22, 2024.

[36] Michael Rosenberg, Jacob White, Christina Garman, and Ian Miers. zk-creds: Flexible anonymous credentials from zksnarks and existing identity infrastructure. In 2023 IEEE Symposium on Security and Privacy (SP), pages 790–808. IEEE, 2023.

[37] Ionuț Roșca, Alexandra-Ina Butnaru, and Emil Simion. Security of ethereum layer 2s. Cryptology ePrint Archive, 2023.

[38] Manuel B Santos. Peco: methods to enhance the privacy of deco protocol. Cryptology ePrint Archive, 2022.

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

[40] Weijie Wang, Annie Ulichney, and Charalampos Papamanthou. {BalanceProofs}: Maintainable vector commitments with fast aggregation. In 32nd USENIX Security Symposium (USENIX Security 23), pages 4409–4426, 2023.

[41] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[42] John Watrous. Parse trees, ambiguity, and chomsky normal form. https://cs.uwaterloo.ca/ watrous/ToC-notes/ToC-notes.08.pdf, 2008. Accessed: March 22, 2024.

[43] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pages 2986–3001, 2021.

[44] Samee Zahur and David Evans. Circuit structures for improving efficiency of security and privacy tools. In 2013 IEEE Symposium on Security and Privacy, pages 493–507. IEEE, 2013.

[45] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. DECO: liberating web data using decentralized oracles for TLS. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pages 1919–1938, 2020.

A Proof of Claim 1

In this section, we provide a detailed proof for the correctness of the algorithm provided in Figure 7.

Proof. We focus on the for-loop starting on line 7 of Figure 7 and show that before each ith iteration, the array prefix $\mathsf{Prods}[0..i-1]$ represents a valid "partial" parse tree for $s[0..\text{leaf\_counter}-1]$ according to G, in the sense that

1. the labels of all Prods[j], $0 \leq j < i$, correspond to valid production rules in the grammar G, meaning that (Prods[j].parent.label, Prods[j].left.label) $\in P$ for "terminal" entries where Prods[j].right = None, and (Prods[j].parent.label, Prods[j].left.label, Prods[j].right.label) $\in P$ for the other "non-terminal" entries,

2. all Prods[j].parent, $0 < j < i$, appear exactly once (with the same index and label) as a child Prods[j'].left or Prods[j'].right for some $0 \leq j' < j$, while the root Prods[0].parent never appears as a child,

3. the stack contains all "unused" right children Prods[j].right that didn't yet occur as a parent, appearing on the stack in increasing order of j from bottom to top.

4. the parents Prods[j].parent.index, $0 \leq j < i$ and the left children of terminal entries Prods[j].left appear in depth-first order, with their index being their sequence number of such a traversal,

5. the labels of the left children of terminal entries Prods[j].left.label concatenate to s[0..leaf_counter-1],

We use an induction argument to prove that these invariants hold. They are trivially true before the first iteration $i = 0$, as leaf_counter is set to zero, the stack is empty, and the variable expected is set to the root node with the start symbol S as label and index 0.

We now show that if all invariants held before the i-th iteration, then they also hold before the $i+1$-st iteration. Going over each of the invariants in order, invariant

1. holds because by the induction hypothesis, Prods[j], $0 \leq j < i$, correspond to valid production rules, and so does Prods[i] by the checks in lines 13 and 20.

2. holds by the induction hypothesis for Prods[j].parent, $0 \leq j < i$. For Prods[i].parent, during the $i-1$st iteration, the expected variable either gets set to the left child of Prods[$i-1$] in line 21, which cannot have appeared yet as a parent before, or to the most recent unused right child popped from the stack in line 17. Line 9 in the i-the iteration assures that Prods[i].parent is equal to the expected node, which is therefore a previously unused child node that appeared in an earlier entry Prods[j], $j < i$.

3. holds by the induction hypothesis and the fact that, if Prods[i] is a non-terminal entry, its right node is pushed to the top of the stack on line 22.

4. holds because, by the hypothesis, Prods[j].parent for $0 \leq j < i$ already appear in depth-first order. If in the i-th iteration, Prods[i].parent is terminal, the expected variable is set to the most recent unused right child at the top of the stack (line 17), and its index is verified to be one more than Prods[i].left (line 18), which is indeed the next node and index in a depth-first order. If Prods[i].parent is non-terminal, then expected is set to Prods[i].left (line 21) which was verified to have index one more than Prods[i].parent (line 11), which is also the next node and index in depth-first order. The check on line 9 in the $i+1$-st iteration ensures that Prods[$i+1$].parent matches the expected node.

5. holds by the induction hypothesis and the fact that, if Prods[i] is a terminal node, the next character in s is verified against the label of the left child (line 14) and the leaf_counter is increased by one (line 15).

After $2n-1$ iterations, we therefore have that all entries in Prods correspond to valid production rules, with each parent apart from the root having appeared exactly once as a left or right child of a previous parent. The 3rd invariant together with the check in line 24 mean that there are no remaining unused right children, and invariants 4 and 5 together with the check in line 23 that leaf_counter == $|s|$ mean that the terminal nodes in depth-first order spell out the string s.

Now consider the binary tree $T = (\mathsf{Verts}, \mathsf{root}, \mathsf{Edges})$ with $\mathsf{Verts} = \{0, \dots, 3n-2\}$, $\mathsf{root} = 0$, and edge $(i, j, \texttt{left})$, $(i, j, \texttt{right}) \in \mathsf{Edges}$ if and only if there exists a $0 \leq k < 2n-1$ such that Prods[k].parent.id= i and Prods[k].left.id= j, respectively Prods[k].right.id= j. The tree T is

- binary, because by the 4th invariant, all Prods[k].parent.id are different;

- connected to the root, because by the 2nd invariant, all parents appear exactly once as a child of a previous node, and

- acyclic, because by the 4th invariant the indices of children are always higher than that of the parent.

Also consider the labeling function label that is defined by label(Prods[i].parent.id) = Prods[i].parent.label for $0 \leq i < 2n-1$ and label(Prods[i].left.id) = Prods[i].left.label for all terminal entries $0 \leq i < 2n-1$. Together with the tree T, this forms a valid parse tree conforming to the grammar G of the string s because

- the root is labeled with the start symbol,

- by the 1st invariant, each node $v \in \mathsf{Verts}$ corresponds to a valid production rule of G,

- and by the 5th invariant, the terminal nodes in depth-first order correspond to the string s.

□

B Definitions for cryptographic primitives

Definition 3 (Static Vector Commitment Scheme)**.** *We define a static vector commitment scheme, denoted* VC *as a tuple of the following algorithms:*

- $\mathsf{pp} \leftarrow \mathsf{VC.KeyGen}(1^k, q)$*: Given the security parameter k and the size q of the maximum length vector to be committed and outputs public parameters* pp *for it.*
- $(\mathsf{com}, \mathsf{aux}) \leftarrow \mathsf{VC.Commit}_{\mathsf{pp}}((v_1, ..., v_m))$*: This algorithm takes as input a vector* $(v_1, ..., v_m)$ *of* $m \le q$ *elements, returns a commitment* com *and auxiliary data* aux.
- $\pi \leftarrow \mathsf{VC.ProveCom}_{\mathsf{pp}}(i, v_i, \mathsf{aux}, \mathsf{com})$*: This algorithm takes at input a position i and corresponding value* v_i*, as well as* aux *information and outputs a proof* π.
- $0/1 \leftarrow \mathsf{VC.VerCom}_{\mathsf{pp}}(i, v_i, \mathsf{com}, \pi)$*: Given a value* v_i*, corresponding location i and a commitment* com*, this algorithm verifies the proof* π *and outputs* 0 *or* 1.

An <u>*aggregatable*</u> *static vector commitment has the following additional algorithms.*

- $\pi \leftarrow \mathsf{VC.ProveAgg}_{\mathsf{pp}}(I, (\pi_i, v_i)_{i\in I}, \mathsf{aux}, \mathsf{com})$*: This algorithm takes at input a set of positions I and corresponding values* v_i *with their respective proofs* π_i*, as well as* aux *information and outputs a proof* π.
- $0/1 \leftarrow \mathsf{VC.VerAgg}_{\mathsf{pp}}(I, (v_i)_{i\in I}, \mathsf{com}, \pi)$*: Given a set of locations I and corresponding values* v_i*, a commitment* com*, this algorithm verifies the proof* π *and outputs* 0 *or* 1.

Definition 4. *A static accumulator or static accumulation scheme, denoted* Acc*, as a tuple of the following algorithms*

- $\mathsf{pp} \leftarrow \mathsf{Acc.KeyGen}(1^k, q)$*: Given the security parameter k and the size q of the maximum length set to be committed, this algorithm outputs public parameters* pp *for it.*
- $(\mathsf{com}, \mathsf{aux}) \leftarrow \mathsf{Acc.Commit}_{\mathsf{pp}}(\{e_1, ..., e_m\})$*: This algorithm takes as input a set of (unique) elements* $(e_1, ..., e_m)$ *of* $m \le q$ *elements, returns a commitment* com *and auxiliary data* aux.
- $\pi \leftarrow \mathsf{Acc.ProveMem}_{\mathsf{pp}}(e, \mathsf{aux}, \mathsf{com})$*: This algorithm takes at input a value e, as well as* aux *information and outputs a proof* π.
- $0/1 \leftarrow \mathsf{Acc.VerMem}_{\mathsf{pp}}(e, \mathsf{com}, \pi)$*: Given a value e and a commitment* com*, this algorithm verifies the proof* π *and outputs* 0 *or* 1.

Definition 5. *We say an accumulator scheme is sound if, the following probability is negligible in the security parameter, for any PPT adversary,* $\mathcal{A}$*:*

$$\Pr[\mathsf{pp} \leftarrow \mathsf{Acc.KeyGen}(1^k, q), (\mathsf{com}, \mathsf{aux}) \leftarrow \mathsf{Acc.Commit}_{\mathsf{pp}}(E), (e, \pi) \leftarrow \mathcal{A}(\mathsf{pp}, 1^k, \mathsf{com}, \mathsf{aux}) : \mathsf{Acc.VerMem}_{\mathsf{pp}}(e, \mathsf{com}, \pi) \wedge e \notin E].$$

Note that Def. 5 assumes correctly executed algorithms Acc.KeyGen and Acc.Commit, i.e. pp and (com, aux) are not generated by the adversary in the security definition. We choose this approach, since in our case, we assume a trusted execution of a setup phase for proving correct parsing and the commitments generated therein are trusted.

C Standard Cryptographic Ideal Functionalities

The universal composability (UC) model, first defined by Canetti in [14] is a strong model for proving security of cryptographic protocols. Proving that a protocol is UC-secure with respect to some functionality means that it can be arbitrarily composed with other instances of the same or other protocols without compromising security. Note that UC security is proven <u>with respect to a functionality</u>. The ideal functionality in this model is a description of the intended interface of a protocol – an input-output API, if you will. One of the most important results of [14] is the composition theorem. Informally, the composition theorem states the following: Let $\mathcal{F}$ be an ideal functionality and π be a protocol that is UC secure with respect to $\mathcal{F}$. Let ϕ be a protocol constructed using $\mathcal{F}$ as a subroutine, such that ϕ is UC secure with respect to another functionality $\mathcal{G}$. Then, if the protocol ϕ' is derived by rewriting ϕ, replacing $\mathcal{F}$ with π, ϕ' is also UC secure with respect to $\mathcal{G}$. In other words, we can build protocols modularly, similarly to writing code, only using the APIs (i.e. ideal functionalities) for complex subroutines, without concerning ourselves with the specifics of the API are implemented.

In this section, we will provide ideal functionalities for various standard cryptographic primitives. For now, we restrict ourselves to the ideal functionality $\mathcal{F}_{\mathrm{ZK}}$, parameterized by a relation R, for showing that the relation R is satisfied by the given inputs.

$\mathcal{F}_{\mathrm{ZK}}$

Parties: $\mathcal{P}$, $\mathcal{V}$, adversary $\mathcal{A}$

Parameters: A relation R, security parameter 1^{λ}.

1. On input (“prove”,sid,x,w) from $\mathcal{P}$, if $R(x,w) = 1$ this functionality sends (“proven”,sid,$\mathcal{P}$,x) to $\mathcal{V}$ and $\mathcal{A}$ and halts. Otherwise, send (“unproven”,sid,$\mathcal{P}$,x) to $\mathcal{V}$ and $\mathcal{A}$ and halt.

Figure 15: An ideal functionality, $\mathcal{F}_{\mathrm{ZK}}$, for zero-knowledge proofs, based on [14].

```
json
    ws element

value
    curlBracLeft membersInObj
    sqBracLeft arrayElements
    quoteLeft string
    numberBase numberExponent
    quoteLeft scalerSpecial

scalar
    quoteLeft string
    numberBase numberExponent
    quoteLeft scalerSpecial

scalarSpecial
    scalerSpecialTerm quoteRight

scalerSpecialTerm
    trueLeft trueRight
    falseLeft falseRight
    nullLeft nullRight

falseLeft
    termF termA
falseRight
    termL falseCompRight
falseCompRight
    termS termE

trueLeft
    termT termR
trueRight
    termU termE

nullLeft
    termN termU
nullRight
    termL termL

termT
    t
termR
    r
termU
    u
termE
    e
termF
    f
termA
    a
termL
    l
termS
    s
termN
    n

members
    memberLHS memberRHS
    memberComma members

memberComma
    member comma

membersInObj
    ws curlBracRight
    members curlBracRight

member
    memberLHS memberRHS

memberKey
    ws stringWS

memberRHS
    ws element

memberLHS
    memberKey colon

arrayElements
    ws sqBracRight
    elements sqBracRight

stringWS
    string ws

elements
    value ws
    elementComma elements

elementComma
    element comma

element
    value ws

escapeU
    'u'

escape
    '"'
    '\'
    '/'
    'b'
    'f'
    'n'
    'r'
    't'
    escapeU fourHexes

twoHexes
    hex hex

fourHexes
    twoHexes twoHexes
```

Figure 16: JSON grammar in Chomsky Normal form, modified from the grammar in **??**

```
sqBracLeft
    ’[’

sqBracRight
    ’]’

curlBracLeft
    ’{’

curlBracRight
    ’}’

comma
    ’,’

quoteLeft
    ’"’

quoteRight
    ’"’

backSlash
    ’\’

colon
    ’:’

point
    ’.’

negation
    ’-’
```

```
string
    quoteChars quoteRight

quoteChars
    quoteLeft characters

characters
    ""
    character characters

character
    ’0020’ . ’10FFFF’ - ’"’ - ’\’
    ’\’ escape

ws
    ""
    ’0020’
    ’000A’
    ’000D’
    ’0009’
    ws ws

hex
    ’0’ . ’9’
    ’A’ . ’F’
    ’a’ . ’f’

digits
    ’0’ . ’9’
    digit digits

digit
    ’0’ . ’9’
```

```
onenine
    ’1’ . ’9’

numberBase
    integer fraction

termExpE
    e
    E

numberExponent
    ""
    termExpE exponent

integer
    digit
    onenine digits
    negation digit
    negation onenine digits

fraction
    ""
    point digits

exponent
    sign digits
    sign digits

sign
    ""
    ’+’
    ’-’
```

Figure 17: JSON grammar in Chomsky Normal form, modified from the grammar in **??**. The production rules uses ‘-’ to denote a set minus and ‘.’ to denote a range.