Practical Proofs of Parsing for Context-free Grammars

Harjasleen Malvai
UIUC/IC3

Gregory Neven
Chainlink Labs

Andrew Miller
UIUC/IC3

Siam Hussain
Chainlink Labs

ABSTRACT

In this work-in-progress, we present a series of protocols to efficiently prove statements about strings in context-free grammars (CFGs). Our main protocol for proving proofs of correct parsing for strings in a CFG flexibly accommodates different instantiations of zero-knowledge proof systems as well as accumulation schemes. While improvements in the modular cryptographic primitives can be carried over for improvements in our protocols, even simpler proof systems, which do not support state-of-the-art techniques such as permutation checks [4, 18] can generate proofs of correct parsing of a string of size n by proving the correctness of a circuit of size $O(cn)$, where c is the cost of verifying a set membership proof in the chosen accumulation scheme.

1 INTRODUCTION

In many cryptographic applications, a party may have access to a committed string which is generated according to some structure. The data sent in response to specific API calls is often used in such applications. For instance, Grubbs et al.[19] showed, that the messages sent during a TLS session commit to the data encrypted within that session. DECO [32] took inspiration from this property, to build solutions that use two-party computation and zero-knowledge proofs to construct a way to create privacy-preserving, but authenticated statements about web-sessions, and especially about API responses. These API responses are structured according to some grammar, most commonly, JSON and XML [15].

Privacy-preserving proofs about structured data motivated, for instance, Luo, et al. [22] and Angel, et al. [4], to create schemes for privacy-preserving parsing of regular expressions.

JSON and XML, however, cannot be parsed as regular expressions and are, in fact, context-free grammars (CFGs). So far, work on proving privacy-preserving statements about strings in CFGs has relied either on (1) assumptions[26, 32] about what is known about the string ahead-of-time, or (2) on naively implementing parsing algorithms in a ZKP system. In the former category, consider for example the partial redaction or revelation schemes of [26, 32]. For instance, these schemes might assume that a JSON API response only contains certain fields, resulting in a grammar that is not really a CFG due to such domain-specific assumptions. Various implementation efforts [2, 3] fall in the latter category of naively implementing ZKP for parsing a context-free grammar. The problem with the naive implementation of such parsing algorithms is that all known algorithms for parsing context-free grammars non-deterministically access elements of the grammar representation and perform other non-deterministic operations such as backtracking over parts of the string being parsed to ensure application of the correct rules. One way that a privacy-preserving implementation of such algorithms could work is by implementing a program which is input-oblivious, i.e., executes all possible branches on a particular input. Stated another way, the most naive implementations rely on exploring all possible paths in the non-determinstic execution of a parsing algorithm – which makes them extremely slow. A few alternatives in the RAM model exist, which haven't yet been implemented and tend to be asymptotically efficient but incur concrete overheads.

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.

Key insight. Observe that the applications that we are concerned with only require that a correct parse tree for a a string in a CFG be generated and paths in this tree be used. Meaning, we only need to verify the correctness of a generated parse tree for a committed string s, not actually emulate generating this tree in a zero-knowledge proof system! It is well-known that the size of the parse tree for a string of length n in a context free grammar has $O(n)$ nodes and edges, with only a small constant overhead [30]. This means that if we generate a parse tree for a string s of length n, and use that as a witness to an algorithm that checks this parse tree, the number of statements to be proven in ZKP becomes significantly smaller than any solutions for context-free grammar parsing in ZKP, proposed before.

Summary of our main construction. 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 $\mathcal{G}$. 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.

Our contribution. The most efficient algorithms for proving the correct parsing of context-free grammars (CFGs) involve looking up elements in pre-processed tables for the grammar of size linear in the grammar. Algorithms for more complex CFG parsing also involve backtracking. When compiling these algorithms to generic zero-knowledge proof systems (e.g. circuits [31], arithmetic intermediate representations [6]), all branches of the parser must be executed in the compiled program, leading to a multiplicative overhead that is linear in the maximum backtracking factor. Thus, if the parser executes a loop where each iteration may require one of k, switch cases, the ultimate number of constraints to represent this program is at least $O(kn)$, where n is the size of the input string. Through our observation that computing the parse tree of an entry has higher complexity than simply checking the correctness of a parse tree, we build up to a final algorithm which can be implemented in $O(n+c)$, where c is the cost of verifying a batch of set membership proofs for elements in the grammar or $O(cn)$ if the scheme only supports one membership proof at a time with the cost of verifying $O(c)$.

Comparing with Reef [4]. Angel, et al. [4] do propose an efficient ZKP-system based 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. Since they so closely integrate with a particular ZKP system and program representation, we also argue that our insight is more extensible and, can be used with any black-box proof system and commitment scheme. Besides, significant parts of the process we implement for verifying the validity of a parse tree for a given string can be sliced up into chunks to make them amenable to recursive proof techniques and parallelization, and hence be combined with the techniques of Reef for further practical improvement.

1.1 Applications

In this section, we provide a brief survey of applications that have motivated the privacy-preserving parsing problem.

Digital identity and anonymous credentials. One place where proofs on private, committed strings often show up is digital identity applications. Consider for example, the active field of research that is anonymous credentials [14, 20, 24, 27]. At a high level, an anonymous credential scheme enables interactions between three kinds of parties: issuers, users and relying parties. An issuer I attests to some statements, stmt about a user U, binds them to the user's public key pk. This process enables U to later make statements to the effect of "the issuer I believes stmt about me", based on which, a relying party R is willing to provide U with some service. One example of such a system might be digital ID cards, issued by a university (the issuer, in this case), that allow its students (the users in this case), to swipe the ID to get a discount at a nearby coffee shop (the relying party).

In terms of privacy properties, all anonymous credential schemes aim for unlinkability, i.e., to separate the issuance process of a credential from the process of using the credential at a relying party. Most works also aim for further privacy properties, in particular, that neither the credential issuance, nor its usage should leak anything about the user other than the statement stmt. As will become clearer in the following discussion, while efficient in the abstract, anonymous credential schemes suffer the major drawback of requiring buy-in from various parties, leading to a chicken and egg problem. Such a problem can be solved with protocols that enable privacy-preserving parsing of broad classes of signed strings.

Anonymous credentials. Another line of work, on sanitizable and redactable signatures (e.g. [7, 28]),extends anonymous credentials by building primitives that may support more flexible showing of stmt in the application above. The idea is that stmt may have some structure and actually contain multiple statements $(\mathsf{stmt}_1,...,\mathsf{stmt}_n)$, only a subset of which may need to be shown to a particular relying party. These primitives, while efficient, are special purpose, require specifications which can have limited flexibility and further, require cooperation from a network of issuers who may not have incentive to provide data to users. This type of framing intuits queries made to API responses, which we discuss below.

DECO: bootstrapping identity from web APIs. DECO, a recent work by Zhang, et al. [32] provides a slightly different, decentralized model, wherein, a server holds data about a user and a third-party issuer (or set of issuers) attests to the fact that the user satisfies some statement, per the web-server, based on its TLS certificate and a TLS session. This model separates the issuance capability from the party that holds the data about a user and uses zero-knowledge proofs and commitments to achieve data privacy and integrity.

Most recently, work in this area such as [25, 27] have demonstrated the benefits of the flexibility offered by zero-knowledge proofs (ZKP). In fact, given that many web servers already output structured responses can be helpful here, in extracting a flexible set of statements from web data. Similarly to how responses to web-APIs can be parsed and queried, perhaps we could also create very flexible statements about users based on data coming from web-servers, using ZKPs?

Proofs over web API responses. Even state-of-the-art proof systems have a high overhead for complex proofs. Also, while many frameworks for programming zero-knowledge proofs have emerged, naively encoding functions, especially non-deterministic ones, puts a heavy burden on provers. This is especially a problem if provers are intended to be a wide variety of internet users with potentially constrained bandwidth and compute resources. Consider the fact that a large proportion of web API responses are JSON and XML. For JSON, for example, there exist query languages such as jQuery [1]. However, to extract the outputs of queries from a JSON, it must first be parsed and then its parse tree traversed specific ways using jQuery. The existence of such query languages which require the use of parsed CFG strings and their usefulness for generating anonymous credentials or decentralized identity, together motivate the need for an efficient proof of correct parsing of a CFG string.

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 Grammars, parsing and parse tree notation

Definition 2.1. 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, $P \subset V \times (V \cup \Sigma)^*$ a set of productions or rules and $S \in V$ the start-symbol.

The standard notation for the production rules for CFGs uses '-' to denote a set minus and '..' to denote a range. For a string w, a parser determines if $w \in \mathcal{G}$ by constructing a parse tree for w. The parse tree represents a sequence of production rules which can then be used to extract semantics.

Given a context free grammar $\mathcal{G}$, let $\mathcal{L}(\mathcal{G})$ denote the language generated by $\mathcal{G}$.

We use the notation $x\|y$ to denote the concatenation of two strings x and y. Given a string x, $|x|$ denotes the bit length of x. Given a string s, the notation $s[:l]$ denotes the bits of s upto and including the lth bit. $s[l+1:]$ denotes the bits of s starting at the $l+1$th bit. For example, for the string $s=101011$, $s[:3]=101$, $s[4:]=011$.

Definition 2.2 (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 $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}')$ [30].*

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 [30].*

2.2 Cryptography background

Universal composability and ideal functionalities. The universal composability (UC) model, first defined by Canetti in [12] 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 with respect to a functionality. 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 [12] 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.

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 whose output 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 $\mathcal{F}_{\mathsf{ZK}}$ in fig. 6.

Vector commitments. A vector commitment scheme, first introduced in [13], 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. We formally define the primitive in Def. 2.3. Note that most works focusing on vector commitments (e.g. [29]) define vector commitments which 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.

Definition 2.3 (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.
- $(\mathsf{com},\mathsf{aux}) \leftarrow \mathsf{VC.Commit}_{\mathsf{pp}}((v_1,\ldots,v_m))$: This algorithm takes as input a vector $(v_1,\ldots,v_m)$ 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 aggregatable 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.

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 [10]). 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 [5].

Definition 2.4. 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.
- $(\mathsf{com},\mathsf{aux}) \leftarrow \mathsf{Acc.Commit}_{\mathsf{pp}}(\{e_1,\ldots,e_m\})$: This algorithm takes as input a set of (unique) elements $(e_1,\ldots,e_m)$ of $m \le q$ elements, returns a commitment com and auxiliary data aux.

[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. [16].

- $\pi \leftarrow \mathsf{Acc.ProveMem}_{\mathsf{pp}}(e,\mathsf{aux},\mathsf{com})$: 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 2.5. We say an accumulator scheme is sound if, the following probability is negligible in the security parameter, for any PPT adversary, $\mathcal{A}$:

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

We only require sound accumulation schemes for our construction. 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 [11, 17].

3 PROTOCOL FOR PROVING CORRECT PARSING

In this section, we build up to a protocol for proving correct parsing of a committed string. To make our construction easier to understand, we define a toy context-free grammar in Chomsky normal form and provide a sample parse-tree.

3.1 A toy grammar

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

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

```
S
    AComma S
    AComma A

AComma
    A Comma

A
    BColon C

BColon
    B Colon

B
    b
    BB

C
    c
    CC

Colon
    ':'

Comma
    ','
```

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

We consider the parse tree for the string `bb: c, b: cc`, given in fig. 1. Observe that the tree in fig. 1 has the following properties:

- The leaves, ordered left to right, concatenate to the string `bb: c, b: cc` and are actually terminals in $\mathcal{G}_{\mathrm{toy}}$.
- All non-leaf nodes are non-terminals in the grammar $\mathcal{G}_{\mathrm{toy}}$.
- 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, specficially: `S` $\longrightarrow$ `AComma A`. This applies to all non-leaf nodes.

Also, since $\mathcal{G}_{\mathrm{toy}}$ 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.

Next, 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 the 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 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.

Intuition for correctness conditions. In order for the parse tree to conform to the grammar G, the following conditions must hold:

- The root of the tree must be labelled with a valid start symbol, i.e. an element of the set of the start symbols S.
- Each non-leaf vertex must be labelled with an element of the set V of non-terminal symbols.
- Each leaf vertex must be labelled with an element of the set of terminal symbols Σ.
- The children of each non-leaf node must, together with that node, form a valid production, i.e. be in the set P. To be precise, if a non-leaf node with label ℓ has children with labels ℓ_1, ℓ_2, then it must hold that $(\ell,(\ell_1,\ell_2)) \in P$.

While the set of conditions stated above shows that a tree is valid according to the grammar G, it is not sufficient to show the validity of the tree itself. In particular, for a graph to be a tree, we need the following:

- The root node has no parents, i.e. in-edges.

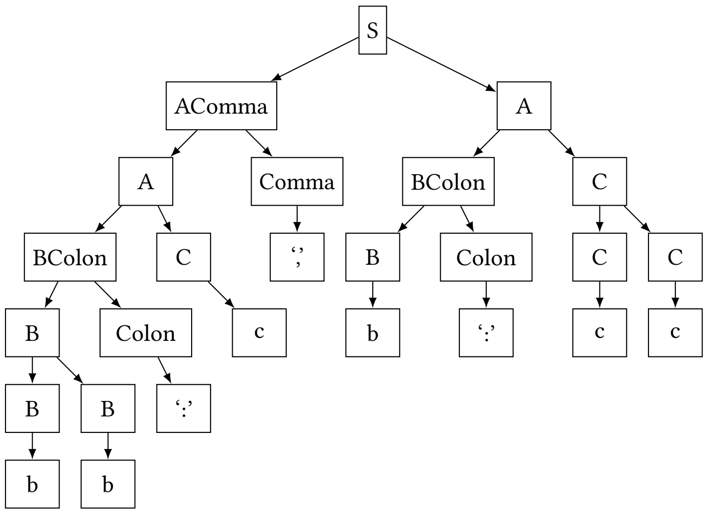

Figure 2: Parse tree for the string `bb: c, b: cc` in $\mathcal{G}_{\text{toy}}$.

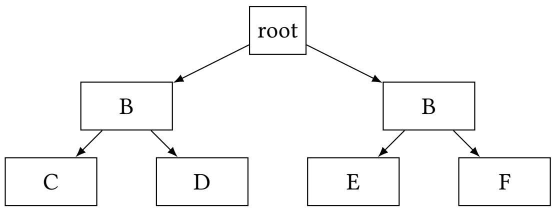

Figure 3: An example of a tree. The vertices of this tree, ordered breadth-first, left-first are labelled (root, B, B, C, D, E, F).

- Every non-root node has one, and only one parent.

Finally, to show that the parse tree in question is indeed the parse tree of a given string s, we need to ensure that:

- The leaf nodes, ordered breadth first from left to right, must have labels that concatenate to form the string s.

In order to formalize these conditions, we must first formalize the description of a tree that we will work with. To this end, we define something we will call a labelled tree representation.

Labelled tree representation. The typical representation of a tree consists of a tuple of vertices and edges (Verts,Edges). We will call (Verts,Edges) the cannonical representation of a tree. The underlying assumption that makes this representation unambiguous is that all vertices have distinct identifiers.

In the case of a parse tree, several vertices might have identical labels, due to multiple occurrences of the same production rule. To understand the ambiguity caused by the loosening of the unique node identifier assumption, consider the tree in fig. 3. There are two nodes with labels B. Suppose we were to simply use the labels of nodes as their identifiers. Then, we could write an unordered set of vertices (root, C, B, D, B, E, F) and an unordered set of edges, ((root, B), (root, B), (B, C), (B, D), (B, E), (B, F)). In this representation, the exact structure of the tree cannot be inferred.

We could try to remedy this with a simple ordering solution, for example, by stating that the vertices are ordered in left-right breadth-first order. However, we still wouldn't be able to unambiguously infer which of the vertices with label B is the parent of any of the nodes C, D, E and F. Thus, we need a way to uniquely identify vertices, separately from their labels.

To this end, we define a modified, but compact tree representation, which we will call a labelled tree representation.

We define a labelled tree representation as a tuple (Labels,Edges), where Labels is the ordered set of node labels, ordered as they appear in a left-right, breadth first traversal of the tree. Edges is an ordered set with elements of the form $(\text{parent,child}) \in [|\text{Labels}|]^2$, where $(i,j) \in \text{Edges}$ means that there exists an edge from the ith node in the breadth-first, left-right traversal to the jth node in that traversal. Also, note that Edges is ordered according to the following relation: $(i,j) < (i',j')$ if ther $i < i'$ or, $i = i'$ and $j < j'$. Note that this is a full ordering on edges in any well-formed tree, since trees contain no cycles and each vertex must have at most one in-edge. The labelled tree representation of the tree in fig. 3 would be as follows: Verts = (root, B, B, C, D, E, F), Labels = $((0,1),(0,2),(1,3),(1,4),(2,5),(2,6))$. This allows us to see that there are two vertices labelled B and there exist edges from root to each of them. We can also infer that the left node with label B has edges to the nodes labelled C and D, and the right node labelled B has edges to E and F.

Correctness of a labelled parse tree Let $\mathcal{G} = (V, \Sigma, S, P)$, be a context-free grammar in Chomsky normal form. Given a string $s \in \mathcal{L}(\mathcal{G})$, let PTree be a parse tree of s, according to $\mathcal{G}$ and we assume that PTree is in labelled tree representation as described above. Therefore, if it is indeed a valid parse tree in labelled representation, it must be the case that PTree := (Labels,Edges) and firstly, satisfies the conditions for being a valid labelled tree of the appropriate size. PTree := (Labels,Edges) being a valid tree amounts to the following statements:

stmt 1 **All edges have validly indexed elements:** For all $(i,j) \in$ Edges, $i,j \in [|\text{Labels}|]$.

stmt 2 **Every non-root node has exactly one parent:** Let the set $C := \{\text{child} : (\cdot,\text{child}) \in \text{Edges}\}$, then checking that every non-root node has exactly parent is equivalent to checking that: (1) $|C| = |\text{Edges}|$ and, (2) $C = [|\text{Labels}|] \setminus \{0\}$. Note that if we consider Edges to be in increasing order of child values, we can also just check that the vector of child values is the ordered set of integers from 1 to $|\text{Labels}|-1$.

stmt 3 **There exist no cycles in this graph i.e. it is a tree:** Since vertices are indexed in breadth-first, left-right order, we know that for any node in our tree representation, its child nodes must have higher indices than the node itself. This means that for all $i \in [|\text{Edges}|]$, $\text{Edges}[i].0 < \text{Edges}[i].1$. Also, by construction each vertex is uniquely indexed, this means that checking this property alone is enough to know that no cycles exist in this graph. Note that since the root node is indexed 0, this check also ensures that **the root node has no parents**.

Note that for now, we don't actually require the ordering property of Edges, it simply comes in handy in efficiently implementing the range checks in stmt 1.

That said, even if $\mathsf{PTree} := (\mathsf{Labels}, \mathsf{Edges})$ is a valid tree, in order to be a valid parse tree for the string s, it must satisfy the following additional conditions relating to the grammar:

stmt 4 **The root node is labelled with a start symbol:** This amounts to checking that $\mathsf{Labels}[0] \in S$.

stmt 5 **Every leaf node is labelled with a terminal:** Let parents be the set $\{i : \exists (i,j) \in \mathsf{Edges}\}$, i.e. the set of non-leaf nodes. Let $\mathsf{parents}^C = [|\mathsf{Labels}|] \setminus \mathsf{parents}$, i.e. $\mathsf{parents}^C$ is the set of indices of nodes which have no children. We require for each $i \in \mathsf{parents}^C$, that $\mathsf{Labels}[i] \in \Sigma$. Note that this and the following condition together ensure that for all $\ell \in \mathsf{Labels}$, $\ell \in V \cup \Sigma$.

stmt 6 **All non-leaf nodes decompose to valid productions:** Let parents be the set of indices for nodes which have children, i.e. for each $i \in$ parents, there exists $(i,j) \in$ Edges for some index j. For all $i \in$ parents, let $\mathsf{children}_i := \{j : (i,j) \in \mathsf{Edges}\}$, i.e. the set of children indices of the node indexed i. Recall that this parse tree is based on a grammar G in Chomsky normal form, so the first property we must check is that (1) for all $i \in$ parents, $|\mathsf{children}_i| = 1$ or 2. Now, (2) if $|\mathsf{children}_i| = 1$, we just check that for $j \in \mathsf{children}_i$, $(\mathsf{Labels}[i], \mathsf{Labels}[j]) \in P$ i.e. a valid production rule, (3) else, we check that $(\mathsf{Labels}[i], \mathsf{Labels}[j], \mathsf{Labels}[k]) \in$ parents such that $j < k$ and $j, k \in \mathsf{children}_i$. Note that this check implicitly ensures that every non-leaf is labelled with a non-terminal.

stmt 7 **This is a parse tree of the string s:** We check that the leaf nodes, in correct order, concatenate to s. More formally, let $T = (i_0, \ldots, i_k)$ where $T = \mathsf{parents}^C$ defined above, and $i_m < i_{m+1}$, for all $m \in [|\mathsf{parents}^C| - 1]$ i.e. T is a sorted version of $\mathsf{parents}^C$. The string $\mathsf{Labels}[i_0] || \ldots | \mathsf{Labels}[i_k]$ equals s.

stmt 8 **The tree has the appropriate size:** Finally, we include an additional check to ensure that this tree is generated correctly. Specifically, we check that if $k = |s|$, then $|\mathsf{Labels}| = 3k - 1$ and that $|\mathsf{Edges}| = 3k - 2$, since we assume $\mathcal{G}$ is in Chomsky normal form.

3.3 Protocol for proving correctness of a parse tree

In the previous section, we laid out the requirements to prove that a parse tree for a string s is correctly generated. Here, we build a protocol for proving these statements about a parse tree in a way that hides the actual tree structure. For now, we assume that both the prover and verifier have the string s. In an updated version of the paper, we will provide a simple modification of the protocol to account for the scenario where the verifier has only a commitment com_s to the string s, while the prover holds the string s itself, the commitment com_s and some auxiliary information aux.

3.3.1 Strawman protocol. Our protocol is a two party protocol with parties prover $\mathcal{P}$ and $\mathcal{V}$. In this strawman, we assume both $\mathcal{P}$ and $\mathcal{V}$ know the grammar $\mathcal{G} = (V, \Sigma, S, P)$ and the string $s \in \mathcal{L}(\mathcal{G})$. We assume $\mathcal{P}$ holds a parse tree $(\mathsf{Labels}, \mathsf{Edges})$, which it claims is the parse tree for s according to $\mathcal{G}$. The intuition behind this simple construction is that $\mathcal{P}$ uses a zero knowledge proof system to prove the requirements stated in stmts 1-9 above. At the end of an honestly run protocol, $\mathcal{V}$ can be convinced of the validity of the parse tree $(\mathsf{Labels}, \mathsf{Edges})$ for s, without having to compute it on its own.

We define the relation $R_1(\mathsf{publicInp}, \mathsf{privateInp})$ such that it parses publicInp as the tuple $(\mathcal{G}, s)$ and privateInp as $(\mathsf{Labels}, \mathsf{Edges})$. We say $R_1((\mathcal{G}, s), (\mathsf{Labels}, \mathsf{Edges})) = 1$ iff all the statements stmt 1 through 8 above, are true for these inputs. Let $\mathcal{F}^1_{\mathsf{ZK}}$ be the ideal functionality $\mathcal{F}_{\mathsf{ZK}}$ in fig. 6 parametrized by the relation R_1.

Then, we devise a strawman protocol with the following phases:

Input Phase:. Both $\mathcal{P}$ and $\mathcal{V}$ receive the grammar $\mathcal{G}$ and the string s as inputs. They also receive a session identifier sid.

Parse and Proof Phase:. $\mathcal{P}$ computes $(\mathsf{Labels}, \mathsf{Edges})$, the parse tree for s and sends the message ("prove", sid, $(\mathcal{G}, s)$, $(\mathsf{Labels}, \mathsf{Edges})$) to $\mathcal{F}^1_{\mathsf{ZK}}$.

Verification:. Upon receipt of a message of the form ("proven", sid, $\mathcal{P}$, $(\mathcal{G}^*, s^*)$) from $\mathcal{F}^1_{\mathsf{ZK}}$, $\mathcal{V}$ checks that $\mathcal{G}^* = \mathcal{G}$ and $s^* = s$, if so, it outputs 1, else it outputs 0. Otherwise, if it receives a message of the form ("unproven", sid, $\mathcal{P}$, $\cdot$) from $\mathcal{F}^1_{\mathsf{ZK}}$, it also outputs 0.

3.3.2 Detailed Protocol 1. In the strawman above, we have abstracted out the details of how the relation R_1 is described. Here, we discuss the implementation of a function NaiveParseTreeChecker, which checks if a set of inputs satisfies the relation R_1. The pseudocode for NaiveParseTreeChecker is given in fig. 4, excluding boxed text.

NaiveParseTreeChecker assumes that the sets P, Σ, V, S and the string s are represented as arrays. In such a case, checking for membership in a set, using function CheckMemNaive, requires traversing the entire array representation of the set. Hence, each set membership check is linear in the size of the set, when implemented in a circuit geared towards a cryptographic proof system.

Overall, using CheckMemNaive as a subroutine, when implemented as an input-oblivious circuit, the following factors contribute to the circuit-size complexity of NaiveParseTreeChecker:

- The check for stmt 4 is linear in $|\mathsf{Labels}|$ and $|\mathsf{Edges}|$, which are, in turn, if computed honestly, of sizes $O(|s|)$.
- Checking stmt 5 similarly costs $O(|S|)$, where S is the set of start symbols.
- For now, we assume that each operation in the `for` loop over the size of Edges (on line 17 in fig. 4) is constant time and thus the loop only adds $O(|s|)$ overhead.
- Several checks require work linear in the size of s to count its length (stmt 4), or check equality with s (stmt 8). This adds cost linear in $O(|s|)$.
- Since each iteration of the `for` loop over the set of labels (see line 33 in fig. 4) in the tree must obliviously implement an `if` statement, whose branches make calls to CheckMemNaive. Each such iteration adds cost $O(|\Sigma| + |P|)$, since CheckMemNaive requires a circuit linear in the size of the set. The number of iterations, which is the size of the set Labels, is linear in the length of s, so, in total, the `for` loop over the length of Labels amounts to an overhead of $O(|s|(|\Sigma| + |P|))$.

In total, the complexity of implementing NaiveParseTreeChecker is $O(|s|(|\Sigma| + |P|) + |S|)$. Next, we consider how to reduce this cost.

3.3.3 Detailed Protocol 2. Now that we have considered the costs of implementing R_1 using CheckMemNaive as a subroutine: a blowup factor of $(|\Sigma|+|P|)$ as well as an additional constant factor $|S|$, where S is the set of start symbols in the grammar.

Besides, the description of G may be long and the verifier has to read the public input. Therefore, we may not want the verifier to have to read G for every verification it conducts, given that it might wish to verify proofs of correct parsing of different strings in G. Further, naively showing membership in a set Z, using a zero-knowledge proof system requires a number of constraints asymptotically at least $\Omega(polylog|Z|)$ (using something called oblivious RAM, see [21]) and concretely often $O(|Z|)$ for smaller sets. At a high level, we now optimize both the proving and verification of the various set membership relationships, we recommend a pre-processing step committing to the terms in $G=(V,\Sigma,S,P)$ using a set accumulator. Then, the prover can provide membership proofs for each required set as (public or private) inputs and run the verifying function as part of the zero-knowledge proof relation.

More concretely, we slightly modify the original protocol, to introduce a global setup phase, described below. We also replace the functionality $\mathcal{F}^1_{\mathrm{ZK}}$ with another functionality $\mathcal{F}^2_{\mathrm{ZK}}$. Let $\mathcal{F}^2_{\mathrm{ZK}}$ be the ideal functionality $\mathcal{F}_{\mathrm{ZK}}$ in fig. 6 parameterized by the relation R_2, detailed in fig. 4, excluding the shaded text, and which we will explain in more detail shortly. At a high level, the relation R_2 is similar to R_1, except, instead of naively checking for set membership, it takes as input the proofs of membership of the requisite entries in the sets V,Σ,S,P of the grammar.

Setup Phase:. Both $\mathcal{P}$ and $\mathcal{V}$ receive as input a set of public parameters $\mathsf{pp}\leftarrow\mathsf{Acc.KeyGen}(1^k,\max(|\mathcal{G}|))$ across all possible grammars which the protocol may use for the given parameter k.

Then, the input phase gets modified as follows:

Input Phase:. Both $\mathcal{P}$ and $\mathcal{V}$ receive the grammar $\mathcal{G}$ and the string s as inputs. They also receive a session identifier sid. They each compute the tuple $((\mathsf{com}_V,\ \mathsf{aux}_V),(\mathsf{com}_\Sigma,\ \mathsf{aux}_\Sigma),(\mathsf{com}_S,,\ \mathsf{aux}_S),(\mathsf{com}_P,\ \mathsf{aux}_P))\leftarrow(\mathsf{Acc.Commit}_{\mathsf{pp}}(V),\ \mathsf{Acc.Commit}_{\mathsf{pp}}(\Sigma),\ \mathsf{Acc.Commit}_{\mathsf{pp}}(S),\ \mathsf{Acc.Commit}_{\mathsf{pp}}(P))$. Let $\mathsf{com}_{\mathcal{G}}$ denote the tuple $(\mathsf{com}_V,\mathsf{com}_\Sigma,\mathsf{com}_S,\mathsf{com}_P)$ and each party stores $\mathsf{com}_{\mathcal{G}}$. $\mathcal{P}$ additionally stores $(\mathsf{aux}_V,\mathsf{aux}_\Sigma,\mathsf{aux}_S,\mathsf{aux}_P)$. Observe that the parties can drop the commitments and associated information for the set of non-terminals V, since their validity is implicitly checked in the process of checking production rules.

Parse and Proof Phase:. $\mathcal{P}$ computes $(\mathsf{Labels},\mathsf{Edges})$, the parse tree for s. $\mathcal{P}$ then runs the following:

- Initialize arrays Π_Σ,Π_P.
- Compute $\Pi_S=\mathsf{Acc.ProveMem}(\mathsf{Labels}[0],\mathsf{aux}_S)$, i.e. the proof of membership of the root node's label in the set S of valid start symbols.
- For each node index i if there exist children indices $(j_1,j_2,\ldots)$, i.e. if there exist (i,j_1), $(i,j_2),\ldots\in\mathsf{Edges}$, then, append the proof of correct production, to Π_P. That is, let $t=(\mathsf{Labels}[i],(\mathsf{Labels}[j_1],\mathsf{Labels}[j_2],\ldots))$, and append the proof Acc. ProveMem (t,aux_P) to Π_P. Else, append $\perp$ to Π_P. Note that this implicitly shows that the node indexed i is labelled by a non-terminal.
- For each node index $i\in[|\mathsf{Labels}|]$, if there exists no entry of the form $(i,\cdot)\in\mathsf{Edges}$, append the proof $\mathsf{Acc.ProveMem}(\mathsf{Labels}[i],\mathsf{aux}_\Sigma)$ to Π_Σ. Else, append $\perp$ to Π_Σ.
- Output (Π_Σ,Π_S,Π_P).

Given the notation we have just introduced, we define the relation $R_2(\mathsf{publicInp},\mathsf{privateInp})$ such that it parses publicInp as the tuple $(\mathsf{com}_{\mathcal{G}},s)$ and privateInp as $(\mathsf{Labels},\mathsf{Edges},\Pi_\Sigma,\Pi_S,\Pi_P)$. We say $R_2((\mathsf{com}_{\mathcal{G}},s),(\mathsf{Labels},\mathsf{Edges},\Pi_\Sigma,\Pi_S,\Pi_P))=1$ iff all the statements stmt 1 through 8 above, are true for the inputs where the grammar $\mathcal{G}$ is committed in $\mathsf{com}_{\mathcal{G}}$ and Π_Σ,Π_S and Π_P attest to the membership of the respective elements of the purported parse tree in elements of the grammar.

At the end of this phase, the prover sends the message ("prove",sid, $(\mathsf{com}_{\mathcal{G}},s)$,$(\mathsf{Labels},\mathsf{Edges},\Pi_\Sigma,\Pi_S,\Pi_P)$) to $\mathcal{F}^2_{\mathrm{ZK}}$.

Verification:. Upon receipt of a message of the form ("proven",sid, $\mathcal{P}$,$(\mathsf{com}^*_{\mathcal{G}},s^*)$), from $\mathcal{F}^2_{\mathrm{ZK}}$, $\mathcal{V}$ checks that $\mathsf{com}^*_{\mathcal{G}}=\mathsf{com}_{\mathcal{G}}$ and $s^*=s$, if so, it outputs 1, else it outputs 0. Otherwise, if it receives a message of the form ("unproven",sid,$\mathcal{P}$,$\cdot$) from $\mathcal{F}^2_{\mathrm{ZK}}$, it outputs 0.

The complexity of this protocol is similar to the previous one, except, at each step which previously required circuits linear in the sizes of Σ, S or P, the required circuit size is now the circuit to run the function Acc.VerMem. Let us denote the circuit complexity of Acc.VerMem as c_{Acc}. Then, the total circuit complexity of implementing R_2 is $O(|s|\times c_{\mathsf{Acc}})$.

Depending on the instantiation, the verification cost of Acc might vary from $O(\log n)$ (e.g., in the case of tree-based structures) to as low as $O(1)$ (e.g. [8]), thus adding only the corresponding circuit complexity c_{Acc}, above.

3.4 Optimizations and extensions

Proving correct parsing on committed strings. The protocols and optimizations considered so far assume that both the prover and verifier receive a string s as an input. An example where such an assumption may be useful in applications such as generating reproducible builds for open-source code [23].

However, if we have a very small client, even in the case of verifying reproducible builds, we may not want this client to read the entire source code (or string) being parsed. In other cases, such as those motivated by privacy-preserving disclosure of credentials [9], or privacy preserving oracles [32], the verifier may only receive a commitment to the string.

In such a case, we modify R_2, described above, to obtain the relation R, to include as public input $(\mathsf{com}_{\mathcal{G}},\mathsf{com}_s)$, and an additional parameter Π_s in the private inputs, where com_s is the commitment to the string s and Π_s is a proof of correct opening of com_s. We then augment the private input, to include s and any extra data aux to prove the correctness of s as the opening for com_s.

Batching accumulator proofs. The circuit we need to implement for the relation used to check correct parsing must, at minimum read the entire input string. However, certain instantiations of the accumulator definition from Def. 2.4 may introduce two more algorithms to make the computation of the entire relation faster: Acc.ProveBatch and Acc.VerifyBatch to respectively prove and verify the membership of a batch of elements in a committed set. For instance, consider the c_{Acc} factor we introduced in calculating the circuit complexity of R_2 as in fig. 4. We may be able to replace the calls to Acc.VerMem for the same commitment, with a single Acc.VerifyBatch call, making the circuit complexity essentially

```
function NaiveParseTreeChecker(($V$,$\Sigma$,$S$,$P$),$s$,Labels,Edges):
function $\boxed{\text{ParseTreeChecker}((\text{com}_{\mathcal{G}},s),(\text{Labels},\text{Edges},\Pi_\Sigma,\Pi_S,\Pi_P))}$:
    # Assumes that the elements of Edges are arranged in
    # increasing order of child nodes, i.e. for $(\cdot,j),(\cdot,k) \in$ Edges,
    # $(\cdot,j)$ appears before $(\cdot,k)$ iff, $j \leq k$.
    output = 1
    $\boxed{(\cdot,\text{com}_\Sigma,\text{com}_S,\text{com}_P) := \text{com}_{\mathcal{G}}}$
    # Check stmt 4: that the inputs are the right length
    output = output $\wedge$ $(3\times|s|-1==|\text{Labels}|)$
            $\wedge$ $(3\times|s|-2==|\text{Edges}|)$
    # Check stmt 5: the root is labelled with a start symbol.
    output = output $\wedge$ CheckMemNaive(Labels[0],$S$)
    $\boxed{\text{Acc.VerMem}(\text{Labels}[0],\text{com}_S,\Pi_S)}$
    # Initialize a dict. mapping ints to arrays of ints, used to
    # check for production rules + identify leaf/non-leaf vertices
    prodDict = EmptyDictionary()
    for i=0...|Edges| - 1:
        (parentIdx, childIdx) := Edges[i]
        # Add this to the set of children for this parent.
        prodDict[parentIdx] = prodDict[parentIdx]
                                        .append(childIdx)
        # This checks the ranges of the indices in edges i.e. stmt 1.
        output = output $\wedge$ $(0 \leq \text{parentIdx} < |\text{Labels}|)$
                $\wedge$ $(0 \leq \text{childIdx} < |\text{Labels}|)$
        # This checks that every non-root node occurs
        # only once as a child, i.e. stmt 2.
        output = output $\wedge$ $(\text{childIdx} == i+1)$
        # Checks stmt 3: vertices indexed breadth-first + no cycles.
        output = output $\wedge$ $(\text{parentIdx} < \text{childIdx})$
    end for
    # Let's initialize a variable to keep count of leaves
    leaves = 0
    for i in 0...|Labels| - 1:
        if prodDict[i] = []: # Leaf case
            # Check stmt 6
            output = output $\wedge$ CheckMemNaive(Labels[i],$\Sigma$)
            $\boxed{\text{Acc.VerMem}(\text{Labels}[i],\text{com}_\Sigma,\Pi_\Sigma[i])}$
            # Partial check for stmt 8
            output = output $\wedge$(Labels[i] == s[leaves])
            leaves = leaves + 1
        else: # Non-leaf case -- check stmt 7
            output = output $\wedge$
            CheckMemNaive((Labels[i],prodDict[i]),$P$)
            $\boxed{\text{Acc.VerMem}((\text{Labels}[i],\text{prodDict}[i]),\text{com}_P,\Pi_P[i])}$
    end for
    # Check (part of) stmt 8:
    output = output $\wedge$ (leaves == |s|)
    return output
```

Figure 4: Pseudocode for simple functions to compute R_1 and R_2.

$O(|s|+c_{\text{batchAccum}})$, where $c_{\text{batchAccum}}$ is the complexity of verifying a batched proof of membership for $|s|$ elements in sets whose size is dictated by the grammar.

Privacy. If the prover were to share the parse tree (Labels,Edges) with the verifier, since all parties know the grammar G, the verifier could verify the validity of the parse tree in the clear. However, this would leak the leaves of the parse tree also, which are equivalent to a string s the prover may want to hide from the verifier in our motivating applications. Further, often the structure of the parse tree itself may leak sensitive information which may be inferred by the verifier about s, even if the set of leaf labels within Labels which correspond to terminals, i.e. s are somehow hidden. Our construction so far hides the entire tuple (Labels,Edges), while leaking the lengths of the vectors Labels and Edges, which, in turn, leak the length of s. Later, we will motivate and discuss a construction where the length of s is also obscured.

3.5 Security intuition

We define the security of privacy-preserving parsing protocols in the UC model. In particular, we say that a secure privacy-preserving parsing protocol emulates the simple ideal functionality $\mathcal{F}_{\text{Parse}}$ assuming authenticated communication channels between parties. For now, we assume that both $\mathcal{P}$ and $\mathcal{V}$ receive $\mathcal{G}$ and s as inputs from the environment. Informally, we argue, that detailed protocol 2, above emulates $\mathcal{F}_{\text{Parse}}$, since an ideal world simulator can work as follows in the various cases:

- If neither $\mathcal{P}$, nor $\mathcal{V}$ is corrupted, the simulator does nothing.
- If only $\mathcal{P}$ is corrupted, the simulator runs $\mathcal{A}$ internally, and if $\mathcal{A}$ does not output valid values satisfying R_2, it inputs a string s', a slightly modified version of s, which will not parse according to $\mathcal{G}$, as the prover's input to $\mathcal{F}_{\text{Parse}}$, leading the honest ideal-world $\mathcal{V}$ to output 0, same as the honest real-world $\mathcal{V}$. In the case that $\mathcal{V}$ outputs 1 in the real-world, $\mathcal{A}$ must have supplied satisfying public inputs $(\text{com}_{\mathcal{G}},s)$ and a parse tree, as well as corresponding membership proofs to $\mathcal{F}_{\text{ZK}}$, showing that $s \in \mathcal{G}$. If $s \notin$ grammar, either our conditions for parse-tree checking are not complete or $\mathcal{A}$ can be used to create an adversary $\mathcal{B}$, which breaks the soundness of Acc from Def. 2.5.
- If only $\mathcal{V}$ is corrupt, in the ideal world, the simulator retrieves the input $\mathcal{G}$,s from "parsed" sent by $\mathcal{F}_{\text{Parse}}$. It internally runs copies of the real world $\mathcal{P}$ and $\mathcal{F}_{\text{ZK}}$, with $\mathcal{P}$ input $\mathcal{G}$,s and provides the requisite messages to the adversarial verifier algorithm and outputs whatever it outputs.
- If both parties are corrupted, simulator simply runs both of them internally, with their respective inputs and $\mathcal{F}_{\text{ZK}}$.

4 CONCLUSION AND FUTURE WORK

We have presented an efficient, modular protocol for proving that a given string has been parsed according to a specific grammar.

We intend to extend this work and update it shortly with the following. Firstly, we plan to provide a more detailed security analysis and a more detailed protocol with privacy, for example, if the input string is only committed. Secondly, we will we will consider protocols for redacting portions of a committed string s and give

$\mathcal{F}_{\textbf{Parse}}$

Parties $\mathcal{P}$, $\mathcal{V}$.

Parameters: A context-free grammar $\mathcal{G}$.

(1) On input (sid,"parse",s) from a party $\mathcal{P}$, check if $s \in \mathcal{G}$, and if so, send the message ("parsed",sid,$\mathcal{G}$,s,1) to $\mathcal{V}$ and $\mathcal{A}$, otherwise send ("parsed",sid,$\mathcal{G}$,s,0) to $\mathcal{V}$ and $\mathcal{A}$.

Figure 5: An ideal functionality, $\mathcal{F}_{\textbf{Parse}}$, to parse a string in a given grammar.

security definitions in this scenario. Further, we will discuss protocols for proving statements about the entirety of a string s, while only parsing parts of it inside a proof system, to further optimize the process for applications where parts of an API response may already be public.

Finally, we intend to provide an evaluation of our techniques and compare them concretely with works such as Reef and other simple, related techniques.

REFERENCES

[1] [n.d.]. jQuery API. https://api.jquery.com. Accessed: March 22, 2024.
[2] [n.d.]. JSON parsing inside circuit. https://github.com/o1-labs/o1js/issues/91. Accessed: March 22, 2024.
[3] [n.d.]. zkJSON. https://github.com/chokermaxx/zkjson/tree/b485c3aa03e928958b67bf977eacb749cb1d7185. Accessed: March 22, 2024.
[4] Sebastian Angel, Eleftherios Ioannidis, Elizabeth Margolin, Srinath Setty, and Jess Woods. 2023. Reef: Fast Succinct Non-Interactive Zero-Knowledge Regex Proofs. Cryptology ePrint Archive (2023).
[5] Niko Barić and Birgit Pfitzmann. 1997. Collision-free accumulators and fail-stop signature schemes without trees. In International conference on the theory and applications of cryptographic techniques. Springer, 480–494.
[6] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2019. Scalable zero knowledge with no trusted setup. In Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39. Springer, 701–732.
[7] Arne Bilzhause, Henrich C Pöhls, and Kai Samelin. 2017. Position paper: The past, present, and future of sanitizable and redactable signatures. In Proceedings of the 12th International Conference on Availability, Reliability and Security. 1–9.
[8] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. 2009. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In Public Key Cryptography–PKC 2009: 12th International Conference on Practice and Theory in Public Key Cryptography, Irvine, CA, USA, March 18-20, 2009. Proceedings 12. Springer, 481–500.
[9] Jan Camenisch, Stephan Krenn, Anja Lehmann, Gert Læssøe Mikkelsen, Gregory Neven, and Michael Østergaard Pedersen. 2015. Formal treatment of privacy-enhancing credential systems. In International Conference on Selected Areas in Cryptography. Springer, 3–24.
[10] Jan Camenisch and Anna Lysyanskaya. 2002. 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. Springer, 61–76.
[11] Matteo Campanelli, Dario Fiore, Semin Han, Jihye Kim, Dimitris Kolonelos, and Hyunok Oh. 2021. Succinct Zero-Knowledge Batch Proofs for Set Accumulators. Cryptology ePrint Archive, Paper 2021/1672. https://eprint.iacr.org/2021/1672 https://eprint.iacr.org/2021/1672.
[12] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. 2002. Universally composable two-party and multi-party secure computation. In Proceedings of the thiry-fourth annual ACM symposium on Theory of computing. 494–503.
[13] Dario Catalano and Dario Fiore. 2013. 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. Springer, 55–72.
[14] David Chaum. 1983. Blind signatures for untraceable payments. In Advances in Cryptology: Proceedings of Crypto 82. Springer, 199–203.
[15] Bryan Cooksey. [n.d.]. Chapter 3: API types and formats. https://zapier.com/resources/guides/apis/data-formats#. Accessed: March 22, 2024.
[16] Ivan Damgård. 2002. On Σ-protocols. Lecture Notes, University of Aarhus, Department for Computer Science 84 (2002).
[17] Nelly Fazio and Antonio Nicolosi. 2002. Cryptographic accumulators: Definitions, constructions and applications. Paper written for course at New York University: www. cs. nyu. edu/nicolosi/papers/accumulators. pdf 24 (2002).
[18] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. 2019. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive (2019).
[19] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. 2017. Message Franking via Committing Authenticated Encryption. In CRYPTO.
[20] Marios Isaakidis, Harry Halpin, and George Danezis. 2016. UnlimitID: Privacy-preserving federated identity management using algebraic MACs. In Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society. 139–142.
[21] Kasper Green Larsen and Jesper Buus Nielsen. 2018. Yes, there is an oblivious RAM lower bound!. In Annual International Cryptology Conference. Springer, 523–542.
[22] Ning Luo, Chenkai Weng, Jaspal Singh, Gefei Tan, Ruzica Piskac, and Mariana Raykova. 2023. Privacy-Preserving Regular Expression Matching using Nondeterministic Finite Automata. Cryptology ePrint Archive (2023).
[23] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. 2017. {CHAINIAC}: Proactive {Software-Update} transparency via collectively signed skipchains and verified builds. In 26th USENIX Security Symposium (USENIX Security 17). 1271–1287.
[24] Kai Rannenberg, Jan Camenisch, and Ahmad Sabouri. 2015. Attribute-based credentials for trust. Identity in the Information Society, Springer (2015).
[25] Deevashwer Rathee, Guru Vamsi Policharla, Tiancheng Xie, Ryan Cottone, and Dawn Song. 2022. Zebra: Anonymous credentials with practical on-chain verification and applications to kyc in defi. Cryptology ePrint Archive (2022).
[26] Chainlink Labs Research. 2023. DECO Research Series #3: Parsing the Response. https://blog.chain.link/deco-parsing-the-response/. Accessed: March 22, 2024.
[27] Michael Rosenberg, Jacob White, Christina Garman, and Ian Miers. 2023. zk-creds: Flexible anonymous credentials from zksnarks and existing identity infrastructure. In 2023 IEEE Symposium on Security and Privacy (SP). IEEE, 790–808.
[28] Kai Samelin and Daniel Slamanig. 2020. Policy-based sanitizable signatures. In Topics in Cryptology–CT-RSA 2020: The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24–28, 2020, Proceedings. Springer, 538–563.
[29] Weijie Wang, Annie Ulichney, and Charalampos Papamanthou. 2023. {BalanceProofs}: Maintainable Vector Commitments with Fast Aggregation. In 32nd USENIX Security Symposium (USENIX Security 23). 4409–4426.
[30] John Watrous. 2008. Parse trees, ambiguity, and Chomsky normal form. https://cs.uwaterloo.ca/ watrous/ToC-notes/ToC-notes.08.pdf. Accessed: March 22, 2024.
[31] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. 2021. Wolverine: fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In 2021 IEEE Symposium on Security and Privacy (SP). IEEE, 1074–1091.
[32] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. 2020. Deco: Liberating web data using decentralized oracles for tls. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. 1919–1938.

A STANDARD CRYPTOGRAPHIC IDEAL FUNCTIONALITIES

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

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

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 6: An ideal functionality, $\mathcal{F}_{\mathbf{ZK}}$, for zero-knowledge proofs, based on [12].