Formatted Encryption Beyond Regular Languages

Daniel Luchaup
University of Wisconsin
Madison, WI, USA
luchaup@cs.wisc.edu

Thomas Shrimpton
Portland State University
Portland, OR, USA
teshrim@cs.pdx.edu

Thomas Ristenpart
University of Wisconsin
Madison, WI, USA
rist@cs.wisc.edu

Somesh Jha
University of Wisconsin
Madison, WI, USA
jha@cs.wisc.edu

ABSTRACT

Format-preserving and format-transforming encryption (FPE and FTE, respectively) are relatively new cryptographic primitives, yet are already being used in a broad range of real-world applications. The most flexible existing FPE and FTE implementations use regular expressions to specify plaintext and/or ciphertext formats. These constructions rely on the ability to efficiently map strings accepted by a regular expression to integers and back, called ranking and unranking, respectively.

In this paper, we provide new algorithms that allow encryption with formats specified by context-free grammars (CFGs). Our work allows for CFGs as they appear in practice, partly a pure grammar for describing syntax, and partly a set of lexical rules for handling tokens. We describe a new relaxed ranking method, structural ranking, that naturally accommodates practical CFGs, thereby empowering new FPE and FTE designs. We provide a new code library for implementing structural ranking, and a tool that turns a simple YACC/LEX-style grammar specification into ranking code. Our experimental analysis of the code shows that the new CFG ranking algorithm is efficient in interesting settings, even when the grammars are ambiguous. For example, we show that one can efficiently rank C programs of size thousands of kilobytes in milliseconds.

Keywords

Format-preserving encryption; format-transforming encryption; ranking

1. INTRODUCTION

Format-preserving encryption (FPE) [2–4] and Format-transforming encryption (FTE) [7], have recently emerged as practically important versions of *formatted encryption*. Loosely speaking, formatted encryption is like conventional encryption in terms of the security goals, but with the additional functionality of transforming plaintexts conforming to one format — say, described by a regular expression (regex) — into ciphertexts conforming to another.

When FPE is used, the plaintext and ciphertext formats are the same. This admits, for example, in-place encryption of database entries that must abide by some strict formatting requirements, such as being a valid credit-card number, social security number, or postal address. FPE is already widely used in the payment industry.

FTE, on the other hand, encrypts a plaintext of a given format to ciphertext of a different format. FTE is a generalization of FPE. Dyer et al. [7] recently introduced FTE and showed its efficacy as a tool for circumventing network monitors that use deep-packet inspection to block protocols such as Tor [6].

Ranking schemes and formatted encryption. At the core of all known instantiations of FPE and FTE schemes are efficient algorithms for *ranking* languages, i.e., for mapping language elements to integers in an invertible fashion. Goldberg and Sipser (GS) [8] introduced the notion of ranking as a technique for optimal language compression. In particular, the GS rank of a word w in a language L is the position of w in the lexicographical ordering of L, given as a non-negative integer. Thus, ranking defines a bijective function (rank), and *unranking* defines the inverse function (unrank) that maps $n \in \{0, 1, \ldots, |L| - 1\}$ to the word $w \in L$ whose rank is n. A pair of ranking and unranking algorithms for a language L constitute a *ranking scheme.*

GS [8] gave a polynomial-time ranking scheme for languages represented by unambiguous context-free grammars (CFGs). They also suggest a method for ranking a regular language, given a DFA for it, but it was Bellare et al. [2] who provided the first detailed algorithm for such a method. (Since they credit the method to GS, we will continue to call it GS ranking.)

Moreover, Bellare et al. [2] show how to instantiate efficient FPE when the format is the regular language L specified by a DFA. Given a blockcipher E that operates over the integers $\{0, 1, \ldots |L|-1\}$ (an *integer cipher*), the FPE ciphertext of plaintext $x \in L$ is defined by $y = \mathsf{unrank}(E_K(\mathsf{rank}(x)))$. For obvious reasons, this is known as *rank-encipher-unrank* FPE.

This scheme is deterministic, as no per-plaintext randomness or internal state is used. On the other hand, the FTE scheme of Dyer et al. [7] uses conventional (randomized or stateful) authenticated encryption (AE) as its underlying encryption primitive in their scheme. To encrypt a plaintext $x \in L_1$, one represents x as a bit string, encrypts it using the AE scheme, treats the resulting AE ciphertext as the base-2 representation of an integer, and unranks this to the appropriate string $y \in L_2$. The Dyer et al. scheme supports ciphertext formats specified by regular expressions (regexes), by way of first converting the regex to an NFA and then to a (minimal) DFA using standard algorithms. The latter NFA-to-DFA step can result in an exponential increase in the number of states of the resulting DFA automaton.

Luchaup et al. [14] provide a unified framework for formatted encryption that captures all of the FPE and FTE schemes so far described, and they go on to give regex-specified FTE and FPE schemes that avoid NFA-to-DFA conversion. To do so, they introduced the notion of *relaxed ranking*, which maps strings in a language L to integers in $\mathbb{Z}_N$ for $N \geq |L|$. The relaxed ranking must also be efficiently invertible on the image of L. They show how to build FTE/FPE schemes using relaxed ranking, and provide an efficient relaxed ranking scheme starting only from the NFA representation of a language.

Context-free-language formats. All the work so far shows how to efficiently support formats defined by regular languages. But there are important settings in which allowing formats to be specified by CFGs would be more useful. For example, network protocol messages, file formats, and (programming) language specifications are often described by CFGs, not regexes. To overcome the restriction to regular languages, one needs an algorithm for efficiently ranking strings in a format described by a CFG. To the best of our knowledge, the only known ranking method using CFGs, that of GS [8], works only for unambiguous CFGs, is far too slow in practice, and lacks a corresponding unranking method.

What's more, while CFGs are more expressive then regexes, they are typically more difficult to specify and use, hence the widespread reliance on tools such as YACC and LEX [13]. In practice, almost no context-free language (CFL) is specified solely based on a CFG. A CFL is typically specified at two levels: a syntax level specified using a CFG; and a lexical level specified using regular expressions. The lexical level defines the symbols, also called *tokens*, used by the CFG. A grammar that uses regexes for tokens can be converted to a pure CFG, by including token definition in the CFG description, but this comes at the cost of programming effort, flexibility, explosion of the number of symbols, and, most importantly, runtime performance. Formatted encryption supporting CFLs should likewise handle this separation.

To sum up, the potential of formatted encryption is limited by the lack of (relaxed-)ranking schemes that efficiently handle context-free languages in practical settings. The main effort of this work is to move beyond this limitations.

Our contributions. We provide the first efficient approach for performing relaxed ranking from a CFG (either pure, or using lexical tokens). Our approach for CFLs follows a general two-step framework first introduced by Luchaup et al. [14], here we first map strings in the language to one of their parse trees (chosen deterministically), and then we perform a (strict) ranking of parse trees. Thus if the grammar is not ambiguous (each string has only one parse tree), then our relaxed-ranking approach becomes strict ranking. For the second step, we give a ranking method for parse trees which is "structural" in the sense that it allows the use of distinct ranking algorithms for trees derived from distinct grammar symbols. In particular, it allows for distinct ranking algorithms for the tokens of the language. As a result our method also works with languages specified using the two levels, syntax and lexical. This is an essential feature for efficiency and for using existing grammar specifications.

In fact, our method allows relaxed ranking of a slightly more general class of languages that we call *cfg-parseable* languages. An example is a format that starts with a byte counter that specifies how long the rest of the data is, and that ends with a checksum. This format is not context-free, or even context-sensitive.

Since ranking has applications beyond formatted encryption, such as compression and random language member generation [8, 11, 16], we also provide the first analysis of the issues involved when we replace ranking with relaxed ranking in those applications. One central matter in this analysis is the *ambiguity* of the grammar. We quantify ambiguity, and then show how it affects the quality of ranking applications, including formatted encryption.

We also provide a library that implements our relaxed ranking for arbitrary CFGs, and we deliver a tool that turns a simple YACC/LEX-like grammar specification into code that performs ranking for that grammar. This is then used as a pluggable component in the FPE/FTE framework from Luchaup et al. [14], yielding formatted encryption of context-free languages. We report on performance of our relaxed-ranking schemes, and show that we can rank C programs up to 5,000 bytes in under one second, or even under 12 ms if we bound the length of lexical tokens.

2. BACKGROUND

In this section we describe the state of the art in Formatted Encryption. We start with a formal definition of ranking and unranking, followed by a description of FTE and FPE (by extension). After that, we introduce relaxed ranking, and explain how FTE and FPE are adapted to use it. We conclude with the limitations of the current work.

Basic notions. A *format* is simply a language L, a set of strings over some alphabet. We will use the terms language and format interchangeably, and for simplicity we assume languages that are finite. In practice one may use infinite languages, but then take as format a *slice* of the language: if L is a language and $n \in \mathbb{N}$, then $L^{(n)} = \{w \in L : |w| = n\}$ is the slice of L that contains all its strings of length n.

A *format specification* describes a format L. A trivial specification is simply to list all elements of L, but for large languages this won't be efficient, so we do not consider it further. Rather, we seek compact and developer-friendly specifications. When L is regular, then options for specification include using a regular expression (regex), a non-deterministic finite automaton (NFA), or deterministic finite automaton (DFA). When L is context-free (but not regular), then a CFG becomes a natural option. Supporting CFG specifications is the goal of our paper.

Following Luchaup et al. [14], a format-transforming encryption (FTE) scheme is a pair of algorithms $(\mathcal{E}, \mathcal{D})$. The encryption algorithm $\mathcal{E}$ may or may not be randomized. It takes as input a pair of format specifications that specify a plaintext format L_p and a ciphertext format L_c, as well as a message $M \in L_p$, and a secret key. It produces a ciphertext $C \in L_c$ or a special error symbol $\perp$. Decryption $\mathcal{D}$ reverses the operation. To indicate the type of specifications supported by a scheme, we will often refer to XXX-specified FTE, where XXX $\in$ {DFA, NFA, regex, CFG}.

An FPE scheme is an FTE scheme for which $L_p = L_c$, meaning that ciphertexts and plaintexts must share the same format. We will use the term FTE to refer to schemes that may or may not be format-preserving.

DFA-specified FTE. Bellare, Ristenpart, Rogaway, and Stegers (BRRS) [2] first formalized the notion of FPE, building off prior work on de novo constructions [18], arbitrary-set enciphering schemes [3], and industry demand for the primitive. They also introduced an FPE scheme that works for any regular language. In BRRS, formats are specified using a deterministic finite automaton (DFA). Their construction makes use of ranking, which was first introduced in the context of language compression by Goldberg and Sipser (GS) [8]. Ranking found subsequent use in applications such as random language member generation [16], biology (c.f. [11]), and now in formatted encryption.

Let L be a language with $|L| = N$. A *ranking scheme* for L is a bijection $\mathsf{rank}_L : L \to \mathbb{Z}_N$ together with its inverse $\mathsf{unrank}_L : \mathbb{Z}_N \to L$. It is common to define the ranking function with respect to some total order $\prec$ on L; for example, the GS ranking takes $\prec$ to be the lexicographical ordering. Given a total order, the ranking function (relative to $\prec$) is defined by $\mathsf{rank}_L(x) = |\{y \in L : y \prec x\}|$.

BRRS give an FPE scheme for arbitrary regular languages using DFA-based ranking. A simple generalization of their FPE scheme to an FTE scheme is the following. Consider two languages X and Y, such that rank_X and unrank_X form a ranking scheme for X, and rank_Y and unrank_Y form a ranking scheme for Y. Assume that $|X| = |Y|$, and a cipher $E : \{0,1\}^k \times \mathbb{Z}_{|Y|} \to \mathbb{Z}_{|Y|}$ for some key length k. (Recall that a cipher is a family of functions such that for any $K \in \{0,1\}^k$ it is the case that $E_K(\cdot) = E(K, \cdot)$ defines a permutation on its domain.) Let the inverse of E_K be E_K^{-1}. Then a plaintext $M \in X$ can be encrypted to ciphertext $C \in Y$ using the *rank-encipher-unrank* construction defined as shown in Figure 1.

```
$\underline{\mathcal{E}_K(M):}$
If $M \notin X$ then Return $\perp$
$r \leftarrow \mathsf{rank}_X(M)$
$c \leftarrow E_K(r)$
Return $\mathsf{unrank}_Y(c)$
```

```
$\underline{\mathcal{D}_K(C):}$
If $C \notin Y$ then Return $\perp$
$c \leftarrow \mathsf{rank}_Y(C)$
$r \leftarrow D_K(c)$
Return $\mathsf{unrank}_X(r)$
```

Figure 1: Encryption and decryption using the rank-encipher-unrank construction.

When $X = Y$, this matches the BRRS FPE scheme. We note that the BRRS scheme uses the GS algorithm for (un)ranking, and this requires X and Y to be specified by DFAs. BRRS also argue that the DFA representation is effectively necessary for regular X, Y, because ranking from given either the regex or NFA representations is PSPACE-complete.

Regex-specified FTE. FTE was first introduced by Dyer, Coull, Ristenpart, and Shrimpton (DCRS) [7]. Their motivating application was avoidance of network censors that identify anti-censorship protocols via regex-based deep-packet inspection. Hence, they targeted FTE with formats specified by regular expressions. They also sought to support arbitrary unformatted data as plaintexts. Like BRRS, they used the DFA-based GS ranking scheme. To accommodate regex specifications, DCRS employed the classic regex-to-NFA-to-DFA conversion process (c.f., [19]). For some regexes, this process leads to poor performance (or even failure) because the NFA-to-DFA conversion results in an exponential increase in automaton size. However, they show experimentally that this behavior is not typical in their use cases.

FTE was revisited by Luchaup, Dyer, Jha, Ristenpart, and Shrimpton (LDJRS) [14], who generalized the DCRS treatment in various ways. LDJRS also provided a new method for regex-specified FTE that completely avoids the NFA-to-DFA conversion. The basis of their new technique is called *relaxed ranking*. Consider a language L and the integers $\mathbb{Z}_N$ for some $N \geq |L|$. A *relaxed-ranking scheme* for L is a pair of functions $\mathsf{Rank}_L : L \to \mathbb{Z}_N$ and $\mathsf{Unrank}_L : \mathbb{Z}_N \to L$ such that for all $x \in L$ it holds that $\mathsf{Unrank}_L(\mathsf{Rank}_L(x)) = x$. See Figure 2. (We capitalize Rank and Unrank to distinguish them from strict ranking with functions rank and unrank.) It follows that Rank is always injective, while Unrank may be surjective in the case that $|L| < N$.

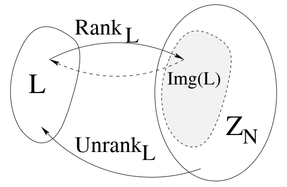
LDJRS gave a recipe for building a relaxed-ranking scheme for a language L, requiring three ingredients (see the middle diagram in Figure 2). First, one specifies an "intermediate" set I for which one has efficient algorithms for $\mathsf{rank}_I : I \to \mathbb{Z}_N$ and $\mathsf{unrank}_I : \mathbb{Z}_N \to I$ where $N = |I|$. Second, a function $\mathsf{map} : L \to I$. Finally, a function $\mathsf{gen} : I \to L$ such that $\forall x \in L$, $\mathsf{gen}(\mathsf{map}(x)) = x$. (It follows that map is injective and gen is surjective). From these, they define $\mathsf{Rank}_L : L \to \mathbb{Z}_N$ by $\mathsf{Rank}_L(x) = \mathsf{rank}_I(\mathsf{map}(x))$, and they define $\mathsf{Unrank}_L : \mathbb{Z}_N \to L$ by $\mathsf{Unrank}_L(a) = \mathsf{gen}(\mathsf{unrank}_I(a))$.

To accommodate relaxed ranking, LDJRS designed an FTE scheme that uses a technique known as *cycle waling*. We give their cycle-walking FTE scheme in the rightmost box of Figure 2, and use $Img(L)$ to denote the image of Rank_L for $L \in \{X, Y\}$. Encryption with cycle walking seeks a point in $Img(Y) \subseteq \mathbb{Z}_N$ which will map to a valid ciphertext under the relaxed ranking. Note that the scheme as written assumes plaintext format X and ciphertext format Y are such that $|X| = |Y|$, but LDJRS give variants that work for more general pairs of formats. We refer the reader to [14] for the details.

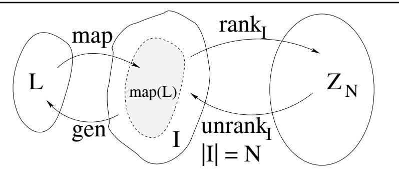
LDJRS give a relaxed ranking scheme for NFAs, which enables efficient regex-specified FTE by first converting a regex to an NFA. They show that NFA-based relaxed ranking often leads to schemes almost as fast as DFA-based ranking schemes without the blow-up in automaton size.

CFG-specified FTE. While regex-specified FTE may be sufficient in some contexts, there are settings in which we would prefer to use context-free grammars (CFGs). CFGs allow expression of richer languages, and are often used in practice to describe the structure of files, web pages, protocol messages, and more. However, none of the BRRS, DCRS or LDJRS schemes provide CFG-specified FTE.

GS sketched a (high-degree) polynomial-time ranking for context-free languages. While theoretically interesting, their ranking only works with unambiguous CFGs and has no matching unranking algorithm. Recall that an ambiguous CFG is one which has multiple derivations for some string in the associated language; moreover, determining whether or not a grammar is ambiguous is undecidable. Thus, the



$\mathsf{Rank}_L : L \to \mathbb{Z}_N$ injective
$\mathsf{Unrank}_L : \mathbb{Z}_N \to L$ surjective
$\forall x : \mathsf{Unrank}_L(\mathsf{Rank}_L(x)) = x$



$\mathsf{map} : L \to I$ injective
$\mathsf{gen} : I \to L$ surjective
$\forall x : \mathsf{gen}(\mathsf{map}(x)) = x$

```
$\mathcal{E}_K(M)$ :                     $\mathcal{D}_K(C)$ :
If $M \notin X$ then                 If $C \notin Y$ then
  Return $\perp$                       Ret $\perp$
$r \leftarrow \mathsf{Rank}_X(M)$                $r \leftarrow \mathsf{Rank}_Y(C)$
Do                                   Do
  $r \leftarrow E_K(r)$                    $r \leftarrow D_K(r)$
Until $r \in Img(Y)$                 Until $r \in Img(X)$
Return $\mathsf{Unrank}_Y(r)$              Return $\mathsf{Unrank}_X(r)$
```

Figure 2: **(Left)** Diagram of relaxed ranking. **(Middle)** Using strict ranking on an intermediate set I to obtain relaxed ranking on L. **(Right)** The cycle-walking construction of FTE using relaxed ranking from [14] for formats X, Y with $|X| = |Y|$.

GS ranking based on CFGs is not well suited to practice. Mäkinen [15] later presented an algorithm for ranking and unranking of Slizard languages (not general CFL). However, the presentation leaves room for interpretation and it appears to contain flaws in some parts of the algorithm.

In summary, there currently exists no efficient mechanism for ranking (relaxed or otherwise) given a CFG representation of a CFL. However, the difficulty in handling CFLs goes beyond the lack of efficient CFG-based ranking methods. Using a monolithic or "pure" CFG for the language would be impractical, even if efficient CFG-based ranking existed. This is because in practice people do not use "pure" CFGs to describe a language. In practice, a language is described at two levels: a lexical level that defines parsing tokens using regexes, and a second level that defines the syntax using a pure CFG that has the tokens as terminals. Although every such *two-level* grammar can be converted to a pure CFG, by absorbing the tokens in the CFG description, this comes at the cost of programming effort, flexibility, explosion of the number of symbols, and runtime performance.

Another difficulty is that a CFG specification is often more complex than a regex specification, and we need a tool to relieve the programmer from the tedious and error prone process of CFG specification.

Finally, some formats used in practice are not context free. For instance, counter based formats of the form:

```
<Item> := <# bytes in Item> BYTE* <Checksum>
```

We are not aware of any ranking method that can handle such formats in a principled way.

In the rest of this paper we show how we address these limitations. First we provide an efficient CFG-based relaxed ranking. Then we show how we adapt it to handle two-level lexer/parser language specifications, including ones that include counters and similar non-context-free embellishments.

3. RELAXED RANKING FOR CFLS

We present an algorithm for relaxed ranking of a context-free language. It will use a two-stage approach, with intermediate objects being parse trees. First we present some background on CFGs.

3.1 Background and Definitions

A context-free grammar (CFG) is a 4-tuple $G = (\mathcal{N}, \Sigma, \mathcal{R}, S)$, where $\mathcal{N}$ is a finite set of non-terminals; Σ is a finite set of terminals, such that $\mathcal{N} \cap \Sigma = \emptyset$; $\mathcal{R}$ is a finite relation $\mathcal{R} \subseteq \mathcal{N} \times (\mathcal{N} \cup \Sigma)^*$; and $S \in \mathcal{N}$ is the start symbol. If $(A, w) \in \mathcal{R}$, we write $A \to w$, and we say that w is derived from A in one step. We extend $\to$ to a derivation relation from words in $(\mathcal{N} \cup \Sigma)^*$ to words in $(\mathcal{N} \cup \Sigma)^*$ as follows. We say that s_2 is derivable from s_1 in one step, and we write $s_1 \to s_2$ iff $\exists X \in \mathcal{N}\ \exists w_1, w_2, w_3 \in (\mathcal{N} \cup \Sigma)^* : s_1 = w_1 X w_3 \wedge X \to w_2 \wedge s_2 = w_1 w_2 w_3$. In the later case, we say that $s_2 = w_1 w_2 w_3$ can be derived from $s_1 = w_1 X w_3$ using the rule $\rho = X \to w_2$, and we write $s_1 \to^{\rho} s_2$; if $w_1 \in \Sigma^*$ we say that $w_1 X w_2 \to w_1 w_2 w_3$ is the leftmost derivation (by expanding the leftmost non-terminal). The transitive closure of $\to$ is $\to^*$. The language accepted by G is the set $L(G) = \{w \in \Sigma^* : S \to^* w\}$. A language is a *context-free language* (CFL) if it is accepted by a CFG. A grammar G is unambiguous iff for all $w \in L(G)$ there is exactly one sequence of leftmost derivations $\rho_1, \rho_2, ..., \rho_k$ that lead from S to w, i.e., $S \to^{\rho_1} s_1 \to^{\rho_2} s_2 \to^{\rho_3} ...s_{k-1} \to^{\rho_k} s_k = w$.

A symbol X in a CFG is called *useless* if it does not occur in any derivation of a word from $L(G)$. A symbol $A \in \mathcal{N}$ is *recursive* if there is a non empty chain of rules such that $A \to^* w_1 A w_2$.

Rule ordering. For every non-terminal $A \in \mathcal{N}$, we assume that the ordered set (enumeration) of rules that have A on the left hand side is $\mathcal{R}_A = \{\rho_1, ..., \rho_{|\mathcal{R}_A|} | \rho_i : A \to w_i \in \mathcal{R}\}$. This enumeration defines an arbitrary but fixed order on A's rules, where $\rho_k <^{\mathcal{R}} \rho_j \iff k < j$ (i.e. ρ_k precedes ρ_j in the ordering $\mathcal{R}_A$). The kth rule in $\mathcal{R}_A$ is denoted by $\mathcal{R}_A[k]$. This assumption will be used in the upcoming relaxed ranking algorithm.

Labeled trees and parse trees. Informally, a *parse tree* (or a *derivation tree*) for a CFG grammar $G = (\mathcal{N}, \Sigma, \mathcal{R}, S)$ is a tree in which every node is labeled with a rule $\rho \in \mathcal{R}$. The rule for a node determines how that node is "expanded" by G, i.e. the number of the node's children and their labels, which must correspond to the non-terminals on the right side of $\mathcal{R}_A[k]$.

To make things precise, we define the following. A *labeled tree* $\mathcal{T}$=(V, E, r_0, λ) over a set Λ is a directed tree where each vertex has a label in Λ. Here (V, E, r_0) is a tree where V is the set of vertexes, $E \subseteq V \times V$ is the set of directed edges, $r_0 \in V$ is the root; Λ is the set of labels; and $\lambda : V \to \Lambda$ associates a label to each vertex. For a vertex $v \in V$, $v[k]$ is the kth child of v. The tree is finite, unless otherwise specified. A *parse-tree* for a CFG grammar $G = (\mathcal{N}, \Sigma, \mathcal{R}, S)$ is a labeled tree $\mathcal{T}$=(V, E, r_0, λ) over $\mathcal{R}$, where:

1. $\lambda : V \to \mathcal{R}$ labels each node v with a rule. If $\lambda(v) = \rho : A \to ... \in \mathcal{R}$, we say that v's type is A and v's rule is ρ. For convenience, we use the notation $v.s$ to denote A, and $v.\rho$ to denote ρ.

2. For every node v, if $v.\rho$ is $A \to s_0A_1s_1A_2...A_ns_n$, where $n \geq 0, s_i \in \Sigma^*$, and $A_i \in \mathcal{N}$, then v must have n children $v_1, ..., v_n$, and each child v_i must have $v_i.s = A_i$.

Define $PT(G)$ to be the set of all parse trees generated by a grammar G, and $PT(G, A) \subseteq PT(G)$ to be the subset of parse trees with type of the root as A.

The yield of a parse tree. Next, we define a function that takes a parse tree as input, and outputs the string which is the yield for the tree. Let T be a parse tree with root r_0, and let v be a parse-tree node. Let $\mathcal{W}(v) \in \Sigma^*$ be the string recognized by v, defined inductively as follows. If $v_1, ..., v_n$ are v's children, $n \geq 0, s_i \in \Sigma^*$, and $A_i \in \mathcal{N}$, and if $v.\rho$ is $A \to s_0A_1s_1A_2...A_ns_n$, then $\mathcal{W}(v) = s_0\mathcal{W}(v_1)s_1\mathcal{W}(v_2)...\mathcal{W}(v_n)s_n$. Overloading notation, define $\mathcal{W}(T) = \mathcal{W}(r_0)$, i.e. the string recognized by T (the *yield* of T). Define $\|T\|$ to be $|\mathcal{W}(T)|$, the length of the yield $\mathcal{W}(T)$. Thus, we can define a function $\mathcal{W} : PT(G) \to \Sigma^*$. Now, for every $A \in \mathcal{N}$, let $\mathcal{W}_A : PT(G, A) \to \Sigma^*$ be defined by $\mathcal{W}_A(T) = \mathcal{W}(T)$. Note that when $A = S$, we have $\mathcal{W}_S : PT(G, S) \to L(G)$.

Minimal parse trees. We will propose a ranking scheme that counts the number of parse-trees that yield strings of a given size. To avoid the problem that there may be an infinite number of such trees, we introduce the concept of minimal parse-trees, and we will count only those.

Let $G = (\mathcal{N}, \Sigma, \mathcal{R}, S)$ be a CFG. We say that $T_1 \in PT(G, A)$ is *reducible* iff T_1 has a subtree $T_2 \in PT(G, A)$ with $\|T_1\| = \|T_2\|$ (equivalently, with $\mathcal{W}(T_1) = \mathcal{W}(T_2)$). Furthermore, we say that T is *minimal* iff T has no reducible subtrees. We define $MPT(G)$ to be the set of minimal parse trees generated by a grammar G, and $MPT(G, A) \subseteq PT(G, A)$ is the set of minimal parse trees with root A. By extension, we define the set of such parse trees with yield-length k to be $MPT(G, A, k) = \{Y \in MPT(G, A) : \|Y\| = k\}$.

We these definitions in place, we state a simple lemma, whose proof we defer to the full version of this paper. In Section 3.2, this lemma will provide assurance that we are counting finite sets.

LEMMA 1. *For all $k \geq 0$, $MPT(G, A, k)$ is finite.*

3.2 Relaxed ranking

Having built up this machinery, we proceed to describe our two-step relaxed ranking approach. Let G be a CFG corresponding to a CFL L_0. We pick a slice size $z \in \mathbb{N}$ and describe the relaxed ranking of the language slice $L = L_0^{(z)}$. We define the intermediate set $I = MPT(G, S, z)$, the set of minimal parse trees derived from S, the start symbol for G, which yield a string of length z. Given this I and L, it remains to define the functions gen and map. For simplicity, in the rest of this section, when we use the term *parse tree*, we mean *minimal parse tree*, unless otherwise specified.

First, let $\mathsf{gen} = \mathcal{W}_S$, i.e. for all $T \in I = MPT(G, S, z)$, $\mathsf{gen}(T) = \mathcal{W}_S(T)$. To define map, we assume that we have a deterministic parser which produces minimal parse trees (we will address this assumption later). Then, for all $w \in L$, we define $\mathsf{map}(w)$ to be a deterministically picked parse tree for w in $MPT(G, S)$. In other words map *is* the deterministic parser. So, to finish our two-step relaxed ranking, it remains to define the strict ranking and unranking algorithms, rank_I and unrank_I, from parse trees to integers.

Just before doing exactly that, we note that ranking a word or string $w \in L$ clearly requires parsing to generate a tree in $MPT(G, S)$. The common parsing methods for an arbitrary grammar can take $\mathcal{O}(l^3|G|)$ time in the worst case, where l is the length of the word being parsed. (Earley's method is said to perform much better in practice.) The worst case cubic time of parsing a general CFG cannot be improved much, and [12] shows why this is the case by providing an efficient reduction of Boolean Matrix Multiplication to CFG parsing.

Strict ranking/unranking of minimal parse trees. We show how to perform ranking by comparing two parse trees whose roots have the same type. While it is possible to define a ordering on all parse trees, this is not necessary for our purpose. Our final goal is ordering of strings in $L(G)$, hence strings derived from S, whose parse trees have some rule of the form $S \to w$ at the root. For this, it suffices to define an ordering only among trees of the same type, but we must take a few precautions to prevent the case where a string may have an infinite number of parse-trees.

Assume that $\prec_A$ is a total order on $MPT(G, A)$ for non-terminal A ($\prec_A$ is induced by the ordering on the rules and described in detail later in the section). For all $X \in MPT(G, A)$, let $\mathsf{rank}_A(X)$ be defined as:

$$\mathsf{rank}_A(X) = |\{Y \in MPT(G, A, \|X\|) : Y \prec_A X\}| \qquad (1)$$

The definition is well founded because for any G the set $MPT(G, A, \|X\|)$ is finite (Lemma 1); which is why we require parsers that produce minimal parse-trees. This is a realistic requirement satisfied by all efficient parsing methods that we know. For a large class of grammars, such as all unambiguous grammars, or grammars without ϵ-productions, all parse trees are minimal, i.e. $MPT = PT$. For other grammars we can easily remove ϵ-productions so that they have only minimal parse trees, but we do not require such a conversion as long as there is a parser that can produce minimal parse trees.

Observe that $\mathsf{rank}_A(X)$ is the position of X among the trees of same yield length in $MPT(G, A, \|X\|)$, rather than all trees in $MPT(G, A)$. This is convenient for language slices of the form $L(G)^{(n)}$.

In the following we show how to obtain a total order for the parse-trees of a grammar. The ordering is based on the length of the yield of the trees, on an ordering of the grammar rules, and on the structure of the trees. In particular:

1. Trees with shorter yields precede trees with longer yields.
2. For the same length, the trees derived from an earlier grammar rule precede the trees derived from a later rule.
3. At the same yield length and grammar rule, the trees are compared based on their ordered children, in a manner similar to lexicographical ordering: we first compare the children on the first position, and if equal then we compare the children on the second position, etc.

For simplicity of presentation, but without lack of generality, it is useful to assume that the grammar has a simpler form, which we call a *weak normal form (WNF)*. Formally, a grammar $G = (\mathcal{N}, \Sigma, \mathcal{R}, S)$ is in weak normal form (WNF) if G has no useless symbols and G's rules can only have one of the following forms:

1. $A \to A_1A_2$, with $A, A_1, A_2 \in \mathcal{N}$

2. $A \to \alpha$, with $A \in \mathcal{N}, \alpha \in \Sigma$
3. $A \to A_1$, with $A, A_1 \in \mathcal{N}$
4. $A \to \epsilon$, with $A \in \mathcal{N}$

We note that WNF is similar to *Chomsky normal form (CNF)*, but less restrictive, because WNF allows two additional rule forms. Any CFG can easily be converted to to WNF, similar to the conversion to CNF, and CNF implies WNF.

Typed Parse-Tree Ordering. Let $G = (\mathcal{N}, \Sigma, \mathcal{R}, S)$ be an arbitrary CFG in WNF, and assume that there is an ordering relation $<^{\mathcal{R}}$ on $\mathcal{R}$ that orders the rules in $\mathcal{R}_A$, for every $A \in \mathcal{N}$. The following definition specifies an ordering relation on the set $MPT(G, A)$, and from this we will define how to rank parse trees.

Definition 1. *For every non-terminal $A \in \mathcal{N}$, we define an ordering relation $\prec_A$ on $MPT(G, A)$ as follows: $\forall X, Y \in MPT(G, A) : Y \prec_A X$ iff $\|Y\| < \|X\|$ or $\|Y\| = \|X\|$ and one of the following conditions is true:*

1. $Y.\rho <^{\mathcal{R}} X.\rho$
2. $Y.\rho = X.\rho = A \to A_1 \quad \wedge Y[1] \prec_{A_1} X[1]$
3. $Y.\rho = X.\rho = A \to A_1A_2 \quad \wedge Y[1] \prec_{A_1} X[1]$
4. $Y.\rho = X.\rho = A \to A_1A_2 \quad \wedge Y[1] = X[1] \quad \wedge Y[2] \prec_{A_2} X[2]$

Although these equations can lead to recursion (if A is recursive), the definition is well founded because X and Y are finite trees, and their children have fewer nodes.

The following lemma, whose proof we defer to the full version, provides assurance that our upcoming ranking scheme works.

Lemma 2. *For every CFG G and non-terminal A in G the order $\prec_A$ is a strict total order on $MPT(G, A)$.*

If $\rho \in \mathcal{R}_A$, the conditions described in Definition 1 are mutually exclusive and we count how many Y's satisfy each of them using the following auxiliary values:

1. $N_\rho(l)$ = number of parse trees that yield a string of length l derived from rule ρ.
   $N_\rho(l) = |\{Y \in MPT(G) : \|Y\| = l \wedge Y.\rho = \rho\}|$
2. $N_A(l)$ = number of parse trees that yield a string of length l derived from A.
   $N_A(l) = |MPT(G, A, l)|$
3. $B_\rho(l)$ = number of parse trees that yield a string of length l derived from a rule of A preceding ρ. This corresponds to condition 1 in Definition 1.
   $B_\rho(l) = \left|\{Y \in MPT(G, A, l) : Y.\rho <^{\mathcal{R}} \rho\}\right|$
4. $B_\rho(l_1, l)$ = number of parse trees that yield a string of length l derived from rule $\rho : A \to A_1A_2$, and whose first child yields a string shorter than l_1.
   $B_\rho(l_1, l) = |\{Y \in MPT(G, A, l) : \|Y[1]\| < l_1\}|$

We use these values to compute $\mathsf{rank}_A(X)$ based on Equation (1), by observing that the conditions in Definition 1 are mutually exclusive. This justifies the correctness of the ranking function shown in Algorithm 1. Observe that $\|X\| \neq \|Y\| \implies \mathsf{rank}_A(X) \neq \mathsf{rank}_A(Y)$. Therefore the restriction of rank_A to $I = MPT(G, S, z)$ is bijective; its inverse is given by $\mathsf{unrank}_A(\_, z)$ in Algorithm 2.

The correctness of unranking in Algorithm 2 uses the following observation: Assume that $\rho, \rho' \in \mathcal{R}_A, \rho <^{\mathcal{R}} \rho'$, $X.\rho = \rho$ and $\|X\| = l$. Then (1) $B_\rho(l) \leq \mathsf{rank}_A(X) < B_{\rho'}(l)$. Using

```
$\mathsf{rank}_\mathsf{A}(X \in MPT(G, \mathsf{A}))$ :
    $\rho \leftarrow X.\rho$;
    $\mathsf{l} \leftarrow \|X\|$;
    switch $(\rho)$ do // check all WNF rule forms
        case $\mathsf{A} \to \mathsf{A}_1$ :    return $B_\rho(\mathsf{l}) + \mathsf{rank}_{A_1}(X[1])$;
        case $\mathsf{A} \to \mathsf{A}_1\mathsf{A}_2$:
            $\mathsf{l}_1 \leftarrow \|X[1]\|$;
            $\mathsf{r}_1 \leftarrow \mathsf{rank}_{\mathsf{A}_1}(X[1])$; // rank of $1^{st}$ child
            $\mathsf{r}_2 \leftarrow \mathsf{rank}_{\mathsf{A}_2}(X[2])$; // rank of $2^{nd}$ child
            return $B_\rho(\mathsf{l}) + B_\rho(\mathsf{l}_1, \mathsf{l}) + \mathsf{r}_1 * \mathsf{N}_{\mathsf{A}_2}(\mathsf{l} - \mathsf{l}_1) + \mathsf{r}_2$;
        case $\mathsf{A} \to \epsilon$ :    return $B_\rho(\mathsf{l})$;
        case $\mathsf{A} \to \alpha$ :    return $B_\rho(\mathsf{l})$;
```

Algorithm 1: Ranking of parse trees

```
$\mathsf{unrank}_\mathsf{A}(rank\ \mathsf{r}, length\ \mathsf{l})$ :
    $\rho \leftarrow \mathbf{max}_{<^{\mathcal{R}}}\{\rho \in \mathcal{R}_\mathsf{A} : B_\rho(\mathsf{l}) \leq \mathsf{r}\}$;
    $\mathsf{r}' \leftarrow \mathsf{r} - B_\rho(\mathsf{l})$;
    switch $\rho$ do // check all WNF rule forms
        case $\mathsf{A} \to \mathsf{A}_1$: return $\mathsf{Tree}(\rho, \mathsf{unrank}_A(\mathsf{r}', \mathsf{l}))$;
        case $A \to A_1A_2$: //child yields?
            // if $r_1, r_2$ = rank of $1^{st}$ and $2^{nd}$ child,
               and $\mathsf{l}_1, \mathsf{l}_2$ = length of children yields,
               then (1) $\mathsf{l}_1 + \mathsf{l}_2 = \mathsf{l}$, and (2) $\mathsf{l}_1$ satisfies
               $\mathsf{r}' = B_\rho(\mathsf{l}_1, \mathsf{l}) + \mathsf{r}_1 * N_{A_2}(\mathsf{l} - \mathsf{l}_1) + \mathsf{r}_2$
            $\mathsf{l}_1 \leftarrow \mathbf{max}\{\mathsf{l}_1 \in [0..\mathsf{l}] : B_\rho(\mathsf{l}_1, \mathsf{l}) \leq \mathsf{r}'\}$;
            $\mathsf{r}'' \leftarrow \mathsf{r}' - B_\rho(\mathsf{l}_1, \mathsf{l})$;
            $r_1 \leftarrow \mathsf{r}''\ /\ N_{A_2}(\mathsf{l} - \mathsf{l}_1)$;
            $X[1] \leftarrow \mathsf{unrank}_{A_1}(r_1, \mathsf{l}_1)$;// $1^{st}$ child
            $r_2 \leftarrow \mathsf{r}''\ \%\ N_{A_2}(\mathsf{l} - \mathsf{l}_1)$;
            $X[2] \leftarrow \mathsf{unrank}_{A_2}(r_2, \mathsf{l}_2)$;// $2^{nd}$ child
            return $\mathsf{Tree}(\rho, X[1], X[2])$;
        case $A \to \epsilon$: return $\mathsf{Tree}(\rho)$;
        case $A \to \alpha$: return $\mathsf{Tree}(\rho)$;
```

Algorithm 2: Unranking of parse trees

this observation it is easy to see that the value $\mathbf{max}_{<^{\mathcal{R}}}\{\rho \in \mathcal{R}_A : B_\rho(l) \leq r\}$ uniquely and correctly identifies the desired rule of A. The only non-trivial case is when this rule is of the form $A \to A_1A_2$. In this case, we must determine l_1, which is the size of the corresponding word derived from A_1. Using the formula in the ranking function and $r' = r - B_\rho(l)$ we get:

$$\begin{aligned}
r' &= B_\rho(l_1, l) + \mathsf{rank}_{A_1}(X[1])N_{A_2}(l - l_1) + \mathsf{rank}_{A_2}(X[2]) \\
&< B_\rho(l_1, l) + \mathsf{rank}_{A_1}(X[1])N_{A_2}(l - l_1) + N_{A_2}(l - l_1) \\
&= B_\rho(l_1, l) + (\mathsf{rank}_{A_1}(X[1]) + 1)N_{A_2}(l - l_1) \\
&\leq B_\rho(l_1, l) + N_{A_1}(l_1)N_{A_2}(l - l_1) \\
&= B_\rho(l_1 + 1, l)
\end{aligned}$$

This guarantees that the selection of l_1 such that $B_\rho(l_1, l) \leq r' < B_\rho(l_1 + 1, l)$ succeeds and is correct. The correctness for the rest of the unranking algorithm is straightforward.

Algorithm 3 computes the values $N_\rho(l), N_A(l), B_\rho(l)$ and $B_\rho(l_1, l)$, using memoization only for N_A and N_ρ. All memoization tables entries are initialized to $\perp$. Although the grammar can be recursive, the functions in Algorithm 3 do not lead to infinite recursions, because of the check and initialization code placed at lines 2–3 in N_ρ and at lines 13–14 in N_A. If, during the first execution of $N_A(l)$, a recursive call to N_A is made using the same value for l, then the second call returns 0. This is correct, because N_A counts the number of minimal parse trees rooted in A, and a minimal parse tree in $MPT(G, A, l)$ should not have another subtree in $MPT(G, A, l)$.

```
$\mathsf{N}_\rho(\mathsf{l} \in \mathbb{N})$ :   // use table _N_ρ initialized with ⊥
    if ($\_\mathsf{N}_\rho[\mathsf{l}] \neq \bot$) then return $\_\mathsf{N}_\rho[\mathsf{l}]$;
    $\_\mathsf{N}_\rho[\mathsf{l}] \leftarrow 0$;          // must be 0, in case of recursion
    switch ($\rho$) do // check all WNF rule forms
        case $\mathsf{A} \to \mathsf{A}_1$ :      $\mathsf{r} \leftarrow N_{\mathsf{A}_1}(\mathsf{l})$;
        case $\mathsf{A} \to \mathsf{A}_1\mathsf{A}_2$: $\mathsf{r} \leftarrow \Sigma_{i=0}^{\mathsf{l}} N_{\mathsf{A}_1}(i) N_{\mathsf{A}_2}(\mathsf{l}-i)$;
        case $\mathsf{A} \to \epsilon$ :       $\mathsf{r} \leftarrow (\mathsf{l} = 0)?1 : 0$;
        case $\mathsf{A} \to \alpha$ :       $\mathsf{r} \leftarrow (\mathsf{l} = 1)?1 : 0$;
    $\_\mathsf{N}_\rho[\mathsf{l}] \leftarrow \mathsf{r}$;
    return $\_\mathsf{N}_\rho[\mathsf{l}]$;

$\mathsf{N}_\mathsf{A}(\mathsf{l} \in \mathbb{N})$ :  // use table _N_A initialized with ⊥
    if ($\_\mathsf{N}_\mathsf{A}[\mathsf{l}] \neq \bot$) then return $\_\mathsf{N}_\mathsf{A}[\mathsf{l}]$;
    $\_\mathsf{N}_\mathsf{A}[\mathsf{l}] \leftarrow 0$;          // must be 0, in case of recursion
    $\_\mathsf{N}_\mathsf{A}[\mathsf{l}] \leftarrow \sum_{\rho \in \mathcal{R}_\mathsf{A}} \mathsf{N}_\rho(l)$;
    return $\_\mathsf{N}_\mathsf{A}[\mathsf{l}]$;

$\mathsf{B}_\rho(\mathsf{l} \in \mathbb{N})$ :
    return $\sum_{\rho' <^{\mathcal{R}} \rho} \mathsf{N}_{\rho'}(l)$;

$\mathsf{B}_{\mathsf{A} \to \mathsf{A}_1\mathsf{A}_2}(\mathsf{l}_1, \mathsf{l} \in \mathbb{N})$ :
    return $\Sigma_{i=0}^{\mathsf{l}_1 - 1} N_{\mathsf{A}_1}(i) N_{\mathsf{A}_2}(\mathsf{l}-i)$;
```

Algorithm 3: Computing N_ρ, N_A, B_ρ and B_ρ. N_ρ and N_A use memoization tables $\_N_\rho$ and $\_N_A$, respectively.

Complexity of the algorithms. Assume that $\|X\| = l$. The ranking and unranking functions visit each of the $\mathcal{O}(l|\mathcal{R}|)$ parse-tree nodes of X exactly once. The complexity of the computation at each node depends on the amount of memoized information for $N_\rho(l), N_A(l), B_\rho(l)$ and $B_\rho(l_1, l)$. Considering numbers as large as $\left|\Sigma^l\right|$, we let $b(l) = \mathcal{O}(l)$ be the space needed to store such a number, $a(l) = \mathcal{O}(l)$ be the time needed to add two such numbers, $m(l)$ be the time needed to multiply two such numbers, and $d(l)$ be the time needed to divide two such numbers. We assume $a(l) \leq m(l) \leq d(l)$.

At one end, all these values are pre-computed for up to a maximum value of $\|X\| = l$. In this case, the tables have $\mathcal{O}(l^2|\mathcal{R}|)$ entries, where each entry holds an integer whose representation may take $\mathcal{O}(l)$ bits, therefore the total amount of space is $\mathcal{O}(l^3|\mathcal{R}|)$, where $|\mathcal{R}|$ is the number of rules in the grammar. Filling in these tables takes $\mathcal{O}(l^3 m(l)|\mathcal{R}|)$ time, where $m(l)$ is the complexity of multiplying two numbers of $\mathcal{O}(l)$ bits.

If all tables are available, then ranking spends $\mathcal{O}(m(l))$ time at each node, for a total of $\mathcal{O}(lm(l)|\mathcal{R}|)$ time. Unranking spends $\mathcal{O}(d(l))$ time at each node, for a total of $\mathcal{O}(ld(l)|\mathcal{R}|)$, where $d(l)$ is the complexity of dividing two numbers of $\mathcal{O}(l)$ bits.

3.3 Ambiguity

Ambiguity is the key factor that determines the quality and usefulness of relaxed ranking, and yet it was not fully explored so far, although it was implicitly used in [14]. Ambiguity is not specific to CFG ranking, but it is particularly relevant, since it is impossible to decide whether a grammar is ambiguous or not. A related line of work [9, 10, 17] relates ambiguity to complexity of languages. In this section, however, we define and analyze ambiguity in the context of relaxed ranking, and explain why relaxed ranking is useful, despite potential inefficiencies caused by ambiguous representations.

The *ambiguity-factor* of a relaxed ranking using function $\mathsf{Unrank}_L \colon \mathbb{Z}_N \to L$ is defined to be the ratio $\beta = N/|L|$. We note that if the relaxed ranking scheme is obtained with an intermediate set I, then $\beta = |I|/|L|$. For instance, an ambiguous grammar has multiple parse trees for some strings and the ambiguity-factor measures how many more trees are than strings (of a given size).

For most of ranking applications, it is easy to see that relaxed ranking can be used as a swap-in replacement for strict ranking, though possibly with slight degradation in performance. The ambiguity-factor quantifies this degradation.

For the following three application areas, consider a finite language slice L, and a relaxed-ranking scheme $\mathsf{Rank}_L \colon L \to \mathbb{Z}_{|I|}$ based on an intermediate set I.

Compression. Relaxed ranking $\mathsf{Rank}_L(x)$ compresses a word $x \in L$ using $\gamma = \lceil \log_2 |I| \rceil$ bits. Decompression follows from $\mathsf{Unrank}_L(\mathsf{Rank}_L(x)) = x$. Compared to the ideal case of ranking using $\alpha = \lceil \log_2 |L| \rceil$ bits, relaxed ranking has an overhead of $\gamma - \alpha$ bits, or approximately $\lceil \log \beta \rceil$ where $\beta = |I|/|L|$ is the ambiguity-factor of the relaxed ranking scheme. Thus *the lower the ambiguity, the better the compression.*

Random member generation. To randomly and uniformly generate a string in a language L, one can pick a random number $n \in \mathbb{Z}_{|L|}$ and then compute $x = \mathsf{unrank}(n)$ using a strict ranking scheme. When we seek to replace unrank with relaxed unranking, we need to avoid distribution biases towards those elements $x \in L$ for which $x = \mathsf{Unrank}_L(n)$ holds for multiple $n \in \mathbb{Z}_{|I|}$. This prevents highly ambiguous elements from being generated more often than the others. One can solve this using rejection sampling: repeatedly pick $n \in \mathbb{Z}_{|I|}$ until $\mathsf{Rank}(\mathsf{Unrank}(n)) = n$. The expected number of trials is exactly the ambiguity-factor $\beta = |I|/|L|$.

Formatted encryption. Replacing ranking with relaxed ranking does not work directly for formatted encryption. Therefore, Listing 2 uses the technique of cycle walking. It easy to see that the expected number of steps in the cycle walk is upper bounded by the ambiguity-factor $\beta = |I|/|L|$, because $|Img(L)| = |L|$. The other algorithmic adaptations for relaxed ranking in [14] are similarly influenced by the ambiguity-factor. Observe that there is no need to decide whether the grammar is ambiguous, the algorithms are protected against this possibility. If the grammar is not ambiguous, then relaxed ranking is in fact strict ranking, and the algorithmic overhead is minimal. For instance the cycle-walk in Listing 2 stops after one step.

Our thesis (backed by experimental data) is that for common applications, *the ambiguity-factor is low and relaxed ranking is a good replacement for strict ranking.*

3.4 Type-Based Customization

Recall that our definition 1 is structural. In other words, the ordering defines $\prec_A$ based on $\prec_{A_1}$ and $\prec_{A_2}$, which implicitly have similar (potentially recursive) definitions. The result is a family of ordering relations, one for each nonterminal A. Algorithm 1 provides a template for the corresponding ranking functions. However, the soundness of Definition 1 requires only that $\prec_{A_1}$ is a correct ordering for $MPT(G, A_1)$, it does not require that $\prec_{A_1}$ be defined precisely as we do. This means that, if some nonterminal B generates trees that are more efficient to compare and rank using a different method, then we can use that method instead to define $\prec_B$ and respectively to implement rank_B. As long as rank_B provides a correct ordering for $MPT(G, B)$, then the overall ranking of $MPT(G, S)$ is still correct. Specifically, if the language derived from B is regular, then ranking for regular languages can be used. Our library and code generation tool (presented in Sec. 4) uses this important optimization to allow ranking of grammars whose tokens are specified as regular expressions.

3.5 Beyond CFGs

This section shows how some language features that are not context-free can be potentially handled by our methodology. If a format uses arbitrary specified delimiters, length specifiers, or checksums, then the language it describes is not context free. Nevertheless, in many cases we can still use our approach to obtain a relaxed ranking scheme for the language, as long as the intermediate set can be ranked, and the user provides efficient implementations for map and gen.

We say that a language L is *cfg-parseable* if there is a CFG G with set I=$MPT(G, S)$, and two functions $\mathsf{map} : L \rightarrow I$ and $\mathsf{gen} : I \rightarrow L$ such that $\forall x \in L : \mathsf{gen}(\mathsf{map}(x)) = x$. Let's consider an example. Assume that the data consists of a sequence of items. Each item starts with a counter which represents the size of the item in bytes, followed by a sequence of bytes, and it ends with a checksum byte.

```
<Data> := <Item> | <Item> <Data>
<Item> := <Count> BYTE* CHECKSUM
<Count> := [0-9]+
```

The data described by this format is not a context free language, because of the counter and the checksum. However, the counter is used only for parsing, and both the counter and the checksum are completely defined by the data. If properly parsed, the content of the data is given by the following context free grammar G

```
<Data> := <Item> | <Item> <Data>
<Item> := BYTE | BYTE <Item>
```

Then, as in the case of ranking a CFG, we take the intermediate set $I = MPT(G, Data)$. We assume that the user has a deterministic parser, that maps each element in the initial language L to a tree in I. The reverse transformation *gen* maps trees from I back to L by inserting the appropriate checksums and the counters.

Note that, for technical reasons, when we build the memoization tables we may want to account for the length of the counter and checksum. In that case we can use the grammar

```
<Data> := <CItem> | <CItem> <Data>
<CItem> := CT <Item> CS
<Item> := BYTE | BYTE <Item>
```

Here CT and CS are just properly sized constant terminals, which are used as placeholders to be replaced with with the proper data by the function gen.

There is a large class of languages L that are not CFG, but which can be efficiently parsed to minimal parse-trees for some CFG G. Parsing discards or replaces those extra elements in the original language (such as counters or checksums) that prevent it from being context free, if such elements can be inferred from the rest of the tree (otherwise our scheme does not apply). The reverse transformation takes a minimal parse-tree and provides its yield annotated with exactly that additional information that is discarded by parsing (such as counters or checksums) and that can be inferred from the tree.

4. IMPLEMENTATION

We implemented a library in C++ and a tool for relaxed ranking with CFGs. The library offers a low-level interface to describe any CFG, and it offers relaxed ranking based on parse trees. To aid developers, our tool supports converting from more user-friendly descriptions of CFGs to the specifications handled by the library. It also supports LALR grammars. See Figure 3 for diagrams depicting the library and tool.

We note that, up to this point, we have for simplicity explained our algorithms assuming CFGs are in weak normal form (WNF), but in fact we have implemented algorithms that work for any CFG.

4.1 Library for ranking CFG parse-trees

Our library offers an API to build an internal representation of any grammar G. The API is low-level, meaning that the user must define a C++ ranker class, using our library, where all grammar components are individually encoded: terminals, non-terminals, and rules. Objects of the resulting class can rank/unrank minimal parse-trees of G, as shown in Figure 3. We used the GNU Multiple Precision Arithmetic Library (GMP [1]) to represent large integers.

As explained in Section 3.4, there may be parts of the grammar for which alternative ranking methods work better. Our library is object-oriented and offers a default implementation of Algorithm 1, which the user can change as they see fit.

One such particular case is so important that we introduced an alternative ranking for it. This is the case of lexical tokens. Consider the grammar G_D in Figure 4. G_D does not have a pure context free specification because the token ID is specified using regexes. The grammar can be turned to a pure CFG by replacing the definition of ID with

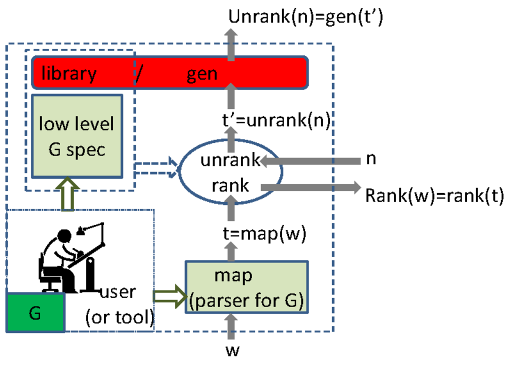
```
ID := 'a'|'b'|...|'Z'|'a'ID | ...|'Z'ID
```

Such a change is undesirable because: (1) people prefer using regexes for lexical tokens, and (2) ranking a regular language is usually much faster using a DFA or NFA method [2, 14] than using the CFG method.

As shown in Figure 3, the user is responsible for producing a deterministic parser. If the user provides a parser, then the ranker object (built using our library) performs relaxed ranking of words in $L(G)$.

4.2 Code Generation Tool

We provide a tool that takes a simple grammar specification and produces: (1) source code for a ranker object based on our API, and (2) a LEX/YACC-based parser for the grammar (provided that the grammar is LALR). The design goal of our tool was to specify a grammar in an intuitive way (such as Backus Normal Form, or LEX/YACC syntax as in Figure 4) and to effortlessly generate from it library code. Figure 3 gives a diagrammatic description of this process. The lower box in Figure 4 shows how to specify the grammar G_D discussed previously for input to our tool. The tool uses a LEX/YACC syntax for its rules. We made this decision because users are already familiar with LEX/YACC, and many grammars already have a LEX/Y-



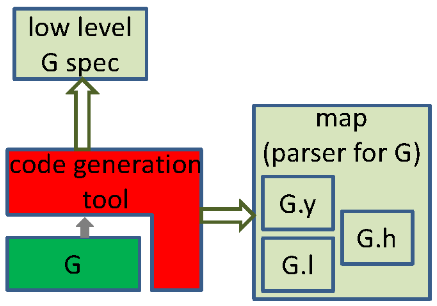


Figure 3: **(Left)** Our library provides an API to specify a CFG and produce an object that ranks/unranks its parse trees. The user must provide code for parsing and encode the low level grammar specification. **(Right)** We provide a code generation tool that uses a high level description of the grammar (in a YACC/LEX syntax) to generate the low level description of the grammar, and YACC/LEX based parsing code.

```
<D> :  INT ID '=' <E>
<E> :  ID | <E> '+' '(' <E> ')'
INT := "int"
ID := [a-zA-Z]([a-zA-Z])*
```

```
%% /*---------- YACC ----------*/
token ID INT
%%
D : INT ID '=' E
E : ID | E '+' '(' E ')'
%% /*---------- LEX -----------*/
"int"                      return INT;
[a-zA-Z]([a-zA-Z])*        return ID;
[=+()]                     return yytext[0];
```

Figure 4: **(Top)** CFG G_D using regex to specify token set ID. **(Bottom)** Specification for G_D as input for the tool.

ACC specification. This way, converting to our tool format is simply a matter of cut and paste.

The tool generates the code for the lexical tokens using an alternative ranking method (see Section 3.4) based on DFAs. It offers the option to specify additional constraints for the tokens; for instance restricting the identifiers sizes to be between 2 and 20 characters. The generated code can be used as is, or further customized by the user.

The tool is necessary because our library's flexibility and power comes from having a low-level interface which, for complex grammars, may be tedious and hard to use. For instance, a slightly modified version of the C99 grammar[1] has 350 lines of LEX/YACC rule definitions. Specifying such a grammar programmatically is error prone. Furthermore, matching it with a parser can also introduce errors. Another source of problems for hand writing code is the avoidance of lexical clashes, discussed in the following section.

4.3 Additional Challenges

Lexical clashes. In grammar G_D from Figure 4, `int` is a keyword and `ID` is an identifier whose name must not be a keyword. However the regular expression for `ID := [a-zA-Z]([a-zA-Z])*` also matches the string `int`. For LEX scanners this is not a problem, because of LEX disambiguating rules. But when we unrank a random number using `ID`'s regex it is possible that the result happens to be the string `int`, because all elements matched by the regex are possible. To stop such clashes between lexical tokens, we must change the regexes. In this case, we must use

$$\texttt{ID} \;:=\; \texttt{i} \mid \texttt{in} \mid \big(\texttt{[a-hj-zA-Z]} \mid \texttt{i[a-mo-zA-Z]} \mid \texttt{in[a-su-zA-Z]} \mid \texttt{int([a-zA-Z])}\big)\texttt{([a-zA-Z])*}\,,$$

which doesn't match `int`. Performing this rewriting by hand is error-prone, and for larger grammars it becomes infeasible. This is a case where using the tool is no longer a convenience, it is a necessity. Our tool automatically handles such clashes using the convention that the longest match wins, or in case of equal size, the first lexical token wins. Internally, it implements regular language difference by DFA complementation and composition, without requiring the user to change any regex.

Comments and white spaces. Many textual formats support use of white space and comments, yet these are removed during parsing and so many inputs (differing only in comments or whitespace) are actually the same element of the CFL. Our tool's current strategy is to ignore comments and white spaces in the grammar.

In more detail, we consider that the strings that have the same parse tree form an equivalence class. The library contains a gen function that produces the yields of parse trees. When white spaces are necessary to separate tokens (e.g., `int x` versus `intx`), the gen function inserts the required white spaces in a canonical way. Because of this, the identity $w = \mathsf{gen}(\mathsf{map}(w))$ may fail. To work around this issue, our ranking works on the canonical element from an equivalence class as follows. If we want to encrypt a string $w \in L$ we first compute $\bar{w} = \mathsf{gen}(\mathsf{map}(w)) \in L$. That is, we first parse w and then get the yield of its parse tree. We select $\bar{w}$ to be a canonical element that has the same parse tree as w, which satisfies $\bar{w} = \mathsf{gen}(\mathsf{map}(\bar{w}))$. Then we use $\bar{w}$ instead of w during encryption. The comments and white spaces of w are not retained through this process, but the semantic meaning of w is preserved by $\bar{w}$.

If the entire input is relevant, then the grammar must be transformed to account for comments or white spaces. Currently, we require the user to perform this grammar change, but it is possible to automate the process.

[1]C99 is an informal name for the 1999 C standard.

5. EXPERIMENTAL RESULTS

Our experiments were designed to answer the following questions about our library and tool:

1. How easy is it to produce the (un)ranking code?
2. How efficient is our method?

To answer the first question we report on our experience using the library and tool. While this is a limited experience, it offers a high-level comparison of different use cases of the library and the tool.

To answer the second question, consider the rank-and-encipher method in Section 2. This method is modular, and its overall performance is determined by: (1) the size of the intermediate set; (2) the performance of relaxed ranking and unranking; (3) the number of repetitions of the loop (i,e, the length of the cycle walk); and (4) the performance of the intermediate integer encryption step. The quality of relaxed ranking only influences the first three items, and so we focus our evaluation on microbenchmarking these items. Because efficient relaxed ranking requires memoization, we also report the memoization time and approximate space.

5.1 Our experience using the tool

We evaluated three usage scenarios: (1) the user writes a new grammar specification using the low level API; (2) the user writes a new grammar specification using our tool; and (3) the user converts an existing LEX/YACC parser to use our tool.

API – New Specification. We used our library's low-level API to manually specify the G_D grammar from Section 4.1. We also wrote a parser to be used by the ranker code. In total, it took about one author one hour; we used LEX/YACC for the parser. This time quickly increased with grammar size. Because of this, we did not attempt manual encoding of much larger grammars.

Tool – New Specification. Using our tool, the specification for the G_D grammar took about 5 minutes and the code was generated in under 1 second. For a new grammar, using our tool is simpler than specifying the grammar using LEX/YACC, because only the rules need to be defined (without the supporting code and data structures involved in real parsers).

Tool – Existing Specification. We downloaded a specification for the C99 language from [5]. We chose C99 because it is a well known example of a complex CFL. The YACC rules did not need any modification. The LEX rules contained embedded code, and it took about 10 minutes to remove it. In the end, we obtained a 350 line specification that contained 68 non-terminals, 63 lexical tokens (defined using regular expressions), 24 constant tokens and 236 rules. Our tool turned this 350 line specification (no empty lines or comments) into a specification consisting of 2,769 lines of C code. This took about 18 seconds. The majority of time was spent checking for lexical token clashes. After the initial run, it took about 30 minutes to debug the initial failures caused by the fact that the regular expressions understood by our library use a 256-byte character set, and string literals and comments expanded to strings that caused scanning problems. After those errors were removed, ranking/unranking and parsing worked correctly.

5.2 Performance

We evaluate relaxed ranking with CFGs by measuring four values: (1) memoization time, (2) memoization space, (3) ranking time, and (4) unranking time. We also report on grammar ambiguity and language size.

5.2.1 *Methodology*

We parameterize our results along two dimensions: grammar and language slice size. If L is a language and $n \in \mathbb{N}$, then L's slice of size n is the set of strings of length n in L, i.e., $L^{(n)} = \{w \in L : |w| = n\}$. Slice size determines (among other things) the dimension of the memoization tables.

Grammars. We explore three grammars:

- C99 is the C grammar mentioned in Section 5.1.
- C99$_2$ is a grammar with the same syntax as C99, but $C99_2$ has identifier and constant tokens of at most two characters long.
- G_D is the grammar defined in Figure 4, Section 4.1.

To explain why we investigated C99$_2$, note that in practice C programs contain reasonably sized identifiers. But this does not necessarily hold for a random program obtained by unranking an arbitrary number. For instance, when we inspected the output of unranking random value generated for a 4000-byte slice, we observed identifiers as long as 68 characters. We used C99$_2$ to provide experience when avoiding such large tokens.

We used slices between 1000 and 4000 bytes (no comments or white spaces counted, in the case of C99 and C99$_2$), at 100 byte intervals.

Tests. For each grammar G (with start symbol S) and slice size n we run the following experiment. We performed relaxed-ranking of $L(G)^{(n)}$ using the intermediate set $I = MPT(G, S, n)$, the set of minimal parse trees T with $\|T\| = n$.

In the first stage, we evaluate the memoization and compute the size of the intermediate set using $N = |I| = N_S(n)$; N_S is given by Algorithm 3 (when $A = S$). When $N_S(n)$ is called for the first time, it has the side effect of filling the memoization tables. We report $|I|$, as well as the total time and space required by memoization. The space is estimated by reading the `data` size from the `/proc` interface, before and after memoization, because it is hard to measure the exact space used by the large integers in the GMP library.

In the second stage, we evaluate ranking and unranking, and we estimate the ambiguity-factor. We repeat ranking and unranking $C = 100$ times. For each $1 \le k \le C$ we let $r = (N-1) \times \lfloor k/C \rfloor$ and compute $w = \mathsf{Unrank}(r)$ which is a string in $L(G)^{(n)}$. Then we compute $r' = \mathsf{Rank}(w)$. If $r \neq r'$ then $r \notin Img(L(G)^{(n)})$ and we say that r is an *outsider*, and w is an ambiguous string. We report the average time for a single ranking and unranking operation, and the number U of outsiders.

It is easy to see that for a uniform distribution as C grows towards N, we can approximate the ambiguity-factor $\beta = N/|L|$ as $\beta \simeq C/(C-U)$.

5.2.2 *Microbenchmarking Results*

We ran all the experiments on a laptop with 32GB RAM and a i7-2720QM CPU with 4 cores running at 2.2GHz. Our implementation is single-threaded, so the number of cores does not affect the results.

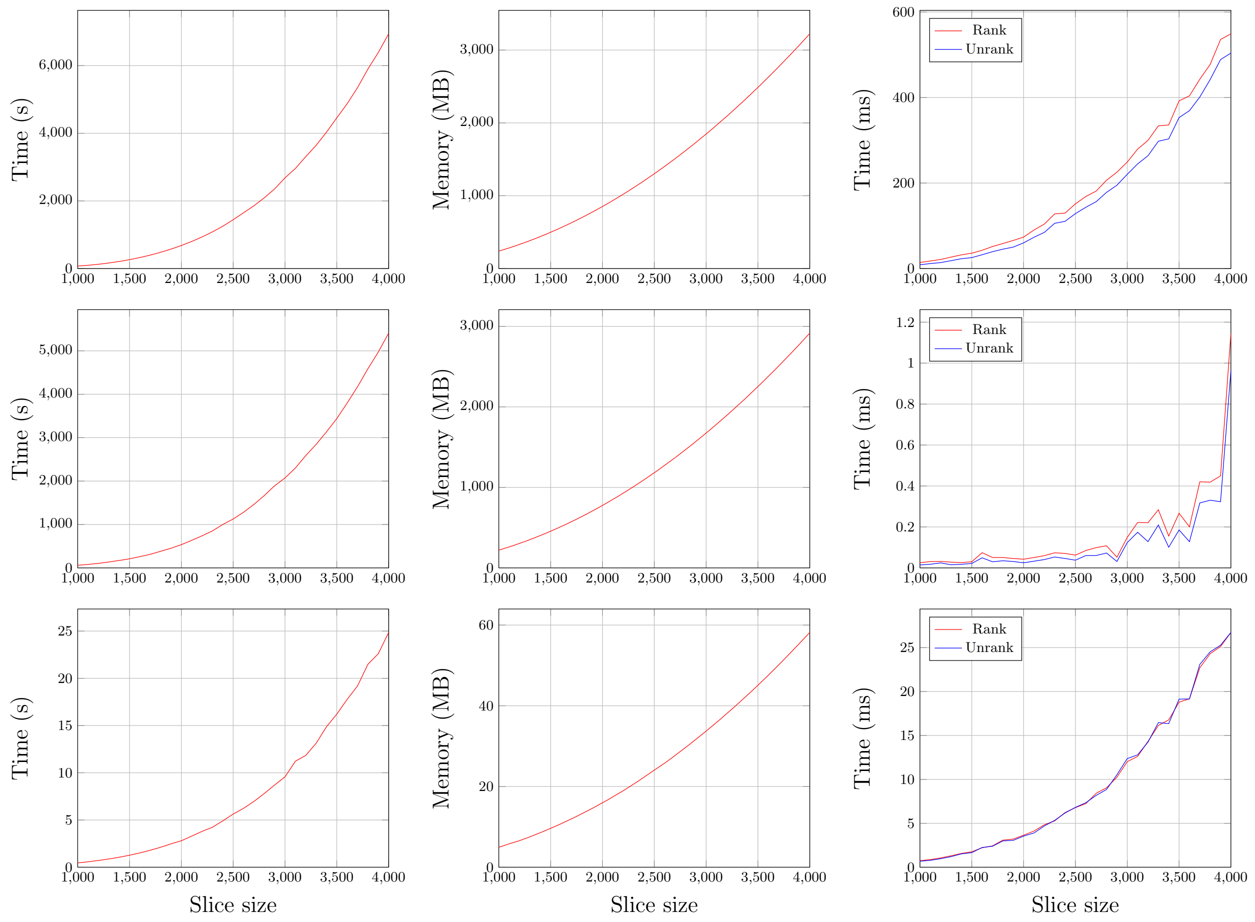


Figure 5: Results for the **(top row)** C99 grammar, **(middle row)** C99$_2$ grammar, and **(bottom row)** GD grammar using slices $n = 1000, 1100, \ldots, 4000$. **(Left column)** The memoization time is the time to execute $N_S(n)$ (using N_S from Algorithm 3, when $A = S$) for the first time. **(Middle column)** The memoization space is estimated using the `/proc` interface. **(Right column)** Ranking and unranking was performed as in Algorithm 1 and Algorithm 2.

The results are shown in Figure 5, where we report on initialization time (first column), memoization space utilization (second), and ranking and unranking times (last column). Note that the ranking/unranking times are measured in milliseconds, and these operations are fast. We also computed the size of the intermediate sets exactly, but we noticed that they can be approximated as $|I| \simeq 10^{1.87\times n}$ for C99, $|I| \simeq 10^{1.67\times n}$ for C99$_2$, and $|I| \simeq 10^{0.83\times n}$ for GD. Thus in all cases the size of the intermediate set I was exponential in the slice size n.

5.2.3 Ambiguity

Although C99 is ambiguous, our experiments detected no outsiders even after we increased the repeat count C from 100 to 10000 trials per slice. The reason is that the density of ambiguous strings is very low.

To better understand this, consider the smaller, but still ambiguous grammar G_A in Figure 6. The lexical token `ID` represents an identifier, and the token `NUM` represents an integer constant; both have arbitrary length. G_A allows a classic example of ambiguity: the statement `if(a)if(b)c;else d;` can be parsed in two ways: `if(a){if(b)c;else d;}`, or `if(a){if(b)c;}else d;`.

```
StatementList:  S | S StatementList
S : AS | IF '('E')' S | IF '('E')' S ELSE S | '{'S'}'
AS : E ';' | ID '=' E ';'
E : ID | NUM
```

Figure 6: An example of an ambiguous grammar G_A

We purposely crafted grammar G_A to be ambiguous and to have few kinds of unambiguous statements. Yet, we detected no outsiders for a slice size of 1000 in $C = 10,000$ trials. One intuitive explanation is that an ambiguous string requires an `if..if..else` construct. The keywords use $2 + 2 + 4 = 8$ bytes out of the 1000 bytes slice size. We could remove the keywords and fill those 8 bytes with identifiers or constants in many ways. Thus for a single ambiguous word we can get potentially many more unambiguous ones, hence a low density of ambiguous words. To test this explanation, we reduced the number of possible lexical tokens, and we changed the regexes of `ID` and `NUM` to match exactly one value each. This should increase the density of ambiguous strings. Indeed, in this case we counted $U = 9$ outsiders in $C = 100$ trials, and estimated $\beta \simeq C/(C - U) \simeq 1.1$. We repeated the experiment by enforcing various bounds for the number of elements that `ID` and `NUM` can match. Table 1

Experiment Parameters				
Limit on # of tokens		Trials	Measured	Inferred
$\|L(\texttt{ID})\|$	$\|L(\texttt{NUM})\|$	C	U	β
1	1	10^2	9	1.1
1	1	10^4	1257	1.14
2	2	10^4	111	1.01
3	2	10^4	33	1.003
3	2	10^5	422	1.004
4	2	10^4	20	1.002
4	3	10^4	14	1.001
10	10	10^5	2	1
15	10	10^5	0	1

Table 1: Counting the number of outsiders for G_A. C is the number of attempts to find an outsider, and U is the number of outsiders found, i.e. $r \neq \mathsf{Rank}(\mathsf{Unrank}(r))$. The notation $|L(\texttt{X})| = 2$ means that token `X` may only take one of two values, say `x1` or `x2`. The ambiguity-factor β is estimated as $\beta \simeq C/(C - U)$

shows the results. It backs the hypothesis that, at least in this case, the more values lexical tokens can take, the lower it is the percentage of ambiguous strings (which require fixed keywords).

Discussion. The experiments indicate that our method for relaxed ranking based on CFGs is both usable and efficient. Once the memoization is performed, ranking and unranking are fast, even for complex languages such as C99, and run in under one second even for slices as large as 4,000 bytes. While theoretically an impediment, in practice ambiguity is not a a problem for commonly used grammars. Even when we used a highly ambiguous grammar, we could not experimentally detect any outsiders unless we artificially bounded the number of values that lexical tokens could take on to be very small numbers. Even in the extreme case when we allowed only one value for each token, the measured ambiguity-factor was less than 2. This means, for instance, that the expected number of cycle walks for FPE is at most 2 in the scheme from Section 2. Thus encryption and decryption will be fast.

Acknowledgements

We thank the reviewers for their helpful comments. This work was supported in part by the US National Science Foundation (NSF) grants CNS-1228782, CNS-1228620, CNS-1064944, CNS-1330308, CNS-1065134, CNS-1253870, CNS-0845610 and CNS-1319061.

6. REFERENCES

[1] The gnu multiple precision arithmetic library. http://gmplib.org/.

[2] M. Bellare, T. Ristenpart, P. Rogaway, and T. Stegers. Format-preserving encryption. In *Selected Areas in Cryptography*, pages 295–312. Springer-Verlag, 2009.

[3] J. Black and P. Rogaway. Ciphers with arbitrary finite domains. In *Topics in Cryptology–CT-RSA 2002*, pages 114–130. Springer Berlin Heidelberg, 2002.

[4] M. Brightwell and H. Smith. Using datatype-preserving encryption to enhance data warehouse security. In *20th National Information Systems Security Conference Proceedings (NISSC)*, pages 141–149, 1997.

[5] Ansi c99 grammar yacc specification. http://www.quut.com/c/ANSI-C-grammar-y-1999.html.

[6] R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[7] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Protocol misidentification made easy with format-transforming encryption. In *Proceedings of the 20th ACM Conference on Computer and Communications Secuirty (CCS 2013)*, November 2013.

[8] A. Goldberg and M. Sipser. Compression and ranking. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, STOC '85, pages 440–448, New York, NY, USA, 1985. ACM.

[9] M. Holzer and M. Kutrib. Descriptional complexity of (un)ambiguous finite state machines and pushdown automata. In *Proceedings of the 4th international conference on Reachability problems*, RP'10, pages 1–23, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] O. H. Ibarra and B. Ravikumar. On sparseness, ambiguity and other decision problems for acceptors and transducers. In *3rd annual symposium on theoretical aspects of computer science on STACS 86*, pages 171–179, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[11] S. Kannan, Z. Sweedyk, and S. Mahaney. Counting and random generation of strings in regular languages. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, SODA '95, pages 551–557, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[12] L. Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1):1–15, Jan. 2002.

[13] J. Levine, T. Mason, and D. Brown. *Lex & Yacc, 2Nd Edition.* O'Reilly, second edition, 1992.

[14] D. Luchaup, K. P. Dyer, S. Jha, T. Ristenpart, and T. Shrimpton. Libfte: A user-friendly toolkit for constructing practical format-abiding encryption schemes. In *Proceedings of the 14th conference on USENIX Security Symposium*, 2014.

[15] E. Mäkinen. Ranking and unranking left szilard languages. Technical report, ISO/IEC JTC1/SC29/WGll/N2467, Atlantic City, 1997.

[16] A. Nijenhuis and H. S. Wilf. *Combinatorial Algorithms.* New York : Academic Press, 1975.

[17] B. Ravikumar and O. H. Ibarra. Relating the type of ambiguity of finite automata to the succinctness of their representation. *SIAM J. Comput.*, 18(6):1263–1282, Dec. 1989.

[18] R. Schroeppel and H. Orman. The hasty pudding cipher. *AES candidate submitted to NIST*, page M1, 1998.

[19] M. Sipser. *Introduction to the Theory of Computation.* Cengage Learning, 2012.