Formal Language Theory for Practical Security – Extended Abstract –

Andreas Jakoby, Jannis Leuther, Stefan Lucks; Bauhaus-Universität Weimar
firstname.lastname[at]uni-weimar.de

***Abstract*—When binary data are sent from one party to another one, the encoding of the data can be described as a "data serialisation" language (DaSeL). Many DaSeLs employ the "length-prefix" pattern for strings, containers and other data items of variable length. This consists of an encoding of the item's length, followed by an encoding of the item itself – without closing brackets or "end" symbols. The receiver must determine the final byte from the length read before. Length-prefix languages are not context-free. Thus, the plethora of tools and methods to specify, analyse, and parse context-free languages appears to be useless for length-prefix languages. This seems to explain why improper specifications of length-prefix languages and buggy hand-written parsers are so often a root cause for security issues and exploits, as, e.g., in the case of the famous Heartbleed bug. One might even be tempted to consider the use of length-prefix languages a security hazard.**

But this consideration would be wrong. We present a transformation of words from "calc-context-free" languages (a superset of context-free and length-prefix languages) into words from proper context-free languages. The transformation actually allows to use tools from context-free languages to deal with length-prefix languages.

Our transformation runs on a Turing machine with logarithmic space. This implies the theoretical result of calc-context-free languages being in the complexity class log$\mathcal{CFL}$. Similarly, deterministic calc-context-free languages are in log$\mathcal{DCFL}$. To run in linear time, one needs to enhance the Turing machine by a stack to store additional data.

I. Introduction

Whenever entities communicate, they use a "language". Formally, a "language" is a set of "strings" over a finite "alphabet". Based on the hierarchy of languages developed by the linguist Noam Chomsky [3], the collaboration of theoreticians and practical compiler writers did shape formal language theory: Today, the computer science and software engineering community understands well how to specify artificial languages, and how to efficiently parse them. Mostly, the focus is on context-free languages, carefully avoiding ambiguity and context sensitivity, and even on subsets which can be parsed with minimal lookahead [9], [8].

It *might seem* that all practical problems for formal language theory have been solved long ago. But actually, important languages have been overlooked, such as "length-prefix" languages, where a variable-length data field is prefixed by a length field. On one hand, such languages are ill-readable for humans. Thus, people didn't feel the urge to study these languages. On the other hand, the length-prefix approach is common for data serialisation languages [6, appendix].

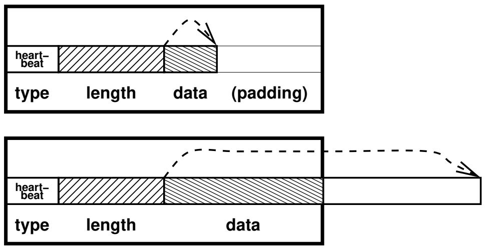

Fig. 1. **Top:** Abstract view at a valid heartbeat package, with its three fields: type ("heartbeat"), length, and data. The outer package holds the heartbeat package and, optionally, some "padding" bytes. The receiver responds the content of the data field.
Bottom: A malicious (and invalid) heartbeat package, claiming more space for the data field than available. A proper implementation must detect that the package is invalid.

Imprecise specifications and buggy hand-written ad-hoc parsers are a root cause for many security issues and practical exploits, such as the famous Heartbleed bug [4], see Figure 1.

Length-prefix languages can be nested, i.e., the variable-length data item can be a container, holding other items in length-prefix encoding. This is somewhat similar to context-free languages, which also support nesting. However, length-prefix languages – and even length-prefix languages without nesting – have been proven not to be context-free, see [6]. In the same paper, the notion of "calc-regular languages" has been introduced, a superset of length-prefix and regular languages.

a) Parsing serialised data: "Textbook parsing", as in formal language theory and compiler design, assumes that the parser is given a word w over a finite alphabet. The parser accepts if w is in the specified language and rejects otherwise. In the case of acceptance, the parser generates a data structure, such as a "parse tree", for further processing. In any case, the parser knows the end of its input. *A parser for a data serialisation language (DaSeL) L* is an online-algorithm, reading from a stream of characters (typically bytes), until it accepts all the characters read so far as a word $w \in L$ – or raises an error if the characters read so far are not the prefix of any word in L. Similarly to textbook parsers, the DaSeL-parser also generates a data structure for further processing. As an online-algorithm, a DaSeL-parser must never read beyond the last character of w. *Thus, every length-prefix DaSeL L is prefix-free,*[1] *and the*

[1]Parsing a DaSeL requires to know when to accept a word w, without reading beyond the last character of w. Unlike languages from textbook parsing, a DaSeL doesn't provide a final "end of word" symbol at the. Thus, no word w' in a DaSeL L can be a prefix of $w \in L$. I.e., if $w, w' \in L$ then there is no nonempty w'' with $w = w'w''$.

empty word is not in L.

b) Recognising the length-prefix: Every word $w \in L$ of a length-prefix language consists of a header h, encoding some length $\ell(h)$, which is followed by the content c of length $\ell(h)$. If we just assume $w = hc$, how will the parser determine the end of h and the beginning of c? We thus write $w = h\boxed{\#}c$ and formally assume a separator "$\boxed{\#}$", which never occurs in h. Some practical length-prefix languages actually use such a separator, e.g., both Bencode [2, chapter 6] and netstrings [1] encode h as a sequence of decimal digits, followed by a colon (":"), followed by a string of characters of the given length.[2] Other length-prefix languages, such as Google protocol buffers [5][3] and the ASN.1 encoding rules [11][4] don't have an explicit separator. Nevertheless for either language it is easy to read a word from each language byte by byte and to determine the last byte of h and the length of c, without reading any byte from c. Thus, even though we formally assume a unique separator "$\boxed{\#}$", our results also apply to languages without such a separator, such as protocol buffers and ASN.1 encoding rules.[5]

c) Contributions: In the current paper, we introduce the notion of "calc-context-free" languages, a superset of calc-regular and context-free languages. We argue that most practical data serialisation languages are calc-context-free. Our main result is a transformation from general length-prefix languages, see Figure 2, into context-free-languages. We present three variants of this transformation:

1) The combination of Algorithm 1 (for the main part of the transformation) and Algorithm 2 (to deal with the special case of nested containers) runs on Turing machines with logarithmic space. Hence, we can assume a two way input tape, i.e. data from the input tape can be read multiple times. Eventually, a valid input is accepted, an invalid input rejected.
2) Algorithm 3 runs on the same Turing machine, except that it requires an additional stack and is restricted to reading the data from an input stream (formally, from a one-way input tape). The algorithm eventually accepts or rejects the entire input.
3) Algorithm 4 has been written to support a streaming mode of operation. It uses a stack, reads data from an input stream and writes parts of the data to an output stream as soon as possible. I.e., it does not need to wait to have read the entire data from the input before writing the output. Hence, Algorithm 4 transforms a calc-context-free language from the input stream into a context-free language on the output stream.

Our complexity-theoretical approach assumes a Turing machine with logarithmic space. If the resulting context-free language is deterministic, then so is the length-prefix language, which is the source language for the transformation. This puts length-prefix languages into the complexity class log$\mathcal{CFL}$, and deterministic length-prefix languages into the complexity class log$\mathcal{DCFL}$.[6]

$$\boxed{L}\quad n\quad \boxed{\#}\boxed{S}\ \overbrace{\text{xxxxx}\ldots\text{x}}^{n\text{-character string}} \qquad (1)$$

$$\boxed{L}\quad n\quad \boxed{\#}\boxed{C}\overbrace{\boxed{\text{xx}}\boxed{\text{x}}\boxed{\text{xxx}}\ldots\boxed{\text{xx}}}^{n\text{ data items}} \qquad (2)$$

Fig. 2. (1): Abstract representation of a word xxxxx...x of length n in length-prefix notation, such as, e.g., for strings of variable length. (2): For containers, an alternative to length-prefix notation, which we also consider a pattern for "length-prefix" language is "count-prefix notation". Here, the number n represents the number of items stored in the container, rather than the length of the container in characters. Note that the items (here xx, x, ...) in the container are also words from a DaSeL, an thus must be parsed recursively.
Note that the symbols "$\boxed{\#}$", "$\boxed{L}$", "$\boxed{S}$", and "$\boxed{C}$" only serve as a convenient notation for us. Specifically, we don't require the length-prefix language to support any such characters. Rather we assume a generic length-prefix language, which can represent any practical length-prefix language, (Bencode, netstrings, protocol buffers, ASN.1 encoding rules, ...). "$\boxed{L}$" formally represents the beginning of a number, "$\boxed{\#}$" is the end. Also, if the language allows to mix length-prefix and count-prefix notation, there must be a mechanism to distinguish between these two cases, and we use "$\boxed{S}$" and "$\boxed{C}$" as the notation to make that distinction.

d) Why does this matter for security?: Whenever a message from a potentially corrupted channel arrives, the receiver must parse the message, optionally check its authenticity/integrity, and then react to the message – or decide to ignore it. For an attacker, a malicious message (or a malicious sequence of messages) is tantamount to a program running on a virtual machine, and the receiver provides the virtual machine for the attacker. As Sassaman et al. put it [10]:

> "The exploit is really a *program* that executes on a collection of the targets computational artefacts, including bugs [...]."

Thus, the design of input languages (including data serialisation languages) is crucial for security. If parsing requires a powerful

[2]Both Bencode strings and netstrings are of the form "h:c" and "h:c,", respectively. The only difference is the final comma in netstrings, which is redundant for parsing and, presumably, is about human readability. The prefix h is a decimal number, and c is a string of exactly h arbitrary characters. Any of the characters in c can be a comma, a colon, a decimal digit, or any other character.

[3]Google protocol buffer length fields consist of a sequence of bytes. The seven least significant bits of each byte specify a "digit" in $\{0, \ldots, 127\}$. The whole sequence of these "digits" is an integer in base-128 representation. This leaves the most significant bit of each byte as a flag for the final byte of h.

[4]The ASN.1 encoding rules' length fields [11] optionally support a meta-length-field, i.e., a single byte, b. The full length field holds another b bytes, which are the encoding of an integer in base-256 representation: the length of the following data field.

[5]Assume a machine M to decide which symbol is the final symbol of h, without reading beyond this symbol. Consider a length-prefix language L over a finite alphabet A with two special symbols $\boxed{\#}, \_ \notin A$, with "_" indicating "ignore". Transform each symbol $s \in A$ into a pair of symbols $(s, b) \in A \times \{\boxed{\#}, \_\}$, where $b = \boxed{\#}$ if s is h's final symbol, and $b = \_$ otherwise. I.e., the value of b is determined by M. Double the value of the encoded length $\ell(h)$. This is a transformation of the length-prefix language L over the alphabet A into a new length-prefix language $L_{\boxed{\#},\_}$ over the alphabet $A \cup \{\boxed{\#}, \_\}$. $L_{\boxed{\#},\_}$ uses $\boxed{\#}$ as an explicit separator.

[6]Formally, log$\mathcal{CFL}$ is the class of decision problems reducible on a deterministic Turing machine with logarithmic space to the problem of deciding membership in a context-free language. The same for log$\mathcal{DCFL}$ and deciding membership in a deterministic context-free language. Since every deterministic context-free language is a context-free language, log$\mathcal{DCFL}$ is a subclass of log$\mathcal{CFL}$. Furthermore, log$\mathcal{CFL}$ is equivalent to the class of decision problems solvable by a uniform family of $\mathcal{AC}^1$ circuits, with all AND-gates of constant fan-in [12], see also [7, p. 137].

machine, such as a Turing machine with linear space (for context-sensitive languages) or even an unconstrained Turing machine (for recursively enumerable languages), the attack surface is huge. By using a less powerful machine, such as a deterministic pushdown automaton for deterministic context-free languages, one can reduce the attack surface greatly. As length-prefix languages are not context-free [6], *are they difficult to parse or even a security hazard?* Our main result provides negative answers to the two above questions: in comparison to context-free languages, one may need an additional wrapper Turing machine of very small (i.e., logarithmic) space.

Another point to consider is the usage of tools: There are well-established techniques to specify context-free languages, to verify that these languages are unambiguous or even deterministic[7]. Additionally, there are "parser generators" which automatically derive the parser from the language specification, thus avoiding the hazards of handwritten parsers. Our main result gives a direction for the development of similar tools for length-prefix languages – not necessarily from scratch, but by adapting existing tools for context-free languages.

e) String-free subsets: Many data serialisation languages (DaSeLs) employ length-prefix notation for strings, cf. [6, Appendix]. Sometimes, the string-free subset of a DaSeL is context-free. E.g., Bencode containers use some form of brackets (characters "l" or "d" as the opening "bracket" for a list or a dictionary and "e" as the closing "bracket"), without length-prefixes. Thus, the string-free subset of Bencode seems to be context-free. Bencode supports unlimited nesting of, e.g., lists within lists. Thus, its string-free subset is not regular. [8]

For some other DaSeLs, even their string-free subset is not context-free. E.g., the ASN.1 encoding rules [11][9] support length-prefix notation for strings, but also for containers and nested data structures (containers in containers ...). I.e., the string-free subset of ASN.1 is still a length-prefix language and thus not context-free [6, Theorem 2 and Remark 3]. Google protocol buffers [5] use length-prefix notation for strings, and "count-prefix" notation for containers. I.e., the prefix of a container doesn't hold the length of the container in bytes, but the number of items stored in that container. If the string-free subset of protocol buffers were context-free, the subset of the subset where all items in a container are of constant size would also be context-free. But for this subset of the subset the count-prefix can be transformed into a length-prefix and the result from [6] applies. Thus, the string-free subset of protocol buffers is not context-free.

f) Organisation: Section II enhances the well-known "Extended Backus Naur Form" (EBNF) to cope with length-prefix languages. The original EBNF has been designed to deal with context-free languages. Section III presents a formal approach to analyse length-prefix languages, including Algorithm 3 on page 7. This shows that typical length-prefix languages are in the complexity class log$\mathcal{DCFL}$ – the complexity class of deterministic context-free languages. Section IV concludes.

II. Calc-EBNF to Specify Length-Prefix Languages

As discussed in the introduction, length-prefix languages are widespread among applications such as data serialisation, where the length of an upcoming message is known before their transmission. A distinct and comprehensible specification is needed to implement these languages securely and to avoid unwanted behaviour and bugs. There are already well established techniques for specifying context-free languages (such as the Extended Backus-Naur Form [13], [14]) which can be extended with additional tools to fit our new class of calc-context-free languages. The following represents an enhanced EBNF, which we call calc-EBNF, designed specifically for calc-context-free languages. As an example, consider a type-length-value representation of messages, which can hold ℓ-byte strings, or ℓ-byte containers holding several messages of their own:

```
message
  := [t←]type  [ℓ←]length  content

content[cp+ℓ< stack-val ∧ t = "string"]
  := byte^ℓ (byte)

content[cp+ℓ< stack-val ∧ t = "container"]
  := [push(cp+ℓ)]  message^ℓ  (byte)  [pop()]
```

Let T be a non-terminal, x a global variable and $f : \Sigma^* \to \mathbb{N} \cup \{\texttt{string}, \texttt{container}\} \cup \perp$ a function where Σ is the set of terminal symbols. Function f can recognise if its input represents either a type field, a length field or neither of both which renders it invalid ($\perp$). The notation $[x \leftarrow]T$ describes that, if they are valid, the contents of T are interpreted by f and stored in the global variable x for future usage. Should $f(T)$ detect that T marks a length field, the contents of this length field are translated into a length $n \in \mathbb{N}$ and stored in x. Vice versa, if $f(T)$ detects a type field, the contents of T are either marking a `string` or a `container`. The corresponding interpretation is extracted and stored in x.

Furthermore, let $b \in \{\text{true, false}\}$ be a Boolean formula which has access to the global variables defined earlier using f. $T[b]$ on the left side of productions denotes that this production rule will only be used if b evaluates to true. The Boolean formula b uses the variables `cp` as well as `stack-val`. Variable `cp` describes the current position of the parser head while parsing, starting at 0 and incrementing by 1 for each terminal symbol (`byte`) read. Additionally, T^m or σ^m with $\sigma \in \Sigma$ refers to a concatenation of T or σ respectively where their total length measured as the *amount of terminal symbols* is m.

The operations `[push(`y`)]` and `[pop()]` interact with the so called container-stack. This stack is necessary to determine the depth of variably nested languages. It is comprised of positive

[7]An ambiguous language specification (i.e., grammar) of an input language is a security hazard: an input string may have more than one meaning. E.g., without common mathematical conventions, the expression "$a + b * c$" could be parsed as either "$a + (b * c)$", or "$(a + b) * c$". Deterministic context-free languages are unambiguous.

[8] One of the reviewers was *"interested to know of practical cases of message formats that really use the full power of calc-context-free, rather than those that are either context-free or TLV [type-length-value, i.e., in length-prefix form] only."* Bencode is such a practical case.

[9]These have been the habitat for Heartbleed, cf. Figure 1.

integers in $\mathbb{N}$, each item describing the size of a container. The topmost value of the stack (`stack-val`) always represents the size of the innermost container in the current language structure. By pushing for each new container, popping for each finished container and using the `stack-val` (for the Boolean formulas in rules 2 and 3), the container-stack is necessary to determine if a message (string or container) inside a container would exceed this container's size limits. Therefore, the parser can detect bad messages before processing any message content.

How all these definitions work together can be demonstrated by rule 2 of the calc-EBNF. The left side is defined as `content[`b`]`, where b = `cp`+ℓ < `stack-val` $\wedge$ `t` = `"string"`. This rule will only be used if b evaluates to true. Prior, variables `t` (type) and ℓ (length) have been extracted by f as shown earlier. Additionally, `cp` denotes the current position of the parser head and `stack-val`, as the top-most value of the stack, is the size of the container the current string is contained in. Here, b is true when

1) the sum of `cp` and the stored length ℓ is smaller than `stack-val` **AND**
2) the extracted type `t` is equal to a `string`.

There are many ways to encode a self-delimiting length field, either with an explicit separator symbol between length field and data field (e.g., ":" in Bencode and netstrings, cf. footnote 2), or without (e.g., in protocol buffers and in BER in ASN.1, cf. footnotes 3 and 4). Calc-EBNF can deal with any such approach. For explicitness, we will use the explicit separator symbol '$\boxed{\#}$' below, but our results apply to any such encoding.

There are also multiple ways to encode the number in the length field (decimal, as in Bencode, radix-127, as in protocol buffers, radix-256, as in BER, ...). For us, this does not matter – we just assume some "oracle", which takes a given string, which is a valid encoding of a number, and returns the number.

The optional terminal (`byte`) in Rules 2 and 3 represents and end-of-content symbol which may be added to increase human readability (like the comma in netstrings). It can not be used to determine the end of the string, of course, as this shall only be done by considering the length prefix value.

This definition of a calc-EBNF is far from being the end product as there are various things to improve upon. For example, the usage of global variables and a global stack that has to be manipulated can be a heavy burden for practical implementations. Additionally, when looking at methods employed by current practical data serialisation languages, Googles protobuf employs an external schema file, which contains information that determines if a (nested) object is either a normal `string` or a `container`. This schema file is therefore closely tied with the type information extracted by the parser and could not be handled by the current specification.

It is worth mentioning the similarity of the calc-EBNF parsing procedure to LL(1) parsing for deterministic context-free languages. All calc-context-free DaSeLs we are aware of can be parsed with a lookahead of 1 and using top-down parsing with the LL technique: parsing **L**eft to right and performing a **L**eftmost derivation of the input.

III. Length-Prefix Language are in log$\mathcal{DCFL}$

Within this section we would like to show that any properly formed sequence (or string) w of a length-prefix language can be transformed into a simple concatenation $w_1 \diamond \cdots \diamond w_k$ of the included non-nested substrings w_i of w. We will present two space-efficient algorithms which reject an input if it is not properly formed and output the required string otherwise. When dealing with a length-prefix language, our first algorithm requires only logarithmic space:

Theorem 1: Assume that the non-nested substrings of a length-prefix language are taken from a deterministic context-free language, then the corresponding length-prefix language is in log$\mathcal{DCFL}$. If the underlying language is context-free, then the corresponding length-prefix language is in log$\mathcal{CFL}$.

Note that the concatenation of strings from a deterministic context-free language or from a context-free language (if the separator "$\diamond$" is not in the original alphabet) does not extend the original language class. If the underlying language is regular, then it is even possible to verify whether particular substrings w_i fulfil the language requirements of the underlying language in constant space instead of generating the output string $w_1 \diamond \cdots \diamond w_k$:

Theorem 2: Assume that the non-nested substrings of a length-prefix language are taken from a regular language, then the corresponding length-prefix language is in $\mathcal{L}$.

For the following algorithms we always assume that if we reach a scenario where we have to determine a header, but no valid header exists (e.g. the header will be empty), the algorithms will reject the inputs.

To analyse space complexity of Algorithm 1 we assume that it never accesses the output r. Therefore we do not have to take the space for r into account to determine the space requirements of this algorithm. If we now consider the values stored within the variables p_0, p_1, p_2, q_0, q_1, q_2, v_1, v_2, u_1, and u_2 we can see that the values of q_0, q_1, q_2, u_1, and u_2 can be computed analogously to the values of p_0, p_1, p_2, v_1, and v_2. Therefore it is sufficient to analyse the required space for n, p_0, p_1, p_2, v_1, and v_2 as well as for evaluating the conditions of the if-statement in Line 9. Note that the values n, p_0, p_1, and p_2 can be stored within space $\mathcal{O}(\log \mathcal{N})$. Now assume that p_1 can be determined in space s_1, p_2 can be determined in space s_2, v_1 can be determined in space s_3, v_2 can be determined in space s_4, and the conditions of the if-statement in Line 9 can be evaluated in space s_5. Then Algorithm 1 uses space $\mathcal{O}(\max\{s_1, \ldots, s_5, \log \mathcal{N}\})$.

To analyse the function computed by Algorithm 1 we define the following recursively defined language $L_{\text{count}}(\ell, d)$:

Definition 3: Given two functions $\ell : \Sigma^* \to \mathbb{N}$ and $d : \Sigma^* \to \{\boxed{S}, \boxed{L}, \boxed{C}\}$ then we define $L_{\text{count}}(\ell, d)$ recursively as follows: Given two strings $c, h \in \Sigma^*$ then

- in the case that $d(h) = \boxed{S}$: '$h\boxed{\#}c$' $\in L_{\text{count}}(\ell, d)$ iff $|h| \in \mathcal{O}(\log \ell(h))$ and $\ell(h) = |c|$,
- in the case that $d(h) = \boxed{L}$: '$h\boxed{\#}c$' $\in L_{\text{count}}(\ell, d)$ iff there exists a partition of $c = w_1 \ldots w_k$ into substrings

Algorithm 1: Space Bounded Transducer with Header-Content Check

```
/* Standard algorithm for detection of languages with fixed length. Token $\boxed{\#}$ denotes the separator between length field $\boxed{L}$
   and content. Content is either a simple string $\boxed{S}$ or a container $\boxed{C}$.                                       */
Input: string $w \in (\Sigma \cup \{\boxed{\#}\})^*$
Output: string $w_0 \diamond w_1 \diamond \ldots \diamond w_k$ (as a stream) where $w_i$ are the content strings over $\Sigma^*$ and $\diamond \notin \Sigma$
1  determine the length $n = |w|$ of $w$
2  initialise $p_0 = 0$ and $r = \lambda$                                   // $\lambda$ denotes the empty string
3  repeat
4      let $p_1$ denote the end of the header starting at $p_0$
5      determine the header value $v_1$ of $w[p_0..p_1]$
6      let $p_2$ denote the end of the content belonging to the header starting at $p_0$
7      determine the content value $v_2$ of $w[p_1+2..p_2]$                 // $w[p_1+1] =$ '$\boxed{\#}$'
8      if $v_1 \neq v_2$ then reject                                        // header and content block do not fit
9      if $w[p_0..p_1]$ identifies that $w[p_1+2..p_2]$ is not nested then
10         if $r = \lambda$ then $r = w[p_1+2..p_2]$
11         else $r =$ '$r \diamond w[p_1+2..p_2]$'
12         $p_0 = p_2 + 1$                                                  // verify the following blocks
13     if $w[p_0..p_1]$ identifies that $w[p_1+2..p_2]$ is nested then
14         $q_0 = p_1 + 2$
15         while $q_0 \leq p_2$ do
                                                          // next level nested blocks fill the content block
16             let $q_1$ denote the end of the header starting at $q_0$
17             determine the header value $u_1$ of $w[q_0..q_1]$
18             let $q_2$ denote end of content belonging to the header starting at $q_0$
19             determine the content value $u_2$ of $w[q_1+2..q_2]$         // $w[q_1+1] =$ '$\boxed{\#}$'
20             if $q_2 > p_2$ then reject                                   // nested content exceeds content bound
21             $q_0 = q_2 + 1$
22         $p_0 = p_1 + 2$                                                  // verify the nested block
23 until $p_0 \geq n$
24 if $p_0 > n$ then reject else return $r$                                 // $p_0 > n$: content exceeds length bound
```

$w_1, \ldots, w_k \in L_{\text{count}}(\ell, d)$, $|h| \in \mathcal{O}(\log \ell(h))$ and $\ell(h) = \sum_i |w_i|$, and

- in the case that $d(h) = \boxed{C}$: '$h\boxed{\#}c$' $\in L_{\text{count}}(\ell, d)$ iff there exists a partition of $c = w_1 \ldots w_{\ell(h)}$ into substrings $w_1, \ldots, w_{\ell(h)} \in L_{\text{count}}(\ell, d)$ and $|h| \in \mathcal{O}(\log \ell(h))$

To find a unique header h we assume that $\boxed{\#}$ never occurs within a header.

Note that this definition determines the general formal form of a count language.

Lemma 4: If $|h| \in \Theta(\log(\ell(h)))$ then $L_{\text{count}}(\ell, d) \notin \text{DSpace}(o(\log \mathcal{N}))$.

Proof Sketch. For simplicity, we focus on the constant function $d(h) = \boxed{S}$. Now we can use a excursion argument to prove that we need at least $\Omega(\log \mathcal{N})$ deterministic space to accept $L_{\text{count}}(\ell, d)$. More precisely we will show that if there exists an s-space bounded one tape deterministic Turing machine (DTM) M accepting $L_{\text{count}}(\ell, d)$ with $s \in o(\log \mathcal{N})$, if M accepts $h\boxed{\#}0^{\ell(h)}$ then it will also accept $h\boxed{\#}0^{\ell(h)+\ell(h)!}$. Our proof will follow the standard proof to show that there exists no $o(\log \mathcal{N})$-space bounded DTM accepting the length language $\{a^n b^n | n \in \mathbb{N}\}$. ◀

Lemma 5: If $\boxed{\#}$ never occurs within any header h, then $L_{\text{count}}(\ell, d)$ is prefix-free, i.e. for every $w \in L_{\text{count}}(\ell, d)$ and every prefix u of w with $|u| < |w|$ it holds that $u \notin L_{\text{count}}(\ell, d)$.

To remove the nested structure from an element of $L_{\text{count}}(\ell, d)$ we define the following string transformation:

Definition 6: Given a string $w \in L_{\text{count}}(\ell, d)$, define the nested structure remover by the following transformation t:

- $t(w) = c$ if $w =$ '$h\boxed{\#}c$' with $d(h) = \boxed{S}$ and $\ell(h) = |c|$,
- $t(w) = t(w_1)$ if $w =$ '$h\boxed{\#}w_1$' with $w_1 \in L_{\text{count}}(\ell, d)$ and either
  - $d(h) = \boxed{L}$ and $\ell(h) = |w_1|$, or
  - $d(h) = \boxed{C}$ and $\ell(h) = 1$, and
- $t(w) = t(w_1) \diamond \cdots \diamond t(w_k)$ if $w =$ '$h\boxed{\#}w_1 \cdots w_k$' with $w_1, \ldots, w_k \in L_{\text{count}}(\ell, d)$ and either
  - $d(h) = \boxed{L}$, $k \geq 2$, and $\ell(h) = \sum_i |w_i|$, or
  - $d(h) = \boxed{C}$ and $k = \ell(h) \geq 2$.

If we deal with $w \in L_{\text{count}}(\ell, d)$ we assume that Algorithm 1 determines the values of p_1, p_2, v_1, and v_2 as follows:

- For a given value p_0 we would like to verify that a prefix w' of $w[p_0..|w|-1]$ is an element of $L_{\text{count}}(\ell, d)$. As we have seen in Lemma 5, the language $L_{\text{count}}(\ell, d)$ is prefix-free and the header of w' is uniquely determined by the first appearance of the $\boxed{\#}$ symbol. Therefore h and p_1 is defined by the position of the first $\boxed{\#}$ symbol (minus 1) and $v_1 = \ell(h)$.

- For $d(h) \in \{\boxed{S}, \boxed{L}\}$ the length of w' is given by $|h| + 1 + \ell(h)$, therefore $p_2 = p_1 + 1 + \ell(h)$ and $v_2 = \ell(h)$ if $w' \in L_{\text{count}}(\ell, d)$.
- For $d(h) = \boxed{C}$ we search for a prefix w'' of $w[p_1 + 2..|w|]$ such that it is a concatenation of $\ell(h)$ elements of $L_{\text{count}}(\ell, d)$. If these elements exist, then according to Lemma 5 these strings are unique and $p_2 = p_1 + 1 + |w''|$ and $v_2 = \ell(h)$. The question of how we can determine w'' space efficiently will be discussed later.
- Moreover, $w[p_1 + 2..p_2]$ is nested iff $d(h) \in \{\boxed{L}, \boxed{C}\}$.

The values of q_1, q_2, u_1, and u_2 are determined analogously.

Lemma 7: If p_1, p_2, q_1, q_2, v_1, v_2, u_1, and u_2 are determined according to ℓ and d, then Algorithm 1 outputs $t(w)$ on input w iff $w \in L_{\text{count}}(\ell, d)$. Otherwise Algorithm 1 rejects w.

Proof Sketch. Assume that $w \in L_{\text{count}}(\ell, d)$, then Algorithm 1 will iterate over the nested construction of w and add '$\diamond w_i$' (of 'w_i' if this is the first substring) to the output r whenever it finds a substring $h\boxed{\#}w_i$ with $d(h) = \boxed{S}$. Therefore, Algorithm 1 will output $t(w)$.

Let us now assume that Algorithm 1 will output a string '$w_1 \diamond \ldots \diamond w_k$' on an input w. Note that the algorithm only outputs '$w_1 \diamond \ldots \diamond w_k$' if it detects substrings $x_i =$ '$h_i\boxed{\#}w_i$' for all $i \in \{1, \ldots, k\}$ with $\ell(h_i) = |w_i|$ and $d(h_i) = \boxed{S}$.

Iteratively let us now follow the inverse process of Algorithm 1. We can see that Algorithm 1 will only detect a sequence $x_1, \ldots, x_l$ where each element x_i has a form '$h_i\boxed{\#}c_i$' if it had detected a sequence $y_1, \ldots, y_m$ such that there exists one $j' \in \{1, \ldots, m\}$ where for all $j \in \{1, \ldots, m\}$ it holds that

$$y_j = \begin{cases} x_j & \text{if } j < j', \\ h'_j\boxed{\#}x_{j'} \ldots x_{j'+l-m} & \text{if } j = j', \text{ and} \\ x_{j+l-m} & \text{if } j > j' \end{cases}$$

here either $d(h'_j) = \boxed{L}$ and $\ell(h'_j) = |x_{j'} \ldots x_{j'+l-m}|$ or $d(h'_j) = \boxed{C}$ and $\ell(h'_j) = l - m + 1$. Continuing this inverse process we can see that it has lead to the form of the input string w. Since all the intermediate strings are elements of $L_{\text{count}}(\ell, d)$, also the string w has to be an element of $L_{\text{count}}(\ell, d)$.

If we assume that we allow to have headers without any content, $\ell(h) = 0$, then we might get the sequence $x_{j'}x_{j'-1}$ within the construction of $h'_j\boxed{\#}x_{j'} \ldots x_{j'+l-m}$, where we have to assume that this denotes the empty string. ◀

In the following we assume that both functions ℓ and d can be computed in logarithmic space (within the length of w) and $|h| \in \mathcal{O}(\log \ell(h))$ for every header h. If we assume that the value of $d(h)$ is given by some kind of a flag and $\ell(h)$ is somehow presented by a binary representation of this value within h, then these two conditions do not give any unnatural restrictions for the representation of h.

If we now consider calc-context-free languages (and let us assume that only the cases $d : \Sigma^* \to \{\boxed{S}, \boxed{L}\}$ occur), then v_1 can be given as the binary representation of the length of the content $w[p_1 + 2..p_2]$. Thus, if this value is upper bounded by n, we can determine v_1 in $\mathcal{L}$ – or reject if $w[p_0..p_1]$ has not the proper form. Furthermore, using v_1 we can determine p_2 and v_2 in $\mathcal{L}$. If we finally assume that the conditions of the if-statement in Line 9 can be evaluated by examining a constant number of specific symbols within the sub-string $w[p_0..p_2]$ (e.g. if we assume that there exists some kind of a flag within this substring indicating whether $w[p_1 + 2..p_2]$ is nested or not), then Algorithm 1 requires only logarithmic space.

Finally we have to investigate the space complexity of determining p_2 if $d(h) = \boxed{C}$. To decide whether a string w has the form $h\boxed{\#}c$ with $d(h) = \boxed{C}$ and $c = w_1 \ldots w_{\ell(h)}$ where $w_i \in L_{\text{count}}(\ell, d)$ we use Algorithm 2.

While analysing the space complexity of Algorithm 2, we can always assume that if any counter value (i.e. the values of i or p) assumes a value larger than n then the algorithm will stop and reject. Thus, if we can determine $d(h)$ and $\ell(h)$ in space $\mathcal{O}(\log n)$ then the space complexity of Algorithm 2 will be in $\mathcal{O}(\log n)$.

To prove the correctness of Algorithm 2 we have to show the following *nested container condition*:

> A string w fulfils the nested container condition iff $w = h\boxed{\#}c$ (recall that we assume that $\boxed{\#}$ does not occur in h) either
> 1) $d(h) \neq \boxed{C}$ and $|c| = \ell(h)$, or
> 2) $d(h) = \boxed{C}$ and $c = w_1 \ldots w_{\ell(h)}$ such that every substring w_i fulfils the nested container condition.

Lemma 8: Given a string $w = h\boxed{\#}c$ with $d(h) = \boxed{C}$, then Algorithm 2 accepts iff w fulfils the nested container condition.

Note that the nested container condition is equivalent to the correct nested form if we assume that we only have to verify the case where $d(h) \in \{\boxed{S}, \boxed{C}\}$. Since Algorithm 1 tests the correctness of the current level whenever the current level starts, this is sufficient to test whether the current level contradicts the language requirements of $L_{\text{count}}(\ell, d)$ or not. Since any string $w \notin L_{\text{count}}(\ell, d)$ has to contain such a level where locally it has its first deviation from the language conditions, Algorithm 1 will reject the input if this substring will be found. This implies the correctness of Algorithm 1 for $L_{\text{count}}(\ell, d)$ if Algorithm 2 is used for passing the substring $w' = h\boxed{\#}c$ with $d(h) = \boxed{C}$.

One can easily see that Algorithm 1 requires time $\mathcal{O}(\mathcal{N}^2)$. But if we can use a stack as an additional data structure then we can speed up the algorithm as described in Algorithm 3.

Let us first focus on the complexity analysis of Algorithm 3. If we assume that the computation of ℓ and d requires linear time then Algorithm 3 runs in time $\mathcal{O}(\mathcal{N} \cdot t_s)$ where t_s is given by the required time to perform the stack operation (i.e. pop, push, and is empty). If we consider the uniform cost measure we can assume that each of this operations can be done within constant time, thus Algorithm 3 will be a linear time algorithm. Otherwise, we have to assume a factor of $s(|w|)$, where $s(|w|)$ denotes the used space for storing the maximum value of v and v'.

Algorithm 2: Verify Nested Containers

```
Input: Input string $w \in (\Sigma \cup \{\boxed{\#}\})^*$
Output: Accept if $w$ has the form $h\boxed{\#}c$ with $d(h) = \boxed{C}$ and $c = w_1 \dots w_{\ell(h)}$ where $w_i \in L_{\text{count}}(\ell, d)$; otherwise reject
determine the length $n = |w|$ of $w$, and $h, c$ such that $w = h\boxed{\#}c$
if $d(h) \neq \boxed{C}$ then reject
$i = \ell(h)$ and $p = |h| + 2$
while $i > 0$ and $p < n$ do
    determine the header value $h'$ starting at $p$
    if $d(h') \neq \boxed{C}$ then $i = i - 1$ and $p = p + |h'| + 1 + \ell(h')$
    else $i = i - 1 + \ell(h')$ and $p = p + |h'| + 1$
if $i \neq 0$ or $p \neq n$ then reject
accept
```

Algorithm 3: Space Bounded Transducer with a Stack for $L_{\text{count}}(\ell, d)$

```
/* Takes a calc-context-free string employing length-prefix notation (cf. Figure 2). The target language is a sequence of
   strings, separated by the token "$\diamond$". The empty string is described by $\lambda$.                                 */
Input: Input string $w \in (\Sigma \cup \{\boxed{\#}\})^*$ (as read once input)
Output: Output string $w_0 \diamond w_1 \diamond \dots \diamond w_k$ (as a stream) where $w_i$ are the content strings over $\Sigma^*$ and $\diamond \notin \Sigma$
initialise: let $S$ be an empty stack; let $r = \lambda$ be an empty word; let $p_0 = 0$ be the current input symbol
read header $h$ starting at $p_0$ and move $p_0$ to the next position after '$h\boxed{\#}$'
determine $t = d(h)$ and $v = \ell(h)$
if $v = 0$ or $t = \boxed{S}$ then
    copy the next $v$ symbols to $r$ and set $p_0 = p_0 + v$
else
    if $t = \boxed{L}$ then $v = p_0 + v$
    push $(v, t)$ on $S$
    while not ($S$ is empty or timeout) do
        pop $(v, t)$ from $S$
        if $t = \boxed{L}$ and $v < p_0$ then reject
        read header $h$ starting at $p_0$ and move $p_0$ to the next position after '$h\boxed{\#}$'
        determine $t' = d(h)$ and $v' = \ell(h)$
        if $t' \in \{\boxed{S}, \boxed{L}\}$ and $t = \boxed{L}$ and $v < p_0 + v'$ then reject
        if $t' = \boxed{S}$ then
            if $r \neq \lambda$ then append $\diamond$ to $r$
            append the next $v'$ symbols to $r$
            set $p_0 = p_0 + v'$ (the position in $w$ after the appended symbols)
            if $t = \boxed{L}$ and $p_0 < v$ then push $(v, t)$ on $S$
            if $t = \boxed{L}$ and $v < p_0$ then reject
            if $t = \boxed{C}$ and $v > 1$ then push $(v - 1, t)$ on $S$
        else
            if $t = \boxed{L}$ and $p_0 < v$ then push $(v, t)$ on $S$
            if $t = \boxed{L}$ and $v < p_0$ then reject
            if $t = \boxed{C}$ and $v > 1$ then push $(v - 1, t)$ on $S$
            if $v' > 0$ then
                if $t' = \boxed{L}$ then $v' = p_0 + v'$
                push $(v', t')$ on $S$
if $S$ is empty then return $r$
else reject
```

The space complexity of Algorithm 3 is given by $s(|w|)$, where $s(|w|)$ denotes the used space for storing the maximum value of v and v' plus the used stack. If we ignore the stack within this analysis, the required asymptotic space is given by required space for storing the maximum value of v and v'. Note that since we have assumed a one-way input tape we can find strings $w \notin L_{\text{count}}(\ell, d)$ such that this storage is linear in $|w|$. If we focus on the weak cost measures we can see that the maximum value of v and v' are always in $\mathcal{O}(\mathcal{N})$ and therefore $s(|w|) \in \mathcal{O}(\log \mathcal{N})$.

We conclude this chapter by a correctness analysis of Algorithm 3.

Lemma 9: Algorithm 3 outputs $t(w)$ on input w iff $w \in L_{\text{count}}(\ell, d)$. Otherwise, Algorithm 3 rejects w.

Proof Sketch. Before we start with the proof we should notice that whenever the algorithm detects a container (it detects a header h' where $d(h') \neq \boxed{L}$) and if it does not reject (because of some length limitations of predecessor containers within the recursive tree)

- first it pushes the type and the remaining open restrictions of the predecessor container (if some open restrictions are left) to the stack S and
- second if the current container is not empty (i.e. if $\ell(h') > 0$), it pushes $d(h')$ and the calculated restrictions of the current container to the stack S.

Together with the pop in Line 10, the first push results in an adaptation of the stack entry from the predecessor container of the current element to remaining open restrictions of this container. If $d(h') = \boxed{L}$ then the current element is not a container and we only have to adapt the stack entry of the predecessor container. Thus we have to skip the second push within this scenario.

The correctness of Algorithm 3 follows from the observation that it follows the recursive construction of an input string $w \in L_{\text{count}}(\ell, d)$. Whenever the recursive construction reaches a substring $w' = h'\boxed{\#}c' \in L_{\text{count}}(\ell, d)$ it verifies that if the predecessor within the recursive tree

1) is length bounded, that the current position does not exceed the space bound of this element
2) counts its next level elements, it reduces this counter by 1.

Furthermore, it finally removes the predecessor from the stack if the current element exactly fills out the remaining length of the predecessor or if the current element is the last counted element within the predecessor container. Since the recursive structure of every element $w \in L_{\text{count}}(\ell, d)$ completes all internal containers, Algorithm 3 will stop and output $t(w)$ on input w.

Before we start with the analysis of the case that $w \notin L_{\text{count}}(\ell, d)$, it should be mentioned that the presented Algorithm 3 assumes to get always a stream of a sufficiently large input. Thus within the algorithm we will always assume that the input length is always correct, if no timeout occur. If we would like to deal with inputs which might not have the proper length, we have to change the condition of the while statement into

not (S is empty or timeout or the end of input w is reached)

Furthermore, we have to add a similar condition about the detected end of the input w to the if statement of Line 29, i.e. we have to change the condition into

S is empty and the end of input w is reached

Now assume that $w \notin L_{\text{count}}(\ell, d)$. If w has no valid header then Algorithm 3 rejects. I.e., our assumption implies that $w = h\boxed{\#}c$ where h is a valid header.

1) If $d(h) = \boxed{S}$, then $w \notin L_{\text{count}}(\ell, d)$ implies that c does not have the proper length $\ell(h)$ – and therefore Algorithm 3 rejects within one of the included length tests.
2) If $d(h) = \boxed{L}$, then $w \notin L_{\text{count}}(\ell, d)$ implies that either
   - c does not have the proper length $\ell(h)$ – and therefore Algorithm 3 rejects within one of the included length tests.
   - $\ell(h) = |c|$ but there exists no subdivision $c = w_1 \dots w_k$ such that every $w_k \in L_{\text{count}}(\ell, d)$. Since $L_{\text{count}}(\ell, d)$ is prefix-free, we can iteratively determine the unique candidates w_i starting from w_1. Algorithm 3 follows this iterative process and will reject if it detects and treats the first substring $w_i \notin L_{\text{count}}(\ell, d)$.
3) If $d(h) = \boxed{C}$, then $w \notin L_{\text{count}}(\ell, d)$ implies that either
   - c does not include $\ell(h)$ containers (analogously to our analysis of Lemma 8 this only occurs if w is one of the rightmost elements within the recursive construction of the whole input), thus either S is not empty when Algorithm 3 leaves the while-loop, or the algorithm does not reach the end of the input during the while-loop. In both cases Algorithm 3 rejects the input in Line 29.
   - c can be subdivided into $\ell(h)$ substrings, but for every subdivision $c = w_1 \dots w_{\ell(h)}$ one of the elements w_i is not an element of $L_{\text{count}}(\ell, d)$. As already mentioned above we can iteratively determine the unique candidates w_i starting from w_1. Algorithm 3 follows this iterative process and will reject if it detects and treats the first substring $w_i \notin L_{\text{count}}(\ell, d)$.

Thus we can conclude that Algorithm 3 rejects every input $w \notin L_{\text{count}}(\ell, d)$. ◀

Within Algorithm 3 we have assumed a one-way input tape. I.e., the pointer to on our input tape moves only in one direction. This is tantamount to an online algorithm which works on an input stream. For completeness, and to ease reading this paper for readers with a focus on the practical application of our approach, we present Algorithm 4 as the the streaming variant of Algorithm 3.

Algorithm 4 reads data symbol-wise from the input stream and writes data as soon as possible to the output stream. It also

Algorithm 4: Streaming variant of Algorithm 3 to parse $L_{\text{count}}(\ell, d)$

```
/* Reads a calc-context-free string employing length-prefix notation (cf. Figure 2) from the input stream and writes the
   individual tokens, as soon as they have been recognised, to the output stream. The nested structure of the
   calc-context-free input stream is reflected by brackets "$\underline{(}$" and "$\underline{)}$" in the output stream. The alphabet of the target
   language must be without the special symbols "$\underline{(}$" and "$\underline{)}$".                                        */
/* See next page for the auxiliary procedures readHeader and copySymbols.                                    */
Input: Input stream, holding a sequence of words from the target language.
Output: Output stream, which consists of context-free tokens plus brackets "$\underline{(}$" and "$\underline{)}$".
define main():
    let $S$ be an empty stack
    while true do
        $p_0 = 0$  // counts the number of symbols read in the current iteration of the loop
        $readHeader(t, v, p_0)$; write "$\underline{(}$" to output  // begin outermost data structure
        if $v = 0$ or $t = \boxed{S}$ then
            $copySymbols(v, p_0)$; write "$\underline{)}$" to output  // end string, or end container of length 0
        else
            if $t = \boxed{L}$ then $v = p_0 + v$
            push $(v, t)$ on $S$
            while not ($S$ is empty) do
                pop $(v, t)$ from $S$
                if $t = \boxed{L}$ and $v < p_0$ then reject
                $readHeader(t', v', p_0)$; write "$\underline{(}$" to output  // begin inner data structure
                if $t' \in \{\boxed{S}, \boxed{L}\}$ and $t = \boxed{L}$ and $v < p_0 + v'$ then reject
                if $t' = \boxed{S}$ then
                    $copySymbols(v', p_0)$; write "$\underline{)}$" to output  // end string
                    if $t = \boxed{L}$ and $p_0 < v$ then push $(v, t)$ on $S$
                    if $t = \boxed{L}$ and $v < p_0$ then reject
                    if $t = \boxed{C}$ and $v > 1$ then push $(v - 1, t)$ on $S$
                else
                    if $t = \boxed{L}$ and $p_0 < v$ then push $(v, t)$ on $S$
                    if $t = \boxed{L}$ and $v < p_0$ then reject
                    if $t = \boxed{C}$ and $v > 1$ then push $(v - 1, t)$ on $S$
                    if $v' > 0$ then
                        if $t' = \boxed{L}$ then $v' = p_0 + v'$
                        push $(v', t')$ on $S$
                if ($t = \boxed{L}$ and $p_0 = v$) or ($t = \boxed{C}$ and $v = 0$) then write "$\underline{)}$" to output  // end container
```

writes an "opening bracket" to the output stream, whenever something is put onto the stack, and a "closing bracket", whenever something is pulled from the stack. The proof for Lemma 9, which shows the correctness of Algorithm 3, carries over for Algorithm 4, except for some trivial corrections. We thus do not provide a separate proof of correctness for Algorithm 4.

IV. Final Remarks

Length-prefix languages are frequently used for data serialisation purposes. A common intuition is that these languages are easy to parse. Alas, these languages are not context-free, and thus, established tools for language specification, analysis and parsing cannot be applied. The ubiquity of implementation errors when such languages are used as input languages is frightening. Too often, these errors cause security issues and exploits. One might even be tempted to consider length-prefix languages to be fundamentally more difficult to parse than context-free languages. The current research has been motivated by our desire to understand this situation and, perhaps, to do a first step towards providing a remedy. We hope to inspire other researchers with this work.

In Section II, we informally introduce calc-EBNF as a specification language for length-prefix languages. In Section III, we describe and analyse several algorithms. Our theoretical approach is based on Algorithm 1 and Algorithm 2. In their combination, these algorithms describe a transduction from length-prefix to context-free languages in logarithmic space. In

Auxiliary procedures *readHeader* and *copySymbols* for the streaming Algorithm 4

```
define procedure readHeader(out type, out length, in out counter):
    // read and determine type and length from input; move input position to next "$\boxed{\#}$"; increase counter
    read symbol $s$ from input; set counter = counter + 1
    set headString = $\lambda$   // $\lambda$ is the empty string
    while $s \neq \boxed{\#}$ do
        concatenate $s$ to headString
        read symbol $s$ from input
        set counter = counter + 1
    determine type = $d(h)$ and length = $\ell(h)$
define procedure copySymbols(in length, in out counter):
    // read length symbols from input and write them to output; increase counter accordingly
    for $i \in \{1, \dots, length\}$ do
        read symbol $s$ from input
        set counter = counter + 1
        write symbol $s$ to output
```

the full paper, we additionally consider the circuit complexity of calc-context-free languages. A slightly restricted version of calc-context-free languages can be recognised in $\mathcal{AC}^1$.

A practical parser for a data serialization language should be online (read its data from an input stream), fast (its running time should be linear in the size of the input word), and it should use low internal memory (i.e., when a word is a container, which consists of many items, the memory required to parse the word should not be dominated by size of the container, but by the maximum memory required to parse any of the items). Thus, we propose Algorithm 3, a stack-based transducer from calc-context-free to context-free languages. Under reasonable assumptions, Algorithm 3 reads data from an input stream and runs in linear time. Algorithm 4 goes a step further, by also writing its output into a stream. Furthermore, the context-free output from Algorithm 4 provides opening and closing brackets, which match the nesting structure of the calc-context-free input.

Our result confirm the intuitution that well-designed length-prefix languages are easy to parse. We argue that the ubiquity of errors and security issues stems from the lack of theoretical foundations for length-prefix languages, and the resulting lack of practical tools to process them. This leaves plenty of open problems for future work, and a strong motivation to perform this work.

Consider specifications of two length-prefix languages L and L' and questions, such as the following. "Is L the empty language?" "Is L unambiguous?" "Is $L = L'$?" "Is $L \subseteq L'$?" Which of these questions is decidable by an efficient algorithm – or decidable at all? This is challenging for theoretical research.

From a more practical point of view, it would be highly desirable to extend the well-known EBNF for context-free languages towards a calc-EBNF. Section II is a starting point. Given a specification of a calc-context-free language L (perhaps using calc-EBNF as a specification language), there is a need for tools to check L for desirable or essential properties, such as unambiguity. Another important goal would be to develop automatic parser generators for calc-context-free languages, similarly to existing ones for context-free languages. Instead of developing a new parser generator from scratch, one should also consider to implement Algorithm 4 as a "wrapper" for an existing parser generator, such as yacc.

References

[1] Dan Bernstein. Netstrings. http://cr.yp.to/proto/netstrings.txt. 1997.

[2] Bittorrent. Protocol specification v1.0. https://wiki.theory.org/index.php/BitTorrentSpecification.

[3] Noam Chomsky. Three models for the description of language. *IRE Trans. Information Theory*, 2(3):113–124, 1956.

[4] Wikipedia contributors. Heartbleed. https://en.wikipedia.org/wiki/Heartbleed. Date of last revision: 21 March 2020 20:57 UTC.

[5] Google Developers. Protocol buffers encoding. https://developers.google.com/protocol-buffers/docs/encoding.

[6] Norina Marie Grosch, Joshua Konig, and Stefan Lucks. Taming the length field in binary data: Calc-regular languages. In *2017 IEEE Security and Privacy Workshops, SP Workshops 2017, San Jose, CA, USA, May 25, 2017*, pages 66–79. IEEE Computer Society, 2017.

[7] David S. Johnson. A catalog of complexity classes. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 67–161. Elsevier and MIT Press, 1990.

[8] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[9] Daniel J. Rosenkrantz and Richard Edwin Stearns. Properties of deterministic top-down grammars. *Information and Control*, 17(3):226–256, 1970.

[10] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Anna Shubina. The halting problems of network stack insecurity. *;login: USENIX magazine*, 36(6), 2011.

[11] International Telecommunication Union. Information technology asn.1 encoding rules: Specification of basic encoding rules (ber), canonical encoding rules (cer) and distinguished encoding rules (der). https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.690-200811-S!!PDF-E&type=items. X.690 (11/2008).

[12] H. Venkateswaran. Properties that characterize LOGCFL. *J. Comput. Syst. Sci.*, 43(2):380–404, 1991.

[13] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20(11):822–823, 1977.

[14] Vadim Zaytsev. BNF was here: what have we done about the unnecessary diversity of notation for syntactic definitions. In Sascha Ossowski and Paola Lecca, editors, *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*, pages 1910–1915. ACM, 2012.