Universidad Autónoma de Madrid

Escuela Politécnica Superior

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO DE FIN DE GRADO

Lenguajes y seguridad

Pedro Valero Mejía

Tutor: Pierre Ganty

Ponente: Eloy Anguiano Rey

Mayo 2016

UNIVERSIDAD AUTÓNOMA DE MADRID

Escuela Politécnica Superior

Double Degreee in Computer Engineering and Mathematics

BACHELOR WORK

LANGUAGES AND SECURITY

Pedro Valero Mejía

Tutor: Pierre Ganty

Ponent: Eloy Anguiano Rey

May 2016

Lenguajes y seguridad

Autor: Pedro Valero Mejía
Tutor: Pierre Ganty
Ponente: Eloy Anguiano Rey

Imdea Software Institute
Validación y Verificación
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Mayo 2016

LANGUAGES AND SECURITY

Author: Pedro Valero Mejía
Tutor: Pierre Ganty
Ponent: Eloy Anguiano Rey

Imdea Software Institute
Validation and Verification
Escuela Politécnica Superior
Universidad Autónoma de Madrid

May 2016

RESUMEN

Resumen Los analizadores sintácticos están presentes en la mayoría de los sistemas informáticos y suponen una pieza fundamental en el proceso de tratamiento de los datos entrantes. Al mismo tiempo, constituyen el punto de entrada para muchos ataques pues, a menudo, estos analizadores son capaces de procesar lenguajes mucho más expresivos de lo requerido por el protocolo en cuestión. Al ser más complejos de lo requerido, mayor es la probabilidad de contener errores que den lugar a importantes vulnerabilidades.

Para lidiar con este problema se han desarrollado diversas herramientas que permiten la generación automática de estos analizadores para determinados tipos de lenguages. Este es el caso de Bison, que permite generar analizadores sintácticos para lenguajes LALR(1); y ANTLR, para lenguajes LL(*). Sin embargo, estos programas también permiten al usuario introducir código arbitrario como método para aumentar la capacidad del analizador y es aquí donde surgen la mayoría de las vulnerabilidades.

En ocasiones, este incremento de la capacidad del analizador podría evitarse llevando a cabo pequeñas modificaciones en el protocolo. En otros casos la capacidad del lenguaje es suficiente y es la falta de conocimientos formales la que lleva a una extensión inecesaria (y contraproducente desde el punto de vista de la seguridad) de la capacidad del analizador.

Este es el caso de algunos aspectos comunes de los protocolos de red como secuencias de datos cuya longitud viene indicada previamente o campos que emplean un delimitador definido en otro. Sorprendentemente, para muchos de estos aspectos, la literatura contiene aseveraciones vagas o incluso contradictorias acerca de la capacidad requerida por parte del analizador.

En este documento afrontamos este problema desde el punto de vista de la teoría de lenguajes formales. Definiremos límites a nivel formal sobre la capacidad requerida para analizar determinados lenguajes estableciendo así el punto de partida para futuras investigaciones. Los resultados positivos aportados están respaldados por la implementación, empleando Flex y Bison, de un analizador sintáctico para el protocolo HTTP, que será descrito al final de este trabajo.

Palabras clave lenguajes independientes de contexto, generadores de analizadores sintácticos, validación de entradas

Abstract

Abstract Parsers are a fundamental building block in the data processing pipeline of software systems. At the same time, they are also an important entry point for attacks, which is attributed to the fact that, very often, these parsers are able to process languages that are more expressive than what is required by the protocol. Since they are more complex than needed then the probability of containing errors, which lead to important vulnerabilities, increases.

As an attempt of solving this problem tools for generating such parsers, for certain languages, have been developed. This is the case of Bison, which allows to generate parser for LALR(1) languages; and ANTLR, for LL(*) ones. However, these programs allow the user to introduce handwritten code as a mechanism to augment the capabilities of the parser and it is here where most of the vulnerabilities rely on.

Sometimes, this increment of the parser's capabilities could be avoided by making small changes in the protocol. Others, the parser is already able to parse the protocol and it is the lack of formal knowledge which leads to a unnecessary (and counterproductive from the point of view of security) increase of the parser's capabilities.

This is the case of some common features in network protocols such as content length fields or dynamically defined delimiters. Surprisingly, for a number of idioms that are common in network protocols, the literature contains vague or contradictory statements about what that required expressiveness would be.

In this document we approach this question from the point of view of formal language theory, laying a foundation for further research on the parsing problem. The positive results given on this work are backed up by the implementation of a HTTP parser, using Flex and Bison, described at the end of this paper.

Keywords context-free languages, parser generator, input validator

Contents

List of Tables

LIST OF FIGURES

1 Introduction

1.1 Motivation

Parsers are a fundamental building block in the data processing pipeline of software systems. The creation of parsers is a common activity, which is facilitated by so-called parser generators such as Bison or ANTLR. Most parser generators take as input grammar rules that are decorated with so-called *semantic actions*. Semantics actions are performed as the input is being parsed. While the grammar rules are restricted to generate an efficiently parseable subclass of context-free languages (LALR(1) for Bison, LL(*) for ANTLR)[1], the semantic actions allow for the execution of arbitrary code.

Typically, semantic actions are used to produce output, construct a data structure like an abstract syntax tree and also, through the use of code assertions, control the parser behavior. Controlling the behavior allows the parser to recognize languages beyond the class of context-free languages (e.g. the language $a^n\, b^n\, c^n$ is easily parsed by adding an integer variable that semantic actions can manipulate). The downside is that the expressiveness of semantic actions make parsers prone to vulnerabilities and form the entry point for attacks [31, 19, 11].

There is a dispute in the literature, especially in that on network protocol parsing, about the extent to which the expressiveness added by semantic actions is actually required. Some authors suggest that they are necessary because "network protocols often have features that cannot be expressed in a context-

[1] https://en.wikipedia.org/wiki/Comparison_of_parser_generators

free grammar" [12]. Other authors suggest that they are merely required for conciseness of expression, because "it is possible to rewrite these grammars to [...] be context-free, but the resulting specification is much more awkward" [9]. But even in cases where the discussion is more specific about technical details ("As long as the length-field has constant width, it is theoretically possible to describe arrays and padding with a context-free grammar." [25]), the arguments made are not backed up by any formalization or proof. As a consequence, the dispute remains open, and users find little guidance for creating a sufficiently expressive, yet not overpowered and unnecessarily vulnerable, parser for a given task.

1.2 Goal of the project

The main goal of this document is to throw some light over the dispute mentioned before by approaching the problem from the point of view of formal language theory. Namely, we ask the question of which common network protocol features can be recognized without making use of semantic actions, i.e., which can be expressed in terms of parser generator input files *without* writing any code beyond the bare grammar. Note that by asking this question we do not perform an analysis of the full parsing process, but consider only the task of input validation; we leave a formal analysis of the task of filling a data structure to future work.

Once a network feature has been proved to be context-free, people writing parsers for such feature will certainly know that it can be done without using semantic actions. By writing a parser without semantic actions, no handwritten code is added to it and thus, it should be considered safe as long as the parser generator is considered as so.

By the other hand, efforts inverted in parsers' security should focus on features that have been proved to not be context-free. Ideally, these features may be reconsidered in order to make them context-free which will drastically reduce the number of vulnerabilities found on network software systems.

We give an answer for number idioms for which we found contradictory statements in the literature: content length fields, selector fields for message types, chunked messages and dynamically defined delimiters. Our answer takes two aspects into account:

1. From the point of view of formal language theory, we investigate which of these idioms can be expressed in terms of a context free (or even regular) language. We also investigate this question for an intersection of context-free languages, which corresponds to running multiple input validators in parallel and isolated from each other.
2. From a practical point of view, we investigate how concisely these idioms can be expressed in the input language of a widely used parser generator: Bison. For this, we implement an input validator for a subset of HTTP without using semantic actions beyond the ones that are strictly necessary according to our language-theoretical development.

Our study yields the following results.

- We show that languages with unbounded length fields are not context-free nor can be expressed as a finite intersection of context-free languages, while languages with fixed-size length fields are even regular and allow for a concise representation using context-free grammars.
- We show that languages with resource type field are context-free and allow for a natural representation using context-free grammars.
- We conclude that chunked messages are CfL under the same constraints as content length fields.
- We remark that languages with dynamically defined delimiters of unbounded length are not context-free. Even though languages with a dynamically defined delimiter of fixed size are even regular, we show that they cannot be represented concisely in a context-free grammar.

In summary, the contribution of this paper is to investigate, for the first time, the expressiveness required for recognizing several idioms that are common in network protocols. It thereby helps lay the formal foundations of a research area of growing importance.

1.3 Structure of the document

The remainder of this paper is structured as follows. Chapter 2 describes the state of the art and describes concepts that will be used throughout the document, Chapter 3 discusses the case of content length fields, whereas Chapter 4 considers the remaining idioms. We outline an HTTP input validator in Chapter 5 and conclude the document with a brief analysis of recent related work, comparing it with ours.

2 State of the Art

2.1 Language's theory

We begin this chapter with some definitions related to formal languages theory [17] that will be used throughout the document.

An *alphabet* Σ is a nonempty finite set of *symbols*. A *word* w is a finite sequence of symbols of Σ where the empty sequence is denoted by ε. A *language* is a set of words and the set of all words over Σ is denoted Σ^*.

A language may be defined by enumerating its components or by providing a list of rules which describes how the words of the language are composed.

We denote by $|w|$ the *length* of a word w. Further define $(w)_i$ as the i-th symbol of w if $1 \leq i \leq |w|$ and ε otherwise. Hence, $w = (w)_1 \dots (w)_{|w|}$.

Since languages are sets, they can be combined to produce new languages. Concatenation of two languages $L_{12} = L_1 \cdot L_2$ is defined as the set of words obtained by concatenating a word of the first language with a word of the second one, preserving the order. Formally we write

$$L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1, \quad w_2 \in L_2\}$$

Boolean operations as intersection and union are defined as usual

$$L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}; \quad L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$$

There are two important operations left that are applied over languages and which are related with the concatenation. The *Kleene star* of a set consists of words obtained by concatenation of an arbitrary number of words of the language, including the concatenation of 0 words which produces the empty word ϵ. In a similar way the *Kleene plus* is defined by making the number of concatenated words strictly positive. By abuse of notation we often write w^+ and w^* to denote $\{w\}^+$ and $\{w\}^*$, respectively.

Given a nonempty subset S of Σ and $i \in \mathbb{N}$, S^i is defined to be the empty set, $\emptyset$, if $i = 0$ otherwise $S^i = \{w \mid |w| = i \text{ and } (w)_j \in S \text{ for all } j\}$. When $S = \{a\}$ we abuse notation and write a^i to denote the word (not the language) such that $\{a^i\} = S^i$.

Within the context of languages theory, a *grammar* is a set of *production rules* describing how to form strings. The set of generated strings is a language. Therefore, a grammar is usually thought of as a language generator. Formally, a grammar is a tuple $G = (V, \Sigma, S, R)$ where V is a finite set of variables, Σ is a finite set of *terminal symbols*, S is the *axiom* or *starting symbol* and R is a finite set of *rules*.

Each rule is of the form `X -> Y` which means that for any occurrence of X (known as the left part for obvious reasons), Y (known as the right part) can be produced. *Terminal symbols* are those that may appear in a string and for which there is no rule for having them as a left part. Thus Σ is also known as the alphabet of the produced language. Respectively *non-terminal symbols* are the rest of variables which cannot appear in a string but are used to build up the grammar.

The four rules given in 2.1 define a grammar where S and A are non-terminal symbols while a and b are terminals. A is also the axiom of the grammar.

$$A \to aA \qquad A \to aS \qquad S \to baS \qquad S \to b \tag{2.1}$$

To generate a string in the language we start with a string consisting of only the *start symbol*. Any production rule may be applied if its left part appears in the current string (for the first step the left part should be the *start symbol*). When applying a rule, the first occurrence in the given string of the left part of the rule is removed. In its place, the right part of the rule is written. The process continues until there are no *non-terminal symbols* left.

An example of derivation of a string for the grammar 2.1 is

Example:

$$A \Rightarrow aA \Rightarrow aaA \Rightarrow aaaS \Rightarrow aaabaS \Rightarrow aaababaS \Rightarrow aaababab$$

so the word "aaababab" belongs to the language generated by the grammar. ◊

In the example we have implicitly defined a *step* as the binary relation $\Rightarrow$ on $(V \cup \Sigma)^*$ given by $u \Rightarrow v$ if there exists a rule $X \to w$ of G such that $u = \alpha\, X\, \beta$ and $v = \alpha\, w\, \beta$ for some $\alpha, \beta \in (V \cup \Sigma)^*$. To extend this concept we define $u \Rightarrow^* v$ if there exists a step sequence $u_0 \Rightarrow u_1 \Rightarrow \ldots \Rightarrow u_n$ such that $u_0 = u$ and $u_n = v$. A step sequence $u \Rightarrow^* w$ is called a *derivation* whenever $u = S$ and $w \in \Sigma^*$.

The language generated by a grammar G is formally defined as

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\} \ .$$

A grammar is said to be *context free* if and only if the left part of each rule consists of a single *non-terminal* symbol. The grammar given in 2.1 is context free. To illustrate the difference we present the grammar 2.2 as an example of a *context-sensitive grammar*.

$$A \to aS \qquad A \to bS \qquad aS \to aaS \qquad aS \to bS \qquad bS \to aS \qquad S \to b \tag{2.2}$$

The following example shows a derivation of a string for the grammar 2.2

Example:

$$A \Rightarrow aS \Rightarrow aaS \Rightarrow abS \Rightarrow abb$$

so the word "abb" belongs to the language generated by the grammar. ◊

A language L is said to be *context-free*, or *CfL* for short, if there exists a context-free grammar G such that $L = L(G)$.

The *size* of a grammar is the sum of the sizes of its production rules, where the size of a rule (X, w) is given by $1 + |w|$.

2.2 LangSec: Language-theoretic Security

Amongst the field of informatics system's security, langsec [3] is a branch of investigation that aims to improve the safety of these systems by tackling the problem from a theoretical point of view. Thus, the messages entering the system are treated as words from a formal language and the grammar associated with said language becomes the primary defense weapon against hostile inputs.

To date there have been many and varied attempts to solve the recent "epidemic of insecurity" [9, 25, 23, 27]. Practically all of these attempts have gone through the development of programs in order to automatically generate the code of the parser, thus trying to avoid the entry of errors on behalf of the programmer.

Facing the lack of sufficiently satisfactory results, LangSec considers that the only path to guarantee the system's safety implies defining a formal language that recognizes the valid entries and so diminishes the problem of input handling by creating a recognizer for that language

Nonetheless, far from being an "Ivory Tower" theory, LangSec is very focused on the achievement of practical results. Based on solid theoretical principles, it develops efficient tools in order to address these and other problems:

- Every element of an informatics system has to accept certain entries and reject those that are badly molded. The problem that arises is usually caused by the fact that this task is completed in an uncontrolled manner and doesn't follow any logic, performing small trials which are scattered throughout the program.

- Every time that an unspecific entry is accepted and processed, the program must engage more resources than those strictly required. This generally implies the storage of more memory, in accordance with the characteristics of the entry, which entails an open door for the attackers. If we restrict the capacities of our program to what is really essential, we reduce the amount of possible error points.

- Some protocols are too complex, which leads to the consequent use of complex parsers that tend to be full of errors.

- The formats that can be imitated by attackers are the biggest threat due to the fact that it is not possible to distinguish a valid message from an attack. If the accepted messages' language could be so broad that it could contain attacks, it would be impossible to reach a minimum level of security.

2.3 Parser generators

A parser generator is a tool that accepts a file specifying a grammar as input and thus produces a program, written in a certain programming language. This program, when compiled, produces a parser for the given grammar. Most parser generators define their own language to write grammars, which tends to be very restrictive.

We can find a huge variety of parser generators [5] with different capabilities in terms of the types of languages that can be accepted by their generated parsers. They also differ in the language in which the parser is written and the format used to describe the grammars. However, there is one thing all these parsers have in common; they use semantic actions.

By semantic action we mean snippets of arbitrary handwritten code that will be executed by the parser under certain conditions while parsing an input.

One of the most known parser generators is Bison, which is usually combined with Flex, where semantic actions are related to a rule and are performed once its rule has been used to reduce the input. Table 2.1 contains a brief overview of some other parser generators, showing that semantic actions are common amongst them.

2.3.1 Safeness

Most network software systems have to parse untrusted inputs that may be attacks. An attacker will try to make the system hang or crash by exploiting its vulnerabilities. From a theoretical point of view, a parser is said to be *safe* when there are no vulnerabilities and thus it cannot be attacked. Since this definition cannot be achieved in practice, we consider a parser as *safe* when we have reasons to expect the parser to not have vulnerabilities.

Writing a parser (or any application in general) by hand is an error prone task which makes the resulting parser extremely vulnerable. Parser generators try to solve this problem by providing a way to *automatically* build safe and bug-free parsers from a given description of the grammar that must be parsed.

Once the parser generator has been extensively used (such is the case of Flex and Bison, for example) it can be considered safe as long as the original bugs have been fixed and no others have been detected recently.

Name	Use of semantic actions
ANTLR4	Semantic actions are pieces of arbitrary code surrounded by curly braces which can be used in a number of places within the grammar. [26]
APG	Under the name of *Callbacks*, handwritten functions are defined to be executed after a rule is used. [30]
Beaver	Same as Bison, semantic actions are specified after each rule. [14]
DRAGON	Each production rule includes an optional method's name to be called once the rule is used. This method should be implemented by the user in the target language. [21]
Irony	There are no semantic actions but the input itself is a piece of C# code. This tool may be seen as a library that facilitates the process of parsing but handwritten C# code may be used everywhere. [2]
Hammer	Same situation as the previous one but using C code. [27]

Table 2.1: Semantic actions in different parser generators

However, as soon as we add handwritten code to the parser (which is done through semantic actions), we can no longer assume that the parser is safe. Even if the parser generator can be trusted, the user may add vulnerabilities within its handwritten code.

2.3.2 Flex and Bison

It is worth focusing on these two tools since these are the ones that are used during the implementation of our HTTP input validator, described at Chapter 5. In this section we will not describe their inner workings, instead we give an overview of how they work.

Both tools receive input files with a certain format, describing what the user wants the parser to do and produces a C code which, when compiled, performs the expected parsing.

Flex

This tool is also referenced to as the *lexer*. It accepts as input a file describing which tokens should be detected, given as regular expressions, and generates a C program to do it. Figure 2.1 shows an example of a Flex's input whose goal is to generate a tokenizer which detects any non-empty sequence of digits as a "NUMBER"; any non-empty sequence of letters as a "WORD" and carriage returns and new lines as so. Any other symbol will be identified as "OTHER".

```
[0-9]^+ {return NUMBER;}
[a-zA-Z]^+ {return WORD;}
"\n" {return CARRIAGE_RETURN;}
"\r" {return NEW_LINE;}
. {return OTHER;}
```

Figure 2.1: Example of a set of rules defined in Flex.

The C program will read the input trying to find the longest string that matches one of the rules. When it is found, the C code written with the rule is executed. The last rule matches anything that doesn't match any of the previous ones.

By abuse of notation, the code shown in Figure 2.1 is usually called lexer and referred to as the program that performs the division of the input into tokens, ignoring the step of producing a C program which implements this procedure.

Bison

Bison is a parser generator which is accepts a grammar as input and produces a C program that checks whether a given message could have been generated by the grammar or not. It is important to take into account that during this process semantic actions are performed.

When used in combination with Flex, the generated parser will split the message into tokens, according to the rules defined within the lexer, and then check if the result follows the given grammar. Thus, the terminal symbols that appear in the Bison's grammar are the tokens detected by Flex. Figure 2.2 shows an example of a grammar. This grammar represents all those messages composed by a "WORD" followed by a non-empty and arbitrarily large list of "NUMBER"s, each one in a different row.

The importance of context-free grammars (and respectively, context-free languages) comes from the fact that as long as a language is context-free it can be

```
axiom: WORD S {}
S: NEW_LINE NUMBER S {}
 | NEW_LINE NUMBER {}
```

Figure 2.2: Example of Bison's grammar.

parsed with a context-free grammar and thus with Flex and Bison without using semantic actions.

2.4 Fuzz testing

Fuzz testing or fuzzing is a software testing technique based on providing invalid, unexpected or random data to the inputs of a computer program looking for crashes, hangs and any kind of memory leak. Basically, it is a form of random testing.

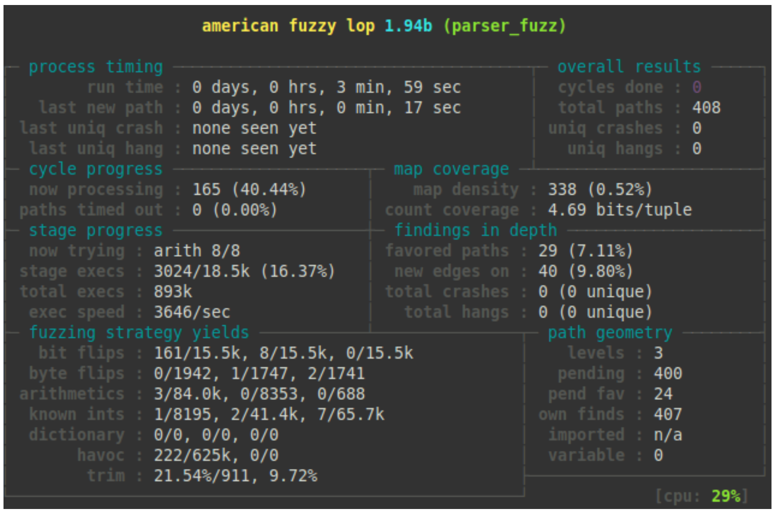
Fuzzing can be employed as white-, gray- or black-box testing. The fuzzer (program that performs the fuzzing over a target) feeds the target program with arbitrary inputs and monitors the program behavior until a memory leak is found. While the target is not crashing, the fuzzer performs small variations to the input to explore different paths.

One of the most famous fuzzers is *American Fuzzy Lop* (American Fuzzy Lop for short) [32] which employs a new type of compile-time instrumentation and genetic algorithms to automatically discover interesting inputs. For cases when the source code of the target is available, AFL incorporates a compiler which allows AFL to perform fuzzying while monitoring the internals of the program. Figure 2.3 shows a running instance of AFL.

2.5 Input validation

Within the context of computer science, the concept of parsing refers to the process of analyzing a string, filling certain data structure and extracting information from it, according to a given grammar.

By using Bison (with or without flex), input validation can only be done by using semantic actions since the data structure have to be defined by the user and so filled "manually" by adding the proper code to implement this task.



```
                    american fuzzy lop 1.94b (parser_fuzz)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 3 min, 59 sec       │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 17 sec       │  total paths : 408    │
│ last uniq crash : none seen yet                      │ uniq crashes : 0      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 165 (40.44%)      │    map density : 338 (0.52%)           │
│ paths timed out : 0 (0.00%)         │ count coverage : 4.69 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : arith 8/8             │ favored paths : 29 (7.11%)             │
│ stage execs : 3024/18.5k (16.37%)   │  new edges on : 40 (9.80%)             │
│ total execs : 893k                  │ total crashes : 0 (0 unique)           │
│  exec speed : 3646/sec              │   total hangs : 0 (0 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 161/15.5k, 8/15.5k, 0/15.5k           │    levels : 3          │
│  byte flips : 0/1942, 1/1747, 2/1741                │   pending : 400        │
│ arithmetics : 3/84.0k, 0/8353, 0/688                │  pend fav : 24         │
│  known ints : 1/8195, 2/41.4k, 7/65.7k              │ own finds : 407        │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 222/625k, 0/0                         │  variable : 0          │
│        trim : 21.54%/911, 9.72%                     ├────────────────────────┘
└─────────────────────────────────────────────────────┘             [cpu: 29%]
```

Figure 2.3: Example of American Fuzzy Lop running over the program "parser_fuzz"

However, throughout this document, we restrict ourselves to input validation, which is slightly different from parsing. Input validation consists in determining whether a string is correct or not according to a given grammar. Parsers also perform input validation indirectly since messages not matching the rules are rejected and only the correct ones are used to fill some data structure. Thus, parser generators are also used to generate input validators.

3 Formal Analysis of Content Length Fields

Content length fields are elements of network protocols such as HTTP and DNS that determine the length of subsequent arrays. As an example, consider the following HTTP POST message.

```
POST /1/notification/list HTTP/1.1\r\n
Content-Length: 47\r\n\r\n
{"header":{},"query":{"count":100},"answer":{}}\r\n
```

Here, the content length field begins after the keyword `Content-Length:` and terminates before the carriage return/newline `\r\n`. Its content, i.e. 47, describes the length of the body of the message, which is the string that follows the double `\r\n` and that ends with `\r\n`. In the case of the HTTP protocol, the content length field may be of arbitrary size; in other cases, such as the `RDLENGTH` field of the DNS protocol [6], the content length field has fixed size.

In this section we characterize content length fields from the point of view of formal language theory. We begin by a formalization that aims to capture their essence, and then characterize the class of languages that are required for expressing them in the bounded and unbounded cases.

3.1 Modeling Content Length Fields

For modeling content length fields, we will work with formal languages over an alphabet Σ. For the example of HTTP, Σ would be the ASCII character set.

We define the *size of the content length field* as the number of digits composing the number that represents the length. As shown next, when modeling content length fields there are slightly differences depending on the boundary of the size of the length field.

3.1.1 Fixed Size

For describing content length fields of finite size $n > 0$ we define the language L_F as follows:

$$L_F(n) = \{w\ x \mid w \in B^n, |x| = \textstyle\sum_{i=0}^{n-1} (w)_{i+1} \cdot b^i\} \ .$$

Here $B = \{0, \ldots, b-1\} \subseteq \Sigma$ for $b > 1$. Intuitively, $L_F(n)$ represents the same number twice, using two different encodings: first b-ary as w and then unary as x, where the relationship between both encodings is given by

$$|x| = \sum_{i=0}^{n-1} (w)_{i+1} \cdot b^i \quad .$$

For example, the word `11abc` consists of the binary representation of $3 = (1 \cdot 2^1) + (1 \cdot 2^0)$ followed by a word (`abc`) of length 3.

Remark 1. *The entire development of this section remains valid when choosing a length encoding in which the most significant digit comes first, e.g., as in big endian encodings.*

3.1.2 Unbounded Size

For describing content length fields of unbounded (and unknown) size, observe that any overlap in the alphabets Σ and B for describing content and its length, respectively, introduces ambiguity as to where the content length field ends. A common approach to remove such ambiguities is to use a delimiter, which is a special symbol $\sharp \in \Sigma \setminus B$ that marks the end of the content length field. We extend the definition of $L_F(n)$ to account for such delimiters:

$$L_F^{\sharp}(n) = \{w \sharp x \mid w \in B^n, |x| = \textstyle\sum_{i=0}^{n-1} (w)_{i+1} \cdot b^i\} \ .$$

We are now in position to define a language for describing content length fields of arbitrary and unbounded size:

$$L_\infty = \bigcup_{i>0} L_F^\sharp(i) \ .$$

Observe that $L_F^\sharp(n)$ is finite for every n but L_∞ is not.

3.2 Analysis of Unbounded-Size Content Length Fields

We now show that L_∞ is not a context-free language. This implies that every input validator based on CfL (like Bison, etc.) has to rely on semantics actions to deal with unbounded content length fields. We obtain this result as a corollary of a more general result, namely that L_∞ is not even a finite intersection of CfLs.

Teorema 2. *L_∞ is not a finite intersection of context-free languages.*

The generalization from context-free languages to finite intersections of context-free languages is

1. *practically relevant*, because the intersection of languages, e.g. $L_1 \cap L_2$, can be validated by running validators for L_1 and L_2 and taking the conjunction of their return values. If the validators for L_1, L_2 are "trusted" then so should be the input validator for $L_1 \cap L_2$. This means that we can build trusted input validator for languages beyond CfL, which departs from some claims emerging in the security literature [28].

2. *non-trivial*, because CfL are not closed under intersection: Consider $L_1 = \{a^n\, b^n\, c^i \mid i, n \geq 0\}$ and $L_2 = \{a^i\, b^n\, c^n \mid i, n \geq 0\}$. Their intersection $L_1 \cap L_2 = \{a^n\, b^n\, c^n \mid n \geq 0\}$ is a textbook example for a language that is not context-free.

3.2.1 Proof Sketch

For the proof sketch, we need to introduce semilinear languages as a tool: a subset of $\mathbb{N}^n$, with $n > 0$, is called *semilinear*, if it can be expressed as a finite union of linear sets. A set $S \subseteq \mathbb{N}^n$ is called *linear* if there exists $\vec{b} \in \mathbb{N}^n$ and a finite subset $\{\vec{p}_1, \dots, \vec{p}_m\}$ of $\mathbb{N}^n$ such that $S = \{\vec{b} + \lambda_1\, \vec{p}_1 + \cdots + \lambda_m\, \vec{p}_m \mid \lambda_1, \dots, \lambda_m \in \mathbb{N}\}$.

Let $w = \langle w_1, \dots, w_n \rangle$ be a tuple of n words. Define a mapping $f_w \colon \mathbb{N}^n \to w_1^* \dots w_n^*$ by $f_w(i_1, \dots, i_n) = w_1^{i_1} \dots w_n^{i_n}$, that is, the output of f_w is a word in which the i-th component of w is repeated a number of times that corresponds to the i-the input to f. We define the preimage of f_w and liftings of f_w from elements to subsets of $\mathbb{N}^n$ in the natural way.

The following result by Latteux [20] establishes a fundamental correspondence between languages that can be expressed as the intersection of CfLs and semilinear sets.

Proposition 3 ([20]). *Let $w = \langle w_1, \ldots, w_n \rangle$, $n > 0$, and $L \subseteq w_1^* \ldots w_n^*$: $f_w^{-1}(L)$ is semilinear if and only if L is the intersection of a finite number of CfLs.*

Since $L_\infty \not\subseteq w_1^* \ldots w_n^*$, Proposition 3 does not directly apply to L_∞. However, as the following lemma shows, it does apply to the subset of words in L_∞ where the content length consists exclusively of 1's and the message's body is obtained by repeating the non-delimiter symbol a. With $a \neq \sharp$, this subset is formally given by

$$L_\angle(a) = L_\infty \cap 1^* \sharp^* a^* \quad .$$

Lemma 4. *$L_\angle(a)$ is not a finite intersection of context-free languages.*

The proof of Lemma 4 relies on the observation that

$$L_\angle(a) = \{w \sharp a^{val} \mid w \in 1^n, val = \textstyle\sum_{i=0}^{n-1} b^i\} \quad .$$

It is routine to check that $\sum_{i=0}^{n-1} b^i = (b^i - 1)/(b-1)$, which gives us the following representation of the preimage of f_w:

$$f_w^{-1}(L_\angle(a)) = \left\{ \left(i, 1, \frac{b^i - 1}{b-1}\right) \mid i \in \mathbb{N} \right\} \quad . \tag{3.1}$$

For showing that this set is not semilinear we make use of the facts that (a) the third component grows exponentially in i, and (b) f_w^{-1} is infinite. Using the definition of semi-linear set, we take two elements in (b) and we construct a third one. We then show that those three elements violate (a) unless they all coincide. The full proof details are given in Appendix A.

With this, the statement of Theorem 2 follows immediately: Assume to the contrary that L_∞ is a finite intersection of context-free languages. Since $1^* \sharp^* a^*$ is context free, $L_\angle(a)$ is also a finite intersection of CfLs, which contradicts Lemma 4.

3.2.2 Beyond Length Constraints

Our definitions of $L_F(n)$ and L_∞ do not put any constraints on the structure of the word w that follows the length field and the delimiter (if any). In practice, however, the word w may need to satisfy constraints beyond those on its length, such as containment in a specific language.

For capturing such constraints consider a language L_∞^C, which is defined along the lines of L_∞, but with the message's body $w \in C$, for an infinite context-free language C (instead of the less restrictive $w \in \Sigma^*$ used in the definition of L_∞). We have the following theorem.

Teorema 5. *L_∞^C is not context-free*

For the proof of Theorem 5 assume that L_∞^C is context-free. Construct from L_∞^C a new language L_∞^a by replacing $w \in C$ by $a^{|w|}$. This replacement can be realized by a so-called *sequential transducer mapping* which, in our case, is a mapping from 2^{Σ^*} into 2^{Σ^*}. Sequential transducers are an input-output automata based formalism: while the automaton reads from its input tape, it outputs symbols on its output tape. In our setting, the automaton outputs what it reads until the first occurrence of '$\sharp$'. After outputting '$\sharp$' the automaton outputs 'a' for each symbols it reads. A detailed definition of sequential transducers is found in the book of Ginsburg [17] where he also shows that context-free languages are closed under sequential transducers mapping [17, Theorem 3.3.1]. We thus conclude that L_∞^a is a context-free language. We can now replace $L_\angle(a)$ by L_∞^a in the proof of Lemma 4 and obtain a contradiction to the initial assumption that L_∞^C is context-free.

3.3 Analysis of Fixed-Size Content Length Fields

Languages $L(n)$ with finite-size content length fields are finite, hence regular and thus also context-free. Next, we study the size of specifications for $L(n)$. In terms of finite state automata, all automata describing $L(n)$ grow exponentially in n. This is because for all automata deciding $L(n)$ there cannot be less than exponentially many reachable states after reading the first n symbols. For otherwise, two distinct numbers are not distinguished by the automaton, hence it cannot decide $L(n)$. We conclude that the encoding of $L(n)$ by a finite state automaton is not useful for practice purposes. However, when $L(n)$ is specified using context-free grammars, we show in this section that it admits a much more compact encoding.

Teorema 6. *Let $n > 0$, there exists a CfG $G(n)$ of size $O(n)$ such that $L(G(n)) = L_F(n)$.*

Proof. We prove the statement by defining the grammar $G(n)$. For simplicity of presentation we assume that length fields are encoded in binary, i.e. $b = 2$ in the definition of $L_F(n)$. The generalization to any $b > 2$ is tedious but straightforward.

The intuition behind the first set of rules of $G(n)$ is to scan the length field from left to right and generate a symbol F_i whenever there is a 1 at position i. I.e., the symbol F_i represents the number 2^i. Zeroes are ignored.

$$S \to X_n \qquad\qquad X_0 \to \varepsilon$$
$$\{X_i \to 0\ X_{i-1} \mid 1 \leq i \leq n\} \qquad\qquad \{X_i \to 1\ X_{i-1}\ F_{i-1} \mid 1 \leq i \leq n\}$$

The intuition behind the next rules is to translate the binary encoding of the length field, given by the F_is, for $i > 0$, into a unary encoding of the length field, given by the terminal symbols. The role of those rules is better understood considering parse tree instead of derivations. In order to avoid cluttering, we do not recall here the definition of parse tree [18]. Each F_i yields parse trees that all have the same shape as shown in the Figure 3.1 for the case F_3. Thus if the root of the tree is labeled F_i it has 2^i leaves.

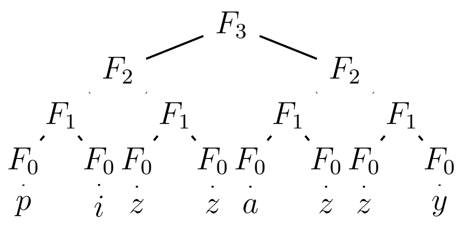


Figure 3.1: Parse tree of the $2^3 = 8$ symbol word "pizzazzy". ("pizzazzy" scores you 49 points at Scrabble®).

Each of those symbols corresponds to one symbol of the payload that follows.

$$\{F_j \to F_{j-1}\ F_{j-1} \mid 1 \leq j \leq n\} \qquad\qquad \{F_0 \to c \mid c \in \Sigma\}$$

It follows by construction that $L(G(n)) = L_F(n)$. Inspecting the definition shows that the size of the rules of each set is fixed and independent from n while the number of rules on each set is exactly n so the size of $G(n)$ is $O(n)$. □

Example: *The derivation below shows that* $11abc \in L_F(2)$ *is also contained in* $L(G_f(2))$.

$$S \Rightarrow X_2 \Rightarrow 1X_1F_1 \Rightarrow 11X_0F_0F_1 \Rightarrow 11F_0F_1 \Rightarrow 11F_0F_0F_0 \Rightarrow^* 11abc$$

$\Diamond$

4 Formal Analysis of Selectors, Chunks, and Dynamic Delimiters

In this section we perform a formal analysis of idioms that are commonly encountered in network protocols and about which there are ambiguous statements in the literature: Selector fields, chunks, and dynamic delimiters.

4.1 Selector Fields

We illustrate selector fields using the example of the record type field of the DNS protocol. If the field `QTYPE` has type `A`, the body of the message is a 32-bit IPv4 address, whereas if it has type `AAAA`, the body is a 128-bit IPv6 address.

```
example.com  A     12.34.56.78
example.com  AAAA  0123:4567:89ab:cdef:0123:4567:89ab:cdef
```

That is, depending on the type of the selector field, the IP address that matches the domain "`example.com`" has IPv4 or IPv6 format. A similar selector field is used in the HTTP protocol to specify whether the message is a response or a request, which affects the possible header fields.

Selector fields hence mandate the parser to use different subsets of grammar rules when parsing the message body. They have been claimed to yield awkward specification in terms of grammars [25]. Here, we argue that selector fields can actually be encoded in a natural way.

4.1.1 Selector Fields as a Grammar

As a starting point we assume a finite set of $n > 0$ words $w_i \in \Sigma_i$, for $1 \leq i \leq n$, specifying the selectors such as A or AAAA. We further assume a set of n grammars $G_i = (V_i, \Sigma_i, S_i, R_i)$ specifying the language of the message body associated to the selector w_i.

Without loss of generality, assume that the non-terminal symbols of the grammars G_i for the individual message bodies are pairwise disjoint, i.e., $V_i \cap V_j = \emptyset$, for all distinct i, j. Then the grammar $G = (V, \Sigma, S, R)$ given by

- $V = \{S\} \cup \bigcup_{i=1} V_i$ where S is a fresh nonterminal that is also the start variable,
- $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$, and
- $R = \{S \rightarrow w_i\ S_i\} \cup \bigcup_{i=1}^{n} R_i$

is a natural account for the context-free language $L(G) = \bigcup_{i=1}^{n} \{w_i\} \cdot L(G_i)$ that specifies the relationship between selector fields and their content.

Example: *The simplified DNS messages given to illustrate selector fields can be parsed by using the grammar from Figure 4.1. This grammar has been built under the assumption that a valid domain is a non-empty string where any symbol whose hexadecimal ASCII code is between 0x21 and 0x7E is allowed.*

```
S → DOMAIN IP
DOMAIN → [0x21,0x7E]+
IP → 'A' IPv4
   | 'AAAA' IPv6
IPv4 → ([0-9][0-9]'.'){4}
IPv6 → ([0-9a-f]{4}'.'){8}
```

Figure 4.1: Grammar for parsing the selector field of a simplified version of DNS protocol.

A derivation of a message for this grammar will be

S ⇒ DOMAIN IP ⇒ example.com IP ⇒ example.com A IPv4 ⇒ example.com A 12.34.56.78

4.2 Chunked Messages

HTTP supports a transfer mechanism in which data is sent in a series of chunks. Chunked transfer is indicated by the header `Transfer-Encoding: chunked`, signaling that the message is split into and thus requires a special treatment. Each chunk contains a variable length field with the size of the chunk.

```
HTTP/1.1 200 OK\r\n
Date: Mon, 22 Mar 2004 11:15:03 GMT\r\n
Content-Type: text/html\r\n
Transfer-Encoding: chunked\r\n
\r\n
29\r\n
<html><body><p>The file you requested is \r\n
5\r\n
3,400\r\n
23\r\n
bytes long\r\n
0\r\n
\r\n
```

4.2.1 Modeling and Analysis of Chunked Transfer Encoding

To model chunked messages, we define the languages L_{c_∞} and L_{c_F} relying on the definitions given in Section 3.1:

$$L_{c_\infty} = (L_\infty)^+, \qquad L_{c_F}(n) = (L_F(n))^+ \ .$$

The difference between these two languages rely on the definition of the length indicator of each chunk.

Teorema 7. *The language L_{c_∞} is not a finite intersection of CfLs. The language $L_{c_F}(n)$ is regular for all $n > 0$ and there exists a grammar $G_{c_F}(n)$ of size $O(n)$ such that $L(G_{c_F}(n)) = L_{c_F}(n)$.*

Proof. The languages L_F and L_∞ have already been studied in Section 3.1; here we focus on the $^+$ operator. To this end, let $a \in \Sigma$ be such that $a \neq 1$ and $a \neq \sharp$. One can easily see that

$$L_\angle(a) = L_{c_\infty} \cap 1^*\sharp^* a^* \ .$$

It is worth pointing out that with the previous intersection we are just considering the first chunk of each message. If the language L_{c_∞} is a finite intersection of CfLs then $L_\angle(a)$ will also be a finite intersection of CfLs, which leads to a contradiction with Lemma 4.

Next, we exhibit a grammar for the language $L_{c_F}(n)$ which is regular since $L_F(n)$ is a finite language. Let $G(n)$ be a context-free grammar for the language $L_F(n)$ and let S_G be its start symbol. Following Theorem 6, the size of $G(n)$ is $O(n)$. Finally, the grammar $G_{c_F}(n)$ is defined by adding to $G(n)$ a fresh start variable S and the following two rules which intuitively concatenate $L_F(n)$ with itself one or more times:

$$S \to S_G \qquad \text{and} \qquad S \to S_G\ S\ .$$

Clearly, $L(G_{c_F})(n) = L_{c_F}(n) = (L_F(n))^+$ and the size of $G_{c_F}(n)$ is $O(n)$. □

4.3 Dynamically Defined Delimiters

Some protocols, such as the MIME protocol, allow to split messages into multiple parts that are separated by user-defined delimiters. We start with the following example [8].

```
MIME-Version: 1.0
Content-type: multipart/mixed; boundary="Mydelimiter"

This is the preamble.  It is to be ignored.
-Mydelimiter

This is implicitly typed plain ASCII text.
-Mydelimiter
Content-type: text/plain; charset=us-ascii

This is explicitly typed plain ASCII text.

-Mydelimiter-
This is the epilogue.
```

In the above example, the delimiter is first declared, `boundary="Mydelimiter"`, and then `Mydelimiter` is used three times, the first two times as `--Mydelimiter` the last time as `--Mydelimiter--`. The challenge in parsing such message is to match correctly the declared delimiter with all its subsequent occurrences. Typically, it is done using semantic actions, where the delimiter is stored in a table whenever declared and then table lookups are used to identify all the subsequent occurrences. The same situation occurs with user-defined variables in a programming language.

4.3.1 Modeling and Analysis of Dynamically defined Delimiters

We now study parsing with user-defined delimiters assuming semantic actions are prescribed. Consider the language given by $L_d = \{w\,w \mid w \in \Sigma^*\}$. Intuitively, L_d captures a use-define relation where the first w corresponds to the declaration while the second corresponds to the use. Because the language L_d is a classical example of a language that is not context-free, we conclude that user-defined delimiters are not context-free.

However, by restricting the language to delimiters of fixed size, say n, we obtain a finite (and thus regular) language of all possible delimiters. Turning back to the language L_d, the restriction yields the following subset of L_d:

$$L_d(n) = \{w\,w \mid |w| = n\} \ .$$

Although $L_d(n)$ is regular, the following theorem, due to Y. Filmus [16], shows it has no "compact" representation when specified as a context-free grammar. This implies it has no "compact" representation by a finite state automaton either.

Teorema 8. *Let $L_d(n) \subseteq \Sigma^*$ with $|\Sigma| = t > 2$ be given by $\{w\,w \mid |w| = n\}$. Every context-free grammar for $L_d(n)$ has size*

$$\Omega\left(\frac{t^{n/4}}{\sqrt{2n}}\right)$$

Recall that $f(n) = \Omega(g(n))$ means that f is lower-bounded by g for sufficiently large n, which implies that context-free grammars for $L(n)$ eventually exhibit exponential growth in n.

5 An Input Validator for HTTP written in Bison

In this section we demonstrate that an input validator for a large subset of HTTP can be written using the parser generator Bison (and Flex), which relies on context-free grammar rules. Our goal is to explore the practical implications of avoiding semantic actions wherever possible, so that the resulting input validator is safe (under the assumption that the parser generator is). We discuss two aspects of our grammar, namely content length fields and selector fields, whose theoretical properties we analyzed in Chapter 3 and Section 4.1, respectively.

5.1 Content Length Field

HTTP features content-length fields of variable, unbounded size. Following Theorem 2 it is not possible to validate unbounded content-length fields while entirely avoiding semantic actions. We see two approaches to tackle this issue:

1. To define an arbitrary but sufficiently large upper bound on the size of the length field and then run several input validators, one for each size. The output of the validator will be the disjunction of these validators' outputs. This approach has the advantage of avoiding semantic actions and the disadvantage of rejecting valid messages that are too large.

2. To create a parser that outputs the size of the content length field by using just one semantic action (a print statement) and then generate a context-free

grammar to validate the content length field according to that value. This approach has the advantage of being able to validate messages of unbounded size and the disadvantage of making (limited) use of semantic actions.

We choose the second approach which is divided into two steps. In the first one, we compute the size n of the content length field. With that information, we know by Theorem 6 that the resulting language is context-free. Thus, in the second step we automatically generate a grammar that corresponds to $G(n)$ defined in Section 3.3.

5.1.1 Computing the Size of the Length Field

Figure 5.1 shows the parser that outputs the size of the content length field where “yyleng” is a Flex internal variable containing the length of the last match. The role of the print statement is to output the size of the content length field.

```
(?i:Content-length:"␣"[0-9]+"\r\n")
  ↪ {printf("%lu\n",yyleng - 18);}

"\r\n\r\n" {exit(0);}
"\r" {}
"\n" {}
. {}
```

Figure 5.1: Parser to extract the size of the content length field.

We may consider this parser to be safe since the handwritten code added to Flex is not interacting directly with the input. As long as we considerer Flex as a safe tool we know the type of variable which is “yyleng” and we know that it will store a value bigger than 18. Thus, the parser is printing a positive number and should be errors-free.

The “exit(0)” command is used to make the parser stop once the headers of the message have been read, avoiding the possibility of finding a matching string in the body of the message.

5.1.2 Generating Validators for Fixed-size Content Length Fields

Once the size n of the content length field is known, we can compute the grammar corresponding to $G(n)$ described in Section 3.3. The main difference is that HTTP relies on a decimal encoding of message length, which requires us to generalize the grammar rules to account for 10 digits:

$$\{X_i \rightarrow j\ X_{i-1}\ (F_{i-1})^j \mid 0 \leq j \leq 9\} \tag{5.1}$$

Figure 5.2 contains code for automatically generating these rules, for $1 \leq i \leq n$. The outer loop iterates over the values of i. The second loop iterates over the j-values while the inner loop writes the symbol (F_{i-1}) j times. The rules

```
for i in `seq 2 $size`
do
   number=$i"_request"
   echo "X$number: '0' X$((i-1))_request {}" >> $OUTPUT
   for j in `seq 1 9`
   do
        echo -n "  | '$j' X$((i-1))_request " >> $OUTPUT
        for k in `seq 1 $j`
        do
          echo -n "F$number " >> $OUTPUT
        done
        echo "{}" >> $OUTPUT
   done
   echo "" >> $OUTPUT
done
```

Figure 5.2: Bash code to generate the rules of Equation (5.1).

$$\{F_j \rightarrow F_{j-1} \ldots F_{j-1} \mid 1 \leq j \leq n\}\ , \tag{5.2}$$

also generalize their counterparts from $G(n)$ to decimal encoding by replacing each F_j by ten F_{j-1} symbols instead of two. This is carried out by the code given at Figure 5.3. It remains to account for the rules

$$\{F_0 \rightarrow c \mid c \in \Sigma\}\ , \tag{5.3}$$

which are implemented by the code shown in Figure 5.4. Any symbol that may appear in the body of a HTTP message must be matched by the non-terminal symbol MCHAR. Since the lexer distinguishes among several symbols and returns a different token for each of them, the rule MCHAR needs to account for all of them. For the sake of simplicity we have used the start conditions of Flex, which allows us to change the rules available in the lexer at runtime. Thus, when the validator is done with the headers, all rules from the lexer but "carriage return", "new line", and "others" are disabled by using the function `void read_msg() {BEGIN(MSG);` where "BEGIN" is a Flex's function and "MSG" is a start condition defined within the lexer.

```
for i in `seq 2 $size`
do
  if [ $i -eq 2 ]
    then
    echo "F2_request: MCHAR MCHAR MCHAR MCHAR MCHAR MCHAR
       ↪ MCHAR MCHAR MCHAR MCHAR {}" >> $OUTPUT
  else
    request=$i"_request"
    echo "F$request: F$((i-1))_request F$((i-1))_request
      F$((i-1))_request F$((i-1))_request F$((i-1))_request
      F$((i-1))_request F$((i-1))_request F$((i-1))_request
      F$((i-1))_request F$((i-1))_request {}" >> $OUTPUT
  fi
done
```

Figure 5.3: Bash code to generate rules of Equation (5.2).

```
echo "X1_request: '0' CARRIAGE_RETURN NEW_LINE
   ↪ headers_request {read_msg();}" >> $OUTPUT
echo "  | '1' CARRIAGE_RETURN NEW_LINE headers_request {
   ↪ read_msg();} MCHAR {}" >> $OUTPUT
[...]
echo "  | '8' CARRIAGE_RETURN NEW_LINE headers_request {
   ↪ read_msg();} MCHAR MCHAR MCHAR MCHAR MCHAR MCHAR
   ↪ MCHAR MCHAR {}" >> $OUTPUT
echo "  | '9' CARRIAGE_RETURN NEW_LINE headers_request {
   ↪ read_msg();} MCHAR MCHAR MCHAR MCHAR MCHAR MCHAR
   ↪ MCHAR MCHAR MCHAR {}" >> $OUTPUT
echo "/* DEFINING TYPES OF DATA */" >> $OUTPUT
echo "MCHAR: CHAR {}" >> $OUTPUT
echo "      | CARRIAGE_RETURN {} " >> $OUTPUT
echo "      | NEW_LINE {}" >> $OUTPUT
```

Figure 5.4: Bash code generating rules for terminals, given by Equation (5.3)

5.2 Selector Fields

HTTP distinguishes between requests and responses, which requires choosing among two different grammars. As explained in Section 4.1 this idiom is context-free so it can be implemented with Bison without semantic actions. However, in HTTP there is no explicit field indicating whether the message is a request or a response. Nevertheless, the distinction can be automatically made by Bison as shown in Figure 5.5. Thus, Bison keeps reading the input without knowing if it is a request or a response until, at some point, only one of the rules "status_line" or "request_line" matches the input. From this point, Bison knows which type of message is reading.

```
axiom: status_line | request_line {YYACCEPT;}
status_line: HTTP VERSION SPACE STATUS_CODE SPACE
   ↪ STATUS_PHRASE CARRIAGE_RETURN NEW_LINE
   ↪ headers_response {}
request_line: METHOD SPACE request-uri SPACE HTTP VERSION
   ↪ CARRIAGE_RETURN NEW_LINE headers_request {}
```

Figure 5.5: Detecting whether the message is a request or a response.

5.3 Multiple Headers

A feature of the HTTP protocol we omitted to discuss so far is the fact that some header fields can only appear up to one time per message. This restriction is easy to implement by generating an additional parser that detects when one of these headers appears twice; as soon as this parser detects a duplicate field, the message is rejected. Figure 5.6 shows how this parser can be implemented with Flex. Although it contains semantic actions we consider it to be safe since the handwritten C code is not interacting with the user's input and it is not using dynamic memory.

The variables "http_date1", "http_date2" and "http_date3" used in Figure 5.6 are defined within Flex as the regular expressions matching the three different formats for writing a date. Figure 5.7 shows the definition of "http_date1" as an example of the three mentioned formats.

```
("C"|"c")("O"|"o")("N"|"n")("T"|"t")("E"|"e")("N"|"n")("T"|
   ↪ "t")"-"("L"|"l")("E"|"e")("N"|"n")("G"|"g")("T"|"t")(
   ↪ "H"|"h")":"("␣")*[0-9]+"\r\n"(.+"\r\n")*("C"|"c")("O"
   ↪ |"o")("N"|"n")("T"|"t")("E"|"e")("N"|"n")("T"|"t")"-"
   ↪ ("L"|"l")("E"|"e")("N"|"n")("G"|"g")("T"|"t")("H"|"h"
   ↪ )":"("␣")*[0-9]+"\r\n" {
  exit(-1);
}

("D"|"d")("A"|"a")("T"|"t")("E"|"e")":"("␣")*({http_date1
   ↪ }|{http_date2}|{http_date3})"\r\n"(.+"\r\n")*("D"|"d"
   ↪ )("A"|"a")("T"|"t")("E"|"e")":"("␣")*({http_date1}|{
   ↪ http_date2}|{http_date3}) {
  exit(-1);
}

("E"|"e")("X"|"x")("P"|"p")("I"|"i")("R"|"r")("E"|"e")("S"|
   ↪ "s")":"("␣")*({http_date1}|{http_date2}|{http_date3})
   ↪ "\r\n"(.+"\r\n")*("E"|"e")("X"|"x")("P"|"p")("I"|"i")
   ↪ ("R"|"r")("E"|"e")("S"|"s")":"("␣")*({http_date1}|{
   ↪ http_date2}|{http_date3}) {
  exit(-1);
}

"\n" {}
"\r" {}
"\r\n\r\n" {exit(0);}
. {}
```

Figure 5.6: Detecting when a field is duplicated.

```
wkday  ("Mon"|"Tue"|"Wed"|"Thu"|"Fri"|"Sat"|"Sun")
two_digit  ([0-9][0-9])
four_digit  ([0-9][0-9][0-9][0-9])
time  ({two_digit}":"{two_digit}":"{two_digit})
date1  ({two_digit}"␣"{month}"␣"{four_digit})
http_date1  ({wkday}",␣"{date1}"␣"{time}"␣GMT")
```

Figure 5.7: Definition of "http_date1".

The rule `"\r\n\r\n" {exit(0);}` at the end of Figure 5.6 makes the parser stop when the end of the headers sections is reached. Thus, the body of the message will not be parsed.

5.4 Evaluation

We perform a micro-evaluation of our input validator according to two criteria, namely accuracy and safety:

- For evaluating accuracy we run the validator on 50 valid HTTP packets captured using Wireshark. Our packets contain both requests and responses. All of the valid packets were accepted as valid.
  We have also hand-crafted invalid messages. For example, to violate message length constraints we modify the value of message length fields in otherwise valid messages. In all cases the validator rejected the message. Moreover, starting from two valid messages (a response and a request) we have exchanged fields that can only appear in one type of messages. Again, we observe that the resulting messages were rejected.
- To support the claim that our HTTP validator is safe, we ran the fuzzer afl on the generated code. After 6 hours, afl completed 14 cycles and explored more than 500 paths by doing more than 34 millions of executions without finding any bug.

We leave the evaluation of the performance of the generated validator for future work since it is out of the scope of this paper. However, we point out that the performance is entirely inherited from Bison and Flex.

6 Related Work

In this section we will focus entirely at recent effort in the context of parser generators for network protocols and put those works in perspective relative to ours. We start with BinPac [25] which is part of a larger project called "The Bro Network Security Monitor"[1]. Binpac implements a parser generator from scratch. We believe that the strength of our approach is in using a tool like Bison (in combination with Flex) that are widely used and have been around for a long time. By further abstaining ourselves to use semantic actions, the resulting parser implementation contains only generated C code that is free of potentially vulnerable user code. The above discussion applies equally to FlowSifter [23], Ultrapac [22], Nail [7] and Gapa [9] since they follow the approach of BinPac by re-implementing a parser generator from scratch. Among the previous parser generators, Gapa and Nail stands out. Gapa strives for safety by making a priority that the resulting parser is safe in the sense that the implementation is written in a memory-safe language. Even though using a memory-safe language is certainly better for safety it does not prevent runtime error (after all, dividing by zero remains always possible). Nail also aim at safety since it does not require the developer to write semantic actions as the authors identify them as potential vulnerabilities. We also prescribe semantics actions. Another line of work [12] relies on the use of the so-called attribute grammars, an extension of context-free grammar that equips rules with attributes that can be accessed and manipulated. For the parser generator, the authors use Bison and encode the attribute aspect of grammars through semantic actions. Since semantic actions are user-defined

[1]`https://www.bro.org/`

the resulting parser is potentially vulnerable. This departs from our no semantic action approach. None of the aforementioned work study the underlying formal language questions as we have done in this work. At best, the authors conjecture results but no rigorous evidence is given.

Deferred Proofs

A

Proof of Lemma 4. According to Proposition 3, if $L_\angle(a)$ is a finite intersection of context-free languages then the set $f_w^{-1}(L_\angle(a))$ is semilinear, which implies that it is a finite union of linear sets. Since the set is infinite, at least one of the linear sets must be infinite. We call this set S. Any element in S will have the form defined in 3.1 so define $\vec{x}, \vec{y} \in S$ as follows:

$$\vec{x} = \left(i_x, 1, \frac{b^{i_x} - 1}{b - 1}\right), \quad \vec{y} = \left(i_y, 1, \frac{b^{i_y} - 1}{b - 1}\right) \quad .$$

Let $\delta = i_y - i_x$, we write

$$\vec{y} = \left(i_x + \delta, 1, \frac{b^{i_x + \delta} - 1}{b - 1}\right) \quad .$$

Without loss of generality, assume $\delta > 0$ (for otherwise swap $\vec{x}$ and $\vec{y}$). Next define

$$\Delta = \vec{y} - \vec{x} = \left(\delta, 0, \frac{b^{i_x}(b^\delta - 1)}{b - 1}\right) \quad .$$

Let $\vec{z} = \vec{x} + 2\Delta$:

$$\vec{z} = \left(i_x + 2\delta, 1, \frac{b^{i_x} - 1 + 2b^{i_x}(b^\delta - 1)}{b - 1}\right) = \left(i_x + 2\delta, 1, \frac{b^{i_x}(2b^\delta - 1) - 1}{b - 1}\right) \quad . \tag{A.1}$$

Since the set S is linear we find that $\vec{z} \in S$ so we can write:

$$\vec{z} = \left(i_z, 1, \frac{b^{i_z} - 1}{b - 1}\right)$$

which means that $i_z = i_x + 2\delta$ obtaining:

$$\vec{z} = \left(i_x + 2\delta, 1, \frac{b^{i_x + 2\delta} - 1}{b - 1}\right) \quad . \tag{A.2}$$

Let us now derive a contradiction using (A.1) and (A.2). For this, we start by deriving the following equivalence:

$$b^{i_x}(2b^\delta - 1) = b^{i_x+2\delta} \text{ iff } 2b^\delta - 1 = b^{2\delta} \quad .$$

Applying the change of variable $t = b^\delta$ we obtain

$$2t - 1 = t^2 \text{ iff } t^2 - 2t + 1 = 0 \quad .$$

Solving the equation give us $t = 1$ iff $\delta = 0$. So the vector $\vec{z}$ obtained basing on the linearity of the set S will belong to S if and only if $\vec{z} = \vec{y} = \vec{x}$. We conclude that the set S is not infinite, in fact, if it is linear then it contains only one element. Thus the set $f_w^{-1}(L_\angle(a))$ can not be written as a finite union of linear set so the language $L_\angle(a)$ is not a finite intersection of context-free languages. $\square$

Glossary

American Fuzzy Lop Tool for fuzzing a program looking for bugs, hangs or any kind of memory leak. 12

Chunked messages Feature present in some protocols that allows a message to be sent as a series of small messages. Each small message sent is known as a *chunk*. 24

Content-Length Field Field present in many protocols whose value describes the size of some part of the message. 15

Context free Grammar A grammar is said to be context free it each production rule has a left part composed by just a non terminal symbol. 7

Grammar A *grammar* is a set of *production rules* describing how to form strings. 6

HTTP Hypertext Transfer Protocol. Application protocol for distributed, collaborative, hypermedia information systems. 3

Input validation Process of analyzing an input to detect whether it matches a given grammar or not. 12

LangSec Branch of investigation that aims to improve the safety of these systems by tackling the problem from a theoretical point of view. 7

Language A *language* is a set of words. 5

Parser Generator Program which takes a formal description of a grammar (e.g. in BNF) and outputs source code for a parser which will recognize valid strings obeying that grammar and perform associated actions. 8

Parsing Process of analyzing an input and filling a data structure with its contents according to a given grammar. 12

Selector field Field that appears in a number of protocols describing the format used to write some part of the message. 23

Semantic action Arbitrary handwritten code written in the target language and used by parser generators to increase the capabilities of the resulting parser. These actions are perform while parsing an input. 1

Bibliography

[1] DCE 1.1: Remote Procedure Call. http://pubs.opengroup.org/onlinepubs/9629399/toc.htm.

[2] Irony - .NET Language Implementation Kit. http://irony.codeplex.com/.

[3] LANGSEC: Language-theoretic Security. "The View from the Tower of Babel". http://langsec.org/.

[4] Start conditions. http://flex.sourceforge.net/manual/Start-Conditions.html.

[5] The catalog of compiler construction tools: Lexer and Parser Generators. http://catalog.compilertools.net/lexparse.html.

[6] R. Aitchison. *Pro DNS and BIND 10*. Apress, Berkely, CA, USA, 1st edition, 2011.

[7] J. Bangert and N. Zeldovich. Nail: A practical tool for parsing and generating data formats. In J. Flinn and H. Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 615–628. USENIX Association, 2014.

[8] N. Borenstein and N. Freed. Mime (multipurpose internet mail extensions): Mechanisms for specifying and describing the format of internet message bodies. RFC 1341, RFC Editor, June 1992.

[9] N. Borisov, D. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo. Generic application-level protocol analyzer and its language. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007*. The Internet Society, 2007.

[10] N. Borisov, D. Brumley, H. J. Wang, and C. Guo. Generic application-level protocol analyzer and its language. Technical Report MSR-TR-2005-133, Microsoft Research, February 2005.

[11] L. Cojocar, J. Zaddach, R. Verdult, H. Bos, A. Francillon, and D. Balzarotti. PIE: Parser identification in embedded systems. In *ACSAC 2015*, 2015.

[12] D. Davidson, R. Smith, N. Doyle, and S. Jha. Protocol normalization using attribute grammars. In M. Backes and P. Ning, editors, *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, volume 5789 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 2009.

[13] D. Davidson, R. Smith, N. Doyle, and S. Jha. Protocol normalization using attribute grammars. In *Computer Security–ESORICS 2009*, pages 216–231. Springer, 2009.

[14] A. Demenchuk. Beaver - a LALR Parser Generator. `http://beaver.sourceforge.net/spec.html`.

[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616, hypertext transfer protocol – http/1.1, 1999.

[16] Y. Filmus. Lower bounds for context-free grammars. *Information Processing Letters*, 111(18):895–898, 2011.

[17] S. Ginsburg. *The Mathematical Theory of Context-Free Languages*. McGraw-Hill, Inc., New York, NY, USA, 1966.

[18] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1st edition, April 1979.

[19] D. Kaminsky, M. L. Patterson, and L. Sassaman. Pki layer cake: New collision attacks against the global x. 509 infrastructure. In *Financial Cryptography and Data Security*, pages 289–303. Springer, 2010.

[20] M. Latteux. Intersections de langages algébriques bornés. *Acta Informatica*, 11(3):233–240, 1979.

[21] B. Lemke. The Dragon Parser Generator. http://www.lemke-it.com/litexec?request=pubdragondoc.

[22] Z. Li, G. Xia, H. Gao, Y. Tang, Y. Chen, B. Liu, J. Jiang, and Y. Lv. Netshield: massive semantics-based vulnerability signature matching for high-speed networks. In S. Kalyanaraman, V. N. Padmanabhan, K. K. Ramakrishnan, R. Shorey, and G. M. Voelker, editors, *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, New Delhi, India, August 30 -September 3, 2010*, pages 279–290. ACM, 2010.

[23] C. R. Meiners, E. Norige, A. X. Liu, and E. Torng. Flowsifter: A counting automata approach to layer 7 field extraction for deep flow inspection. In A. G. Greenberg and K. Sohraby, editors, *Proceedings of the IEEE INFOCOM 2012, Orlando, FL, USA, March 25-30, 2012*, pages 1746–1754. IEEE, 2012.

[24] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: A yacc for writing application protocol parsers. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 289–300. ACM, 2006.

[25] R. Pang, V. Paxson, R. Sommer, and L. L. Peterson. binpac: a yacc for writing application protocol parsers. In J. M. Almeida, V. A. F. Almeida, and P. Barford, editors, *Proceedings of the 6th ACM SIGCOMM Internet Measurement Conference, IMC 2006, Rio de Janeriro, Brazil, October 25-27, 2006*, pages 289–300. ACM, 2006.

[26] T. Parr. ANTLR4 GitHub Repository. http://www.antlr.org/.

[27] M. Patterson and D. Hirsch. Hammer parser generator. https://github.com/UpstandingHackers/hammer, May 2016.

[28] A. S. Sergey Bratus, Meredith L. Patterson. The bugs we have to kill. *;login:*, 40(4), Aug. 2015.

[29] M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, Boston, MA, 1996.

[30] L. D. Thomas. APG... an ABNF Parser Generator. http://www.coasttocoastresearch.com/apg.

[31] E. Vela and D. Lindsay. Dns protocol. http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf.

[32] M. Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.