On-Line Synthesis of Parsers for String Events

João Saffran

UFMG, Brazil

Haniel Barbosa

UFMG, Brazil

Fernando Magno Quintão Pereira

UFMG, Brazil

Srinivas Vladamani

Cyral Inc., USA

Abstract

A *string event* is the occurrence of a specific pattern in the textual output of a program. The capture and treatment of string events has several applications, such as log anonymization, error handling and user notification. However, there is no systematic approach to identify and treat string events today. This paper formally defines string events and brings forward the theory and practice of a general framework to handle them. The framework encompasses an example-based user interface to specify string patterns plus a grammar synthesizer that allows efficiently parsing such patterns. We demonstrate the effectiveness of this framework by using it to implement *Zhefuscator*, a system that redacts occurrences of sensitive information in database logs. Zhefuscator is implemented as an extension to the Java Virtual Machine (JVM). It intercepts patterns of interest on-the-fly and does not require interventions in the source code of the protected program. It can infer log formats and capture string events with minimal performance overhead. As an illustration, it is up to 14x faster than an equivalent brute-force approach, converging to a definitive grammar after observing less than 10 examples from typical logs.

Email addresses: `joaosaffran@gmail.com` (João Saffran), `hbarbosa@dcc.ufmg.br` (Haniel Barbosa), `fernando@dcc.ufmg.br` (Fernando Magno Quintão Pereira), `srini@cyral.com` (Srinivas Vladamani)

1. Introduction

We define a *string event* as the occurrence of some pattern of interest in the output of a program. Events can be produced automatically, for instance, as part of a log, or due to interactions between programs and users, as in a chat system. Examples of events of interest include the output of sensitive information that must be redacted or occurrences of notifications requiring immediate attention. Since there is no unified framework for capturing and treating string events, each software application handles them in specific ways. Nevertheless, the building blocks to construct such infrastructure are already in place: grammar synthesis [1, 2, 3] and function interception [4, 5]. This paper uses this body of knowledge to create a framework that handles string events for applications running in the Java Virtual Machine, as a way to anonymize sensitive data in logs.

The advent of Data Protection Laws in several countries [6, 7, 8] has bestowed great importance onto the capacity to treat and explain the output produced from black-boxes software (Section 2.1). However, this task is challenging (Section 2.2), since the chain of characters produced by such black boxes is unbounded. The efficient detection of string events requires the synthesis of a language's grammar from a potentially unlimited number of examples.

Contributions. We describe an on-line grammar synthesis algorithm that incrementally over-approximates a grammar for any language (Section 3). Our grammars fit into a format henceforth called *Heap-Chomsky Normal Form* (Section 3.1), a restriction of Chomsky Normal Form. Heap-CNP grammars recognize, indeed, a regular language; hence, they can be represented as regular automata. Therefore, these grammars are never ambiguous and admit LL(1) parsers. LL(1) parsers can run in linear time on the input size [9], and admit formal proofs of correctness, as recently shown by Edelmann *et al* [10]. We have implemented a system that uses our theory to anonymize sensitive information in logs, while treating the log generator as a black-box (Section 4).

Summary of Results. We implemented the above techniques in a tool, the

Zhefuscator, that redacts sensitive data in SQL queries found in logs created by Java-based systems. Zhefuscator implements a form of reactive programming, which, in the words of Ramson and Hirschfeld [11, p12-2], "*consist of two parts: detection of change and reaction to change.*" Detection is the topic of Section 3, whereas reaction is discussed in Section 4. In Section 5 we evaluate properties of this tool. We summarize the results of this evaluation as follows:

- Section 5.1 shows that we can construct a grammar for typical database logs (MySQL and PostgreSQL) after observing less than 10 examples of outputs. Exercising Zhefuscator on more complex logs, e.g., files in the `/var/log` directory of MacOS, then convergence requires more examples, but still a small proportion compared to the size of the log. Our worst case performance required 170 examples in a log containing 6,579 entries.

- Section 5.2 demonstrates that our on-line approach can be up to 14x faster than a brute-force event detection system that does not synthesize grammars. Performance is important because our techniques are meant to be used in tandem with a running application. If it's overhead is prohibitive, then chances are that users would not employ it. Furthermore, the more complex is the language that generates the logs, the larger is the improvement of Zhefuscator over its trivial counterpart.

- Section 5.3 shows that our event handler does not add statistically significant overhead onto 11 out of 15 benchmarks from DaCapo [12], when building a grammar for the entire output of each benchmark. Furthermore, in the four benchmarks where overhead is noticeable, in only one case (`luindex`), it reaches 50%.

Software. Zhefuscator is open software, distributed through the GPLv3 license, and publicly available at `https://github.com/lac-dcc/Zhe`. As of today, it is embedded in products of at least one data-protection company: `Cyral Inc.` (`https://www.cyral.com/`).

2. Motivation and Challenges

Section 2.1 provides motivation for the automatic treatment of string events and Section 2.2 discusses the challenges related to this endeavor.

2.1. String Events in the Context of Data Protection

In this paper, we call a *generator* a computer program that produces a string t_i at each time slot i. Software that produces logs, like database servers and operating systems, or content providers, such as e-mail and news services, can be understood as generators. Usually, when part of the output of a generator is analyzed, this analysis is performed *off-line*, i.e., after such text has been produced and stored. However, there are situations in which such analysis must be carried out *on-line*, i.e., while it is being produced.

Data protection laws are one of the forces driving the need for on-line analyses. As an example, the *General Data Protection Regulation* (GDPR)[1], valid in the European Economic Area since 2016, requires companies to anonymize personal data, whenever this data is amenable to be used in ways not foreseen by the company's terms of use [8]. Discussions involving the European GDPR have inspired similar laws in other regions, such as the *California Consumer Privacy Act*[2], taking effective since January of 2020 in the American state of California, and the *General Law of Personal Data Protection* [13], taking effect in August of 2020 in Brazil.

Data protection laws bear an impact on log generation, since logs should not leak personal data. However, many software systems have been designed and implemented before the advent of these laws. Adapting these systems to accommodate privacy is an expensive endeavor inasmuch as such adaptation entails modifications in legacy code. However, in this paper, we demonstrate that it is possible to filter logs while they are produced, by projecting this problem onto the general framework of string events. The appearance of sensitive

[1] `https://eugdpr.org/`
[2] AB-375 Privacy: personal information: businesses.(2017-2018)

information in a log is a string event. Given the right framework, this event can be detected and treated on-the-fly. Nevertheless, the creation and deployment of this framework involves theoretical and practical challenges, which we discuss in the next section.

2.2. Event Recognition: Challenges

Handling string events while treating the event generator as a black box is challenging for three reasons, which we discuss in this section. To make this presentation more concrete, we relate the challenges to the following real-world problem, which Zhefuscator solves:

Example 1 (Concrete Problem). *Consider a log-producing database server running on the Java Virtual Machine. The grammar that describes the log syntax is unknown. Logs might contain SQL queries. Some queries contain sensitive information. Design a system that intercepts strings in the log, before they are printed, and anonymizes particular literals embedded in the SQL queries. A literal is any constant in the SQL query, e.g., integer values, quoted strings, and so on. The users specifying which data must be elided are not necessarily programmers.*

Challenge 1 (Grammar Synthesis). *How to efficiently identify SQL queries within the log, when the log grammar is not known?*

Each generator has its own log format. Part of this log uses the SQL syntax. If we call L the language of log strings, then each string $t \in L$ might contain SQL and non-SQL substrings, as Example 2 shows. In this combination of two languages, we call L the *host language* and SQL the *event language*.

Example 2. *Figure 1 shows part of a log taken from an actual application (literals have been replaced with fake surrogates). Strings in the target language, SQL, are shown in red. This log contains five examples, one per line. Each example is produced by the generator in successive moments in time. A solution to Challenge 1 amounts to synthesizing a parser for this log.*

```
      82 Query SELECT * FROM Clts WHERE SSN='078-05-1120' 0
      83 Init DB grossi
11 8:02 84 Query SELECT * FROM Byrs WHERE name='J.Generics' 1
      85 Connect mysqldumpuser@localhost on
12 8:11 86 Query DELETE * FROM Clts WHERE name='J.Generics'
```

Figure 1: Snippet of log with five examples.

Requiring a parser for the host language L would complicate the deployment of the obfuscator, as this requirement forces users to be aware of L's format. It is possible to separate host and event languages via a brute-force approach considering every token of the host language as the potential starting point of a sentence in the event language. However, as we show in Section 5.2.1, this approach does not scale well with the number of tokens in the string $t \in L$. The generator produces an infinite stream of strings; hence, Challenge 1 involves inferring a grammar *in the limit*, that is, from an infinite number of examples. Even though this problem is undecidable even for regular or superfinite languages, as shown by Edward Gold [14], we can efficiently build unambiguous grammars that recognize, in a scalable manner, the subset of the host language defined by all examples seen up to a point. We detail this process in Section 3.

Challenge 2 (Interface). *Which interface should users who are not programmers use to specify sensitive patterns?*

Obfuscating the log in Figure 1 requires knowing which SQL literals must be redacted. It is up to users of the obfuscator to specify such literals. However, information can be sensitive when used in some types of queries, and innocuous when used in others, as Example 3 illustrates.

Example 3. *Consider an instance of the concrete problem (Ex. 1) that requires redacting occurrences of* `SSN` *in the pattern:* `SELECT * FROM Clts WHERE SSN='?'`*. Occurrences of* `SSN` *in other patterns, such as* `DELETE FROM Clts WHERE SSN='000-00-0000'`*, must be preserved.*

When building Zhefuscator, we first considered defining a domain specific language (DSL) to let users specify patterns to obfuscate[3]. Our experience shows that this option is not ideal: it prevents users of the log-producing system—usually non-programmers—from using our tool. In Section 4.1 we describe a programming-by-examples approach, inspired by the Parsimony IDE [2], which provides users a simple but effective interface for specifying sensitive data. From this interface, we derive an *event grammar*, that specifies which queries should have their literals redacted. This grammar feeds Zhefuscator with knowledge to distinguish sensitive and innocuous queries. It will redact every literal within the former group, while preserving occurrences of the same literal in the latter. What distinguishes one type of query from the other? Syntax! And this syntax is specified by the user, when building the event grammar (following steps yet to be introduced in Section 4.1). Notice that the user will never have to deal with the format of the tokens, e.g., the SSN format in Example 3. All that she must do is to highlight examples of sensitive queries.

Challenge 3 (Engineering). *How to intercept the generator's output without changing its implementation?*

Challenge 3 is an engineering problem specific to the log-generation application. In Section 4.2 we describe a solution for systems running on the Java Virtual Machine. In contrast to our solutions to the other challenges, the approach adopted in Section 4.2 is not general—a natural consequence of the fact that Challenge 3 is technology specific.

3. First Challenge: Grammar Synthesis

Context-free grammars.. Let $G = \langle S, N, T, P \rangle$ be a *context-free grammar*, with *non-terminals* N, *terminals* T, a *start symbol* $S \in N$ and *production rules* $P \subseteq N \times (N \cup T)^*$. The set $V = N \cup T$ is G's *vocabulary*. A *sentence* is a string

[3] This language, which we currently call `ZheLang` is publicly available at `https://github.com/joaosaffran/zhe-lang`. `ZheLang`, when used as a tool to specify string events, is more expressive than the techniques that we explain in this paper. However, it requires some knowledge of parsing and Boolean logic, which our example-based approach avoids. It is our intention to describe `ZheLang` in future work.

of terminals. A sentence t is *generated* from a grammar G if there is a sequence of applications of production rules that transforms S in t. This sequence of applications is called a *derivation*. In a *leftmost derivation* the leftmost non-terminal is always reduced first. The concatenation of strings p and q is $p \bullet q$. If t and t' are strings, and t is a substring of t', we write $t \in subs(t')$. A context-free grammar G is in *Chomsky normal form* if all of its production rules are of the form $A ::= BC$, $A ::= \mathtt{a}$, or $S ::= \epsilon$, in which A, B and C are non-terminals, $\mathtt{a}$ is a terminal and S is the start symbol. The language that G *recognizes*, denoted $lang(G)$, is the set of all strings generated from G. Given a string t, it can be *generated ambiguously* by a grammar G if G allows two different derivations that generate t. If G generates any string ambiguously, then G is *ambiguous*.

String events.. Let L be a language. A *text* over L is a sequence of strings $t_0, t_1, \ldots$, such that $t_i \in L$. A *generator* for L is a Turing Machine that generates this text. We say that t_i is the *text generated at time* i. We allow $t_i = t_j, i \neq j$. No function from time to strings is assumed; however, we assume that on the limit the text covers L. Notice that the existence of a generator, coupled with this last assumption, implies that L is recursively enumerable. From these notions, we define string events as follows:

Definition 1 (String Event). *A string event $\langle s, G_e, t_i, L\rangle$, parameterized by a context-free grammar G_e, which we call the event grammar, occurs at time $i, i > 0$, on the text t_i produced by a language L, which we call the host language, if there exists $s \in lang(G_e)$, such that $s \in subs(t_i)$.*

Example 4 (String Event). *Let the host language L be the language that contains the string representations of every natural number, and only these strings. Let the event grammar G_e be a grammar that recognizes palindromes with more than one digit on the language of positive decimal numbers. Tokens, in this case, are single digits. Consider the text over L in which $t_i =$ "i", for $i \in \mathbb{N}^+$, i.e., the text is* "1", "2", $\ldots$, "10", "11", $\ldots$. *A string event occurs on $t_{1223} =$ "1223", because* "22" $\in subs($"1223"$)$ *is a palindrome.*

3.1. Synthesizing the Grammar for the Host Language

As seen in Definition 1, capturing string events involves detecting occurrences of substrings produced by a context-free grammar G_e within text pertaining to a recursively enumerable language L. We call a grammar G that recognizes L, i.e., $L = lang(G)$, *the host grammar*. In the context of handling string events from a black-box event generator, as explained in Section 2.2, we cannot assume that the host grammar is known. Thus, it is necessary for L to be discovered while string events are being captured. Moreover, only examples of strings that are part of the language, denoted "positive examples", are available to do so. As demonstrated by Gold [14], this problem is undecidable for most classes of languages, including context-free.

3.2. On-Line Grammar Synthesis

The intuition behind Gold's result is simple: since L is being determined by positive examples, whichever grammar has been synthesized up to time m can fail to parse an example $t_n, n > m$. However, up to time m, it is always possible to build a grammar G_m that recognizes $t_1, \ldots, t_m$: in the worst case, G_m contains m production rules, one for each string $t_i, 1 \leq i \leq m$. Therefore, Gold's conclusions indicate that a grammar for L should be recognized by an *on-line* algorithm, which builds successive grammars $G_1, \ldots, G_m$ up to time m, such that $\{t_1, \ldots, t_m\} \subseteq lang(G_m), 1 \leq i \leq m$.

The Language Separation Problem.. In this paper, we assume that the event grammar G_e that encodes string events is known[4]. Therefore, to capture string events we must be able to distinguish occurrences of strings from $lang(G_e)$ within the input text. From these observations, we define the *language separation* problem as follows:

---

[4]Section 4.1 discusses the approach that we have chosen to let users specify events. Notice that users do not need to provide G_e explicitly: they specify events through examples valid in G_e, which is assumed to be already known by the language synthesis system.

```
1 # An infinite sequence of strings:
2 val text: String stream
3
4 # Parameters of the implementation
5 val TOKENIZE: String -> Token list
6
7 fun add_example
      (tokens: Token list, current_grmr: Grammar) =
8   if successfull_parse(current_grmr, tokens)
9   then current_grmr  # Success!
10  else
11   let
12    val new_grammar = fill_holes(tokens)
13   in
14    merge(current_grmr, new_grammar)
15   end
16
17 fun build_grammar((example::text): String stream, grammar: Grammar) =
18  let
19   val new_grammar = add_example(TOKENIZE example, grammar)
20  in
21   build_grammar(text, new_grammar)
22  end
23
24 # Start language separation with the simplest sketch grammar:
25 val grammar:Grammar = build_grammar(text, R1 ::= ε)
```

Figure 2: The language separation procedure.

Definition 2 (Language Separation Problem). *Let $T = \{t_1, \ldots, t_m\}$ be a set of strings pertaining to an unknown host language L. Given $G_e = \langle s_e, N_e, T_e, P_e\rangle$, find grammars $G_m = \langle s_m, N_m \cup N_e, T_m \cup T_e, P_m \cup P_e\rangle$ such that $\{t_1, \ldots, t_m\} \subseteq lang(G_m), 1 \le i \le m$.*

Our language separation algorithm is outlined in Figure 2 as a program written in ML syntax. The entry point of this program is function `build_grammar`, which receives `text`, the infinite sequence of strings $t_1, t_2, t_3, \ldots$ corresponding to the language to be recognized. The function `build_grammar` operates in a classic *counterexample-guided inductive synthesis* (CEGIS) [15, 16] loop, in which a learner proposes solutions and a verifier checks them, providing counterexamples for failures. In our context the learner produces grammars that recognize the examples seen so far and the verifier checks whether they can generate the subsequent examples.

For each string `example` in the `text` stream, `build_grammar` refines a `grammar` that recognizes `example`. Thus, the `grammar` variable at line 25 of Figure 2 refers to the grammar that recognizes `text` on the limit, that is, after an infinite number of examples have been produced. Notice, nevertheless, that even though `build_grammar` never halts, it produces a new grammar each time it is recursively invoked (Line 21). Function `build_grammar` uses an auxiliary routine `add_example`. This procedure checks if the current grammar can parse a string in `text` (Line 8). If it can, nothing else happens (Line 9). However, if parsing fails, then `add_example` refines the current grammar (Lines 10-15). The next section describes this refinement.

On the `TOKENIZE` Function. In this work, we do not focus on synthesis of lexers. Instead, we rely on a predefined lexer, the `TOKENIZE` function, which transforms examples into sequences of tokens. Said function is invoked at Line 19 of Figure 2. Examples of tokens are $int = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ and $time = int : int$. Our solution to language separation (Fig. 2) is parameterized by this function. The tokenizer might bear an impact on the number of examples necessary to synthesize a definitive grammar for the host language. It can also modify the speed of the algorithms that we shall discuss in the next section. In Section 5.2.4 we analyze these two facts empirically.

3.3. Grammar Synthesis from Examples

Whenever `build_grammar` fails for a new example t_i, we use the function `fill_holes` to produce a grammar G_i that recognizes it. This function is invoked at Line 12 of Figure 2, and its implementation is given in Figure 3. We shall be explaining this code in the rest of this section. Notice that the auxiliary function `build_hcnf` contains comments mentioning two "Rules". These rules will be the explained shortly.

```
# Build a grammar in Heap-CNF that recognizes "tokens"
fun build_hcnf(n:int, [token]: Token list): Grammar =
      R_n ::= token        # Rule 1
    | build_hcnf(n:int, token::Rest: Token list): Grammar =
      R_n ::= R_2n R_2n+1   # Rule 2
      R_2n ::= token       # Rule 1
      build_hcnf(2×n+1, Rest)  # R_2n+1 ::= ...

fun fill_holes(tokens: Token list): Grammar = build_hcnf(1, tokens)
```

Figure 3: The grammar synthesizer.

To build a parser for the host language L, thus solving the Language Separation Problem, we apply a programming-by-examples [17] approach. For each example t_i we synthesize a grammar G_i that generates it. Then we merge this grammar into a previously synthesized grammar G that generates the previous examples, thus obtaining a new grammar G such that $\{t_1, \ldots, t_i\} \subseteq G$. Each grammar G_i synthesized for generating the given example t_i is in Heap-CNF, a restrictive form of CNF defined as follows:

Definition 3 (Heap-CNF). *A Heap-CNF grammar has restrictions on the non-terminals and the production rules. Non-terminals are R_1, R_2, $R_3, \ldots,$ R_{2^n-2}, R_{2^n-1}, for some arbitrary n. The allowed production rules are*

- $R_{2^{k+1}-2} ::= \mathtt{a}$,
- $R_{2^k-1} ::= R_{2^{k+1}-2}R_{2^{k+1}-1}$, *and*
- $R_{2^k-1} ::= \mathtt{a}$

in which $\mathtt{a}$ *is a terminal and* $k \in \{1, \ldots, n\}$. *Since non-terminals are numbered in the same way as data in the heap data structure, we call this restricted version of Chomsky Normal Form, Heap-CNF.*

We restrict ourselves to Heap-CNF grammars for three reasons. First, given two grammars in this format, it is possible to merge them in linear time on the number of non-terminals, thus producing a new Heap-CNF grammar, as we will see in Section 3.3.1. Second, they are not ambiguous (Theorem 5). Finally, they admit LL(1) parsing (Theorem 7). We shall leverage the two latter properties to demonstrate that our solution to the language separation problem is correct. The two latter properties are a consequence of Heap-CNF grammars encoding regular languages. Indeed, a Heap-CNF language can be described by a regular automaton. Nevertheless, we shall call them grammars, as we are using them to synthesize parsers.

The grammar G_i is built by successively increasing its vocabulary and by "filling holes", i.e. adding production rules to a partial grammar while $t_i = t_i^1 t_i^2 \cdots t_i^n$ is traversed. Initially the partial grammar contains only the starting non-terminal R_1 and no terminals or production rules. At each iteration, the grammar is augmented to generate the first token in the sequence, which is then removed from it. The grammar is also expanded so that it can be further augmented to generate the remaining tokens. This is represented by the application of the following two expansions, which add production rules to G_i:

$$\begin{aligned} R_k ::= ? &\overset{\text{(Rule 1)}}{\Rightarrow} & & R_k ::= t_i^j, \text{ if } t_i^j \text{ is the last token of } t_i \\ & & & \text{or } G_i \text{ contains } R_{k+1} \\ R_k ::= ? &\overset{\text{(Rule 2)}}{\Rightarrow} & & R_k ::= R_{2k}R_{2k+1},\ R_{2k} ::= ?,\ R_{2k+1} ::= ? \\ & & & \text{otherwise} \end{aligned}$$

in which R_k is a non-terminal not yet associated with a production rule.

The first rule allows the consumption of the first remaining token in t_i. It can be applied when the respective non-terminal is not the last one in G, except if there is only one token left to be consumed. Otherwise, the second rule is applied, which introduces two new non-terminals in the grammar: one for consuming the first remaining token, via Rule 1, and another to continue the process for generating the subsequent tokens in t_i. This process continues until the grammar that parses t_i is obtained. Function `fill_holes` (Figure 3), which implements this procedure, takes as input a sequence of tokens t_i and yields a grammar G_i in Heap-CNF that consumes said sequence, as stated below.

Theorem 1 (Correctness). *Function `fill_holes` (Fig. 3) produces a grammar G_i that recognizes an example $t_i = t_i^1 \cdots t_i^n$ in n steps with $2n - 1$ non-terminals.*

Proof 1. *The proofs of all lemmas and theorems of this paper are provided as supplementary material.*

Example 5. *Figure 4 illustrates `fill_holes` "stepwisely" building a grammar that generates the first example from Figure 1. Note that the characters '82' and '0' were tokenized to int and the SQL query to s_e. At Step 1 the partial grammar consists only of the starting non-terminal R_1. Given that the list of tokens to generate contains more than one element, `fill_holes` applies rule 2 (line 5 of Figure 1), producing the partial grammar in Step 2 with two new undefined*

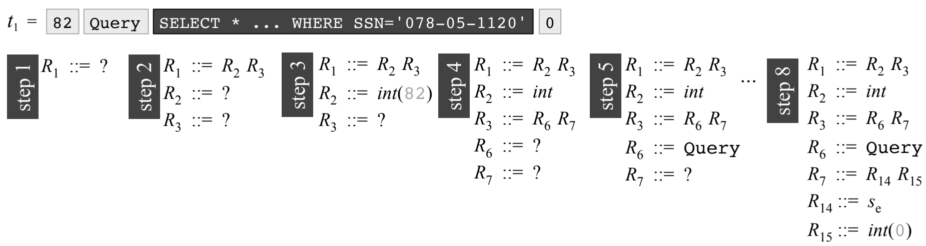

Figure 4: Grammar inference via `fill_holes`.

non-terminals. Rule 1 is then applied to generate the first token in the list (line 6), producing the grammar in Step 3. The ***`fill_holes`*** *algorithm proceeds to recursively build a grammar to generate the remaining tokens, applying rules 2 and 1 in sequence, until it reaches the case when there is only one token to be generated. This triggers a final application of rule 1 (line 3), yielding the grammar in Step 8.*

The grammar synthesis has the following properties:

Lemma 1 (`fill_holes` yields Heap-CNF). *Given an example t_i, the resulting grammar G_i produced by* ***`fill_holes`*** *that generates t_i is in Heap-CNF.*

Theorem 2. *Given an example $t_i = t_i^1 t_i^2 \cdots t_i^n$, the resulting grammar G_i produced by* ***`fill_holes`*** *is such that $R_{2^n-1} ::= t_i^n$.*

Theorem 2 and Lemma 1 perfectly define the structure of grammars produced by `fill_holes`, as stated below:

Corollary 1. *Given an example $t_i = t_i^1 t_i^2 \cdots t_i^n$, the resulting grammar G_i produced by* ***`fill_holes`*** *is such that*

$$\begin{aligned} R_{2^{k+1}-2} \quad &::= \quad t_i^k,\ k \in \{1, \ldots, n-1\} \\ R_{2^k-2} \quad &::= \quad \begin{cases} R_{2^{k+1}-2} R_{2^{k+1}-1} & k \in \{1, \ldots, n-1\} \\ t_i^n & k = n \end{cases} \end{aligned}$$

Figure 5 illustrates the structure of a derivation of a string of 5 tokens from the Heap-CNF grammar that would be produced by `fill_holes`.

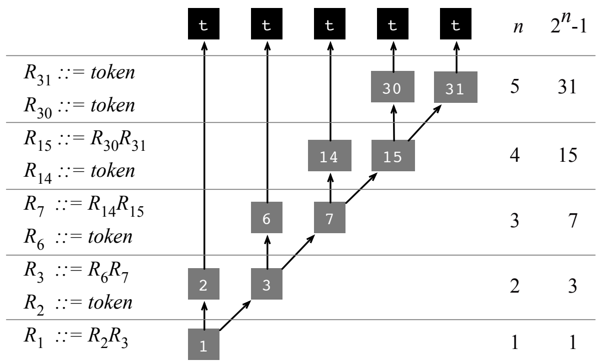

Figure 5: The format of the leftmost derivation tree of a Heap-CNF grammar to produce a string with 5 tokens.

The s_e Token.. Throughout this paper, we have been treating the string event as a single token. As an example, in Figure 4, we represent it as s_e, the starting symbol of the even grammar G_e. However, the string event is not a single token; rather, it is a complex sentence pertaining to $lang(G_e)$. Consequently, the sentence represented by s_e does not even need to be formed by the same tokens as the host language. In other words, the tokens in s_e do not, necessarily, need to be recognizable by the `TOKENIZE` function adopted in our implementation of `build_grammar`. That function recognizes tokens from the host language, not from the event language.

Recognizing s_e is necessary when augmenting the current grammar via the `fill_holes` routine. To perform this task, we resort to a brute-force approach: we try to recognize the largest subsentence $s \in subs(t_i)$ within the active example t_i so that $s \in lang(G_e)$ (See Definition 1). This heuristic is $O(|G_e||t_i|)$, because we can imagine a scenario in which every prefix of t_i is also a prefix of some—incomplete—sentence in $lang(G_e)$.

Nevertheless, the brute-force search tends to fail already in the first token, at least in the setting in which we use it: redaction of SQL queries embedded in an unknown language. For instance, consider that the event language is a subset of SQL performing the so called *CRUD* operations, i.e., *SELECT*, *UPDATE*, *CREATE* and *DELETE*. Only sentences that start with one of these four tokens can be part of the event language. Therefore, as soon as the brute-force algorithm stumbles on a different token, it can stop searching immediately. Typical data-representation languages, such as YAML, XML or JSON bear similar properties, meaning that valid sentences in these languages start with a limited number of token combinations.

Furthermore, it is important to consider that the brute force approach is only necessary to augment the current grammar. Whenever line 8 of Figure 2 succeeds, no brute-force heuristics are used. As we shall see in Section 5.1, in a typical SQL or PostgreSQL log, four to nine samples—among an unbounded number of examples—are enough to give us a definitive grammar to solve the language separation problem.

3.3.1. Merging grammars

Once `fill_holes` produces a grammar G_i for a new example t_i, this grammar is merged into the current grammar G, as it can be seen at Line 26 of Figure 2. We define the merging of two Heap-CNF grammars as follows:

Definition 4 (Grammar Merging). *Let* $G = \langle R_1, N, T, P \rangle$ *and* $G_i = \langle R_1, N^i, T^i, P^i \rangle$ *be two Heap-CNF grammars.* $G' = \langle R_1, N \cup N^i, T \cup T^i, P \cup P^i \rangle$ *is the grammar that merges* G *and* G_i*.*

Our goal is that G' be also in Heap-CNF and still generates $lang(G)$ and $\texttt{lang}(G_i)$. This is achieved by combining the production rules of G and G_i, i.e. by defining the production rules of G' as $P \cup P^i$. Since G and G_i are in Heap-CNF they have the same non-terminals up to R_{2^k-1}, in which R_{2^k-1} is the maximum non-terminal in either G or G_i. So each non-terminal in G' up to R_{2^k-1} will generate the combined tokens of G and G', while every non-terminal beyond R_{2^k-1} has the same production rules of the grammars it comes from, thus generating the same strings that grammar generates.

Example 6. *Figure 6 shows the grammars produced for the first, third and fifth lines of Figure 1. The tokenization applied to the characters is illustrated in the derivation trees. Each grammar but the first is formed by the merging of a* *current grammar plus the grammar newly built to match the latest example in the log. Since grammars share non-terminals up to a given index, the union of the production rules has the effect of adding tokens as alternatives to a given non-terminal. For example, in the grammar that generates the first example,* R_{15} *generates the terminal* int*, while in the second grammar it generates the non terminals* $R_{30}R_{31}$*, so merging these two grammars results in a grammar in which* $R_{15} ::= R_{30}R_{31} \mid \text{int}$*. This observation ensures that the merged grammar will be able to generate both the examples generated by the two original grammars.*

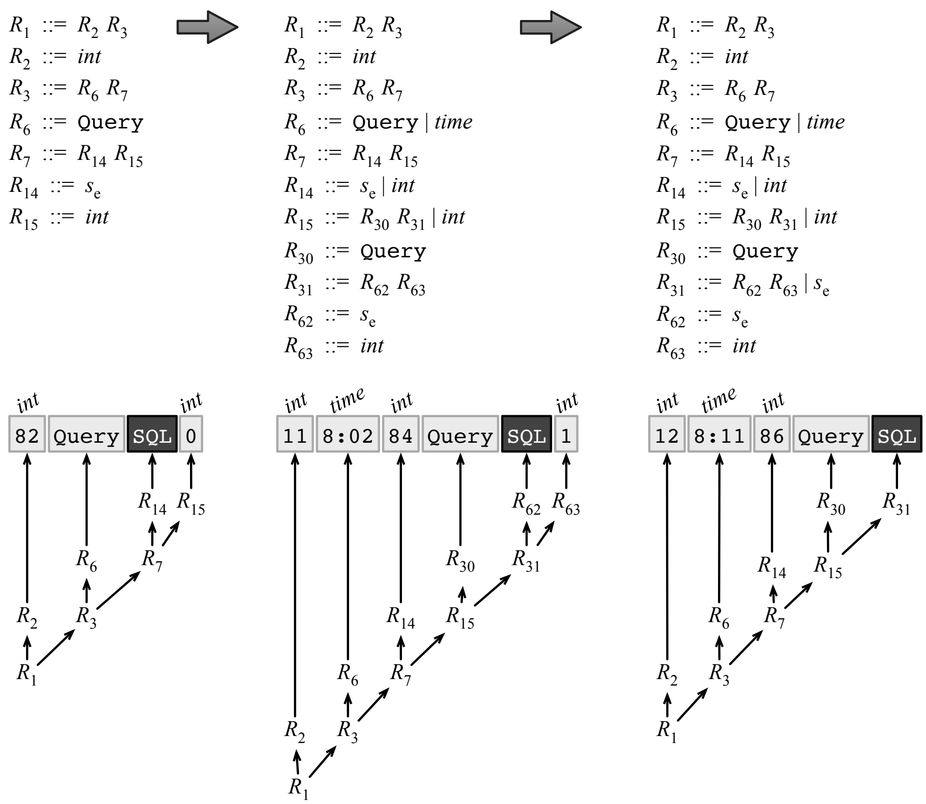

Figure 6: Grammars produced from the examples in Figure 1.

Notice that the final grammar that results from merging multiple grammars recognizes a language that is larger than the union of all the examples seen thus far. For instance, the final grammar in Example 6 recognizes the string "*int* `Query` s_e `Query` s_e", which encodes two SQL queries.

Lemma 2 (Merging). *If G is the grammar that results from merging two Heap-CNF grammars G' and G", then G is Heap-CNF, and $lang(G') \cup lang(G")$ $\subseteq$ $lang(G)$*

Theorem 3. *The procedure `build_grammar` (Fig. 2) constructs grammars in Heap-CNF.*

Theorem 4 (Semantics). *Let $G_1, G_2, \ldots, G_n$ be the grammars constructed by function `build_grammar` (Fig. 2) for input strings $t_1, t_2, \ldots, t_n$. Grammar $G_i, 1 \leq i \leq n$ recognizes every input $t_i, 1 \leq i \leq n$.*

Lemma 3 (Size Complexity). *Let G_n be the grammar constructed by function `build_grammar` (Fig. 2) after observing inputs $t_1, t_2, \ldots, t_n$. The size of G_n is $O(N)$, where N is the number of tokens in $t_1, t_2, \ldots, t_n$.*

Theorem 5 (Determinacy). *Let G_n be the grammar constructed by function `build_grammar` (Fig. 2) after observing inputs $t_1, t_2, \ldots, t_n$. G_n is not ambiguous.*

Corollary 2 (Time Complexity). *Let G_n be the grammar constructed by function `build_grammar` (Fig. 2) after observing inputs $t_1, t_2, \ldots, t_n$. G_n recognizes $t_i, 1 \leq i \leq n$ with O(N) derivations, where N is the number of tokens in t_i.*

3.3.2. Limitations: False Positives

The procedure `build_grammar` synthesizes a grammar G that over-approximates the host language L. By over-approximation, we mean that there might exist strings that belong to *lang*(G), but that do not belong to L. This observation leads to the notion of *false positives*, which we define as follows:

Definition 5 (False Positive). *Let G_n be the grammar synthesized by `build_grammar` after observing n examples from the host language L. We call a false positive an example t_{fp} such that $t_{fp} \notin L$, $t_{fp} \in lang(G_n)$ and t_{fp} contains a string event (Definition 1).*

Example 7 (False Positive). *Consider the third grammar seen in Figure 6. This grammar recognizes four strings with four tokens: (i)* int ***query*** s_e int*; (ii)* int time s_e int*; (iii)* int ***query*** int int*; (iv)* int time int int*. Only sentences in the format (i) fit the examples seen in Figure 6. Sentence (ii) does not correspond to any example and contains a string event (marked by s_e).*

A false positive will lead to the treatment of a string event that, in principle, should be ignored. In the context of this work, the system to be described in the next section will redact information that is not sensitive. Such action is innocuous in the settings where said system is deployed. Furthermore, the logs that we evaluate in Section 5 never lead to false positives. Therefore, we have decided to take no account of false positives in this work. There are two more reasons that led us to ignore them. First, the number of events is unbounded; hence, strings that are false positives up to a certain instant in time might become true positives later. Second, we follow Parsimony's approach [2] when specifying the event grammar, as we discuss in Section 4.1. Parsimony does not support negative examples—a potential way to avoid some false positives.

4. Case Study: the Zhefuscator

We have used the grammar inference techniques discussed in Section 3 to implement a system that redacts sensitive information present in program logs. This system is called the Zhefuscator. Zhefuscator receives as input an *event language*, given as a grammar G_e, and a running instance of the Java Virtual Machine (JVM). Notice that the Zhefuscator does not need the source code of the program under execution in the JVM—this program is treated as a black box. Figure 7 provides an overview of this tool. In the rest of this section, we discuss particular details of its implementation.

4.1. Second Challenge: User Interface

Zhefuscator is meant to be used by professionals who are not necessarily programmers. Therefore, to simplify the task of specifying string events, we provide users with an example-based interface, in which users select substrings

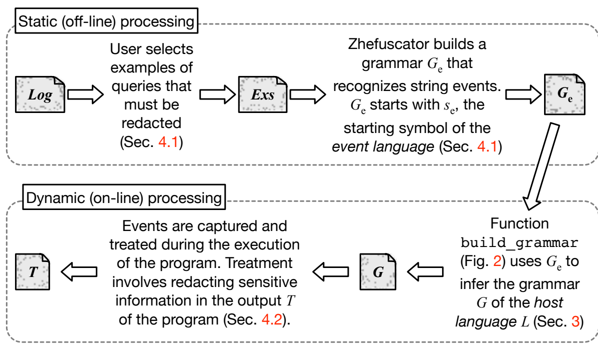

Figure 7: Zhefuscator: event handler for the JVM.

from log entries, and a base grammar, G_b, which will be used as a basis for building the event grammar G_e. As a helper to the user, once a substring l is marked for being redacted, all other substrings in the log entries that are recognized by the same rule from G_b which recognizes l are highlighted. That rule is then added to G_e. Through this iterative procedure, the event grammar G_e is built from the basic grammar G_b according to the example substrings selected by the user. Note that users do not deal directly neither with the basic nor with the event grammar: they only deal with textual examples, from which they must choose samples. Currently, we use the SQL grammar as the base grammar, but our implementation is not specific to any grammar. Just keep in mind that if the base grammar does not recognize the example substring selected to be redacted, this example will be ignored in the final event grammar. To determine which data must be redacted, users follow the procedure `markup`, in Figure 8.

Theorem 6 (Markup). *Grammar G_e produced by* `markup` *(Fig. 8) recognizes a subset of $lang(G_b)$ or the empty language.*

As seen in the proof of Theorem 6, grammars G_e and G_b start with the same initial symbol s_e. This symbol is used to compose the instance of the language separation problem (Definition 2) that routine `build_grammar` solves.

Procedure `markup`(G_b : *Base Grammar*, L_f : *Log Example*)

1. Let G_e be an empty grammar.
2. The user selects a literal l to be redacted, which occurs in a given example from L_f.
3. Zhefuscator uses the event grammar to extract the largest string $s \in lang(G_b)$ that contains l.
4. A grammar G'_e, formed with the production rules of G_b necessary to recognize s is constructed.
5. The terminal symbol T that recognizes l is marked to be redacted, T must occur within the rule that recognizes s.
6. G_e is augmented with the rules in G'_e.
7. Every sentence $s' \in L_f$ that G_e recognizes is highlighted.
8. If there are more literals in L_f that must still be obfuscated, the user goes back to step 2.

Figure 8: The `markup` procedure that determines which literals must be redacted.

4.2. Third Challenge: Engineering

This section describes details concerning the engineering of the Zhefuscator—a language-specific problem. For reasons related to the business model in which the authors of this paper are involved, Zhefuscator has been implemented in Java, and deployed onto the Java Virtual Machine. Therefore, it intercepts and treats string events produced by programs written in any programming language that runs on the JVM, including Java, Scala, Kotlin, Clojure and many others. In what follows, we discuss particular aspects of the implementation of this tool.

Parsing. Zhefuscator uses the theory seen in Section 3 to build parsers incrementally. These parsers are constructed via the ANTLR [18] parser generation tool. This tool takes as input a grammar that specifies a language and generates as output source code for a recognizer of that language. Procedure `build_grammar` gives ANTLR a new grammar whenever it fails to parse the current text example. ANTLR produces LL(*) parsers, which suits the needs of build_grammar, because Heap-CNF grammars are always Left-to-right, Leftmost derivation and can be parsed with one token of lookahead, as the Theorem 7 states. In terms of implementation, we update the grammar by relying on the JVM's ability to load classes dynamically. The JVM does not need to be restarted in this process. The new grammar is compiled into Java bytecodes by a separate thread, and, as we will see in Section 5, such updates take negligible time.

Theorem 7 (LL). *Any Heap-CNF grammar is LL(1).*

Corollary 3. *There are languages whose grammars cannot be synthesized by Zhefuscator.*

The proof of Theorem 7 mentions that Heap-CNF grammars recognize languages with a finite number of possible derivation trees. In fact, strictly speaking, a Heap-CNF language is finite, as the grammar is not recursive. However, in practice, Zhefuscator deals with infinite languages. Infiniteness comes from the lexer. The procedure build_grammar is parameterized by a string tokenizer. In the context of Zhefuscator's implementation, this tokenizer is given by ANTLR. The regular language used to recognize tokens can accept an unbounded number of strings. In Section 5.2.4 we evaluate the impact of the tokenizer on the performance of Zhefuscator.

Method interception.. Zhefuscator uses Java Agents to intercept calls to the *System.out.** singleton object. The Java Agent API [19] provides support to the dynamic instrumentation of JVM applications. Intercepted strings are first fed to build_grammar, and then redacted. The first action might result in an expansion of the host language's grammar. The second might lead to modifications in the output of the program. Literals that must be redacted are specified using the technique discussed in Section 4.1.

String Obfuscation.. Zhefuscator performs the redaction of sensitive information via asymmetric cryptography. A sensitive literal l is replaced with a new string l_s, which can be later used as a key to retrieve the true value of l from a classified table. Currently, we use *Advanced Encryption Standard* (AES) to ensure safe redaction of values.

4.3. Discussion

The developments explained in this section are necessary to make the ideas introduced in Section 3 practical. We do not claim them as contributions, given that the interface and implementation that we adopted have been already discussed in previous work. Our choice for these aspects of our work are pragmatical. On the one hand, the interface discussed in Section 6.3 and the implementation discussed in Section 4.2 were effective enough to realize the ideas discussed in this paper. However, this choice comes with limitations, which we discuss in the rest of this section.

4.3.1. Lack of Negative Examples

The main limitation of our example-based approach is a lack of negative examples. This limitation is also present in Parsimony [2]; hence, it has naturally persisted in our implementation of it. We opted to avoid negative examples because it is our understanding that in most of the cases where Zhefuscator is useful, negative examples are unnecessary. In other words, database logs tend to follow simple formats, with a small set of sentences of interest. Nevertheless, if necessary to handle more complex formats, then Zhefuscator might produce false positives. In the context of this work, as explained in Section 3.3.2, false positives might cause the redaction of sentences that do not contain sensitive information.

4.3.2. Expressiveness

Additionally, an example-based interface lacks resources that would be promptly available in a domain-specific language, such as the ability to specify logical combinations of events. For instance, users could be interested in enabling certain events only after particular events of interest have been detected. Our current interface lacks such sequencing operations. Users interested in such ability are encouraged to use `ZheLang`, a DSL that we have defined for the treatment of string events. Nevertheless, `ZheLang` is not the focus of this paper.

5. Evaluation

We have implemented the techniques discussed in this paper onto an actual on-line obfuscator, which we call the *Zhefuscator*. Zhefuscator is open source and can be used to redact queries produced by database logs. This section investigates the following research questions related to this implementation, as well as the techniques that support it:

- **RQ1—Convergence**: how many examples are necessary to produce grammars for languages typically used by SQL logging systems?
- **RQ2—Effectiveness**: are the parsers derived from the synthesized grammars effective?
- **RQ3—Practicality**: what is the runtime overhead of Zhefuscator when deployed onto a database system dealing with a heavy workload?

We chose these three particular questions to demonstrate that the theory developed in Section 3, and its implementation described in Section 4, once combined into a concrete tool, lead to a system that is not only novel, but also practical.

Runtime Setup. Every result reported in this section has been produced on an 8-core Intel(R) Core(TM) i7-3770 at 3.40GHz, with 16GB of RAM running Ubuntu 16.04.

5.1. RQ1—Convergence

Methodology. To answer RQ1 we measure how many times the predicate `successfull_parse`, invoked at Line 8 of Figure 2, fails before we produce a definitive grammar for a certain log generator. We perform this analysis on logs from two database systems and from the OSX operating system. Logs are given as a *text* of examples t_i, as defined in Section 3. Each t_i is the entire output produced by the generator, be it a database, be it the operating system, at time unit i. To determine the parts of the log that should be obfuscated, we chose, from each one, four examples, following the steps enumerated in Figure 8. We chose the first four sentences that did not fit into the same SQL production rule. However, this choice bears no impact on the results reported in this paper. Convergence does not depend on it, and the time to redact strings (running time will be evaluated in the next section) is the same for the different approaches that we compare.

5.1.1. Logs from Database

On this experiment, we have generated logs from two different SQL Databases: MySQL version 14.14 Distribution 5.7.27 and PostgreSQL version 9.2.24. Workloads for these two databases were produced by the 9 real-world web applications emulated by OLTP-Bench [20], which include systems such as Wikipedia, Twitter and an ordinary seats system.

Discussion. Figure 9 shows the average prefix necessary to synthesize a grammar in different database systems. Zhefuscator requires approximately eight examples to infer a grammar for the logs produced by MySQL, and five for those produced by PostgreSQL. In the former collection, logs contain an average of 662K lines; in the latter, 1,867K. This experiment indicates that, for typical database logs, the grammar inference procedure of Section 3 tends to converge to a definitive parser after five to eight examples. Furthermore, these examples are a very small portion of the entire log: in every case, we had a definitive grammar after observing less than 0.01% of the whole log file.

5.1.2. Logs from the Operating System

This experiment uses the logs produced by default by MacOS version 10.14.6 in the `/var/log` directory. Contrary to the examples that use the databases, these logs are very different one from the other (the format of sentences is not shared across them). This fact will be made clear once we analyze how many examples are necessary for synthesizing a definitive grammar—this number varies substantially across the logs. We gathered four logs from five distinct OS users,

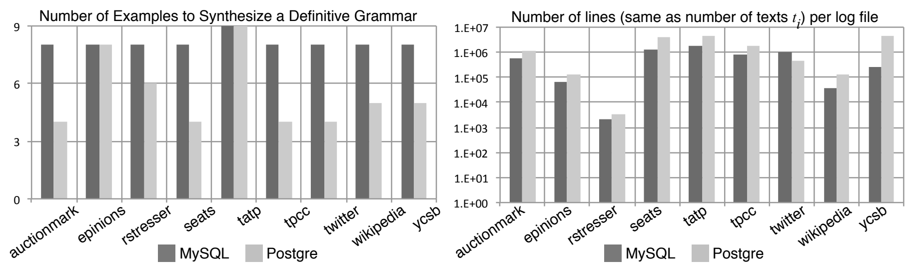

Figure 9: Average prefix size necessary to synthesize a grammar for different log files produced by either MySQL or PostgreSQL.

whose usage pattern corresponds to the profile of professional programmers. The logs used in this experiment are:

- `corecaptured.log`: logs operations of the network hardware. On average, these logs have 174K lines.
- `wifi.log`: logs network traffic. On average, they contain 9K lines.
- `system.log`: logs the operations executed in the whole system. On average, they contain 4K lines.
- `fsck_apfs.log`: logs file system operations, and contain 4K lines on average.

Discussion. Figure 10 shows the average prefix necessary to synthesize grammars for the OSX logs. The number of required examples is higher than what has been observed in Section 5.1.1. The ratio of examples per log size is also higher. In one case (`user3:system`) we had a log with only five lines, whose grammar demanded three examples. This case is an anomaly, due to the small log size. The largest prefix consisted in 170 examples, for a log with 6,579 samples (`user1:system`). In general, the ratio of examples per sample is still very low. For instance, our largest logs (`corecap tured`) have almost 200K lines on average, and yet our on-line grammar inference engine finds a grammar that recognizes all these samples after observing 57 to 64 examples.

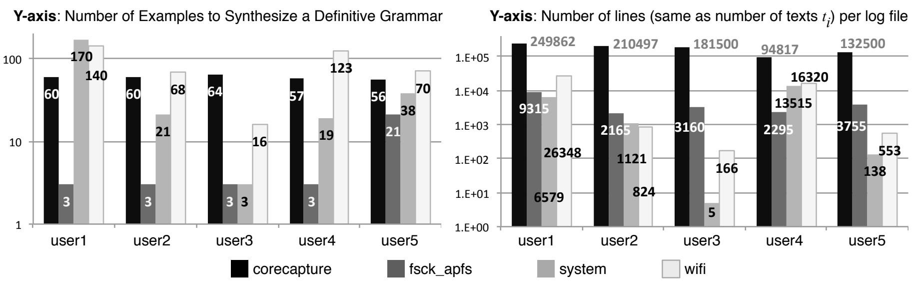

Figure 10: Average prefix size necessary to synthesize a grammar for different MacOS logs.

5.2. RQ2—Effectiveness

This section evaluates the practicality of the grammars synthesized by Zhefuscator. To this effect, we shall answer the five questions below. BF refers to the *Brute Force* approach, which searches event patterns exhaustively within text examples:

1. Section 5.2.1: how does Zhefuscator compare to BF to parse one individual example for which a parser has not already been synthesized.

2. Section 5.2.2: how does Zhefuscator compare to BF to parse 1,000 examples in an actual log file produced by a MySQL database.

3. Section 5.2.3: how does Zhefuscator compare to BF to parse 1,000 examples in artificially generated logs of different sizes.

4. Section 5.2.4: how does the tokenizer change the runtime of Zhefuscator.

5.2.1. Parsing Effectiveness

There exists a trivial approach to solve the Language Separation Problem introduced in Definition 2: given an example t_i in the host language, we start a search for an event s, an SQL query in our context, at every token of t_i. If two events can start at the same token, we choose the longest one. This solution is called the *brute-force* approach. The developments in Section 3 are attractive inasmuch as they lead to a faster solution to language separation than the brute-force technique. In this section, we compare the parsing speed of both approaches.

Before we discuss our methodology, two observations are in order. First, when Zhefuscator's current grammar is not able to recognize the active example, it behaves in a similar manner as the brute force approach: it must scan the SQL query, assuming that it can start at any token. In addition to this, it must augment the current grammar using the techniques discussed in Section 3. Second, when Zhefuscator's parser is able to recognize the active example, parsing happens via $O(N)$ productions, where N is the number of tokens. Yet, the number of characters per token varies, and the lexer's runtime must be taken into consideration. Thus, the overall runtime is $O(M)$, where M is the number of characters in the active example. The brute force approach might expand $O(N^2)$ productions; however, such worst case seldom happens. Most of the tokens in a valid example cannot be the prefix of any SQL query. Therefore, although naïve, the brute force approach is still likely to outperform Zhefusctor for examples with few characters.

Methodology. The brute-force approach becomes less practical as the number of characters in the examples t_i of the host language L increases. To investigate at which point the grammars synthesized by `build_grammar` become more efficient, we have used the logs seen in Section 5.1.1. To obtain examples of varying sizes, we either split or concatenate lines from these logs; hence, producing strings of different lengths.

Discussion. Figure 11 compares the brute-force with our synthetic grammars. Our grammars are more asymptotically efficient than the brute-force approach. After multiple merging operations, a Heap-CNF grammar still recognizes a sentence in $O(N)$ derivation steps, where N is the size of the sentence. The brute force approach, in turn, will always require $O(N^2)$ steps. Figure 11 shows that for examples between 128 and 256 characters (about 16 tokens) our approach becomes consistently better than the trivial brute-force parsing. In Section 5.2.2 we observe the effect of this improvement when applied onto an actual log.

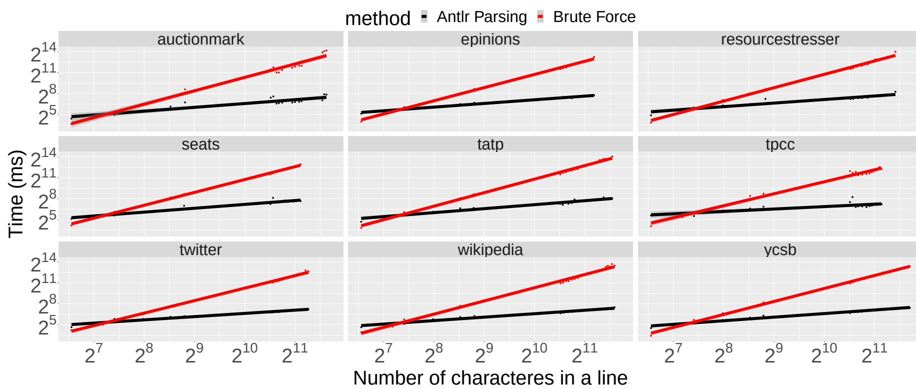

Figure 11: Time comparison of brute force approach and the ANTLR parser.

5.2.2. Effectiveness on an Actual Log File

In Section 5.2.1, we compared the average time taken by the Zhefuscator and the brute force approach to parse one example. However, the benefit of our parser synthesizer becomes more evident once we analyze its effect when amortized onto a long chain of examples. In this section, we analyze this effect via *skyline* charts. These charts show the time taken per individual example in the log. For this experiment, we chose the log produced by the MySQL implementation for the AuctionMark application. We emphasize that the choice of log, for this experiment, is immaterial: all the logs produced by MySQL follow the same format, and Zhefuscator's parser needs to be augmented only 8 times for all of them. AuctionMark has been chosen simply because it is the first benchmark in OLTPBench.

Methodology. We compare both the approaches, Zhefuscator and the brute force, when given the first 1,000 examples in the log that MySQL produces for AuctionMark. For each example, we count only the time to recognize strings—redaction is not accounted for, because it applies the same algorithm, the same number of times, in both the approaches. Notice that choosing more than 1,000 examples will not change the results reported in this section, because Zhefuscator builds a definitive parser after observing 19 entries in the log file.

Discussion. Figure 12 shows the result of this experiment, juxtaposing the

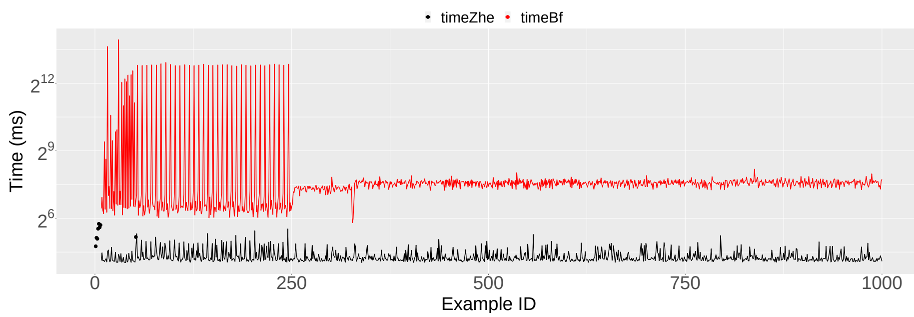

Figure 12: Skiline comparison between Zhefuscator and the brute force approach to parse 1,000 examples from the AuctionMark log.

skyline produced by the brute force approach and by Zhefuscator. The log file contains two distinct parts. The first 250 examples are system configuration commands, and have 1,021 characters, on average. The last 750 examples are various SQL queries, and contain 106 characters on the average. Using the C tokenizer, we obtain 105 tokens, on the average, considering the 1,000 examples in the log file. Under this circumstance, the performance gap between Zhefuscator and the brute force approach is noticeable.

Zhefuscator spends, on the average, 26.15 milliseconds per example, with a standard deviation of 17.86 ms. This number includes the extra time Zhefuscator needs to augment the current parser—an action that happened 8 times in this experiment. The brute force approach spends 586.63 milliseconds per example, with a standard deviation of 1,830.33 ms. Zhefuscator is 22.5x faster, per example, than the brute force approach. However, this experiment uses an ideal scenario for Zhefuscator: a long stream of homogeneous textual examples. In the next section, we shall analyze the behavior of Zhefuscator under more unfavorable conditions.

5.2.3. Increased Effectiveness via Amortized Cost

The logs produced by MySQL and PostgreSQL are formed by long individual examples (more than 100 tokens on average). However, these examples are all similar; hence, as already observed in Figure 9, Zhefuscator synthesizes a definitive parser after observing a very short subset of them. To stress out the performance of Zhefuscator, in this section we analyze its behavior when dealing with more complex logs, which we have produced artificially.

Methodology. To produce the logs, we use six different types of tokens: booleans, integers, doubles, strings, dates and sets of comma-separated integers within curly brackets, e.g., $\{2, 3, 5, 7\}$. We generate four types of logs. Each log contains a random number of tokens between 0 and $R \in \{4, 8, 16, 40\}$, before and after an SQL query. We use always the query "`SELECT` *string* `FROM string` WHERE `id = int`". With $R = 4$, we have $4^6 + 4^6 = 8,192$ possible example formats; with $R = 8$, we have $8^6 + 8^6 = 524,288$, and so on. Therefore, `fill_holes` will be invoked a much larger number of times than in the setup used in the previous section.

Discussion. Figure 13 shows the result of this experiment. Whereas BF shows homogeneous behavior—its runtime per example varying only slightly—Zhefuscator has two types of responses. Such responses depend on the current parser recognizing or not the active example. When recognition is possible, parsing is fast; otherwise, the parser must be augmented with new productions, and we observe a runtime spike, which is marked in Figure 13 with a black dot. Said spikes are compulsory for the initial examples. However, as the current grammar increases, sentence recognition becomes more common, and spikes tend to disappear. As a consequence, the more events are observed, the larger is the performance improvement of Zhefuscator over the brute force approach.

Figure 14 shows average time per example, plus standard deviations observed for Zhefuscator and for the brute force approach. The figure shows two results for Zhefuscator: the first considers only the time when parsing succeeds; the second considers, in addition, the time taken by `fill_holes`, when Zhefuscator fails. In the former scenario, Zhefuscator always outperforms the brute force approach. In the latter, it always loses. The conclusion is that, once it reaches a steady state, Zhefuscator's $O(N)$ parser is consistently a better option than BF's $O(N^2)$ algorithm. However, if necessary to augment the current parser too often, our technique loses its attractiveness. In this particular experiment, `fill_holes` performs worse than in Section 5.2.2, because the host language is

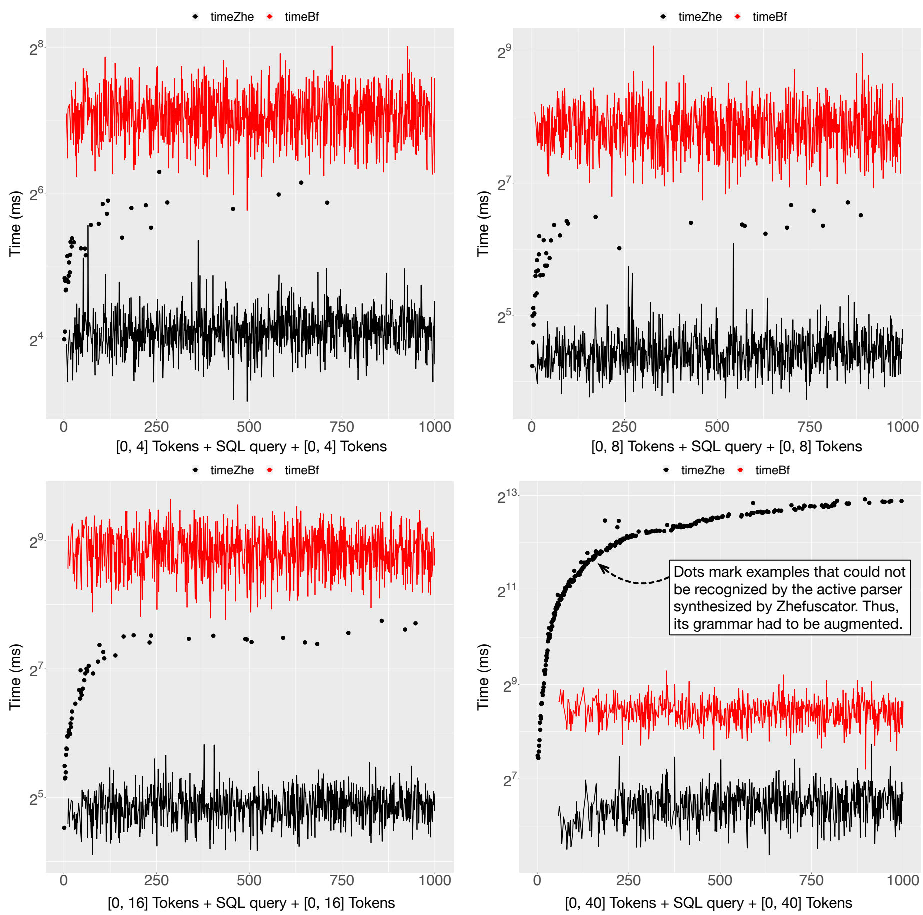

Figure 13: Runtime comparison between Zhefuscator and the brute force approach to parse 1,000 examples of artificially generated logs. Black dots mark invocations of `fill_holes`.

much more complex.

5.2.4. *Impact of the Tokenizer on Runtime*

The `add_example` routine, which is invoked by `build_grammar` (Figure 2, Lines 7-15) is parameterized by a tokenizer. The tokenizer is a function that converts the input text into tokens. The tokenizer is just an artifact of our implementation: users of our system will never have to deal with it. The implementation of Zhefuscator can use any tokenizer that ANTLR supports. As we have hinted in Section 3.2, the tokenizer impacts both the number of examples

Format	[0,4]+SQL+[0,4]		[0,8]+SQL+[0,8]		[0,16]+SQL+[0,16]		[0,40]+SQL+[0,40]	
Tk/Ex	29.99		38.26		54.10		298.43	
	parsing only	fill_holes: 35	parsing only	fill_holes: 39	parsing only	fill_holes: 51	parsing only	fill_holes: 264
avg (Zhe)	17.53	18.40	22.08	23.67	29.47	34.05	88.88	1,085.01
std (Zhe)	3.45	6.32	4.51	10.09	5.00	23.70	21.15	2,004.49
avg (BF)	137.15		232.14		464.65		347.20	
std (BF)	31.41		55.38		103.24		60.48	

Figure 14: Average time and standard deviation (per example, in milliseconds) that Zhefuscator (Zhe) and the brute force approach (BF) take to analyze the artificial logs. "Parsing only" reports runtimes for examples in which Zhefuscator's current parser succeeds without having to synthesize a new grammar. "`fill_holes`: XX" includes the time of "parsing only", plus the time to augment the current parser. XX reports the number of times Zhefuscator had to augment the current parser (via the `fill_holes` routine).

as well as the runtime of Zhefuscator. In this section, we analyze this impact by verifying the behavior of Zhefuscator when parameterized by two different lexers.

Methodology. We have tried Zhefuscator with two different lexers. Both were taken from public projects that use ANTLR—they have not been implemented as part of this research.

Discussion. Although the choice of tokenizer might modify the number of examples necessary to reach a definitive grammar, the two tokenizers that we have used led to the same prefix size in Figures 9 and 10. This happens because C and SQL have many similar tokens, including identifiers—the most common in the examples. However, the impact on runtime is different. Using the C tokenizer, Zhefuscator takes 26.15 milliseconds, on average (STD = 17.85ms), per example from the AuctionMark log (Figure 12), including the eventual time taken to augment the grammar. Using the SQL version, this time drops down to 18.81 milliseconds, with a standard deviation of 3.33ms. The latter is faster because the SQL lexer uses a smaller automaton than the C lexer.

5.3. RQ3—Practicality

The techniques described in Section 3 have a computational cost. The goal of this section is to measure such cost. This empirical evaluation shall allow us to claim that the overhead of Zhefuscator, when deployed onto typical Java applications, is low enough to be practical.

Methodology. It is difficult to measure the overhead of Zhefuscator in our experimental setup involving actual deployments of MySQL and PostgreSQL. This difficulty comes from the fact that logging, at least in that particular setting, is a rare event. Log entries are produced only when users enter queries in the database. In this scenario, the overhead of Zhefuscator is negligible. Thus, to probe this overhead in a more heavily loaded scenario, we shall proceed with two experiments. In Section 5.3.1 we measure the runtime overhead that event handling imposes onto a single invocation of the *System.out.println* routine used to output log information in a database server. This evaluation provides some insight into the absolute overhead of event handling; however, it does not give us much information about how Zhefuscator would impact user experience, for the time of handling one single string event is very fast. To circumvent this limitation, in Section 5.3.2 we measure the overhead that Zhefuscator imposes onto batch computations, i.e., that perform a fixed number of steps. In this case, we focus on the Java Dacapo benchmark suite [12].

5.3.1. Overhead of Treating one String Event

To measure the overhead of treating one string event, we have built a system that reads a log file and outputs it line by line using the *System.out.println* method from the Java Standard Library. For maximum stress, we assume that every SQL literal must be redacted. In this experiment, we adopt the same logs from the MySQL databases used in Section 5.1.1.

Discussion. Figure 15 presents the results of this evaluation. Each log was evaluated ten times; hence, each box plot contains ten samples. The figure makes it clear that Zhefuscator's event handler has an overhead over individual method invocations. This overhead can be as high as two orders of magnitude, as observed in `resourcestresser`. However, this cost accounts for a very small proportion of the runtime of a typical database system. In the case of `resourcestresser`, the average time to redact every literal in the log is 0.03sec per invocation of *System.out.println*. This time includes the invocation of `build_grammar` (Fig. 2) and the obfuscation of literals. Obfuscation includes the time to encrypt literals using AES.

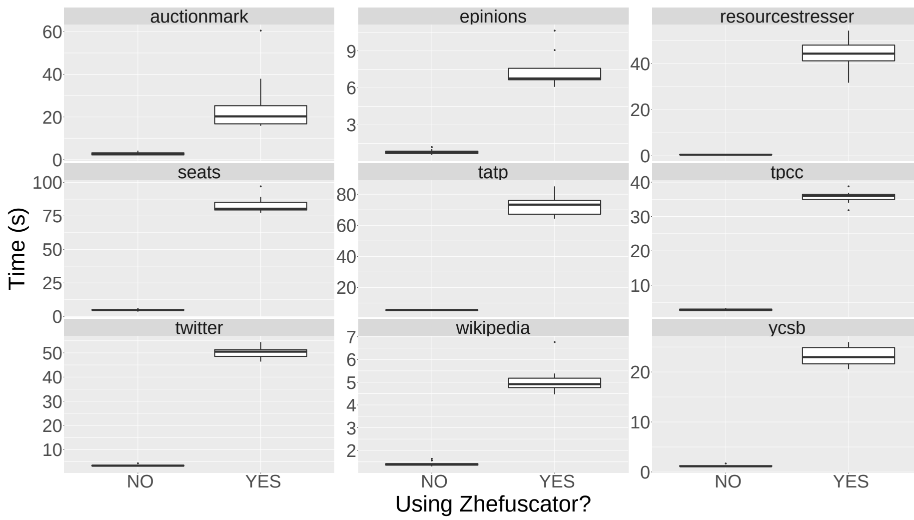

Figure 15: Overhead of Zhefuscator on an extreme case: a system that only outputs different database logs.

5.3.2. Deploying on Java Dacapo

In this experiment, we measure the overhead of building a grammar for every output produced by the programs in the DaCapo Benchmark Suite. DaCapo's logs do not contain SQL queries; hence, in this section, we are measuring the time to build grammars, but not the time to redact queries.

Discussion. Figure 16 compares the runtime of DaCapo without and with interventions from Zhefuscator. Figure 17 shows accompanying data: p-values, number of log events and number of production rules in the final grammar that we synthesize. The p-value provides us with some notion of statistically significant runtime difference: the lower the p-value, the more noticeable is the gap in runtime between the two versions of each DaCapo program. Typically, p-values below 0.05 are considered statistically significant. These p-values have been obtained via a T-Test applied on the same data used to produce Figure 16. The T-Test provides us with an idea on how different are a "control" and a "test" groups. In our setting, the control group is formed (in Figure 16) by applications that do not run the Zhefuscator. The test group, in contrast, is formed by the same applications using the Zhefuscator. The lower the p-value returned by the T-Test, the more statistically significant is the different between these two groups.

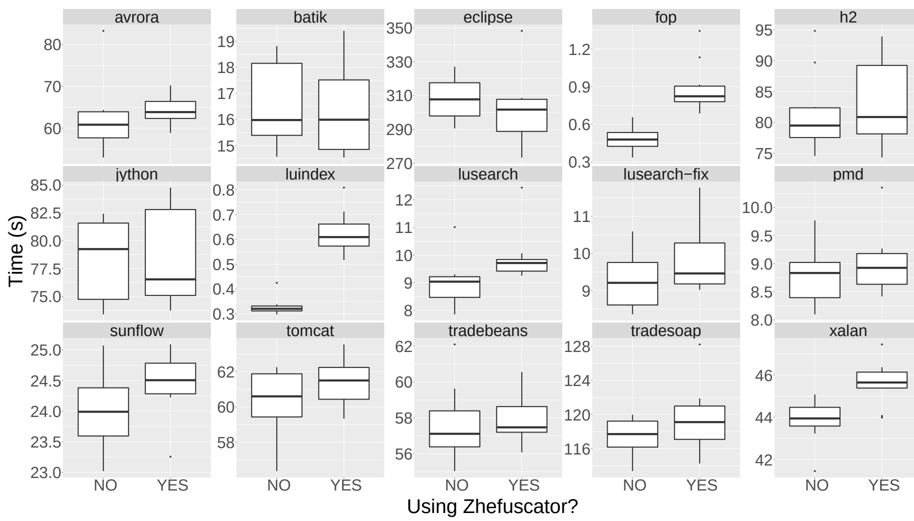

Figure 16: The overhead of Zhefuscator on Dacapo.

	avrora	batik	eclipse	fop	h2	jython	luindex	lusearch	lusrch-fix	pmd	sunflow	tomcat	tradebeans	tradesoap	xalan
P-values	0.47	0.68	0.38	0	0.54	0.99	0	0.03	0.17	0.52	0.07	0.13	0.66	0.16	0
Log Lines	13	22	25	13	24	94	13	45	45	13	13	19511	31	31	14
Productions	30	60	10	16	68	30	16	46	46	30	46	56	54	54	46

Figure 17: The overhead of Zhefuscator on Dacapo. The lower the p-value, the more statistically significant the overhead.

The runtime overhead of Zhefuscator, even when deployed onto a batch system, tends to be small. In 11, out of 15 cases, we could not perceive any statistically significant runtime difference. The largest runtime gap that we have observed was in `fop`; however, this is the benchmark that runs for the shortest time. Thus, this overhead, due to the initialization of Zhefuscator's agent, tends to be amortized in systems that run for more time. The largest absolute overhead was observed in `xalan`: 1.7 seconds on average, over a system that runs for 44 seconds on average.

6. Related Work

Much theory concerning the recognition of languages on the limit has been designed and discussed in the literature. Section 6.1 discusses this theory, to give the reader some perspective on the foundations of the present work. We also notice that much of the developments in this work bear resemblances to programming fuzzing. Yet, whereas fuzzing is concerned with recognizing a language that describes the input of a program, our paper deals with the inverse problem: we recognize a language that describes the output of the program. Section 6.2 discusses work related to fuzzing. Additionally, there exists a vast body of literature concerned with the synthesis of grammars from examples. This is the approach that we use in Section 4.1 to equip the Zhefuscator with a user interface. In Section 6.3 we discuss work related to the synthesis of grammars from examples. Finally, our theory, once implemented into an actual tool, yields a reactive system. Events, in this case, are the occurrence of particular patterns in Strings. Section 6.4 explores other reactive systems of similar nature.

6.1. Inductive Grammar Synthesis

The notion of *language identification in the limit*, which we have used as a motivation for our on-line grammar inference algorithm, was introduced by Edward Gold in the mid sixties [14]. Much research evolved from Gold's initial problem formulation. The main developments in the field are due to Angluin and her collaborators [21, 22, 23, 24]. Nevertheless, several research groups have formalized grammar inference for specific types of languages [25, 26, 27, 28, 29]. Since the nineties, decidability for inference of grammars for several classes of languages is already known [30]. Usually, the language thus produced is deterministic, although Eman *et al.* have shown how to derive probabilistic automata on the limit [31]. The identification of string events fits into the framework of language inference in the limit; however, in this paper, we do not try to guess the right host language L that contains said events. Instead, we try to infer a grammar G that recognizes string events in any prefix of this language. Notice that G might also recognize strings that do not belong into L. This possibility has no practical implications in the context of this paper: we are interested in finding string events, not in recognizing exactly the host language that contains it.

Recent progress in the field of machine-learning has imbued Gold's original program with renewed attractiveness. For an overview of how machine-learning techniques are used to solve language recognition in the limit, we recommend Bennaceur *et al.*'s [32]. The literature contains several examples of how statistical inference techniques are used to learn a language in the limit, such as the work of Li *et al.* [33], who employ a genetic-based algorithm to learn the structure of XML documents. Or, along a different direction, the work of Graben *et al.* [34], who have developed an interactive system to gradually learn a simple language of English numerals. We contend that such techniques, although effective in their contexts, are not ideal fits to our problem—online language recognition—because they require slow, exploratory-based algorithms, which would be too heavy for our needs.

6.2. Program Fuzzing

In this paper, we are interested in approximating a grammar that characterizes the output of a program. The inverse problem has received more attention in the programming language community: to infer a grammar that describes the input of a program. This kind of inference is useful in testing via software fuzzing, as demonstrated by Bastani *et al.* [35] and Blazytko *et al.* [36], for instance. The many approaches described in the literature [35, 36, 37, 38] differ from our work in many ways. First, there is the obvious difference in direction: we infer grammars for program outputs, not inputs. Second, these techniques typically rely on negative examples to refine the inferred grammar, whereas negative examples play no role in our formulation. Finally, there is a difference in purpose: we are not interested in testing a program; rather, our intention is to intervene in the program already in production.

6.3. Interactive Grammar Inference

There exists prior work about the construction of parsers for programming languages based on examples [39, 40, 41, 1, 2]. Such systems synthesize and refine grammars, one example at a time. Much of the inspiration behind our approach to select which literals must be redacted (see Section 4.1) came from Parsimony [2], an IDE for example-guided synthesis of lexers and parsers. This line of work is an instance of a much broader field known as *programming-by-examples* (PBE) [42]. Zhefuscator is not a framework to support programming by example. It infers grammars on-the-fly that recognize examples produced automatically by a machine, not a person. Therefore, the speed to synthesize a parser is an essential requirement of our work—more than clarity, or the efficiency of the parser itself. That is the reason why we have opted to produce Heap-CNF grammars: it is fast to generate and merge them.

6.4. String Events

This paper is not the first work to deal with the on-line detection of string events. Research along this direction was mostly concerned with security. String events have been handled, for instance, in the context of intrusion detection [43, 44], dynamic taint analysis [45, 46] and on-the-fly spam identification [47]. Nevertheless, if we do not claim primacy, we claim generality. All these previous works would identify string events in very specific situations, e.g., as particular patterns embedded in an SQL query, in the case of tainted flow analysis [45], or as a combination of specific tokens within a network package, in the case of intrusion detection [43]. This paper is the first work to provide a general framework that, in a way, "learns" a language, and recognizes string events embedded into it.

7. Conclusion

This paper has presented a theoretical framework to detect string events. Said events are described by a language whose grammar is known. They occur within a potentially infinite text, defined by a host language, whose grammar is unknown. We showed how to synthesize a grammar G that recognizes any prefix of the infinite text stream. By defining a specific restriction of Chomsky Normal Form, the Heap-CNF, we guarantee that G is non-ambiguous (Theorem 5) and admits LL(1) parsing (Theorem 7). We have shown, empirically, that this theory can be implemented into an efficient log anonymization system, the Zhefuscator, which redacts sensitive information from the output of programs, while treating these programs as black-box software. We have tested the Zhefuscator onto logs from databases (MySQL and PostgreSQL), operating systems (OSX) and Java benchmarks (DaCapo). In every case, the performance overhead of this system is very small.

Future work. We speculate that recent developments in the programming languages community can be used to strengthen the theory and the practice discussed in this paper. First, concerning formalization, our theorems are not mechanically verified. This shortcoming is due to the lack of a general framework to reason about properties of LL(1) parsers. However, Edelmann *et al.* [10] have showed how to build LL(1) parsers with derivatives and zippers that are correct by construction.

Second, Zhefuscator is parameterized by a tokenizer, which our current implementation borrows from ANTLR. The fact that users have no way to specify a lexer in our system can be considered a limitation of our current implementation. Thus, it would be desirable to give users the possibility to define their own tokenizers without exposing them to minutia related to automata theory. Recent work by Chen *et al.* [48] has provided a clear interface for this purpose, which is based on examples supported by a natural language (NL) description of regular expressions. We believe that NL-based specifications will be able to improve purely example-based approaches that have recently been shown to be effective to specify regular expressions [49, 50]. This research direction is even more promising once we consider the availability of efficient string solvers such as CVC4Sy [51] or Z3-Str [52], which supports a wide range of logical theories, including strings and regular expressions.

References

[1] A. Leung, J. Sarracino, S. Lerner, Interactive parser synthesis by example, in: PLDI, ACM, New York, NY, USA, 2015, pp. 565–574. doi:10.1145/2737924.2738002.

[2] A. Leung, S. Lerner, Parsimony: An IDE for example-guided synthesis of lexers and parsers, in: ASE, IEEE Press, Piscataway, NJ, USA, 2017, pp. 815–825.

[3] K. Nakamura, M. Matsumoto, Incremental learning of context free grammars, in: ICGI, Springer-Verlag, London, UK, 2002, pp. 174–184.

[4] R. Lämmel, C. Stenzel, Semantics-directed implementation of method-call interception, IEE Proceedings - Software 151 (2) (2004) 109–128. doi:10.1049/ip-sen:20040080.

[5] U. Zdun, Supporting incremental and experimental software evolution by runtime method transformations, Sci. Comput. Program. 52 (1-3) (2004) 131–163. doi:10.1016/j.scico.2004.03.005.

[6] S. Abiteboul, J. Stoyanovich, Transparency, fairness, data protection, neutrality: Data management challenges in the face of new regulation, J. Data and Information Quality 11 (3) (2019) 15:1–15:9. doi:10.1145/3310231.

[7] B. Custers, A. M. Sears, F. Dechesne, I. Georgieva, T. Tani, S. van der Hof, EU Personal Data Protection in Policy and Practice, 1st Edition, T.M.C. Asser Press, The Netherlands, 2019.

[8] P. Voigt, A. v. d. Bussche, The EU General Data Protection Regulation (GDPR): A Practical Guide, 1st Edition, Springer-Verlag, Berlin, Heidelberg, 2017.

[9] P. M. Lewis, R. E. Stearns, Syntax-directed transduction, J. ACM 15 (3) (1968) 465–488. doi:10.1145/321466.321477.

[10] R. Edelmann, J. Hamza, V. Kuncak, LL(1) parsing with derivatives and zippers, in: Conference on Programming Language Design and Implementation (PLDI), ACM, New York, NY, USA, 2020, p. to appear.

[11] S. Ramson, R. Hirschfeld, Active expressions: Basic building blocks for reactive programming, The Art, Science, and Engineering of Programming 1 (2). doi:10.22152/programming-journal.org/2017/1/12.
URL http://dx.doi.org/10.22152/programming-journal.org/2017/1/12

[12] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al., The dacapo benchmarks: Java benchmarking development and analysis, in: ACM Sigplan Notices, ACM, New York, US, 2006, pp. 169–190. doi:10.1145/1167515.1167488.

[13] A. Erickson, Comparative analysis of the EU's GDPR and brazil's LGPD: Enforcement challenges with the LGPD, Brooklyn Journal of International Law 2 (9) (2019) 859–.

[14] E. M. Gold, Language identification in the limit, Information and Control 10 (5) (1967) 447–474. doi:10.1016/S0019-9958(67)91165-5.

[15] A. Solar-Lezama, R. M. Rabbah, R. Bodík, K. Ebcioglu, Programming by sketching for bit-streaming programs, in: V. Sarkar, M. W. Hall (Eds.), Conference on Programming Language Design and Implementation (PLDI), ACM, 2005, pp. 281–294. doi:10.1145/1065010.1065045.
URL http://doi.acm.org/10.1145/1065010.1065045

[16] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, V. A. Saraswat, Combinatorial sketching for finite programs, in: J. P. Shen, M. Martonosi (Eds.), Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, 2006, pp. 404–415. doi:10.1145/1168857.1168907.
URL http://doi.acm.org/10.1145/1168857.1168907

[17] S. Gulwani, Automating string processing in spreadsheets using input-output examples, in: T. Ball, M. Sagiv (Eds.), Conference on the Principles of Programming Languages (POPL), ACM, 2011, pp. 317–330.

[18] T. Parr, K. Fisher, LL (*): the foundation of the antlr parser generator, Sigplan Notices 46 (6) (2011) 425–436. doi:10.1145/1993316.1993548.

[19] W. Binder, J. Hulaas, P. Moret, Advanced java bytecode instrumentation, in: Proceedings of the 5th international symposium on Principles and practice of programming in Java, ACM, 2007, pp. 135–144. doi:10.1145/1294325.1294344.

[20] D. E. Difallah, A. Pavlo, C. Curino, P. Cudre-Mauroux, Oltp-bench: An extensible testbed for benchmarking relational databases, Proceedings of the VLDB Endowment 7 (4) (2013) 277–288. doi:10.14778/2732240.2732246.

[21] D. Angluin, Finding patterns common to a set of strings (extended abstract), in: STOC, ACM, New York, NY, USA, 1979, pp. 130–141. doi:10.1145/800135.804406.

[22] D. Angluin, Inductive inference of formal languages from positive data, Information and Control 45 (2) (1980) 117–135.

[23] D. Angluin, Inference of reversible languages, J. ACM 29 (3) (1982) 741–765. doi:10.1145/322326.322334.

[24] D. Angluin, C. H. Smith, Inductive inference: Theory and methods, ACM Comput. Surv. 15 (3) (1983) 237–269. doi:10.1145/356914.356918.

[25] P. García, M. Vázquez de Parga, D. López, On the efficient construction of quasi-reversible automata for reversible languages, Inf. Process. Lett. 107 (1) (2008) 13–17. doi:10.1016/j.ipl.2007.12.004.
URL `http://dx.doi.org/10.1016/j.ipl.2007.12.004`

[26] C. Globig, S. Lange, On case-based representability and learnability of languages, in: AII, Springer-Verlag, London, UK, UK, 1994, pp. 106–120.
URL `http://dl.acm.org/citation.cfm?id=647712.735260`

[27] D. López, J. M. Sempere, P. García, Inference of reversible tree languages, Trans. Systems, Man, and Cybernetics, Part B 34 (4) (2004) 1658–1665. doi:10.1109/TSMCB.2004.827190.

[28] P. Peris, D. López, Transducer inference by assembling specific languages, in: ICGI, Springer, Berlin, Heidelberg, 2010, pp. 178–188. doi:10.1007/978-3-642-15488-1_15.

[29] Y. Sakakibara, Grammatical inference: An old and new paradigm, in: ALT, Springer-Verlag, Berlin, Heidelberg, 1995, pp. 1–24.
URL `http://dl.acm.org/citation.cfm?id=647713.735416`

[30] Y. Sakakibara, Recent advances of grammatical inference, Theor. Comput. Sci. 185 (1) (1997) 15–45. doi:10.1016/S0304-3975(97)00014-5.

[31] S. S. Emam, J. Miller, Inferring extended probabilistic finite-state automaton models from software executions, ACM Trans. Softw. Eng. Methodol. 27 (1). doi:10.1145/3196883.
URL `https://doi.org/10.1145/3196883`

[32] A. Bennaceur, R. Hähnle, K. Meinke, G. Goos, J. Hartmanis, J. Leeuwen, D. Hutchison, Machine Learning for Dynamic Software Analysis: Potentials and Limits, Springer, 2018.

[33] Y. Li, C. Dong, X. Chu, H. Chen, Learning DMEs from positive and negative examples, in: DASFAA, 2019, pp. 434–438.

[34] P. B. Graben, R. Römer, W. Meyer, M. Huber, M. Wolff, Reinforcement learning of minimalist numeral grammars, in: CogInfoCom, IEEE, 2019, pp. 67–72.

[35] O. Bastani, R. Sharma, A. Aiken, P. Liang, Synthesizing program input grammars, ACM SIGPLAN Notices 52 (6) (2017) 95–110. doi:10.1145/3140587.3062349.

[36] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, T. Holz, GRIMOIRE: Synthesizing structure while fuzzing, in: SEC, USENIX Association, Berkeley, CA, USA, 2019, pp. 1985–2002. URL `http://dl.acm.org/citation.cfm?id=3361338.3361475`

[37] M. Höschele, A. Zeller, Mining input grammars with AUTOGRAM, in: ICSE-C, IEEE Press, Piscataway, NJ, USA, 2017, pp. 31–34. doi:10.1109/ICSE-C.2017.14.

[38] Z. Wu, E. Johnson, W. Yang, O. Bastani, D. Song, J. Peng, T. Xie, REINAM: Reinforcement learning for input-grammar inference, in: ESEC/FSE, ACM, New York, NY, USA, 2019, pp. 488–498. doi:10.1145/3338906.3338958.

[39] A. Bartoli, G. Davanzo, A. De Lorenzo, M. Mauri, E. Medvet, E. Sorio, Automatic generation of regular expressions from examples with genetic programming, in: GECCO, ACM, New York, NY, USA, 2012, pp. 1477–1478. doi:10.1145/2330784.2331000.

[40] R. C. Miller, Lightweight structure in text, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA, aAI3051028 (2002).

[41] T. Lau, S. A. Wolfman, P. Domingos, P. Domingos, D. S. Weld, Programming by demonstration using version space algebra, Mach. Learn. 53 (1-2) (2003) 111–156. doi:10.1023/A:1025671410623.

[42] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, A. Turransky (Eds.), Watch What I Do: Programming by Demonstration, MIT Press, Cambridge, MA, USA, 1993.

[43] T. Abbes, A. Bouhoula, M. Rusinowitch, On the fly pattern matching for intrusion detection with snort, Annales des Télécommunications 59 (9-10) (2004) 1045–1071. doi:10.1007/BF03179710.

[44] M. Roesch, Snort - lightweight intrusion detection for networks, in: LISA, USENIX Association, Berkeley, CA, USA, 1999, pp. 229–238. URL `http://dl.acm.org/citation.cfm?id=1039834.1039864`

[45] K. Chen, D. Wagner, Large-scale analysis of format string vulnerabilities in debian linux, in: PLAS, ACM, New York, NY, USA, 2007, pp. 75–84. doi:10.1145/1255329.1255344.

[46] W. Chang, B. Streiff, C. Lin, Efficient and extensible security enforcement using dynamic data flow analysis, in: CCS, ACM, New York, NY, USA, 2008, pp. 39–50. doi:10.1145/1455770.1455778.

[47] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, I. Osipkov, Spamming botnets: Signatures and characteristics, SIGCOMM Comput. Commun. Rev. 38 (4) (2008) 171–182. doi:10.1145/1402946.1402979.

[48] Q. Chen, X. Wang, X. Ye, G. Durrett, I. Dillig, Multi-modal synthesis of regular expressions, in: PLDI, ACM, New York, NY, USA, 2020, pp. 487–502. doi:10.1145/3385412.3385988.

[49] R. Pan, Q. Hu, G. Xu, L. D'Antoni, Automatic repair of regular expressions, PACMPL 3 (OOPSLA) (2019) 139:1–139:29. doi:10.1145/3360565. URL `https://doi.org/10.1145/3360565`

[50] A. Bartoli, A. De Lorenzo, E. Medvet, F. Tarlao, Automatic search-and-replace from examples with coevolutionary genetic programming (2019). doi:10.1109/TCYB.2019.2918337.

[51] A. Reynolds, H. Barbosa, A. Nötzli, C. Barrett, C. Tinelli, cvc4sy: Smart and fast term enumeration for syntax-guided synthesis, in: I. Dillig, S. Tasiran (Eds.), Computer Aided Verification (CAV), Part II, Vol. 11562 of Lecture Notes in Computer Science, Springer International Publishing, Cham, 2019, pp. 74–83.

[52] Y. Zheng, X. Zhang, V. Ganesh, Z3-Str: A Z3-based string solver for web application analysis, in: ESEC/FSE, ACM, New York, NY, USA, 2013, p. 114–124. doi:10.1145/2491411.2491456.

[53] D. J. Rosenkrantz, R. E. Stearns, Properties of deterministic top down grammars, in: STOC, ACM, New York, NY, USA, 1969, p. 165–180. doi:10.1145/800169.805431.

Proofs of Lemmas and Theorems

This appendix contains proofs of Lemmas, Theorems and Corollaries present in the paper "On-Line Synthesis of Parsers for String Events".

Theorem 1. Function `fill_holes` (Fig. 3) produces a grammar G_i that recognizes an example $t_i = t_i^1 \cdots t_i^n$ in n steps with $2n-1$ non-terminals.

Proof 2. *The proof is by induction on the size $|t_i|$ of the example. On the **Base Case**, we have that $t_i = token$; hence, $|t_i| = 1$. `fill_holes` produces $R_1 ::= token$, which recognizes t_i trivially. On the **Inductive Case**, we assume that $t_i = token \bullet$ `Rest`. By induction, we have that `fill_holes` generates a grammar with starting symbol R_{2i+1} that recognizes `Rest` in $n-1$ steps (Line 7 of Figure 3). The extended grammar recognizes t_i:*

$$\begin{array}{lll} R_n & ::= & R_{2n}R_{2n+1} \\ R_{2n} & ::= & token \\ R_{2n+1} & ::= & \ldots \end{array}$$

By induction, we know that R_{2n+1} starts production rules with $2(n-1)-1$ non-terminals. Adding R_n and R_{2n}, we have that the resulting grammar contains $2n-1$ non-terminals.

Lemma 2.. If G is the grammar that results from merging two Heap-CNF grammars G' and $G"$, then G is Heap-CNF, and $lang(G') \cup lang(G") \subseteq lang(G)$

Proof 3. *We demonstrate the lemma analyzing each one of the four cases involved in the process of merging two Heap-CNF grammars. We let $R_i' ::= P'$ be the production rule that corresponds to R_i in G_i. Similarly, we let $R_i" ::= P"$ be the production rule that corresponds to R_i in $G_i"$. We let* tk *be a token:*

- $P' = tk_1' \mid \ldots \mid tk_n'$ *and* $P" = tk_1" \mid \ldots \mid tk_n"$. *In this case, we have that* $R_i ::= tk_1' \mid \ldots \mid tk_n' \mid tk_1" \mid \ldots \mid tk_n"$, *which is still Heap-CNF.*
- $P' = R_{2i}R_{2i+1} \mid tk_1' \mid \ldots \mid tk_n'$ *and* $P" = tk_1" \mid \ldots \mid tk_n"$. *In this case, we have that* $R_i ::= R_{2i}R_{2i+1} \mid tk_1' \mid \ldots \mid tk_n'\ tk_1" \mid \ldots \mid tk_n"$, *which is still Heap-CNF.*
- $P' = tk_1' \mid \ldots \mid tk_n'$ *and* $P" = R_{2i}R_{2i+1} \mid tk_1" \mid \ldots \mid tk_n"$. *In this case, we have that* $R_i ::= R_{2i}R_{2i+1} \mid tk_1' \mid \ldots \mid tk_n'\ tk_1" \mid \ldots \mid tk_n"$, *which is still Heap-CNF.*

- $P' = R_{2i}R_{2i+1} \mid tk'_1 \mid \ldots \mid tk'_n$ *and* $P" = R_{2i}R_{2i+1} \mid tk_1" \mid \ldots \mid tk_n"$. *In this case, we have that* $R_i ::= R_{2i}R_{2i+1} \mid tk'_1 \mid \ldots \mid tk'_n \mid tk_1" \ldots \mid tk_n"$, *which is still Heap-CNF.*

Notice that if we have a token tk_x *that appears in both lists:* $tk'_1 \mid \ldots \mid tk'_n$ *and* $tk_1" \mid \ldots \mid tk_n"$, *then this token will appear only once—by definition—in the corresponding list of the merged grammar.*

Theorem 3.. The procedure `build_grammar` (Fig. 2) constructs grammars in Heap-CNF.

Proof 4. *The proof of Theorem 3 is the junction of two facts: (i) function* `fill_holes` *(Fig. 3) builds only grammars in Heap-CNF; and (ii) the merging of grammars (Def. 4) yields Heap-CNF grammars. To demonstrate Fact-i, notice that* `fill_holes` *only produces rules in the format* $R_i ::= token$, *or* $R_i ::= R_{2i}R_{2i+1}$; *hence, the grammar is in Heap-CNF. Fact-ii follows from Lemma 2.*

Theorem 4.. Let $G_1, G_2, \ldots, G_n$ be the grammars constructed by function `build_grammar` (Fig. 2) for input strings $t_1, t_2, \ldots, t_n$. Grammar $G_i, 1 \leq i \leq n$ recognizes every input $t_i, 1 \leq i \leq n$.

Proof 5. *The proof works by induction on the number of examples* t_i. *In the* ***base case***, `build_grammar` *fails compulsorily in the attempt to parse* t_1, *because its current grammar recognizes only the empty string, i.e.:* $R_1 ::= \epsilon$. *Failure happens in the conditional at Line 8 of Figure 2. A new grammar* G_1 *will be constructed for* t_1 *by routine* `expand_grammar`, *via function* `fill_holes`. *By Theorem 1,* G_1 *recognizes* t_1. *In the* ***inductive step***, *we have a grammar* G_k, *that recognizes every example* $t_1, \ldots, t_k$. *When* `build_grammar` *is given a new example* t_{k+1}, *two scenarios are possible:*

- G_k *recognizes* t_{k+1}; *hence, the conditional at Line 19 of Figure 2 is true.*
- G_k *fails to recognize* t_{k+1}. *In this case, a new grammar* G' *will be constructed by* `fill_holes`, *and the resulting grammar* $G_{k+1} = \texttt{merge}(G_k, G')$ *recognizes* $t_1, \ldots, t_{k+1}$, *by Lemma 2.*

We let $\texttt{merge}(G_k, G')$ *above be the grammar that results from merging* G_k *and* G'.

Lemma 3.. Let G_n be the grammar constructed by function `build_grammar` (Fig. 2) after observing inputs $t_1, t_2, \ldots, t_n$. The size of G_n is $O(N)$, where N is the number of tokens in $t_1, t_2, \ldots, t_n$.

Proof 6. *The `fill_holes` procedure only augments the rightmost node of a derivation tree. In other words, given a sentence of n tokens, `fill_holes` produces a grammar with*[5]*:*

- $2n - 1$ *non-terminal symbols;*
- $2n - 1$ *production rules;*
- n *terminal symbols;*

The `merge` routine never adds new terminals or non-terminals to a grammar; hence, it maintains its asymptotic size complexity.

Theorem 5.. Let G_n be the grammar constructed by function `build_grammar` (Fig. 2) after observing inputs $t_1, t_2, \ldots, t_n$. G_n is not ambiguous.

Proof 7. *As a consequence of Lemma 3, the rightmost derivation tree of a Heap-CNF grammar always has height $n - 1$ and $O(N)$ nodes. Only one rightmost derivation tree is possible, which Figure 5 illustrates. The rightmost token is always recognized by a production from non-terminal R_{2^n-1}.*

Corollary 2.. Let G_n be the grammar constructed by function `build_grammar` (Fig. 2) after observing inputs $t_1, t_2, \ldots, t_n$. G_n recognizes $t_i, 1 \leq i \leq n$ with O(N) derivations, where N is the number of tokens in t_i.

Proof 8. *This corollary follows from Lemma 3, plus the fact, already mentioned in the proof of Theorem 5, that only one rightmost derivation tree is possible. Thus, the grammar built by `fill_holes` recognizes a sentence with n tokens with $2n - 1$ derivations.*

Theorem 6.. Grammar G'_e produced by `markup` (Fig. 8) recognizes a subset of $lang(G_e)$ or the empty language.

[5] We treat s_e, the starting symbol of the event grammar, as a single token.

Proof 9. *The proof works by induction on the number of times Step 2 in Procedure* `markup` *runs. In the* ***base case*** *(Step 1), we have that G'_e recognizes the empty language. In the* ***inductive step****, we assume that G'_e recognizes a subset of lang(G_e) after n iterations of Step 2. In the next iteration, Steps 3 and 4 ensure that G_e" recognizes a subset of lang(G_e). The junction of G'_e and G_e" uses only production rules of G_e; hence, it must recognize a subset of the language that G_e recognizes. Furthermore, because these two grammars start with s_e, the initial symbol of G_e, the resulting grammar after the junction also starts with s_e.*

Theorem 7.. Any Heap-CNF grammar is LL(1).

Proof 10. *This fact follows from the observation that Heap-CNF grammars are not recursive. Therefore, no left recursion is possible, and the language that these grammars recognize has a finite number of possible derivation trees. The one token of lookahead follows from Definition 3 and Corollary 1, because the position of a token in the derivation tree is uniquely determined by the position of that token in the input string.*

Corollary 3.. There are languages whose grammars cannot be synthesized by Zhefuscator.

Proof 11. *A formal language is called an LL(k) language if it has an LL(k) grammar. The set of LL(k) languages is properly contained in that of LL(k+1) languages, for each k greater than or equal to zero [53]. Therefore, there exist context-free languages that are not LL(1). This restriction mean that even on the limit, Zhefuscator would not be able to synthesize perfect grammars for some languages. However, up to any number n of events, Zhefuscator will synthesize a grammar G_n that recognizes every $t_1, \ldots t_n$, and potentially other strings, as discussed in Section 3.3.2.*