`Lushu`: Obfuscating Sensitive Data via Language Recognition

Alexander T. M. Holmquist[1]

[1] Federal University of Minas Gerais (UFMG)

Abstract. *The synthesis of grammars to recognize sentences from examples is a problem that has several practical applications, including the identification and encryption of sensitive information in computer systems. Existing techniques tend to create very large grammars, having a number of terminals symbols proportional to the number of words in the example sentences. This work proposes a technique to merge grammar terminals into regular expressions. The technique uses a lattice built from a partial ordering of regular expressions. This lattice, and the language identification algorithm it entails, were used to build* `Lushu`*, a data protection tool that encrypts sensitive information produced by the Java virtual machine. A comparison between* `Lushu` *and* `Zhefuscator`*, a tool of similar purpose, demonstrates that the technique proposed in this work is not only efficient in terms of time, but also in space, producing grammars up to 10 times smaller than the current state of the art.*

1. Introduction

Language Recognition from Examples (LRE) is a classic problem in computer science. Given a set of sentences (henceforth called *examples*), we must find a grammar that recognizes them. LRE has two versions, varying based on the cardinality of the set of examples. If the set can contain infinitely many examples, then the problem is called *Language Recognition at the Boundary*. [Gold(1967)] demonstrated that this version of LRE is undecidable, even for restricted classes of languages such as regular languages. On the other hand, if the set of examples is finite, then LRE has a trivial solution: the sentences can be arranged in a prefix tree (as in [Huffman(1952)]). A solution to the finite problem can be used to approximate a solution to the infinite problem: given a sample of examples, a trivial grammar is built to recognize that language. When new examples are found, the grammar is updated.

Although trivial grammars can be constructed to recognize finite sets of sentences, these grammars tend to be large. The minimization of such grammars is a computationally difficult problem (PSPACE) [Meyer and Stockmeyer(1972)]. Thus, trivial grammars tend to use a number of terminal symbols proportional to the number of different strings contained in the examples. This work proposes a mechanism to reduce these grammars. The idea of this work is to use lattices—algebraic structures built on partial orders—to group strings into tokens. The proposed technique finds common structures among regular expressions, using, for that purpose, a domain-specific language that defines lattices. The regular expressions are built gradually, from the observation of examples. The proposed algorithm approximates the target language incrementally. Thus, new examples can be used to augment current grammar.

Application: Obfuscation of Sensitive Data. To demonstrate that the language identification algorithm proposed in this work is useful, we will show how to use it to build

a sensitive data encryption system. For that purpose, this work expands on the recent work by [Saffran et al.(2021)]. In 2021, [Saffran et al.(2021)] proposed a system for encrypting sensitive database logs. This system intercepts Java virtual machine (JVM) calls that produce strings, and creates a grammar that recognizes the language formed by all these strings. Users can mark parts of sentences that are confidential. The markup functionality allows the grammar to recognize sentences defined as secrets, and encrypt such sentences before they are published. The grammars produced by [Saffran et al.(2021)]'s tool (the `ZheFuscator`) grow quickly: they suffer from the aforementioned expansion problem. The technique proposed in this work, however, keeps this growth under control. It groups terminal symbols into regular expressions, which can be gradually updated as more examples are seen.

Results. A tool analogous to `Zhefuscator` has been implemented. The tool, called `Lushu`, intercepts JVM calls, and encrypts sensitive information present in strings issued by such calls. The interception process is invisible to users: it does not require program recoding, and has a low performance impact, as Section 4 will show. `Lushu` users must specify, via a `YAML` description, a partial order between regular expressions. `Lushu` uses this specification to build a lattice: the structure used to merge tokens into regular expressions. The current implementation of `Lushu` is similar to its original inspiration, `Zhefuscator`, in terms of performance: both tools cause a statistically negligible performance on the systems they decorate. However, as will be seen in Section 4, `Lushu` produces grammars that can be up to 10 times smaller than those produced by `Zhefuscator` to recognize the same language.

2. Overview

The ideas presented in this work arose from a practical problem, which exists in the context of a data security company, *Cyral Inc*[1]. This problem consists of obfuscating personal data contained in database logs. In order to motivate the work, this section explains this problem, and relates it to a theoretical one: the identification of infinite languages.

2.1. Data Protection Laws

Recently, personal data processing regulations were created. As an example, we have the GDPR [GDPR(2018)] in Europe, the LGPD [Lei Geral de Proteção de Dados(2018)] in Brazil, and the CCPA [CCPA(2020)] in California. These laws are not identical. Each law adapts to the needs of the region it encompasses. However, there is at least one consistent requirement: companies that hold data personal data shall protect the privacy of such data. This requirement is a challenge to fulfill for two reasons:

1. Data is manipulated by complex systems which were already in production before the laws started being enforced.
2. Data can be manipulated as untyped strings. In this context, it is not simple to distinguish what is sensitive data and what is not.

[1] `https://cyral.com/`

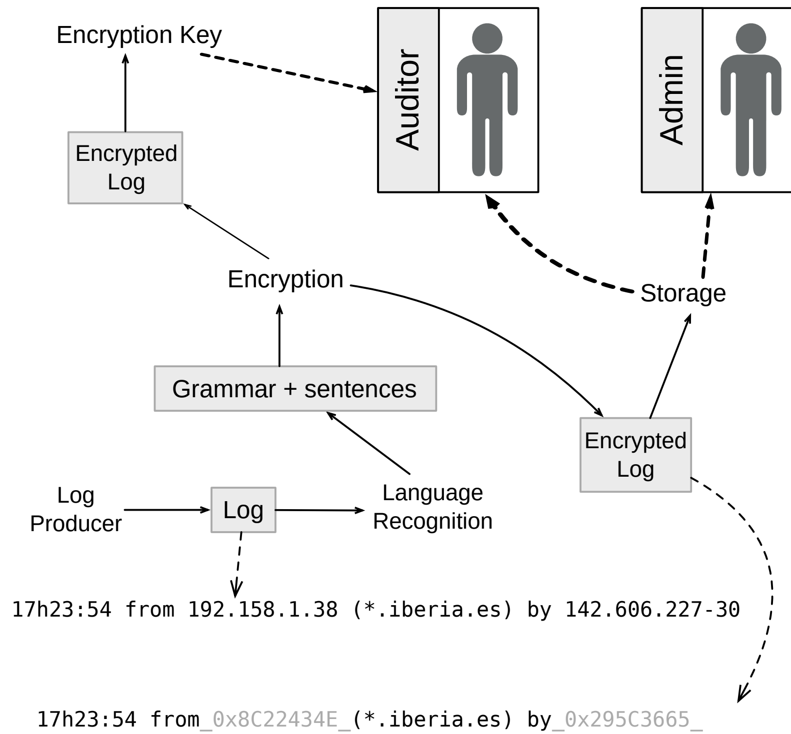

Figure 1. Obfuscation of log records in database systems. The log producer is seen as an automaton capable of producing sentences indefinitely.

2.2. Log Producers

Database systems, in general, produce reports that can be analyzed by administrators or information auditors. These systems periodically send logs to a log collector. There are specialized tools to dispatch and collect these logs: Fluentd [Fluentd(2023)], Logstash [Logstash(2023)], etc. Usually the final destination of the logs is an analysis tool like Splunk [Splunk(2023)], Sumo Logic [Sumo Logic(2023)] or Datadog [Datadog(2023)]. System administrators and data analysts use these logs to make decisions.

Every log has an origin: an application such as a database server. After the application issues a log, it goes through several paths, called *data paths*, until it reaches the hands of the analysts. Logs often contain sensitive data. Data protection laws require that the privacy of these records be preserved, either in its final form (as seen by the analyst), or in the data path, or even in the source (application). In other words, a database administrator, as opposed to an accredited auditor, should not have access to personal data. The ideas of this work can be used to obfuscate sensitive data in the source. Figure 1 illustrates this operation.

2.3. From Logs to Infinite Languages

In this work, a log producer is seen as an automaton capable of generating an infinite number of sentences. Sentences are sequences of strings (also called tokens). A system that encrypts sensitive information produced by such a producer needs to solve three challenges, namely:

Specification: Users need to be able to determine which tokens, within a sentence, contain private data. This specification must be sensitive to the context of the sen-

tence, because tokens described in a similar format may, in certain contexts, be confidential, and in others, be public.

Identification: the encryption system needs to be able to recognize the language formed by all the sentences issued by the log producer, so that confidential information may be recognized in the suitable contexts.

Compression: the growth of the grammar used to recognize the language of logs needs to be controlled, otherwise an explosion in the number of production rules may compromise execution time and memory consumption of the system being protected.

Solutions to the problem of distinguishing secrets in their context already exist in the literature. This work adopts the approach proposed by [Saffran et al.(2021)]: users use a text markup language for specifying sensitive information in examples of logs, and grammars in the *Heap-CNF* format are used to approximate the language used by the log producer to emit sentences. *Heap-BNF* are regular grammars which have a number of production rules linear on the size of the largest sentence produced by the log producer. Such grammars contain a terminal for each different symbol emitted by the producer. To avoid an explosion of the number of symbols, [Saffran et al.(2021)] defines a system of primitive types formed by integers, floating point numbers and alphanumeric strings. Users can specify the context in which tokens should be wrapped in such general categories, and the context in which tokens form tokens independent terminals. Still, grammars can grow quickly, for these contexts do not capture all types of noise, as will be seen in Section 4. The aim of this work is to further compress such grammars. The next section explores these ideas.

3. The `Lushu` System

The purpose of this section is to show how the abstract vision depicted in Figure 1 can be implemented on the Java Virtual Machine (JVM). Figure 2 provides a more concrete version of Figure 1. The tool shown in Figure 2 is called "`Lushu`". To attach `Lushu` to applications, one just needs to add a Java class defined in the `Lushu` bytecode which extends `java.io.PrintStream`. This class can be distributed as a `Jar` package, and requires minimal source code change. This methodology works at the level of Java bytecodes, and has some virtues:

1. Does not require knowledge of source code: applications are instrumented as black boxes.
2. Can be used on applications written in any language that runs on top of the JVM besides Java: Kotlin, Scala, Groovy, etc.
3. No Java programming required: sensitive data are marked via examples, the lattice is derived from a list of tokens and the classes that comprise `Lushu` are invisible to users.

3.1. Regular Expression Lattice

The main idea of this work is to use a semilattice to merge strings into regular expressions. A semilattice is an algebraic structure, built around a partially ordered set (poset), which we define as follows:

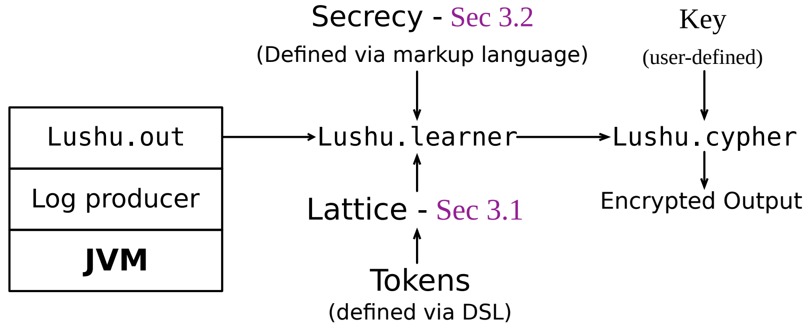

Figure 2. Integration between `Lushu` and the Java Virtual Machine.

Definition 1. *Let S be a partially ordered set based on the $\leq$ relation. We define the semilattice $\langle S \cup \{\top\}, \vee \rangle$, such that:*

- *For every pair $\{e_0, e_1\} \subseteq S$, $(e_0 \vee e_1) \in S$ is the least upper bound of e_0 and e_1, that is:*
  - $e_0 \leq e_0 \vee e_1$
  - $e_1 \leq e_0 \vee e_1$
  - *if $e_0 \leq e$ **e** $e_1 \leq e$, then $e_0 \vee e_1 \leq e$*
- *$e \leq \top$, for any element $e \in S$*

Example 1. *The set of regular languages $S = \{[a\text{-}z]+, [0\text{-}9]+, [a\text{-}z0\text{-}9]+\}$, plus the operator $\vee$ (character union in regular expressions – see Definition 2) forms a lattice. $[a\text{-}z]+ \vee [0\text{-}9]+ = [a\text{-}z0\text{-}9]+$, and that $\top = [a\text{-}z0\text{-}9]+$.*

`Lushu` users do not interact with lattices directly. Instead, they define the set of regular expressions S that can be used to build tokens. Every regular expression E in S has two constraints:

1. $E = [c_1 c_2 \ldots c_k]\{l, u\}$, where each $c_i, 1 \leq i \leq k$ is a character, and $\{l, u\} \in N, l \leq u$, is the range of acceptable sizes.
2. If E_0 and E_1 are regular expressions in S, then the intersection of characters recognized by E_0 and E_1 is empty.

Example 2 demonstrates such a set. Note that `Lushu` users only need to define S. The least upper bound operation is defined as follows:

Definition 2 (Least Upper Bound)**.** *Let $E_0 = [a_1 \ldots a_k]\ \{l_0, u_0\}$ and $E_1 = [b_1 \ldots b_k]\{l_1, u_1\}$. We define $E_0 \vee E_1 = [a_1 \ldots a_k\ b_1 \ldots b_k]\{l, u\}$, where $l = min(l_0, l_1)$ and $u = max(u_0, u_1)$.*

Example 2. *Figure 3 (a) shows the definition of a set of four regular expressions. Each regular expression is made up of a set of characters plus a range of possible lengths that these expressions can have. For example, the first expression,* alpha*, denotes any sequence of one to eight lowercase letters. For simplicity, Figure 3 omits the sublattices formed by subsets of* alpha *and* Alpha *with different size ranges. The sublattice of* digit *is shown in full.*

3.1.1. Heap-CNF Grammars

Lattices enable us to merge tokens of a trivial grammar into regular expressions. This process will be explained via a series of examples. The first example shows what a trivial

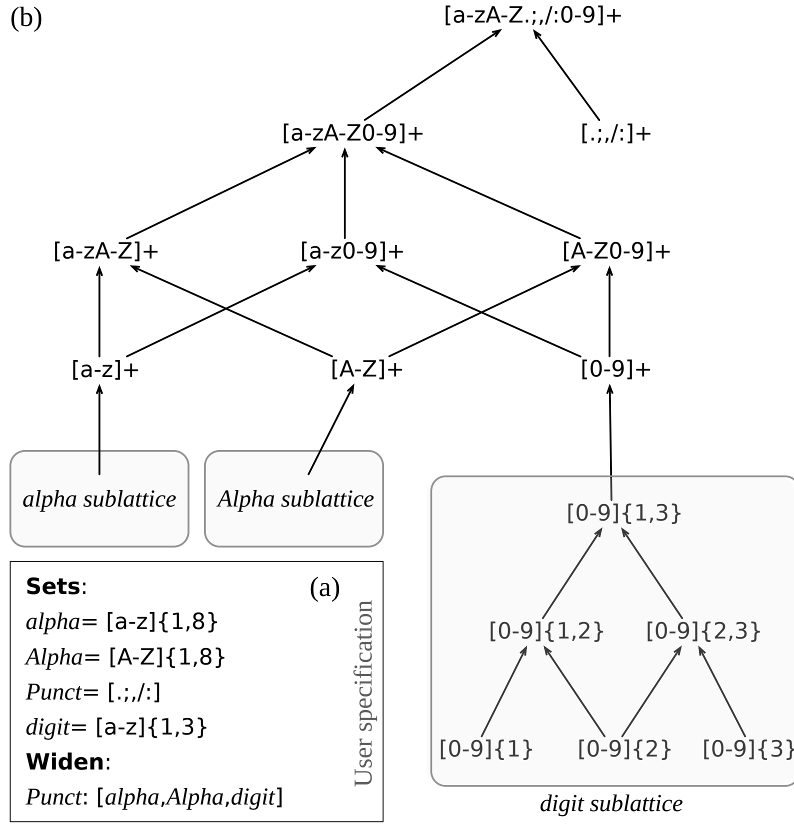

Figure 3. Lattice that recognizes CPF numbers and IP addresses.

grammar is and why it grows so fast.

Example 3. *Figure 4 (b) shows the trivial grammar produced to recognize the two sentences seen in Figure 4 (a). Note that a terminal is created for each token. In other words, no attempt is made to join tokens into regular expressions. Figure 4 (c) shows the corresponding grammar that would be produced by the* `Zhefuscator` *[Saffran et al.(2021)] tool. This tool recognizes some primitive types, such as integers and floating point numbers. In this case, the first two tokens,* `7282` *and* `7283`*, are grouped as instances of the integer type.*

The grammars shown in Example 3 follows a known format called *Heap-CNF* (*Heap-Chomsky Normal Form*) [Saffran et al.(2021)]. `Lushu` recognizes grammars in this format. Grammars in Heap-CNF format are trivial grammars that recognize a set finite number of L sentences. Tokens at position $k, k \geq 1$, of sentence L are recognized from the non-terminal R_{2k}. To keep this work self-contained, Definition 3 revisits this concept.

Definition 3 (Heap-CNF ([Saffran et al.(2021)])). *A Heap-CNF grammar of height n has the following non-terminal symbols: $R_0, R_1, \ldots, R_{2n-2}, R_{2n-1}$, and production rules that match one of the following three patterns, where the symbol $\mathtt{a}_x$ is a wildcard for any terminal:*

1. $R_{2k} ::= \mathtt{a}_1 \mid \mathtt{a}_2 \mid \ldots \mid \mathtt{a}_p$ *se* $k < n$
2. $R_{2k-1} ::= R_{2k}\ R_{2k+1} \mid \epsilon$

(a) First *log* :

```
7282 1:15 May 1st
```

Second *log* :

```
7283 10:25 05 2nd
```

(b)

```
R1 ::= R2 R3
R2 ::= `7282` | `7283`
R3 ::= R4 R5
R4 ::= `1:15` | `10:25`
R5 ::= R6 R7
R6 ::= `May` | `5`
R7 ::= `1st` | `2nd`
```

(c)

```
R1 ::= R2 R3
R2 ::= <int>
R3 ::= R4 R5
R4 ::= `1:15` | `10:25`
R5 ::= R6 R7
R6 ::= `May` | `5`
R7 ::= `1st` | `2nd`
```

(d)

```
R1 ::= R2 R3
R2 ::= <Digit>{4,4}
R3 ::= R4 R5
R4 ::= <Digit>{1,2}
R5 ::= R6 R7
R6 ::= <Punct>+
R7 ::= R8 R9
R8 ::= <Digit>{2,2}
R9 ::= R10 R11
R10 ::= xx
R11 ::= R12 R13
R12 ::= <Digit>{1,1}
R13 ::= <Alpha>{2,2}
```

Where **xx** is <digit ∨ alpha ∨ Alpha>{2,3}

Figure 4. (a) Two sentences emitted by the log producer. (b) Trivial grammar that recognizes the two sentences. (c) Grammar produced by `Zhefuscator`. (d) Grammar produced by `Lushu`.

3. $R_{2n-1} ::= \mathtt{a}_1 \mid \mathtt{a}_2 \mid \ldots \mid \mathtt{a}_p$

3.1.2. Using the Lattice to Merge Tokens

In the discussion that follows, S is the set defined by the `Lushu` user. In Figure 3 (a), this set is formed by the four regular expressions mentioned in Example 2. The tool maintains a Heap-CNF grammar G capable of recognizing all sentences issued by the log producer up to the current point in time. Whenever a new sentence V is received, the following actions take place:

1. The V sentence is partitioned into a list T of regular expressions in S.
2. T is merged into G via the lattice defined by S and the union operator for regular expressions.

The remainder of this section describes these two actions.

Splitting Sentences We split sentence V into tokens using the lattice defined by the set of regular expressions S. A *token* is defined as the longest sequence of characters that compose a regular expression other than the top of the lattice. In other words, if v is a contiguous sequence of characters from V that compose a token, v has the following properties:

1. for each character $c \in v$, let $s \in S$ be the expression that contains it. The expression that recognizes v is the least upper bound of all expressions s.
2. we define the interval $e\{l, u\}$ as $\{|s|, |s|\}$, if $l \leq |s| \leq u$. Otherwise, we use $e+$.
3. adding one more character to v makes s the top of the lattice.

Example 4 illustrates this process.

Example 4. *Figure 5 (a) shows how to partition the first sentence emitted by the log producer in Figure 4. Each token in the target sentence can be represented via one of the base sets seen in Figure 3 (a); except for the token* `May`*, because it contains both lowercase and uppercase letters. In this case, the token is recognized by the least upper bound of the sets* alpha *and* Alpha*.*

(a)

where **ww** = (alpha ∨ Alpha){3,3}

digit{4,4}	digit{1,1}	Punct{1,1}	digit{2,2}	**ww**	digit{1,1}	alpha{1,1}
7282	1	:	15	May	1	st

(b)

7283	10	:	25	05	2	nd
digit{4,4}	digit{2,2}	Punct{1,1}	digit{2,2}	(digit){2,2}	digit{1,1}	alpha{1,1}

(c)

digit{4,4}	digit{1,2}	Punct{1,1}	digit{2,2}	**xx**	digit{1,1}	alpha{1,1}

where **xx** = (alpha ∨ Alpha ∨ digit){2,3}

Figure 5. (a-b) Partitioning sentences into tokens using the lattice seen in Figure 3 (b). (c) Final tokens, obtained via least upper bound operations.

Merging Sentences After a sentence is received and partitioned into tokens, it needs to be incorporated into the current grammar. Generally speaking, this operation is equivalent to applying a reduction on all sentences issued by the log producer up to a given moment. The reduction operation consists of applying the least upper bound operation, token by token. The next example illustrates such an operation.

Example 5. *Figure 5 (c) shows the regular expressions that result from joining the regular expressions in Figures 5 (a) and (b). Figure 6 shows the grammar that recognizes the concatenation of all the regular expressions in Figure 5 (c).*

3.1.3. The Widening Operation

It is desirable that some join operations lead directly to the top of the lattice, rather than following the Definition 2. For example, users might want the string `1:15` to be recognized as the concatenation of three regular expressions, namely: $digit\{1,1\}Punct\{1,1\}$

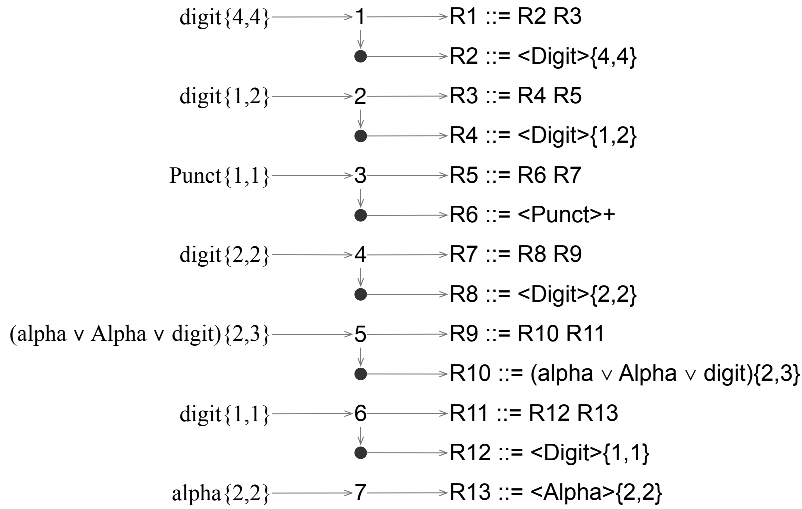

Figure 6. Process of building a Heap-CNF grammar that recognizes the concatenation of several regular expressions.

$digit\{2, 2\}$ instead of the regular expression $(digit \vee Punct)\{4, 4\}$. We must prohibit the combination of the expressions *digit* and *Punct*. `Lushu` allows users to specify forbidden joins via a *widening* operation. The **Widen** block of a specification indicates which joins lead directly to the top of the lattice. Example 6 illustrates the semantics of this block.

Example 6. *Figure 3 (a) contains a* ***Widen*** *block. This block indicates that the least upper bound of* Punct *and any another regular expression is the top of the lattice. Thus, elements like* $[a - z.; / :]+$ *(which would be* $alpha \vee Punct$*) do not exist in the lattice seen in Figure 3 (b).*

3.2. Marking Sensitive Data

The `Lushu` user determines tokens that should be obfuscated from log examples. With an example in hand, the user uses pairs of XML elements of the type `<s>` and `<\s>` to delimit sensitive tokens. Any substring between `<s>` and `<\s>` is considered sensitive, as long as it appears in the position where the marked text is located. This step takes place before activating `Lushu` on the virtual machine Java. Once activated, any occurrences of tokens with the specified format are encrypted. If tokens of this type are merged into a more complex expression, any string recognized by this new super expression will also be encrypted. Example 7 illustrates the example markup `Lushu` needs to obfuscate the log seen in Figure 1.

Example 7. *Figure 7 shows text with* `Lushu` *markup. This markup determines that regular expressions in the format of CPF numbers and IP addresses must be obfuscated, but only when they occur as the third and fifth words of sentences. Thus, if a CPF number occurs as the second token of a log (which happens in logs of type* `DEBUG`*), then that CPF number will not be encrypted.*

4. Evaluation

The purpose of this section is to discuss the four research questions below, where "decorated system" is the JVM instance whose output is intercepted by `Lushu`:

User secrecy markup on examples:

```
7282, from <s> 631-306-734/74 </s> at <s> 192.158.1.38 </s>
DEBUG 127.0.0.1 7283, from <s> 631-306-734/74 </s>
```

Original text emitted by JVM:

```
15278, from 435-249-572/77 at 192.158.1.38
15279, from 165-022-110/03 at 74.199.177.134
DEBUG 127.0.0.1 15280, from 74.199.177.134
```

Text emitted by JVM after being obfuscated by Lushu :

```
15278, from 0x83740293 at 0x88293874
15279, from 0x36838473 at 0x3A8D75E9
DEBUG 127.0.0.1 15280, from 0x3A8D75E9
```

Figure 7. Markup and obfuscation of sensitive information.

QP1: What is the impact of Lushu on the execution time of the decorated system?
QP2: What is the impact of Lushu on the memory consumption of the decorated system?
QP3: How effective is Lushu for compressing grammars, when compared to its predecessor, Zhefuscator?
QP4: How fast does the Lushu grammar converge, when given real database logs?

Software: JVM version: 17.0.6. Kotlin Version: 1.7.10. Operating system: Ubuntu 20.04.6 LTS. PostgreSQL Version: 14.7.

Hardware: CPU: Intel i7-1065G7 @ 3.90GHz, 4 cores. Primary Memory: Samsung, 16GB @ 3200 MT/s.

Benchmarks: This section uses two benchmarks: a program that generates random logs (henceforth random log generator), and real logs produced by PostgreSQL. The random log generator is capable of producing 1,858,950 different permutations of a predefined set of words. Each log contains a CPF number, a date, a timestamp, a random integer, or a combination of these.

Example 8. *Examples of logs emitted by the random log generator:*

```
A new product review was submitted by Nathan
  on 2023-05-10 22:18:34.
An order was placed by customer WDTPAPuv on
  2023-05-10 22:18:34.
A new message was received from Mason on
  2023-05-10 22:18:34.
The user 219.260.870-06 deleted their
  account on 2023-05-10 22:18:34.
```

The PostgreSQL logs allow us to evaluate Lushu in a more realistic scenario. These logs were generated while running the pgbench tool against the database. pgbench emulates the activity of a real database[2]. Although the records that make up these logs

[2] https://www.postgresql.org/docs/current/pgbench.html (May 15th, 2023).

can have an arbitrary number of tokens, their format is less diverse than the random logs. Example 9 shows some records.

Example 9. *Examples of the contents of* actual_log *in PostgreSQL logs format:*

```
statement: UPDATE pgbench_accounts SET abalance
  = abalance + -1147 WHERE aid = 28677;
statement: SELECT abalance FROM pgbench_accounts
  WHERE aid = 52176;
duration: 0.081 ms
```

4.1. QP1: Execution Time

To measure the impact of `Lushu` on decorated applications, we measured its performance over a JVM instance that only prints logs, without any further processing. The number of records issued is a parameter. In this experiment, the number of logs produced by each run varies between 10^0 and 10^6. In this setup, the JVM instance is fully dedicated to emitting logs: no processing takes place between issuing one record and the next.

Discussion: Figure 8 compares the execution time of different applications, with and without the interception performed by `Lushu`. The running times shown are the arithmetic mean of the times measured in 10 independent runs. The running time tends to grow linearly with the number of logs. Applications whose logs are intercepted by `Lushu` are slower. However, this impact is only noticeable if the application outputs many logs in a row—an unlikely scenario for most real applications. The increasing distance between the curves is probably due to *buffering* effects, and does not seem to be related with the computational cost of `Lushu`.

4.2. QP2: Memory Consumption

This experiment is similar to the one described in Section 4.1, except that we measure primary memory consumption instead of execution time. The memory reported in this section is the difference between the memory reserved for the Java process (`Runtime.totalMemory()`) and the memory reserved for object allocation (`Runtime.freeMemory()`). The numbers obtained are the average of ten samples. Before each sample, the JVM's garbage collector is invoked.

Discussion: Figure 9 shows the impact of `Lushu` on memory consumption. The impact of `Lushu` is remarkable; however, it is constant in the two sampled scenarios. Memory consumption grows only 2% between issuing one and a million logs, becoming constant after about one thousand records. The initial 2% growth is due to the growth of the grammar used by `Lushu`.

4.3. QP3: Grammar Compression Rate

We compare the ability to compress grammars of `Lushu` with its predecessor, `Zhefuscator`. To simulate the behavior of `Zhefuscator`, we use a lattice that distinguishes sequences of digits from other tokens. Thus, any characters from the alphabet, except numbers, is not merged into regular expressions. In this experiment, we intercept 1 to 2001 records

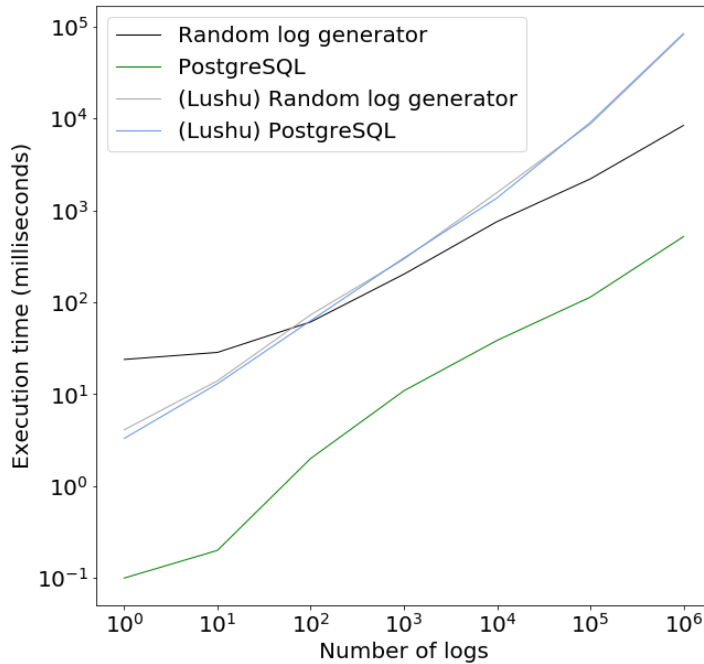

Figure 8. Impact of `Lushu` interception in the execution time of the log producer.

issued by the random log generator. For each number of records, we compute the average size of the `Lushu` grammar in ten different runs. The size of the grammar is measured as the number of terminal symbols it contains.

Discussion: Figure 10 shows the size of the grammar (in number of terminals) generated by `Zhefuscator` and `Lushu`. `Lushu`'s grammar converges with less than 100 logs, while `Zhefuscator`'s grammar does not converge even after observing 2001 records. The grammar size of `Zhefuscator` after 2001 logs is more than 10 times larger than that of *Lushu*. `Zhefuscator` convergence requires that all terminal symbols are observed—a scenario that does not occur in this experiment.

4.4. QP4: Convergence Rate for Real Logs

`Zhefuscator`'s design was based on the premise that logs real tend to have a very well-defined format. However in this case, the grammar produced by `Lushu` also converges much faster. Consider, for example, the convergence rate in logs produced by `pgbench` from PostgreSQL. After consuming 32 lines, `Lushu` produces a grammar with 69 non-terminal symbols and 118 terminal symbols. After consuming a total of 155 lines, the number of terminals only increases by one. Having reached 119 terminal symbols, the size of the grammar remains constant, even after consuming more than a million records.

5. Related Work

This work presents a form of *inductive language synthesis*. This problem seeks to build an automaton that recognizes a language from positive and negative examples of sentences.

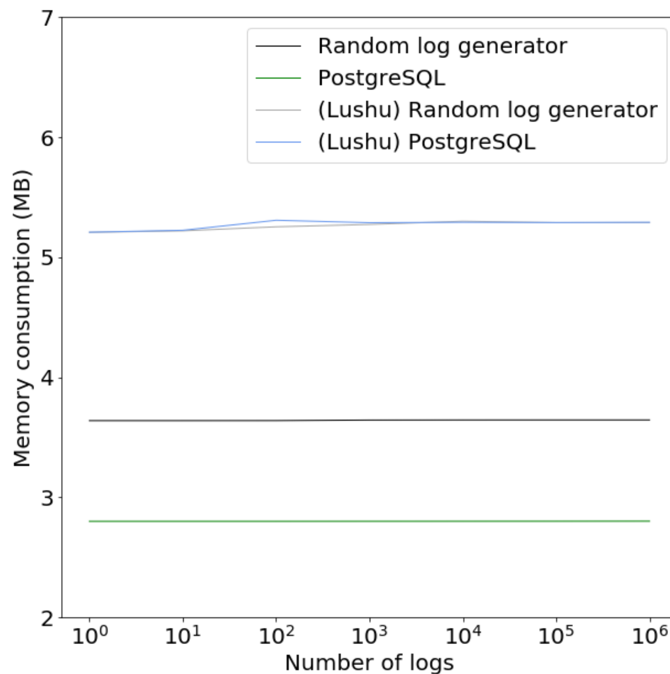

Figure 9. Impact of `Lushu` interception in the memory consumption of the log producer.

The former belong to the language; the latter don't. Inductive language synthesis has its origins in the work of [Gold(1967)]. The version of the problem studied by [Gold(1967)] aims to recognize the language "in the limit", that is, from a potentially endless stream of positive examples. In that version, a "recognizer" is built incrementally from the seen examples. Our work fits into this theoretical framework.

The inductive synthesis of languages can incorporate negative examples, which are sentences that do not belong to the language. Much of the related literature is based on the work of [Angluin(1987)]. [Angluin(1987)] presents a framework formed by a learner (as in the work of [Gold(1967)]) and by a teacher: the source of information. The learner must identify a regular language that the teacher knows. The learning process follows a trial-and-error strategy. The learner presents a deterministic finite automaton (DFA) to the teacher. If their conjecture is correct, the teacher answers affirmatively. Otherwise, the teacher shows them a counterexample—a string that the automaton does not recognize, or recognizes but is not in the objective language. [Angluin(1987)]'s work (and the many works she inspired) depends on the existence of the teacher role. The present work differs from the [Angluin(1987)] framework in that it does not encompass the teacher role and does not use negative examples.

The techniques presented in this work are similar to the work of [Shcherbakov(2016)]. Similarities include incremental learning and the availability only of positive examples. The algorithm proposed by [Shcherbakov(2016)] is based on alignment of sequences: the same principle used, for example, in genetic sequencing. The algorithm proposed in the

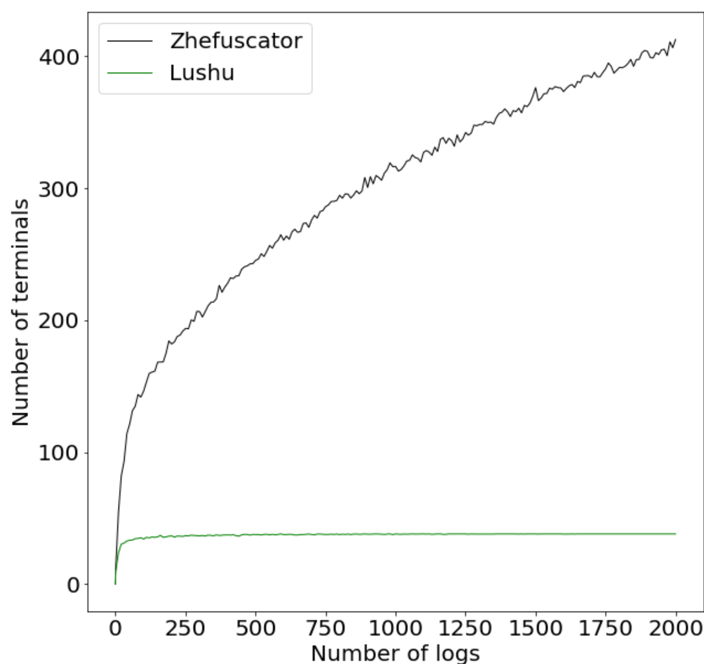

Figure 10. Comparison between the size of the grammars produced by `Lushu` and `Zhefuscator`.

present work is based on a lattice, which is extracted from a user-built specification. In that sense, [Shcherbakov(2016)]'s technique is more automatic: while our work presupposes the lattice and the examples, the technique of [Shcherbakov(2016)] only needs the examples. However, [Shcherbakov(2016)]'s algorithm is quadratic in the size of the examples: if the longest example has $O(N)$ characters, and there are $O(X)$ examples, the algorithm has complexity $O(X \times N^2)$. The algorithm presented in this work has lower complexity: $O(X \times N)$.

6. Conclusion

We have presented a system for compressing grammars that recognize a finite sequence of example sentences. The proposed technique consists in joining tokens that occur in the same positions of different sentences into regular expressions. Tokens are joined according to the structure of a semilattice. This semilattice emerges from a user-defined specification of character sets. The proposed ideas were incorporated into an interception system that automatically encrypts sensitive information, known as `Lushu`. This system is free software, available under the GPL 3.0 license. Although it can already be used in practical situations, there are still several directions along which `Lushu` could be improved. In particular, the proposed ideas are only able to reduce tokens, but not entire production rules. It is expected that this limitation will be overcome by future developments of this work.

References

[Angluin(1987)] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and computation* 75, 2 (1987), 87–106.

[CCPA(2020)] CCPA. 2020. https://oag.ca.gov/privacy/ccpa

[Datadog(2023)] Datadog. 2023. https://www.datadoghq.com/

[Fluentd(2023)] Fluentd. 2023. https://www.fluentd.org/

[GDPR(2018)] GDPR. 2018. https://gdpr-info.eu/

[Gold(1967)] E Mark Gold. 1967. Language identification in the limit. *Information and control* 10, 5 (1967), 447–474.

[Huffman(1952)] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers* 40, 9 (September 1952), 1098–1101.

[Lei Geral de Proteção de Dados(2018)] Lei Geral de Proteção de Dados. 2018. https://www.gov.br/cidadania/pt-br/acesso-a-informacao/lgpd

[Logstash(2023)] Logstash. 2023. https://www.elastic.co/logstash/

[Meyer and Stockmeyer(1972)] A. R. Meyer and L. J. Stockmeyer. 1972. The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space. In *Proceedings of the 13th Annual Symposium on Switching and Automata Theory (Swat 1972) (SWAT '72)*. IEEE Computer Society, USA, 125–129. https://doi.org/10.1109/SWAT.1972.29

[Saffran et al.(2021)] João Saffran, Haniel Barbosa, Fernando Magno Quintão Pereira, and Srinivas Vladamani. 2021. On-line synthesis of parsers for string events. *Journal of Computer Languages* 62 (2021), 101022. https://doi.org/10.1016/j.cola.2021.101022

[Shcherbakov(2016)] Andrei Shcherbakov. 2016. A Branching Alignment-Based Synthesis of Regular Expressions. In *AIST (Supplement)*. Springer, Berlin, Germany, 315–328.

[Splunk(2023)] Splunk. 2023. https://www.splunk.com/

[Sumo Logic(2023)] Sumo Logic. 2023. https://www.sumologic.com/