Lushu: Ofuscação de Dados Sigilosos via Reconhecimento de Linguagens a partir de Exemplos

Anonymous Author(s)

Resumo

A construção de gramáticas para reconhecer sentenças a partir de exemplos é um problema que possui diversas aplicações práticas, incluindo a identificação e ciframento de informações sigilosas em sistemas computacionais. Técnicas atualmente em uso para resolver esse problema tendem a criar gramáticas muito grandes, com um número de símbolos terminais proporcional à quantidade de palavras nas sentenças de exemplo. Este artigo propõe uma técnica de fusão de terminais em expressões regulares. Tal técnica utiliza um reticulado construído a partir de um ordenamento parcial de expressões regulares. Tal reticulado, e o algoritmo de identificação de linguagens que ele enseja, foi utilizado para construir `Lushu`, uma ferramenta de proteção de dados pessoais que cifra informações sigilosas produzidas pela máquina virtual Java. Uma comparação entre `Lushu` e `Zhefuscator`, uma ferramenta de propósito similar, demonstra que a técnica proposta neste trabalho não somente é eficiente em termos de tempo, mas também de espaço, produzindo gramáticas até 100 vezes menores que o atual estado da arte.

Keywords: obfuscation, language recognition, compiler optimizations

ACM Reference Format:
Anonymous Author(s). 2022. Lushu: Ofuscação de Dados Sigilosos via Reconhecimento de Linguagens a partir de Exemplos. In *XXVI Brazilian Symposium on Programming Languages (SBLP 2022), October 6–7, 2022, Virtual Event, Brazil.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3561320.3561322

1 Introdução

O Reconhecimento de Linguagens a partir de Exemplos (RLE) é um problema clássico em ciência da computação. Dado um conjunto de sentenças (doravante chamadas *exemplos*), o problema pede a gramática que reconheça todas essas *strings*. RLE possui duas versões, que dependem da cardinalidade do conjunto de exemplos. Caso o conjunto possa conter infinitos exemplos, então o problema é chamado de *Reconhecimento de Linguagens no Limite*. Gold [6] demonstrou que essa versão de RLE é indecidível, mesmo para classes restritas de linguagens, como as linguagens regulares. Por outro lado, caso o conjunto de exemplos seja finito, então RLE possui solução trivial: as sentenças podem ser arranjadas em uma árvore de prefixos (à la Huffman [7], por exemplo). Uma solução para o segundo problema pode ser usada para aproximar uma solução para o primeiro: dado um prefixo da sequência de exemplos, uma gramática trivial é construída para reconhecer aquela linguagem. Uma vez encontrados novos exemplos, a gramática deve ser atualizada.

Embora gramáticas triviais possam ser construídas para reconhecer conjuntos finitos de sentenças, essas gramáticas tendem a ser grandes. A minimização de tais gramáticas é um problema computacionalmente difícil (PSPACE) [10]. Assim, gramáticas triviais tendem a usar uma quantidade de símbolos terminais proporcional à quantidade de *strings* diferentes contidas nos exemplos. Este artigo propõe um mecanismo para reduzir essa quantidade de *tokens*. A ideia deste trabalho é utilizar reticulados—estruturas algébricas construídas sobre ordens parciais—para agrupar *strings* em tokens. A técnica proposta encontra estruturas comuns entre expressões regulares, usando, com tal propósito, uma linguagem de domínio específico que define reticulados. Essas expressões regulares são construídas gradativamente, a partir da observação de exemplos. O algoritmo proposto aproxima a linguagem alvo de forma incremental. Assim, novos exemplos podem ser usados para aumentar a gramática corrente.

Aplicação: Ofuscação de Dados Sigilosos. Para demonstrar que o algoritmo de identificação de linguagens proposto neste artigo é útil, demonstraremos como utilizar tal algoritmo para construir um sistema de ciframento de dados sigilosos. Com tal propósito, este artigo expande o recente trabalho de Saffran et al. [11]. Em 2021, Saffran et al. propuseram um sistema de encriptação de relatórios (*logs*) produzidos por bancos de dados. Esse sistema intercepta chamadas da máquina virtual Java (JVM) que produzem *strings* públicas, e cria uma gramática que reconhece a linguagem formada por todas essas *strings*. Usuários podem marcar partes de sentenças que são sigilosas. Tal marcação permite à gramática criada reconhecer sentenças definidas como segredos, e cifrar tais sentenças antes delas serem publicadas. As gramáticas produzidas pela ferramenta de Saffran et al. (o `ZheFuscator`) crescem rapidamente, sofrendo do mesmo problema de expansão anteriormente mencionado. A técnica proposta neste

SBLP 2022, October 6–7, 2022, Virtual Event, Brazil

ACM ISBN 978-1-4503-9744-5/22/10…$15.00
https://doi.org/10.1145/3561320.3561322

trabalho, contudo, permite manter esse crescimento sob controle, agrupando símbolos terminais em expressões regulares, as quais podem ser gradativamente atualizadas, conforme mais exemplos sejam vistos.

Compêndio de Resultados. Uma ferramenta análoga ao `Zhefuscator` foi implementada. Tal ferramenta, chamada `Lushu`, intercepta chamadas da JVM, e cifra informações sigilosas presentes em *strings* emitidas por tais chamadas. Esse processo de interceptação e ciframento é invisível para usuários: ele não exige recodificação de programas, e possui baixo impacto de desempenho, conforme a Seção 4 irá mostrar. Usuários de `Lushu` devem especificar, via uma descrição `JSON`, uma ordem parcial entre expressões regulares. `Lushu` usa tal especificação para construir um reticulado: a estrutura usada para amalgamar *tokens* em expressões regulares. A atual implementação de `Lushu` é similar a sua inspiração original, o `Zhefuscator`, em termos de desempenho: ambas as ferramentas causam um desempenho estatisticamente negligível sobre os sistemas que decoram. Contudo, conforme será visto na Seção 4, `Lushu` produz gramáticas que podem ser até 100 vezes menores que aquelas produzidas por `Zhefuscator` para reconhecer a mesma linguagem.

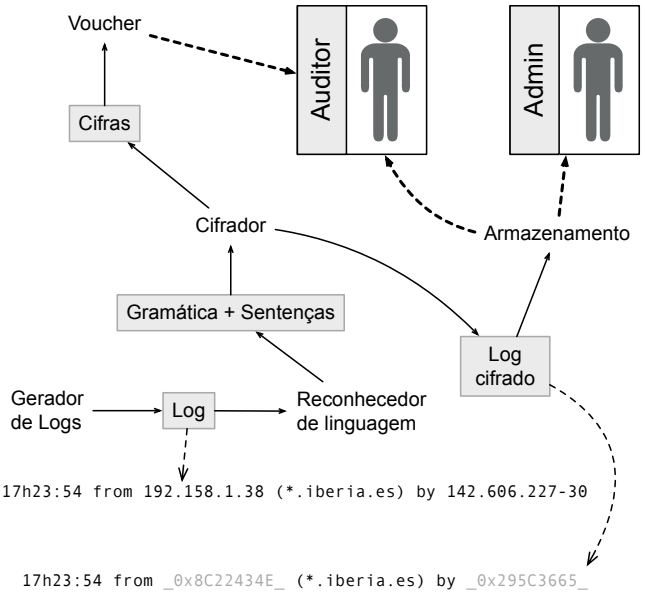

Figura 1. Ofuscação de registros de *log* em sistemas de bancos de dados. Neste contexto, o gerador de *logs* é visto como um autômato capaz de produzir sentenças infinitamente.

2 Visão Geral

As ideias apresentadas neste trabalho surgiram de um problema prático, que existe no contexto de uma empresa de segurança de dados, a *Cyral Inc*[1]. Este problema consiste em ofuscar dados pessoais em relatórios de *logs* de bancos de dados. A fim de motivar o trabalho, esta seção explica tal problema, e o relaciona com um problema teórico: a identificação de linguagens infinitas.

2.1 Leis Gerais de Proteção de Dados

Recentemente, foram criadas leis que regulamentam o tratamento de dados pessoais. Como exemplo, temos a Lei Geral de Proteção de Dados (LGPD) [8] no Brasil, a GDPR [5] na Europa, e a CCPA [2] na Califórnia. Essas leis não são idênticas, adequando-se a particularidades de cada região. Porém uma obrigação parece ser consistente entre elas: empresas que abrigam dados pessoais devem proteger a privacidade desses dados. Essa obrigação é desafiadora por duas razões:

1. Dados são manipulados por sistemas complexos, que já se encontravam em regime de produção antes do advento das leis de proteção de informação.
2. Dados podem ser manipulados como *strings* não tipadas, e, neste contexto, não é simples distinguir o que é dado sigiloso e do que não é.

2.2 Geradores de Logs

Sistemas de bancos de dados, em geral, produzem relatórios que podem ser analisados por administradores ou auditores de informação. Esses sistemas enviam registros periodicamente para um coletor de *logs*. Existem ferramentas especializadas para despachar e coletar esses *logs*: Fluentd [4], Logstash [9], etc. Geralmente o destino final dos registros enviados é uma ferramenta de análise como o Splunk [13], Sumo Logic [14] ou Datadog [3]. Administradores de sistema e analistas de dados utilizam esses *logs* para tomar decisões.

Todo *log* tem uma origem: uma aplicação como um servidor de banco de dados, por exemplo. Depois que a aplicação emite um registro, ele atravessa caminhos diversos, denominados *caminhos de dados*, até chegar às mãos dos analistas. *Logs* frequentemente contêm dados sensíveis. Leis de proteção de dados exigem que a privacidade desses registros seja preservada, seja em sua forma final (como vista pelo analista), seja no caminho de dados, ou mesmo na origem (aplicação). Em outras palavras, um administrador de banco de dados, ao contrário de um auditor credenciado, não deveria ter acesso a dados pessoais. Nesse contexto, as ideias usadas neste artigo serão usadas para ofuscar dados pessoas em registros. A figura 1 ilustra tal funcionamento.

2.3 De Logs para Linguagens Infinitas

Neste trabalho, um gerador de *logs* é visto como um autômato capaz de gerar uma quantidade infinita de sentenças. Sentenças são sequências de *strings* (também chamados *tokens*). Um sistema que cifre informações sensíveis produzidas por um tal gerador precisa de resolver três desafios, a saber:

[1] https://cyral.com/

Especificação: usuários precisam ser capazes de determinar quais *tokens*, dentro de uma sentença, contêm informação sigilosa. Essa especificação deve ser sensível ao contexto da frase, pois *tokens* descritos por similar formato podem, em certos contextos, serem sigilosos, e em outros, serem públicos.

Identificação: o sistema de ciframento precisa ser capaz de reconhecer a linguagem formada por todas as sentenças emitidas pelo gerador de *logs* até um certo momento, para que informação sigilosa possa ser reconhecida nos contextos adequados.

Compressão: o crescimento da gramática utilizada para reconhecer a linguagem de *logs* precisa ser controlado, pois, de outro modo, a explosão de regras de produção pode comprometer o tempo de execução e o consumo de memória do sistema que está sendo protegido.

Soluções para o problema da especificação de segredos e identificação de linguagens já existem na literatura. Este artigo adota a abordagem proposta por Saffran et al. [11]: usuários usam uma linguagem de marcação de texto para especificar informações sigilosas em exemplos de *logs*, e gramáticas no formato *Heap-BNF* são utilizadas para aproximar a linguagem utilizada pelo gerador de *logs* para emitir sentenças. Gramáticas *Heap-BNF* são regulares, contendo um número de regras de produção linearmente proporcional ao tamanho da maior sentença produzida pelo gerador de *logs* até um certo momento. Tais gramáticas contém um terminal para cada símbolo diferente emitido pelo gerador. Para evitar uma explosão de símbolos, Saffran et al. define um sistema de tipos primitivos formado por números inteiros, números de ponto flutuante e *strings* alfa-numéricas. Usuários podem especificar o contexto em que *tokens* devem ser agrupados em tais categorias gerais, e o contexto em que *tokens* formam símbolos terminais independentes. Ainda assim, gramáticas podem crescer rapidamente, conforme será visto na Seção 4. O objetivo deste artigo é comprimir ainda mais tais gramáticas; e, dessa forma, propor soluções para o desafio de compressão. A próxima seção explora tais ideias.

3 O Sistema Lushu

O objetivo desta seção é mostrar como a visão abstrata retratada na Figura 1 pode ser implementada sobre a Máquina Virtual Java (JVM). A Figura 2 provê uma visão concreta das abstrações vistas na Figura 1. Denomina-se "Lushu"a ferramenta mostrada na Figura 2. Para acoplar Lushu sobre aplicações, basta adicionar a tais aplicações uma nova classe Java, que extende `System.out`. Essa classe é um pacote `Jar`, e não exige ré-escrita de código fonte. Essa metodologia funciona ao nível de *bytecodes* Java, e tem algumas virtudes:

1. Não requer conhecimento do código fonte: aplicações são instrumentadas como caixas pretas.

Figura 2. Integração de Lushu à Máquina Virtual Java.

2. Pode ser usada sobre aplicações escritas em qualquer linguagem que executa sobre a JVM, além de Java: Kotlin, Scala, Groovy, etc.
3. Não requer programação em Java: sigilos são especificados via exemplos, o reticulado é deverivado de uma lista de *tokens* e as classes que compõem Lushu são invisíveis para os usuários.

3.1 Reticulado de Expressões Regulares

A ideia principal deste trabalho é usar um (semi)-reticulado para amalgamar *strings* em expressões regulares. Um semi-reticulado é uma estrutura algébrica, construída em torno de um conjunto parcialmente ordenado, que definimos da seguinte forma:

Definição 3.1. Seja S um conjunto parcialmente ordenado por uma relação $\leq$. Define-se o semi reticulado limitado $\langle S \cup \{\top\}, \vee\rangle$, tal que:

- Para todo par $\{e_0, e_1\} \subseteq S$, tem-se que $(e_0 \vee e_1) \in S$ é a menor quota superior de e_0 e e_1, isto é:
  - $e_0 \leq e_0 \vee e_1$
  - $e_1 \leq e_0 \vee e_1$
  - se $e_0 \leq e$ e $e_1 \leq e$, então $e_0 \vee e_1 \leq e$
- $e \leq \top$, para qualquer elemento $e \in S$

Exemplo 3.2. O conjunto de linguagens regulares $S = \{$[a-z]+, [0-9]+, [a-z0-9]+$\}$, mais o operador $\vee$ (união de caracteres em expressões regulares – vide Definição 3.3) forma um reticulado. Tem-se que [a-z]+ $\vee$ [0-9]+ = [a-z0-9]+, e que $\top$ = [a-z0-9]+.

Usuários de Lushu não interagem com reticulados diretamente. Em vez disso, eles definem o conjunto S de expressões regulares que podem ser usadas para formar *tokens*. Cada expressão regular E em S têm duas restrições:

1. $E = [c_1 c_2 \ldots c_k]\{l, u\}$, onde cada $c_i, 1 \leq i \leq k$ é um caracter, e $\{l, u\} \in \mathbb{N}, l \leq u$, é o intervalo de tamanhos aceitáveis.
2. Se E_0 e E_1 são expressões regulares em S, então a interseção de caracteres reconhecidos por E_0 e E_1 é vazia.

O Exemplo 3.2 demonstra um tal conjunto. Notem que os usuários de Lushu definem somente S. A operação de menor quota superior é definida da seguinte forma:

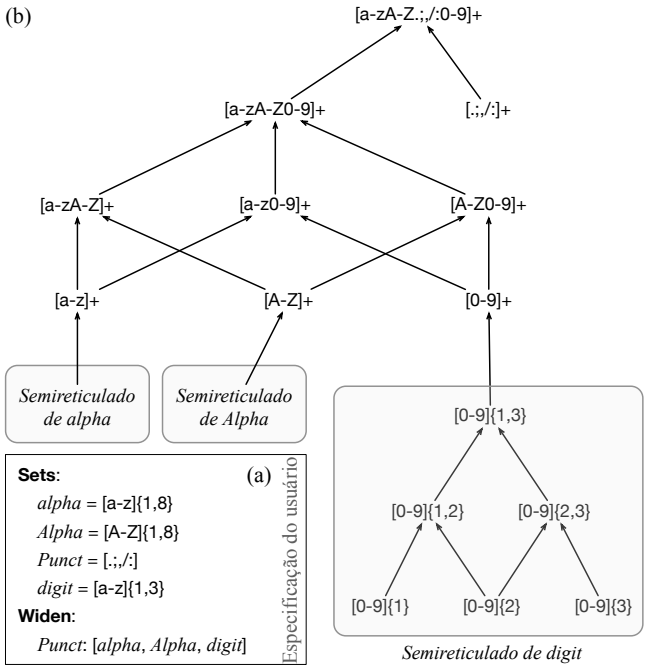

Figura 3. Reticulado usado para reconhecer CPFs e IPs.

Definição 3.3 (Menor Quota Superior)**.** Seja $E_0 = [a_1 \dots a_k]\{l_0, u_0\}$ e $E_1 = [b_1 \dots b_k]\{l_1, u_1\}$. Define-se $E_0 \vee E_1 = [a_1 \dots a_k\ b_1 \dots b_k]\{l, u\}$, onde $l = \min(l_0, l_1)$ e $u = \max(u_0, u_1)$.

Exemplo 3.4. A Figura 3 (a) mostra a definição de um conjunto de quatro expressões regulares. Cada expressão regular é formada por um conjunto de caracteres, mais um intervalo de tamanhos possíveis que essas expressões podem ter. Por exemplo, a primeira expressão, *alpha*, denota qualquer sequência de uma a oito letras latinas minúsculas. Por simplicidade, a Figura 3 omite os subreticulados formados por subconjuntos de *alpha* e *Alpha* com diferentes tamanhos de intervalos. O subreticulado de *Digit* é mostrado inteiramente.

3.1.1 Gramáticas Heap-CNF.

Reticulados permitem que *tokens* de uma gramática trivial possam ser amalgamados em expressões regulares comuns. Esse processo será explicado via uma série de exemplos. O primeiro exemplo mostra o que é uma gramática trivial, como ela surge, e porque ela pode crescer tanto.

Exemplo 3.5. A Figura 4 (b) mostra a gramática trivial produzida para reconhecer as duas sentenças vistas na Figura 4 (a). Note que um terminal é criado para cada *token*. Em outras palavras, não é feita qualquer tentativa de juntar *tokens* em expressões regulares. A Figura 4 (c) mostra a gramática correspondente que seria produzida pela ferramenta `Zhefuscator` [11]. Essa ferramenta reconhece alguns tipos primitivos, como inteiros e números de ponto flutuante. Nesse caso, os dois primeiros tokens, a saber, `7282` e `7283` poderiam ser agrupados como instâncias do tipo inteiro.

(a) Primeiro registro de *log*:

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

Segundo registro de *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}
```

Sendo **xx** igual a <digit∨alpha∨Alpha>{2,3}

Figura 4. (a) Duas sentenças emitidas pelo gerador de *logs*. (b) Gramática trivial produzida para reconhecer as duas sentenças. (c) Gramática produzida por `Zhefuscator`. (d) Gramática produzida por `Lushu`.

As gramáticas mencionadas no Exemplo 3.5 seguem um formato conhecido como *Heap-CNF* (*Heap-Chomsky Normal Form*) [11]. `Lushu` reconhece gramáticas neste formato. Gramáticas em formato Heap-CNF são gramáticas triviais que reconhecem um conjunto finito de sentenças L (e possivelmente mais sentenças). *Tokens* na posição $k, k \geq 1$, de uma sentença de L são reconhecidos a partir do não terminal R_{2k}. Para manter esse artigo auto-contido, a Definição 3.6 revisita esse conceito.

Definição 3.6 (Heap-CNF (Saffran et al.))**.** Uma gramática Heap-CNF de altura n possui os seguintes símbolos não-terminais: $R_0, R_1, \dots, R_{2n-2}, R_{2n-1}$, e regras de produção que seguem um dos seguintes três padrões, onde o símbolo a_x representa um terminal qualquer:

1. $R_{2k} ::= \mathsf{a}_1 \mid \mathsf{a}_2 \mid \dots \mid \mathsf{a}_p$ se $k < n$
2. $R_{2k-1} ::= R_{2k}\ R_{2k+1} \mid \epsilon$
3. $R_{2n-1} ::= \mathsf{a}_1 \mid \mathsf{a}_2 \mid \dots \mid \mathsf{a}_p$

3.1.2 Junção de Tokens via Reticulado.

Na discussão que se segue, seja S o conjunto definido pelo usuário de `Lushu`. Na Figura 3 (a), esse conjunto é formado pelas quatro expressões regulares mencionadas no Exemplo 3.2. Uma vez definido o conjunto S, `Lushu` funciona da seguinte forma: a ferramenta mantém uma gramática G, no formato Heap-CNF, capaz de reconhecer todas as sentenças emitidas pelo gerador de *logs* até o momento corrente. Uma vez recebida uma nova sentença V, as seguintes ações acontecem:

1. A sentença V é particionada em uma lista T de expressões regulares existentes em S.
2. T é amalgamado em G via o reticulado definido por S mais a operação de união de expressões regulares.

O restante desta seção descreve essas duas ações.

Particionamento de Sentenças. A divisão de uma sentença V em *tokens* utiliza o reticulado que surge a partir

do conjunto de expressões regulares S. Um *token* é definido como a maior sequência de caracteres que forma uma expressão regular que não seja o topo do reticulado. Em outras palavras, se v é uma sequência contígua de caracteres de V que forma um *token*, então v tem as seguintes propriedades:

1. para cada caracter $c \in v$, seja $s \in S$ a expressão que o contém. A expressão que reconhece v é a menor quota superior da junção de todas as expressões s.
2. define-se o intervalo $e\{l, u\}$ como $\{|s|, |s|\}$, se $l \leq |s| \leq u$. Doutro modo, usa-se e+.
3. a adição de mais um caracter a v torna s o topo do reticulado.

O Exemplo 3.7 ilustra esse particionamento.

Exemplo 3.7. A Figura 5 (a) mostra o particionamento em *tokens* da primeira sentença emitida pelo gerador de *logs* na Figura 4. Cada *token* presente na sentença alvo pode ser representado via um dos conjuntos base visto na Figura 3 (a); exceto o *token* `May`, pois ele envolve letras minúsculas e maiúsculas. Nesse caso, o token é reconhecido pelo menor limite superior dos conjuntos *alpha* e *Alpha*.

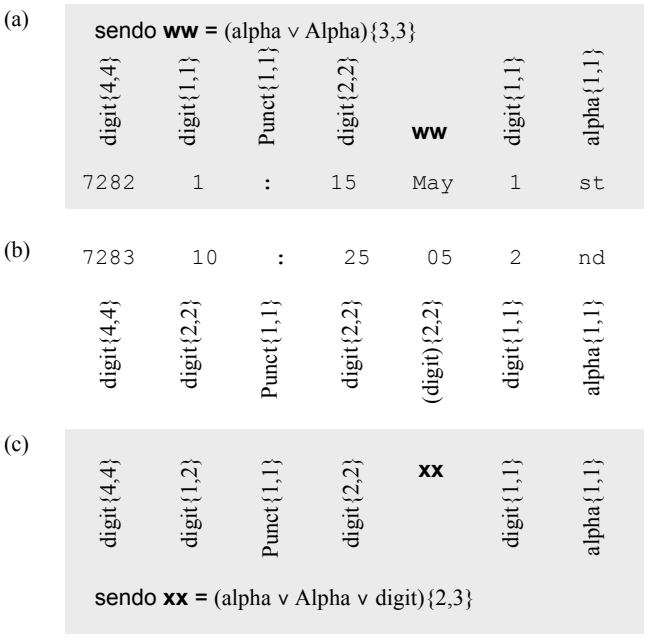

Figura 5. (a-b) Particionamento de sentenças em *tokens* usando o reticulado visto na Figura 3 (b). (c) Junção das sequências de *tokens*, via aplicação do operador de menor quota inferior.

Amalgamento de Sentenças. Após uma sentença ser recebida e particionada em *tokens*, ela precisa ser incorporada à gramática corrente. Em linhas gerais, essa operação é equivalente a aplicar uma redução sobre todas as sentenças emitidas pelo gerador de *logs* até um dado momento. A operação de redução é a menor quota inferior aplicada *token* a *token*. O próximo exemplo ilustra tal operação.

Exemplo 3.8. A Figura 5 (c) mostra as expressões regulares que resultam da junção das expressões regulares nas Figuras 5 (a) e (c). A Figura 6 mostra a gramática que reconhece a concatenação das expressões regulares na Figura 5 (c).

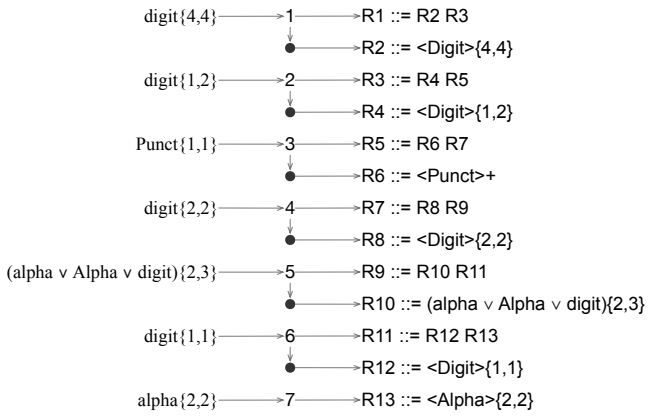

Figura 6. Construção de uma gramática Heap-CNF que reconhece a concatenação de uma série de expressões regulares.

3.1.3 A Operação de Alargamento.

É desejável que algumas operações de junção levem diretamente ao topo do reticulado, em vez de seguir a Definição 3.3. Por exemplo, usuários podem querer que a sequência `1:15` seja reconhecida como a concatenação de três expressões regulares, a saber: $digit\{1,1\}Punct\{1,1\}digit\{2,2\}$, em vez de ser reconhecida como a expressão regular $(digit \vee Punct)\{4,4\}$. Nesse caso, a junção das expressões *digit* e *Punct* deve ser proibida. `Lushu` permite que usuários especifiquem junções proibidas via uma operação de *alargamento*. O bloco **Widen** de uma especificação indica quais junções levam diretamente ao topo do reticulado. O Exemplo 3.9 ilustra a semântica deste bloco.

Exemplo 3.9. A Figura 3 (a) contem um bloco **Widen**. Esse bloco indica que a menor quota superior da junção de *Punct* e qualquer outra expressão regular é o topo do reticulado. Assim, elementos como $[a-z.;/:]+$ (que seria $alpha \vee Punct$) não existem no reticulado visto na Figura 3 (b).

3.2 Especificação de Sigilo

O usuário de `Lushu` determina *tokens* que devem ser ofuscados a partir de exemplos de *logs*. De posse de um exemplo, o usuário utiliza pares de elementos XML do tipo `<s>` e `<\s>` para delimitar *tokens* sigilosos. Qualquer sub-sentença entre `<s>` e `<\s>` é considerada sensível, desde que apareça na posição em que se encontra o texto marcado. Esse passo se dá antes da ativação de `Lushu` sobre a máquina virtual Java. Uma vez ativado, quaisquer ocorrências de *tokens* com o formato especificado são cifradas. Caso *tokens* desse tipo

sejam amalgamados em expressões mais complexas, qualquer *string* reconhecida por essa nova super-expressão será também cifrada. O Exemplo 3.10 ilustra a marcação para que Lushu consiga obfuscar o *log* da Figura 1.

Exemplo 3.10. A Figura 7 mostra um texto com marcação de sigilo. Tal marcação determina que expressões regulares no formato de CPFs e endereços IPs devem ser ofuscadas, mas somente quanto elas ocorrem como o terceiro e o quinto *tokens* de sentenças. Assim, caso um CPF ocorra como o segundo *token* de uma sentença (o que de fato ocorre em sentenças do tipo DEBUG), então aquele CPF não será cifrado.

Texto com marcações de sigilo feitas pelo usuário:

```
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>
```

Texto original emitido pela 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
```

Texto emitido pela JVM após ofuscação via Lushu:

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

Figura 7. Marcação de informação sigilosa e ofuscamento de registros de *log*.

4 Avaliação Experimental

O objetivo desta seção é discutir as quatro questões de pesquisa abaixo, em que o "sistema decorado"é a instância da JVM cuja saída é interceptada por Lushu:

QP1: Qual é o impacto de Lushu no tempo de execução do sistema decorado?

QP2: Qual é o impacto de Lushu no consumo de memória do sistema decorado?

QP3: Qual é a efetividade de Lushu para comprimir gramáticas, quando comparado com a ferramenta precursora, o Zhefuscator?

QP4: Qual é a taxa de convergência para a construção de uma gramática para um *log* real de banco de dados?

Software: Versão da JVM: 17.0.6. Versão de Kotlin: 1.7.10. Sistema operacional: Ubuntu 20.04.6 LTS. Versão do PostgreSQL: 14.7.

Hardware: CPU: Intel i7-1065G7 @ 3.90GHz, 4 núcleos. Memória primária: Samsung, 16GB @ 3200 MT/s.

Benchmarks: Esta seção usa dois benchmarks: um gerador de *logs aleatórios*, e um arquivo de *log* real produzido por PostgreSQL. O gerador de *logs* aleatórios é capaz de produzir 1.858.950 diferentes combinações de palavras. Cada *log* contém um número de CPF, uma data, um carimbo de tempo, um inteiro aleatório, ou uma combinação desses termos.

Exemplo 4.1. Exemplos de *logs* emitidos pelo gerador de *logs* artificiais:

```
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.
```

Os arquivos de *logs* provenientes de PostgreSQL nos permitem avaliar Lushu em um contexto mais real. Os *logs* foram gerados via a ferramenta pgbench, que emula a atividade de um banco de dados real[2]. Embora os registros que compõem esses *logs* possam ter uma quantidade arbitrária de *tokens*, a diversidade deles é menor que aquela vista nos *logs* aleatórios. O Exemplo 4.2 mostra alguns registros.

Exemplo 4.2. Exemplos do conteúdo de *actual_log* no formato dos *logs* de PostgreSQL:

```
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: Tempo de Execução

Para medir o impacto de Lushu sobre as aplicações decoradas, nós medimos seu desempenho sobre uma instância da JVM que simplesmente imprime registros de *logs*, sem qualquer outro processamento. A quantidade de registros emitidos é parametrizável. Neste experimento, a quantidade de registros produzidos por cada gerador de *logs* varia entre 10^0 e 10^6 registros. Não ocorre qualquer processamento entre a emissão de um registro e o próximo.

Análise: A Figura 8 compara o tempo de execução das diferentes aplicações, com e sem a interceptação realizada por Lushu. Os tempos de execução apresentados são a média aritmética dos tempos medidos em 10 execuções independentes. O tempo de execução tende a crescer linearmente com o número de *logs*. As aplicações cujos *logs* são interceptados por Lushu são mais lentas. Contudo, este impacto só é perceptível se a aplicação emite muitos *logs* em sequência—um cenário improvável para a maioria das aplicações reais. A crescente distância entre as curvas é provavelmente devida a efeitos de *buffering*, e não parece ser relacionada ao custo computacional de Lushu.

4.2 QP2: Consumo de Memória

Este experimento é similar àquele descrito na Seção 4.1, exceto que medimos o consumo de memória primária ao invés

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

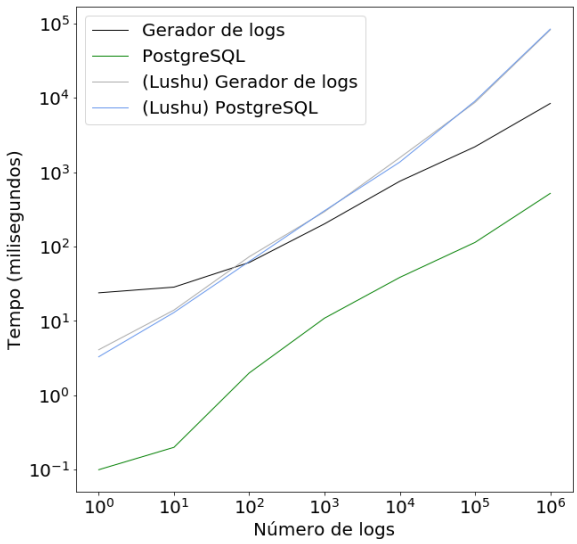

Figura 8. Impacto de Lushu no tempo de execução.

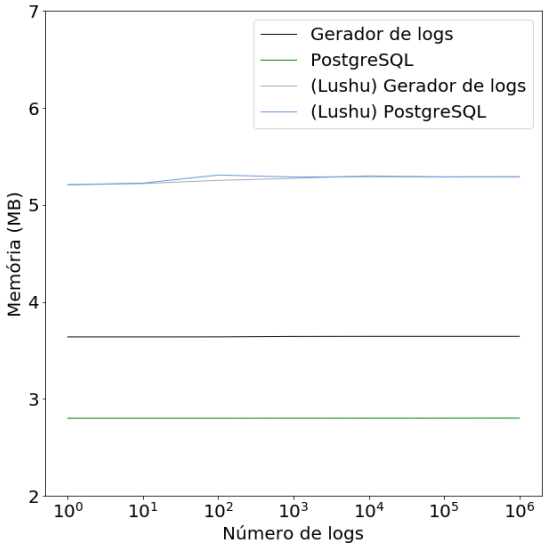

Figura 9. Impacto de Lushu no consumo de memória.

do tempo de execução. A memória reportada nesta seção é a diferença entre a memória reservada para o processo Java (`Runtime.totalMemory()`) e a memória reservada para alocação de objetos (`Runtime.freeMemory()`). Os números obtidos são a média de dez amostras. Antes de cada amostra, o coletor de lixo da JVM é invocado.

Análise: A Figura 9 mostra o impacto de Lushu no consumo de memória. O impacto de Lushu é notável; contudo, tal impacto é constante nos dois cenários amostrados. O consumo de memória cresce somente 2% entre a emissão de um registro até um milhão de *logs*. Esse consumo se torna constante a partir de mil registros. O crescimento inicial de 2% é devido ao crescimento da gramática utilizada por Lushu.

4.3 QP3: Taxa de Compressão de Gramáticas

Comparamos a capacidade de comprimir gramáticas de Lushu com seu predecessor, `Zhefuscator`. Para simular o comportamento do `Zhefuscator`, utilizamos um reticulado que distingue sequências de dígitos dos demais *tokens*. Assim, caracteres do alfabeto, exceto números, não são amalgamados em expressões regulares. Neste experimento são interceptados entre 1 a 2001 registros emitidos pelo gerador de *logs* aleatório. Para cada número de registros é tomada a média de tamanho de gramáticas para dez execuções do gerador aleatório. O tamanho da gramática é medido como o número de símbolos não terminais que ela contém.

Análise: A Figura 10 mostra o tamanho da gramática (em número de terminais) gerada por `Zhefuscator` e Lushu. A gramática de Lushu converge com menos de 100 *logs*, enquanto a gramática do `Zhefuscator` não converge após observar 2001 registros. O tamanho da gramática do `Zhefuscator` depois de 2001 *logs* é mais de 100 vezes maior que a de *Lushu*. A convergência do `Zhefuscator` requer que todos os símbolos terminais sejam observados—cenário que não ocorre neste experimento.

4.4 QP4: Taxa de Convergência com *Log* Real

O projeto de `Zhefuscator` se baseava na premissa que *logs* reais tendem a ser muito regulares. Em um tal cenário, também a convergência da gramática produzida por Lushu é muito mais rápida. Consideremos, por exemplo, a taxa de convergência em *logs* produzidos pelo pgbench de PostgreSQL. Após consumir 32 linhas, Lushu produz uma gramática com 69 símbolos não-terminais e 118 símbolos terminais. Após 155 linhas, o número de terminais aumenta somente uma vez. Tendo alcançado 119 símbolos terminais, o tamanho da gramática permanece constante, mesmo após o consumo de mais de um milhão de registros.

5 Trabalhos Relacionados

Este artigo traz uma forma de *síntese indutiva de linguagens*: esse problema busca construir um autômato que reconheça uma linguagem a partir de exemplos positivos e negativos de sentenças. Os primeiros pertencem à linguagem; os segundos não pertencem. A síntese indutiva de linguagens tem suas origens no trabalho de Gold [6]. A versão do problema estudada por Gold busca reconhecer a linguagem "no limite", isso é, a partir de uma sequência potencialmente infinita de

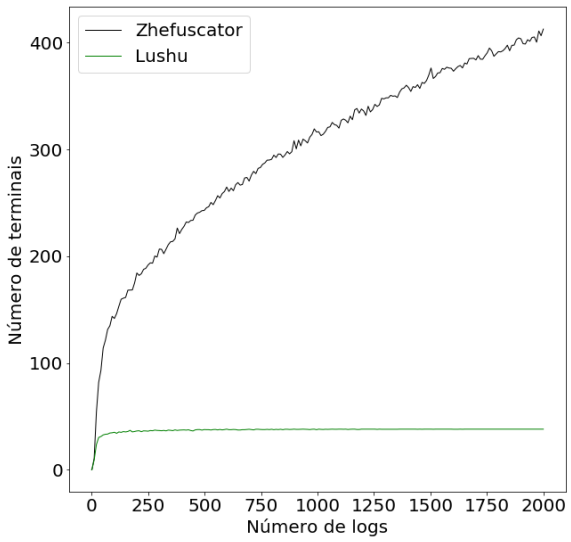

Figura 10. Comparação entre o tamanho da gramática gerada por `Lushu` e `Zhefuscator`.

exemplos positivos. Naquela versão, um "reconhecedor" é construído incrementalmente, a partir dos exemplos vistos. Nosso trabalho cabe neste arcabouço teórico.

A síntese indutiva de linguagens pode incorporar exemplos negativos, isso é, sentenças que não pertencem à linguagem. Muito da literatura relacionada baseia-se no trabalho de Angluin [1]. Angluin apresenta um arcabouço formado por um "aprendedor"(como no trabalho de Gold [6]) e por um "professor": a fonte de informação. O "aprendedor"deve identificar uma linguagem regular que o professor conhece. O processo de aprendizagem segue uma estratégia de tentativa-e-erro. O "aprendedor"apresenta uma autômato finito determinístico (AFD) ao professor. Se sua conjectura estiver correta, o professor responde afirmativamente. De outro modo, o professor gera um contra-exemplo—uma cadeia de caracteres que o autômato não reconhece, ou reconhece mas não está na linguagem objetivo. O trabalho de Angluin (e os muitos trabalhos que ela inspirou) depende da existência do professor. O presente trabalho difere do arcabouço de Angluin na medida em que ele não prevê a figura do professor e não utiliza exemplos negativos.

As técnicas apresentadas neste artigo se assemelham ao trabalho de Shcherbakov [12]. Semelhanças incluem a aprendizagem incremental e a existência de somente exemplos positivos. O algoritmo proposto por Shcherbakov é baseado em alinhamento de sequências: o mesmo princípio usado, por exemplo, em sequenciamento genético. O algoritmo proposto no presente trabalho é baseado em um reticulado, o qual é extraído de uma especificação construída pelo usuário. Nesse sentido, a técnica de Shcherbakov é mais automática: enquanto nosso trabalho pressupõe o reticulado e os exemplos, a técnica de Shcherbakov precisa somente daqueles últimos. Porém, o algoritmo de Shcherbakov é quadrático no tamanho dos exemplos: caso o maior exemplo tenha $O(N)$ caracteres, e existem $O(X)$ exemplos, o algoritmo tem complexidade $O(X \times N^2)$. O algoritmo apresentado neste artigo tem complexidade mais baixa: $O(X \times N)$.

6 Conclusão

Este artigo apresentou um sistema para compressão de gramáticas que reconhecem sequências finitas de exemplos de sentenças. A técnica proposta gira em torno da ideia de juntar em expressões regulares os *tokens* que ocorrem nas mesmas posições das diferentes sentenças. Esse amalgamento de *tokens* é guiado por um reticulado. Tal reticulado emerge de uma especificação de conjuntos de caracteres que o usuário produz. As ideias propostas foram incorporadas em um sistema de ciframento de informações sigilosas, conhecido como `Lushu`. Tal sistema é software livre, disponível via licença GPL 3.0. Entretanto, embora já possa ser utilizado em situações práticas, ainda há várias direções ao longo das quais `Lushu` poderia sofrer melhorias. Em particular, as ideias propostas comportam somente a compressão de *tokens*, não de regras de produção. Espera-se que tal limitação possa ser contornada em desenvolvimentos futuros deste trabalho.

Referências

[1] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and computation* 75, 2 (1987), 87–106.
[2] CCPA. 2020. https://oag.ca.gov/privacy/ccpa
[3] Datadog. 2023. https://www.datadoghq.com/
[4] Fluentd. 2023. https://www.fluentd.org/
[5] GDPR. 2018. https://gdpr-info.eu/
[6] E Mark Gold. 1967. Language identification in the limit. *Information and control* 10, 5 (1967), 447–474.
[7] 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.
[8] Lei Geral de Proteção de Dados. 2018. https://www.gov.br/cidadania/pt-br/acesso-a-informacao/lgpd
[9] Logstash. 2023. https://www.elastic.co/logstash/
[10] 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
[11] 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
[12] Andrei Shcherbakov. 2016. A Branching Alignment-Based Synthesis of Regular Expressions. In *AIST (Supplement)*. Springer, Berlin, Germany, 315–328.
[13] Splunk. 2023. https://www.splunk.com/
[14] Sumo Logic. 2023. https://www.sumologic.com/