Sintetizador de Gramáticas para Obfuscação de Dados em Sistemas de Logs

João Saffran, Fernando Magno Quintão Pereira, Haniel Barbosa

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Av. Antônio Carlos, 6627 – 31.270-010 – Belo Horizonte – MG – Brazil

{joaosaffran,fernando,hbarbosa}@dcc.ufmg.br

Resumo. *Um evento de strings é a ocorrência de um padrão de texto produzido na saída de um programa. A captura e tratamento destes eventos de strings podem ser usados em diversas aplicações, como por exemplo, anonimização de logs, tratamento de erros e notificação dos usuários de um programa. Porém, atualmente não existe uma maneira sistemática para identificar e tratar eventos de string, cada sistema lida com este problema de maneira ad-hoc. Este trabalho formaliza o conceito de eventos de string e propõem um framework baseado em síntese de gramáticas para identificar e tratar esses eventos. Este framework é composto por: i) uma interface baseada em exemplos para especificar padrões de texto; e ii) um algoritmo de síntese de gramáticas com parsers eficientes para reconhecer estes padrões. Nós demonstramos a eficiência desta abordagem implementando o Zhefuscator, uma extensão da Java Virtual Machine (JVM). Esta ferramenta detecta padrões em texto conforme são produzidos, e os obfusca, a fim de proteger a privacidade dos dados de usuários do sistema.*

Link to Code: https://github.com/lac-dcc/Zhe

1. Introdução

O avanço da Lei Geral de Proteção de Dados, em vários países, inclusive no Brasil, tem aumentado a importância de tratar eventos de string produzidos por *software*, ainda que tais programas sejam observados como caixa-pretas. Mas, esta tarefa continua sendo um desafio, uma vez que um programa pode produzir um número infinito de strings. A detecção destes eventos requer que seja sintetizada uma gramática com potencialmente infinitos exemplos.

Eventos de strings podem ser definidos como a ocorrência de algum padrão de interesse em um texto produzido por um programa. Estes eventos podem ser produzidos de maneira automática, como por exemplos em logs de um programa ou devido à interação entre programas e seus usuários. Exemplos de padrões de interesse incluem a produção de informação sensível dos usuários que precisam ser anonimizadas. Como não existe uma maneira unificada para capturar e tratar este tipo de evento, cada aplicação lida com este problema de maneiras específicas. Contudo, as ferramentas necessárias para produzir tal arcabouço já existem de maneira consolidada, a saber: i) síntese de gramáticas [Nakamura and Matsumoto 2002]; ii) interceptação de funções [Lämmel and Stenzel 2004]. Esse artigo usa estes conceitos para propor uma abordagem utilizada em tal *framework* capaz de lidar com eventos de *strings*. Esta abordagem é implementada em uma ferramenta capaz de anomimizar logs produzidos por aplicações que executam na máquina virtual java (JVM).

Neste trabalho, implementamos a técnica proposta em uma ferramenta, chamada Zhefuscator, capaz de esconder dados sensíveis em *queries* SQL encontradas em logs produzido por aplicações Java. Zhefuscator implementa uma forma de programação reativa [Ramson and Hirschfeld 2017]. Zhefuscator lida com dois problemas: i) uma forma de detectar eventos; ii) uma forma de reagir a estes eventos. Ambos estes tópicos serão explorados na seção 2. Na seção 3 será discutido o funcionamento do Zhefuscator e sua implementação. Os detalhes da apresentação da ferramenta serão discutidos na seção 4. O Zhefuscator é disponível como um projeto de código aberto, distribuido com a licença GPLv3 e atualmente ele é utilizado por pelo meno uma empresa de proteção de dados.

2. Visão Geral

Na seção 2.1 apresenta motivações para o tratamento automático de eventos de *string* e na seção 2.2 são apresentados os desafios para realizar esta tarefa.

2.1. Eventos de *String* no contexto da Lei Geral de Proteção de Dados (LGPT)

Neste trabalhos nos chamamos um *gerador* um programa que produz *string* t_i em cada momento i no tempo. Software que produz logs, como servidores de bancos de dados e sistemas operacionais, podem ser entendidos como geradores. Geralmente, quando a saída de um programa é analisado, esta análise ocorre de maneira *off-line*, por exemplo, após os logs serem produzidos e armazenados. Porém, existem situações que esta análise deve ocorrer *on-line*, enquanto a saída está sendo produzida.

Leis de proteção de dados são um dos fatores que requerem esta análise *on-line*. Por exemplo a *General Data Protection Regulation* (GDPR)[1], válida na *European Economic Area* desde 2016, requer que as empresas anonimizem dados pessoais quando estes dados potencialmente podem ser usados de maneiras não especificada nos termos da empresa[Voigt and Bussche 2017]. A criação da GPDR motivou a criação de leis similares na Califórnia e no Brasil.

Leis de proteção de dados têm impacto na geração dos dados, já que logs não devem vazar informações pessoais dos usuários do sistema. Porém, muitos programas foram concebidos e implementados antes do surgimento destas leis. Ajustar tais sistemas a esses requisitos de privacidade custa caro para empresas, uma vez que isso requer modificações em código legado. Entretanto, neste artigo, nós demonstramos que é possível filtrar logs enquanto estes estão sendo produzidos, projetando este problema no *framework* de tratamento de eventos de *strings*. A ocorrência de dados sensíveis é um evento de *strings*. Dado o *framework* correto, estes eventos podem ser detectados e tratados *on-line*. Mesmo assim, a criação e a instalação de tal arcabouço envolve problemas práticos e teóricos, que serão discutidos na próxima seção.

2.2. Reconhecimento de Eventos de *String*: Desafios

Lidar com eventos de *string* enquanto tratando o evento gerador como uma caixa preta é desafiador por três razões, as quais serão discutidas nesta seção. Para tornar esses problemas mais concretos, nós os relacionamos com o seguinte exemplo, que é resolvido pelo Zhefuscator:

Example 1 (Problema Concreto) *Considere um servidor de banco de dados produzindo logs executando na JVM. A gramática que descreve a sintaxe do log é desconhecida. O*

[1] `https://eugdpr.org/`

```
     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'
```

Figura 1. Trecho do log com 5 exemplos.

Log pode conter queries SQL. Algumas queries contêm informação sensível. Projete um sistema que intercepte strings do log, antes delas delas serem copiadas para a saída padrão do programa, e anonimize literais específicos contidos nas queries SQL. Um literal é uma constante em uma query SQL. Exemplos incluem inteiros e strings.

Challenge 1 (Síntese de Gramáticas) *Como identificar eficientemente queries SQL dentro do log, quando não sabemos a gramática do log?*

Cada programa tem seu próprio formato de log. Partes deste log usam a sintaxe SQL. Se chamarmos de L a linguagem que define as *strings* do log, então cada *string* $t \in L$ pode conter *substrings* que são SQL e *substrings* que não são SQL. Nesta combinação de duas linguagens nos chamamos L de *linguagem hospedeira* e SQL de *linguagem alvo*.

Example 2 *A figura 1 mostra parte do log gerado por um programa gerador (alguns literais foram trocados por dados falsos). Strings na linguagem alvo, SQL, são mostrados em vermelho. Esse log contém 5 exemplos, 1 por linha. Cada exemplo foi produzido pelo gerador em momentos sucessivos no tempo. Uma solução para o desafio 1 é sintetizar um parser para este log.*

Requerer um *parser* para a linguagem hospedeira L seria uma complicação para a instalação do obfuscador, pois esse requisito força o usuário a ter conhecimento do formato de L. É possível separar as linguagem alvo e hospedeira utilizando uma abordagem de força bruta. Essa abordagem considera todo *token* pertencente à linguagem hospedeira como um potencial *token* para início da de uma sentença da linguagem de eventos. Entretanto, tal abordagem não escala com o números de *tokens* nas *strings* $t \in L$. O Gerador produz um fluxo infinito de *strings*; portanto, o Desafio 1 envolve inferir uma gramática *no limite*, isso é, a partir de um número infinito de exemplos. Apesar deste problema ser indecidível, até para linguagens regulares ou *superfinitas*, como mostrado por Edward Gold [Gold 1967], nós podemos construir gramáticas não ambíguas que reconhecem, de maneira escalável, um subconjunto da linguagem hospedeira definida por todos os exemplos vistos até aquele ponto.

Challenge 2 (Interface) *Qual interface deve ser utilizada pelo usuário para definir padrões de texto que representem dados sensíveis?*

Obfuscar o log na Figura 1 requer que saibamos quais literais da linguagem SQL devem ser obfuscados. Cabe ao usuário do obfuscador definir quais padrões anonimizar. Porém, uma determinada informação pode ser sensível quando usada em certas *queries*, mas não em outras, como o Exemplo 3 ilustra.

Example 3 *Considere uma instância do problema concreto (Ex. 1) que requer obfuscar as ocorrências de* `CPF` *no padrão:* `"SELECT * FROM Clts WHERE CPF='?'"`. *Ocorrência de* `CPF` *em outros padrões, como* `DELETE FROM Clts WHERE CPF='000.000.000-00'`, *devem ser preservadas.*

Challenge 3 (Engenharia) *Como interceptar a saída do gerador, sem mudar sua implementação?*

O Desafio 3 é especifico para cada gerador. Na Seção 3.3 nós descrevemos uma solução para sistemas executando na JVM. Em contraste com as soluções propostas para os outros desafios, esta não é geral. Em outras palavras, soluções para o Desafio 3 são específicas para cada tecnologia.

3. Arquitetura do Zhefuscator

Nesta seção iremos discutir os módulos necessários para a implementação do Zhefuscator. Seção 3.1, discute o procedimento de marcação de exemplos. Seção 3.3 apresenta o algoritmo de síntese de gramática. Seção 3.3, detalha as especificações do Zhefuscator.

3.1. Procedimento de Marcação

O Zhefuscator requer que o usuário forneça exemplos para que ele possa identificar o que é sensível na linguagem alvo. Para ajudar o usuário a determinar quais *substrings* anonimizar, o Zhefucator implementa o seguinte procedimento:

Procedure `markup`($G_e : \mathit{Event\ Grammar}, L_f : \mathit{Log\ Example}$)

1. Dado que G'_e é uma gramática vazia.
2. O usuário seleciona um literal $l \in L_f$ que deve ser obfucado.
3. Zhefuscator usa a gramática da linguagem de evento para extrair a *substring* mais longa $s \in G_e$ que contém l.
4. Uma gramática G''_e, formada com as regras de produção de G_e necessária para reconhecer s, é construída.
5. O terminal T que reconhece l é marcado para ser obfuscado.
6. G'_e é aumentada com as regras de G''_e.
7. Toda sentença $s' \in L_f$ que G'_e reconhece é marcada.
8. Se há mais literais em L_f que ainda precisam ser obfuscados, o usuário volta para o passo 2.

Figura 2. O Procedimento de marcação que ajuda o usuário a determinar quais literais devem ser obfuscado.

3.2. Síntese de Gramática

Pela definição de eventos de *strings*, a captura destes envolve detectarmos ocorrência de *substrings* produzidas pela gramática de eventos G_e contido na linguagem alvo L. Nós chamamos de G a gramática que reconhece L. No contexto de reconhecer eventos de *strings*, originados a partir da execução do gerador caixa preta, como explicado na Seção 2.2, não podemos assumir que a linguagem alvo é conhecida. Por causa disso, é necessário inferir a linguagem L conforme os eventos são capturados.

Nosso algoritmo que aproxima a gramática G que reconhece a linguagem hospedeira L esta descrito na Figura 3. O ponto inicial do algoritmo é a função `build_grammar` que recebe como entrada um texto (`text`), uma sequência infinita de *strings* $t_1, t_2, t_3, \ldots$, pertencentes a linguagem hospedeira a ser reconhecida. A função `build_grammar` opera utilizando um *loop* da baseado em *counterexample-guided inductive*

```
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)
```

Figura 3. Procedimento de separação de linguagem.

synthesis(CEGIS) [Solar-Lezama et al. 2005, Solar-Lezama et al. 2006] , na qual um inferidor produz gramáticas que reconhecem os exemplos vistos até um determinado momento e um verificador checa se a gramática proposta é capaz de verificar os exemplos subsequentes.

Para cada *string* `example` no texto de entrada `build_grammar` refina a gramática que reconhece `example`. Portanto, a variável `grammar` na linha 25 da Figura 3 se refere a gramática que reconhece `text` no limit, ou seja, após um infinito número de exemplos serem gerados. Apesar da função `build_grammar` nunca terminar, ainda assim, ela produz uma nova gramática sempre que é invocada recursivamente (linha 21). Função `build_grammar` usa uma rotina auxiliar chamada `add_example`. Este procedimento checa se a gramática corrente é capaz de reconhecer uma *string* em `text`(Linha 8). Se conseguir nada acontece (Linha 9). Caso contrário, `add_example` refina a gramática corrente.

O processo de refinamento implementado no procedimento `add_example` consiste em realizar a união de duas gramáticas. Esta operação é implementada no algoritmo da Figura 3, utilizando a união das regras de produção de cada uma das gramáticas. Contudo, isto só é possível se ambas as gramáticas seguirem um conjunto de regras de numeração específicas de suas regras de produção. Esta restrições nos levam a propor o *Heap-CNF* uma forma de gramática que segue a seguinte definição:

Definition 1 (Heap-CNF) *Uma gramática Heap-CNF possui restrição nos não terminais e nas regras de produção. Não terminais são* R_1, R_2, $R_3, \ldots,$ R_{2^n-2}, R_{2^n-1}, *para um* n *arbitrário. As regras de produção permitida são:*

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

tal que $\mathtt{a}$ *é um terminal e* $k \in \{1, \ldots, n\}$. *Já que os não terminais são numerados da mesma maneira que um estrutura de dados heap, nós chamamos esta versão restrita da forma normal de Chomsky, Heap-CNF.*

3.3. Implementação da JVM

Nesta seção descrevemos os detalhes de implementação do Zhefuscator, principalmente relacionados ao *parsing* da linguagem dos logs, a interceptação de eventos na JVM e a obfuscação de *strings*.

Parsing. Zhefuscator usa a teoria descrita na Seção para construir parsers incrementalmente. Estes *parsers* são construídos utilizando o *ANTLR*[Parr and Fisher 2011]. Esta ferramenta recebe como entrada a gramática gerada pela função `build_grammar`(Figura 3) e gera como saída o código fonte de um *parser* $LL(*)$ que é utilizado para separar a linguagem alvo e hospedeira.

Interceptação de Eventos da JVM Zhefuscator utiliza *Java Agents* para interceptar as chamadas dos métodos presentes no *Singleton System.out.**. A API de *Java Agents* [Binder et al. 2007] provê suporte para instrumentação dinâmica de aplicações que executam na JVM. As *strings* interceptadas são passadas para o procedimento `build_grammar` e por fim são anonimizadas, caso necessário. A primeira destas ações pode gerar uma expansão da gramática que descreve a linguagem hospedeira, já a segunda pode modificar a saída do programa. Os literais que devem ser obfucados foram definidos pelo usuário utilizando a técnica descrita na Seção 3.1

Obfuscação de *Strings*. O Zhefuscator realiza a obfuscação de informação sensíveis utilizando criptografia assimétrica. Um literal sensível, l, é substituído por uma nova *string* l_s, de forma que, posteriormente, o valor de l possa ser recuperado via uma chave de encriptação. Atualmente o Zhefuscator utiliza *Advanced Encryption Standard* (AES) para obfuscar valores sensíveis.

4. Salão de Ferramentas

Para demonstrar o funcionamento do Zhefuscator, nós iremos utilizá-lo para obfuscar logs produzidos por um servidor de banco de dados escrito em Java. Este servidor faz a intermediação com banco de dados MySQL e PostgreSQL.

Para realizar as demonstrações será necessário uma cabo VGA ou HDMI com conexão a um projetor. Iremos instalar o Zhefuscator e o banco de dados do autor que apresentar o trabalho. A audiência do salão de ferramentas poderá definir subconjuntos de SQL que devem ser anonimizados selecionando exemplos de trechos de logs.

5. Trabalhos Relacionados

Síntese Dedutiva de Gramáticas. A noção de *Indentificação de linguagens no limite*, que foi utilizado como motivação para o nosso algoritmo de inferência de gramáticas *on-line*, foi introduzida por Edward Gold nos anos 60 [Gold 1967]. Muita pesquisa evoluiu a partir da formulação inicial de Gold. O principal desenvolvimento foram graça a Angluin e seus colaboradores [Angluin 1979, Angluin 1980, Angluin 1982, Angluin and Smith 1983]. Ainda assim, vários grupos de pesquisa formalizaram inferência de gramáticas para tipos específicos de linguagens [García et al. 2008, Globig and Lange 1994, López et al. 2004, Peris and López 2010, Sakakibara 1995]. Desde os anos noventa, decidibilidade para inferência de gramáticas em diferentes tipos de linguagens já é conhecida para diferentes linguagens [Sakakibara 1997]. A identificação de eventos de *string* cabe dentro do *framework* de inferência de linguagens no limite; entretanto, neste artigo, não tentamos adivinhar a linguagem hospedeira correta L que contém estes eventos. Ao invés disso, tentamos inferir uma gramática G que reconhece os eventos de *string* em qualquer prefixo desta linguagem. Nota-se que G pode reconhecer *strings* que não pertencem a L. Essa possibilidade não tem implicações

práticas no contexto deste trabalho: nós estamos interessados em encontrar eventos de *strings*, não reconhecer a linguagem hospedeira que reconhece este evento.

Eventos de *Strings*. Este trabalho não é o primeiro a lidar com detecção *on-line* de eventos de *strings*. Este tipo de abordagem já foi utilizado, por exemplo, no contexto de detecção de invasores [Abbes et al. 2004, Roesch 1999], *dynamic taint analysis* [Chen and Wagner 2007, Chang et al. 2008] detecção de spam *on-the-fly* [Xie et al. 2008]. Contudo, estes trabalhos anteriores identificaram eventos de *strings* em situações bastante específicas, por exemplo, padrões particulares em uma *query* SQL no cenário de *tainted flow analysis* [Chen and Wagner 2007], ou como uma combinação específica de tokens em pacotes de rede, no caso de detecção de intrusos [Abbes et al. 2004]. Este trabalho é o primeiro a propor um *framework* geral que "aprende" uma linguagem e reconhece eventos de *strings* dentro dela.

6. Conclusão

Este artigo apresentou uma arcabouço para detectar eventos de *string*. Um evento de string consiste na ocorrência de uma determinada sequência de caracteres em um texto. Estes eventos são descritos por uma linguagem na qual a gramática é conhecida. Eles ocorrem em texto potencialmente infinito, definido por uma linguagem hospedeira, cuja gramática é desconhecida. Nós mostramos como sintetizar uma gramática G que reconhece qualquer prefixo de um texto infinito. Essas gramáticas seguem um formato que chamamos *Heap-CNF*, uma restrição da forma Normal de Chomsky. Gramáticas neste formato não são ambíguas, e admitem *parsers* do tipo LL(1)—logo, podem ser analisadas de maneira eficiente. Demonstramos impiricamente que essa teoria pode ser implementada em um anonimizador eficiente de logs, o Zhefuscator. Essa ferramenta anonimiza dados sensíveis na saída de programas, enquanto tratas estes programas como caixa preta.

Agradecimentos. Este trabalho foi possível graças a uma doação feita pela empresa *Cyral Inc.* ao Departamento de Ciência da Computação da UFMG.

Referências

Abbes, T., Bouhoula, A., and Rusinowitch, M. (2004). On the fly pattern matching for intrusion detection with snort. *Annales des Télécommunications*, 59(9-10):1045–1071.

Angluin, D. (1979). Finding patterns common to a set of strings (extended abstract). In *STOC*, pages 130–141, New York, NY, USA. ACM.

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

Angluin, D. (1982). Inference of reversible languages. *J. ACM*, 29(3):741–765.

Angluin, D. and Smith, C. H. (1983). Inductive inference: Theory and methods. *ACM Comput. Surv.*, 15(3):237–269.

Binder, W., Hulaas, J., and Moret, P. (2007). Advanced java bytecode instrumentation. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 135–144. ACM.

Chang, W., Streiff, B., and Lin, C. (2008). Efficient and extensible security enforcement using dynamic data flow analysis. In *CCS*, pages 39–50, New York, NY, USA. ACM.

Chen, K. and Wagner, D. (2007). Large-scale analysis of format string vulnerabilities in debian linux. In *PLAS*, pages 75–84, New York, NY, USA. ACM.

García, P., Vázquez de Parga, M., and López, D. (2008). On the efficient construction of quasi-reversible automata for reversible languages. *Inf. Process. Lett.*, 107(1):13–17.

Globig, C. and Lange, S. (1994). On case-based representability and learnability of languages. In *AII*, pages 106–120, London, UK, UK. Springer-Verlag.

Gold, E. M. (1967). Language identification in the limit. *Information and Control*, 10(5):447–474.

Lämmel, R. and Stenzel, C. (2004). Semantics-directed implementation of method-call interception. *IEE Proceedings - Software*, 151(2):109–128.

López, D., Sempere, J. M., and García, P. (2004). Inference of reversible tree languages. *Trans. Systems, Man, and Cybernetics, Part B*, 34(4):1658–1665.

Nakamura, K. and Matsumoto, M. (2002). Incremental learning of context free grammars. In *ICGI*, pages 174–184, London, UK. Springer-Verlag.

Parr, T. and Fisher, K. (2011). LL (*): the foundation of the antlr parser generator. *Sigplan Notices*, 46(6):425–436.

Peris, P. and López, D. (2010). Transducer inference by assembling specific languages. In *ICGI*, pages 178–188, Berlin, Heidelberg. Springer.

Ramson, S. and Hirschfeld, R. (2017). Active expressions: Basic building blocks for reactive programming. *The Art, Science, and Engineering of Programming*, 1(2).

Roesch, M. (1999). Snort - lightweight intrusion detection for networks. In *LISA*, pages 229–238, Berkeley, CA, USA. USENIX Association.

Sakakibara, Y. (1995). Grammatical inference: An old and new paradigm. In *ALT*, pages 1–24, Berlin, Heidelberg. Springer-Verlag.

Sakakibara, Y. (1997). Recent advances of grammatical inference. *Theor. Comput. Sci.*, 185(1):15–45.

Solar-Lezama, A., Rabbah, R. M., Bodík, R., and Ebcioglu, K. (2005). Programming by sketching for bit-streaming programs. In Sarkar, V. and Hall, M. W., editors, *Conference on Programming Language Design and Implementation (PLDI)*, pages 281–294. ACM.

Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S. A., and Saraswat, V. A. (2006). Combinatorial sketching for finite programs. In Shen, J. P. and Martonosi, M., editors, *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415. ACM.

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

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