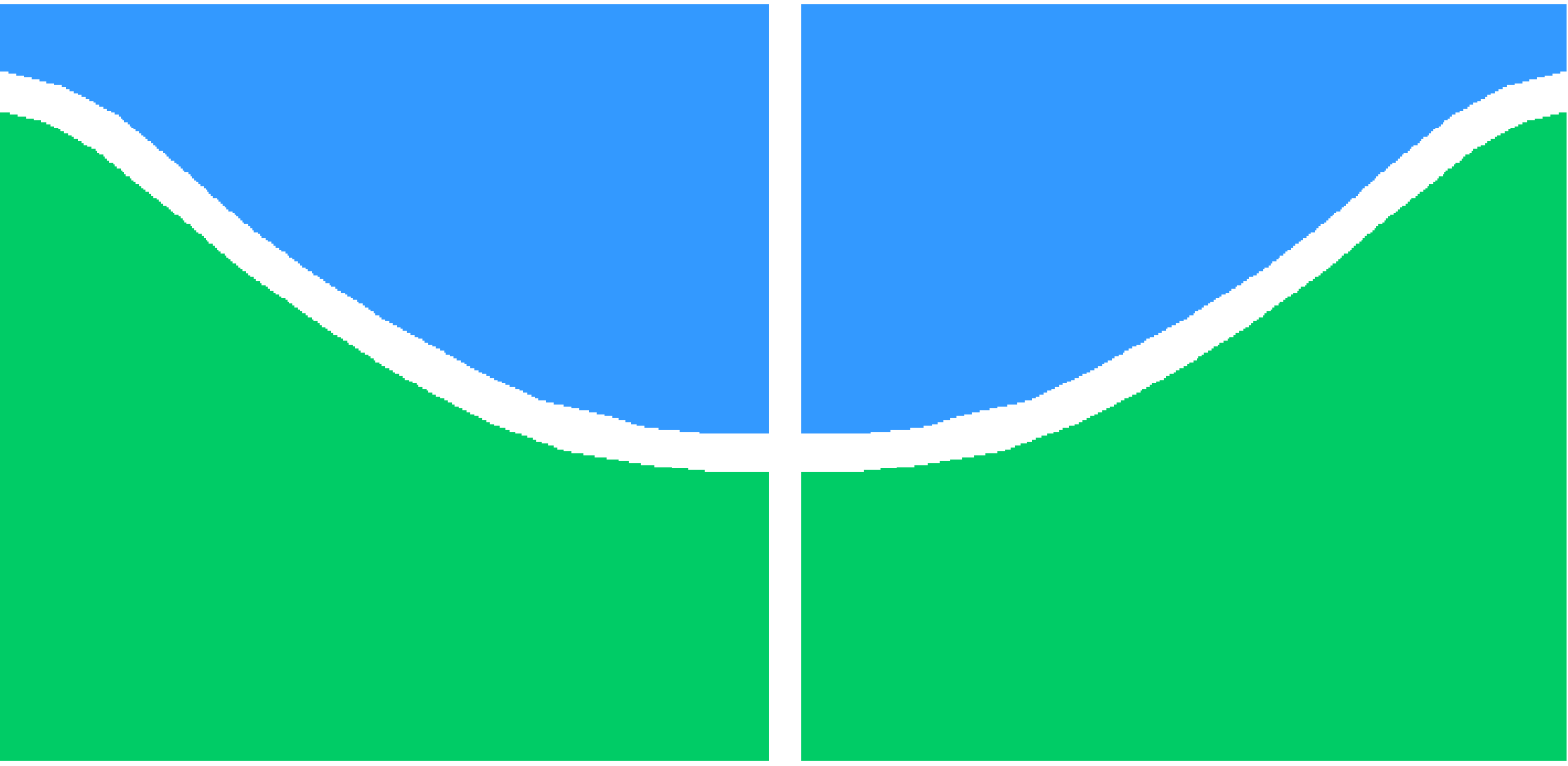

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Linguagem para Consulta de Código Fonte: Um Pilar para Ferramentas de Análise Estática de Código

Autor: Charles Daniel de Oliveira

Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeiras

Brasília, DF

2014

Charles Daniel de Oliveira

Linguagem para Consulta de Código Fonte: Um Pilar para Ferramentas de Análise Estática de Código

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeiras

Brasília, DF

2014

Charles Daniel de Oliveira

Linguagem para Consulta de Código Fonte: Um Pilar para Ferramentas de Análise Estática de Código/ Charles Daniel de Oliveira. – Brasília, DF, 2014

Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeiras

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2014.

1. analise-estatica. 2. C-Parser. I. Prof. Dr. Luiz Augusto Fontes Laranjeiras. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Linguagem para Consulta de Código Fonte: Um Pilar para Ferramentas de Análise Estática de Código

Charles Daniel de Oliveira

Linguagem para Consulta de Código Fonte: Um Pilar para Ferramentas de Análise Estática de Código

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Trabalho aprovado. Brasília, DF, 24 de junho de 2014:

Prof. Dr. Luiz Augusto Fontes Laranjeiras
Orientador

Prof. Dr. Paulo Roberto Miranda Meirelles
Convidado 1

Prof. Msc. Hilmer Rodrigues Neri
Convidado 2

Brasília, DF
2014

Agradecimentos

Agradeço primeiramente ao professor Luiz Augusto Fontes Laranjeira, entusiasta matemático, com fortes interesses na área de Segurança e Redes, por ter estabelecido contato com pesquisadores do NIST, o que possibilitou minha estadia de um ano em solo americano, realizando pesquisas em Análise Estática de Código para segurança e me motivando para o desenvolvimento deste trabalho.

Agradeço ao Paul E. Black, pesquisador do NIST há 15 anos, fã de Star Trek e KFC, por ter paciência e me apresentar aspectos importantes da Análise Estática de Código e desenvolvimento de Software Seguro.

Agradeço também os professores que fizeram questão de lecionar uma boa aula com o objetivo único do aprendizado dos alunos, sem eles a FGA não teria os resultados que tem hoje.

"I have not failed,
I have just found
10,000 ways that won't work".
Thomas A. Edison.

Resumo

O presente documento destina-se ao desenvolvimento do EzParser, um software capaz de ler um código fonte em C e disponibilizar uma interface de consulta para desenvolvedores de ferramentas de Análise Estática de Código. O EzParser utiliza o Bison/Yacc para ler e validar determinado código fonte escrito emC. Após a análise sintática, é realizado um processo de reestruturação do código fonte em uma base de dados Postgresql. Uma vez pesistidos, os dados podem ser consultados em forma de linguagem de consulta estruturada (SQL), ao invés das consultas recursivas ao usar ASTs. Por fim, o EzParser disponibilizará de uma DSL (Domain Specific Language) que possibilitará ao desenvolvedor escrever regras de extração e obter informações sobre o código fonte.

Palavras-chaves: C/C++. análise-estática. parser. segurança. qualidade. consulta de código

Abstract

This document details the development of EzParser, a parser capable of reading code written in C and provide an easy interface so that static code analysis tool can make use of it. Basically, EzParser makes use of Bison/Yacc during the process of reading and validating a piece of source code in C. Once the syntax analysis is done, there will be a process of re-organizing the source code data into a Postgresql database. After persistence is done, the data can be queried by SQLs, instead of endless recursive calls to the AST. After that, EzParser will be able to provide a DSL which a static code analysis tool developer writes rules, submit them to EzParser and gets the specific data.

Key-words: C/C++. static analysis. parser. security. quality. code querying

Lista de abreviaturas e siglas

.QL	Dot Query Language
AST	Abstract Syntax Tree
Bison	Gerador de parser da GNU
CWE	Common Weakness Enumeration
NVD	National Vulnerability Database
DSL	Domain Specific Language; Linguagem de Dominio Especifico
NIST	National Istitute of Standards and Techonology
OWASP	Organization
SQL	Structured Query Language
Yacc	Yet Another Compiler Compiler, gerador de tokens para o Bison

Sumário

Introdução

O computador foi inventado pelo homem há quase uma centena de anos e hoje pode ser considerado como a invenção mais inovadora desde o fogo. Mais fortemente incentivado pelas 2ª Guerra Mundial, o computador teve como propósito inicial servir de calculadora para auxiliar no cálculo da trajetória de fogetes ao serem lançados em campo inimigo. Naquela época a resolução de *bugs* era feita manualmente, ao analisar relés e cartões perfurados era possível, literalmente, encontrar insetos que estivessem impedindo o funcionamento de alguma válvula ou dispositivo físico.

Entretanto, somente anos depois de sua invençãoo, essa máquina chegou ao lar de civis, pessoas que não estão envolvidas em guerras ou as pesquisas das grandes universidades. Graças a dois alunos aficcionados por tecnologia (Bill Gates and Steve Jobs) o computador como conhecemos hoje foi tido como bem doméstico, para uso nas tarefas caseiras como calculadoras, jogos, editores de texto entre milhares de outras utilidades. Existem hoje aproximadamente 2 bilhões de computadores no mundo(HOW..., 2014), entre eles servidores, desktops, tablets, laptops e smatphones.

Ao contrário dos primórdios da computação, hoje em dia não é mais factível abrir o disco rígido de um computador para verificação de *bugs* em determinado programa. A dimensão de Software tomou proporções gigantescas a ponto de ser comum cada computador rodar programas que ultrapassam dezenas de milhões de linhas de código (CODEBASES..., 2013), tornando-se muito complicada a tarefa de checagem da qualidade de código e verificacao de falhas.

A Análise Estática de Código surge para solucionar esse problema, agindo com automatização das operações manuais, antes realizadas por humanos, como a conformidade com padroes de projeto, qualidade de código, revisão de código, entre outros diversos usos que são observados ao escrever código com o objetivo de ser integrado à um Software maior.

O objetivo do presente projeto é estudar a Análise Estática de Código de um modo geral e aprofundar a técnica chamada Consulta de Código, diferente da utilização de árvores de sintaxe abstratas que necessitam de entendimento um pouco mais aprofundado em compiladores para seu uso.

Inicialmente serão mostrados aspectos gerais de Análise Estática de Código, apresentando terminologia comum, as principais áreas da Engenharia de Software que fazem seu uso e algumas ferramentas existentes no mercado e academia. Posteriormente o estudo aprofundado sobre Consulta de Código será apresentado na forma de um *background* acadêmico, sem fazer qualquer comparação entre as tecnologias apresentadas. Por fim, será

apresentada a proposta do presente projeto de TCC.

Parte I

Análise Estática de Código

1 Aspectos Gerais

Neste capítulo serão detalhadas características da Análise de Código, um ramo da Engenharia de Software voltado para obter informações acerca de um Software (ANDREY, 2012) a partir do código fonte.

1.1 Qualidade de código

Assim como outras engenharias, é fundamental ter qualidade no principal produto em foco, que no caso da Engenharia de Software é o código. Pode-se melhorar a qualidade de código utilizando-se boas práticas de projeto, trabalhando-se em duplas, desenvolvendo-se testes antes de código entre várias outras técnicas (MARTIN, 2008).

Como parte do processo de garantia da qualidade de Software, são realizadas análise do código fonte produzido, gerando-se métricas de qualidade. As métricas funcionam como indicadores númericos que representam o estado de um Software (MEIRELLES, 2013).

1.2 Termos comuns

Não há uma especificação formal dos termos da Análise Estática de Código. Como visto em (STATIC..., 2013), e pelo período de experiência no NIST, segue uma listagem de termos mais comuns quando se fala de análise:

- Falso Positivo: o analisador aponta um trecho de código com determinada característica que não existe naquele trecho. É um erro do analisador.
- Falso Negativo: o analisador não aponta um trecho de código com determinada característica que deveria ser apontado. É um erro do analisador ou erro de especificação da característica.
- Testes Sintéticos: são códigos gerados com o único fim de serem submetidos aos analisadores estáticos. Contém diversos recursos que a linguagem proporciona a fim de testar a flexibilidade do analisador. A vantagem dos testes sintéticos é o conhecimento do que deve ser apontado pelo analisador, facilitando a categorização de verdadeiros/falsos positivos/negativos.
- Testes reais: são códigos fontes de Software reais, que são usados para avaliar a qualidade do analisador em termos de escalabilidade e prática em um projeto real.

1.3 Futuro

Hoje existem iniciativas que tem por objetivo de registrar as falhas de Software (NVD..., 2014) voltadas pra segurança com o intuito de estudá-las e apresentar pesquisas para se ter um entendimento maior sobre elas. Pode-se dizer que um dos frutos desse registro em massa de falhas de Software ocasionou na criação de uma outra frente de pesquisa, que é a categorização dos tipos de fraquezas existentes (CWE..., 2014). Nesse último, existe uma hierarquia numerada caracterizando cada fraqueza de Software. Desde *buffer overflow* até ausência de autenticação.

Espera-se que no futuro, a descrição de fraquezas de Software sejam detalhadas a ponto de ser possível a submissão de tais descrições como insumo para analisadores estáticos, que por sua vez, terão maturidade suficiente para vasculhar todo o projeto de código, apresentando baixa taxa de falsos positivos que também aplica-se a qualidade de Software, como mencionado em (ALVES; HAGE; RADEMAKER, 2011).

2 Lista de ferramentas

Existem diversas ferramentas de Análise Estática de Código, comerciais e livres. Abaixo segue uma breve lista:

Ferramenta	**Link**
Analizo	www.analizo.org/
CodeSonar	www.grammatech.com/codesonar
Coverity	coverity.com
GoAnna	redlizards.com
HP Fortify	to.ly/zn1R (short)
KlocWork	www.klocwork.com
LDRA	www.ldra.com/en
PMD	pmd.sourceforge.net
Semmle(Odasa)	semmle.com

Tabela 1 – Lista Resumida de Ferramentas de Análise Estática de Código

Para uma lista extensiva de ferramentas, visite (STATIC. . . , 2014) e (SOURCE. . . , 2014).

3 Consulta de código

Segundo (ALVES; HAGE; RADEMAKER, 2011), a aplicação da Consulta de Código é bastante ampla na área de Análise de Software: Analise Arquitetural, Engenharia Reversa, Verificação de Consistência, Verificação de Padrão de Código entre outros. O uso dessas tecnologias é proporcionado devido ao uso do paradigma extrair-abstrair-apresentar:

- Extrair: pegue o código fonte e mapei-o para alguma estrutura de armazenamento
- Abstrair: aplique operações e consultas nessa estrutura para obter resultados
- Apresentar: apresentar os resultados de forma gráfica

Esse seria o comportamento ideal das ferramentas de Consulta/Analisadores, porém a realidade é diferente. O paradigma é muito difícil de ser seguido porque cada ferramenta de consulta trabalha apenas com o escopo de uma linguagem em específico, e a integração com outras linguagens é um processo trabalhoso.

As ferramenta de Consulta de Código dependem da complexidade do problema. Um exemplo seria o de encontrar todas as chamadas de funções com nome *calc* em um trecho de código escrito em C. Bastaria digitar a seguinte linha de comando no Linux (considerando que o código segue o minimo em boas práticas):

- $: grep calc(. -R

Essa busca ainda é bastante falha pois não considera a possibilidade de *calc* ser chamada através de um ponteiro de função ou se *calc* tiver seu nome mascarado por uma cláusula *#DEFINE*. O programador poderia passar horas escrevendo o melhor regex que abrangesse todas as possíveis regras de chamada de uma função em C.

Existem diversas ferramentas que fazem exatamente o que o exemplo anterior ilustrou e mais variados recursos que surgem com a necessidade de obtenção de informação sobre um código. Abaixo segue uma breve listagem de ferramentas que são/usam Consulta de Código.

Parte II

Motivação

4 Por que da escolha do tema

4.1 Experiência no NIST

A experiência que tive no NIST, Gaithersburg, MD-USA, foi crucial para a motivação do presente Trabalho de conclusão de curso. No início de 2013, participei do *workshop Static Analisys Tools Exposition* (SATE) IV, um evento organizado pelo NIST que teve sua primeira edição no ano de 2009.

O SATE (SATE..., 2013) consiste na apresentação de resultados de experimentos em Análise Estática de Código voltada pra segurança, em que pesquisadores do NIST definem uma suíte de testes de código fonte, sintéticos e reais, e abrem inscrições para os desenvolvedores de ferramentas de dos analizadores terem a chance de submeter essa suíte de testes em suas ferramentas, que por sua vez fazem o melhor possível, em termos de customização da ferramenta, para gerar um bom resultado. Passado um periodo de alguns meses, os resultados, geralmente no formato de relatório XML, são repassados ao NIST (são muitos, cerca de 4GB).

Pesquisadores do NIST realizam uma amostragem dos relatórios e a partir da quantidade amostrada, fazem-se avaliações humanas sobre os resultados das ferramentas, isto e, providos da suíte de teste e do relatório disponibilizado pela ferramenta, é possivel avaliar se a ferramenta realmente detectou um *bug* (verdadeiro positivo) ou se gerou um lixo (falso positivo).

Terminadas as avaliações da amostra, o resultado é passado aos responsáveis pelas ferramentas na forma de feedback. No dia do evento SATE, os responsáveis tem oportunidade de mostrar seus resultados e trocar conhecimento com os pesquisadores do NIST com propósito de ambas as partes sincronizarem suas pesquisas.

4.2 Ausência de ferramentas brasileiras

Percebendo como o assunto era tratado nos Estados Unidos, ficou evidente que Análise Estática de Código ainda dá seus primeiros passos no Brasil.

A única ferramenta que se tem conhecimento com origem nacional é o Analizo (ANALIZO, 2014), uma ferramenta de Análise Estática de Código multilinguagem (Java e C/C++) voltada para extração de métricas de qualidade e geração de gráficos de dependência.

Acredito que essa área tem forte espaço no futuro e por isso decidi investir na

ideia de criar uma ferramenta que de base para criação de novas ferramentas de Análise Estática. É muito complicado manter um *parser* de qualquer linguagem atualizado, visto que uma linguagem evolui com o tempo. Considerando que a maioria das ferramentas de análise são baseadas em um *parser* para extrair as informações de código pode-se supor que um dos fatores bloqueantes para criação de novas ferramentas seja a complexidade de se extrair informações do código.

Existindo uma ferramenta *open-source* que disponibilize as informacoes do código de maneira facilitada, atravéz de consultas semelhantes à SQL, pode ser que seja fomentada a criação de novos analisadores estáticos.

Parte III

Desenvolvimento

5 Prática

5.1 Problema

Como citado em (HAJIYEV; VERBAERE; MOOR, 2006), o entendimento do código fonte é vital para muitas tarefas da Engenharia de Software, e ferramentas de consulta de código fonte são construídas para apoiar essas tarefas, permitindo programadores explorar as relações existentes entre as diferentes partes do código. Ambientes integrados de desenvolvimento modernos provem suporte à Consulta de Código, porém de forma bastante limitada, apenas para consulta de strings e, algumas vezes, referências de funções.

Como apresentado no capítulo 3, existem diversas ferramentas para consulta. Há ausência para Consultas de Código fonte escrito em C. Também foi notada uma certa ausência do termo segurança na descrição das ferramentas e isso mostrou-se um campo vago para exploração. Existem hoje diversas ferramentas de Análise Estática de Código voltadas para segurança (SOURCE. . . , 2014), mas não foi encontrado estudos sobre relação de qual delas trabalha com Consulta de Código e quais não trabalham.

5.2 Proposta

A proposta é a criação de uma ferramenta, o EzParser, que tem como entrada código fonte escrito em C e, como saida, uma base de dados que permita consultas referentes ao código de entrada. Estão embutidos na ferramenta um parser de código em C (Yacc + Bison), *scripts* em PHP e uma engine para interpretar as consultas.

O parser é responsável por validar a sintaxe do C, que apesar da simplicidade da linguagem, pode trazer lógica complexa e difícil de ser processada. Os scripts serão criados para auxiliarem no tratamento de strings, na fase de leitura da gramática do C para criação do parser e do banco de dados. A engine será escrita em C++ e terá características descritas a seguir.

A DSL será uma linguagem simples, baseada em SQL e na .QL, para consulta do código fonte de apenas um arquivo escrito em C, sem inclusão de bibliotecas externas. A intenção é mapear a consulta para retornar os seguintes recursos da linguagem de forma precisa:

- *Statements*
- Variáveis/Acesso de *buffers*:
  - Leitura ou escrita

  - *Stack* ou *Heap*

- Fluxo do programa:
  - Laços
  - Condicionais
  - Recursão
  - *Goto*
- Funções
- *Input/Output*
- Expressões
- Expressões Aritméticas
- Expressões Booleanas
- Argumentos de Funções
- Derreferência de Ponteiro
- Ponteiro de Função

5.3 Resultados esperados

Tendo-se a base de dados em mãos, espera-se consultar o código fonte para obter informações para qualquer uso. Um bom exemplo seria a extração de métricas a partir da consulta de elementos do código, como número de funções por arquivo, número de argumentos por função, número de linhas dentro de cada função entre outros. Outro bom exemplo de uso seria a extração de possíveis vulnerabilidades de códigos, consultando todos os accessos de *buffers*, fazendo-se consultas recursivas para encontrar os possíveis valores do índice até o momento em que foi usado para acessar o *buffer*.

Pelo fato do código fonte ser mapeado em um banco de dados, haverá lentidão maior no início do processo, porem é esperada melhor performance na extração de informações do código. Ainda por estar sob o formato de banco de dados, espera-se diminuir a complexidade da forma de extração dessas informações e ao mesmo tempo proporcionar ao desenvolvedor de ferramentas de Análise Estática de Código um meio de personalizar as consultas, sem ter que se preocupar com estruturas complexas da Árvore de Sintaxe Abstrata.

5.3.1 Especificação inicial da DSL

A linguagem terá características da .QL e SQL. Abaixo segue um trecho de .QL usado para consultar os atributos públicos de todas as classes que não sejam *final*:

```
FROM field f
WHERE f.hasModifier("public") AND NOT(f.hasModifier("final"))
SELECT f.getDeclaringType().getPackage(),
f.getDeclaringType(),
f
```

Muito semelhante à SQL, possuirá um espaco para especificação da seleção de entidade (*FROM*), das cláusulas booleanas para filtros (*WHERE*) e dos dados a serem consultados (*SELECT*).

Vale a pena ressaltar que a abstração *field* foi uma implementação específica da .QL. No caso da DSL do EzParser, haverá ambas abstrações como também entidades encontradas na gramática da linguagem C, disponível em anexo.

6 Considerações

6.1 O que o projeto não é

O EzParser náo é um Analisador Estático, *code grapper*, nem muito menos um compilador. Focando apenas na fase intermediária da compilação de um programa em C, o EzParser é fundamentalmente uma plataforma de consulta de código referente a um projeto em específico.

Por questões de tempo, o escopo precisou ser restrito à leitura de somente um arquivo de código fonte em C, afim de evitar problemas de complexidade com diretivas de pré-processamento, inclusão de arquivos, chamada de funções extra-arquivo entre outras dependências que a separação em arquivos acarreta.

6.2 Desenvolvimento para o TCC2

Para o TCC2 o foco será voltado basicamente implementar a linguagem de consulta sobre um determinado código em C. Espera-se que a consulta de código utilizando-se a linguagem seja pelo menos mais rápida que a consulta utilizando-se de ASTs carregadas diretamente na memória.

Como requisito não funcional, espera-se também que a forma com que um determinado código é consultado seja de fácil entendimento para qualquer desenvolvedor, sem a necessidade de aprender uma nova linguagem de programação

Referências

ALVES, T. L.; HAGE, J.; RADEMAKER, P. A comparative study of code query technologies. p. 12, April 2011. ISSN 0924-3275. Citado 2 vezes nas páginas 22 e 25.

ANALIZO. 2014. Disponível em: <http://www.analizo.org/>. Citado na página 29.

ANDREY, K. Static code analysis. 2012. Disponível em: <http://www.codeproject.com/Tips/344663/Static-code-analysis>. Citado na página 21.

CODEBASES - Millions of lines of code. 2013. Disponível em: <http://www.informationisbeautiful.net/visualizations/million-lines-of-code/>. Citado na página 17.

CWE: Common Weakness Enumeration. 2014. Disponível em: <http://cwe.mitre.org/>. Citado na página 22.

HAJIYEV, E.; VERBAERE, M.; MOOR, O. de. Scalable source code queries with datalog. p. 26, 2006. Citado na página 33.

HOW many computers were sold this year. 2014. Disponível em: <http://www.worldometers.info/computers/>. Citado na página 17.

MARTIN, R. C. Clean code: A handbook of agile software craftsmanship. 2008. Citado na página 21.

MEIRELLES, P. R. M. Monitoramento de métricas de código-fonte em projetos de software livre. 2013. Disponível em: <http://www.teses.usp.br/teses/disponiveis/45/45134/tde-27082013-090242/publico/tesePauloMeirelles.pdf>. Citado na página 21.

NVD: National Vulnerability Database. 2014. Disponível em: <http://nvd.nist.gov/>. Citado na página 22.

SATE: Static Analysis Tool Exposition. 2013. Disponível em: <http://samate.nist.gov/SATE4.html>. Citado na página 29.

SOURCE Code Security Analyzers. 2014. Disponível em: <http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html>. Citado 2 vezes nas páginas 23 e 33.

STATIC Code Analysis. 2013. Disponível em: <https://www.owasp.org/index.php/Static_Code_Analysis>. Citado na página 21.

STATIC Source Code Analysis Tools for C. 2014. Disponível em: <http://spinroot.com/static/>. Citado na página 23.

Anexos

ANEXO A – Gramática do C-2011

```
%{
#include <cstdio>
#include <iostream>
#include <AST.h>
#include <CRules.h>
#include <c_grammar_2011.tab.h>  // to get the token types that we return
using namespace std;

// stuff from flex that bison needs to know about
extern "C" int yylex();
extern "C" FILE *yyin;
extern CRules * crule;
void yyerror(const char *s);
%}

/* Declare data types to be used */
%union {
double number;
char * word;
AST * ast;
}

%token IDENTIFIER I_CONSTANT F_CONSTANT STRING_LITERAL FUNC_NAME SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN
%token TYPEDEF_NAME ENUMERATION_CONSTANT

%token TYPEDEF EXTERN STATIC AUTO REGISTER INLINE
%token CONST RESTRICT VOLATILE
%token BOOL CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE VOID
%token COMPLEX IMAGINARY
%token STRUCT UNION ENUM ELLIPSIS
```

```
%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%token ALIGNAS ALIGNOF ATOMIC GENERIC NORETURN STATIC_ASSERT THREAD_LOCAL

%start translation_unit
%%

primary_expression
: IDENTIFIER
| constant
| string
| '(' expression ')'
| generic_selection
;

constant
: I_CONSTANT
| F_CONSTANT
| ENUMERATION_CONSTANT
;

enumeration_constant
: IDENTIFIER
;

string
: STRING_LITERAL
| FUNC_NAME
;

generic_selection
: GENERIC '(' assignment_expression ',' generic_assoc_list ')'
;

generic_assoc_list
: generic_association
| generic_assoc_list ',' generic_association
;
```

```
generic_association
: type_name ':' assignment_expression
| DEFAULT ':' assignment_expression
;

postfix_expression
: primary_expression
| postfix_expression '[' expression ']'
| postfix_expression '(' ')'
| postfix_expression '(' argument_expression_list ')'
| postfix_expression '.' IDENTIFIER
| postfix_expression PTR_OP IDENTIFIER
| postfix_expression INC_OP
| postfix_expression DEC_OP
| '(' type_name ')' '{' initializer_list '}'
| '(' type_name ')' '{' initializer_list ',' '}'
;

argument_expression_list
: assignment_expression
| argument_expression_list ',' assignment_expression
;

unary_expression
: postfix_expression
| INC_OP unary_expression
| DEC_OP unary_expression
| unary_operator cast_expression
| SIZEOF unary_expression
| SIZEOF '(' type_name ')'
| ALIGNOF '(' type_name ')'
;

unary_operator
: '&'
| '*'
| '+'
| '-'
| '~'
```

```
| '!'
;

cast_expression
: unary_expression
| '(' type_name ')' cast_expression
;

multiplicative_expression
: cast_expression
| multiplicative_expression '*' cast_expression
| multiplicative_expression '/' cast_expression
| multiplicative_expression '%' cast_expression
;

additive_expression
: multiplicative_expression
| additive_expression '+' multiplicative_expression
| additive_expression '-' multiplicative_expression
;

shift_expression
: additive_expression
| shift_expression LEFT_OP additive_expression
| shift_expression RIGHT_OP additive_expression
;

relational_expression
: shift_expression
| relational_expression '<' shift_expression
| relational_expression '>' shift_expression
| relational_expression LE_OP shift_expression
| relational_expression GE_OP shift_expression
;

equality_expression
: relational_expression
| equality_expression EQ_OP relational_expression
| equality_expression NE_OP relational_expression
```

```
;

and_expression
: equality_expression
| and_expression '&' equality_expression
;

exclusive_or_expression
: and_expression
| exclusive_or_expression '^' and_expression
;

inclusive_or_expression
: exclusive_or_expression
| inclusive_or_expression '|' exclusive_or_expression
;

logical_and_expression
: inclusive_or_expression
| logical_and_expression AND_OP inclusive_or_expression
;

logical_or_expression
: logical_and_expression
| logical_or_expression OR_OP logical_and_expression
;

conditional_expression
: logical_or_expression
| logical_or_expression '?' expression ':' conditional_expression
;

assignment_expression
: conditional_expression
| unary_expression assignment_operator assignment_expression
;

assignment_operator
: '='
```

```
| MUL_ASSIGN
| DIV_ASSIGN
| MOD_ASSIGN
| ADD_ASSIGN
| SUB_ASSIGN
| LEFT_ASSIGN
| RIGHT_ASSIGN
| AND_ASSIGN
| XOR_ASSIGN
| OR_ASSIGN
;

expression
: assignment_expression
| expression ',' assignment_expression
;

constant_expression
: conditional_expression
;

declaration
: declaration_specifiers ';'
| declaration_specifiers init_declarator_list ';'
| static_assert_declaration
;

declaration_specifiers
: storage_class_specifier declaration_specifiers
| storage_class_specifier
| type_specifier declaration_specifiers
| type_specifier
| type_qualifier declaration_specifiers
| type_qualifier
| function_specifier declaration_specifiers
| function_specifier
| alignment_specifier declaration_specifiers
| alignment_specifier
;
```

```
init_declarator_list
: init_declarator
| init_declarator_list ’,’ init_declarator
;

init_declarator
: declarator ’=’ initializer
| declarator
;

storage_class_specifier
: TYPEDEF
| EXTERN
| STATIC
| THREAD_LOCAL
| AUTO
| REGISTER
;

type_specifier
: VOID
| CHAR
| SHORT
| INT
| LONG
| FLOAT
| DOUBLE
| SIGNED
| UNSIGNED
| BOOL
| COMPLEX
| IMAGINARY
| atomic_type_specifier
| struct_or_union_specifier
| enum_specifier
| TYPEDEF_NAME
;
```

```
struct_or_union_specifier
: struct_or_union '{' struct_declaration_list '}'
| struct_or_union IDENTIFIER '{' struct_declaration_list '}'
| struct_or_union IDENTIFIER
;

struct_or_union
: STRUCT
| UNION
;

struct_declaration_list
: struct_declaration
| struct_declaration_list struct_declaration
;

struct_declaration
: specifier_qualifier_list ';'
| specifier_qualifier_list struct_declarator_list ';'
| static_assert_declaration
;

specifier_qualifier_list
: type_specifier specifier_qualifier_list
| type_specifier
| type_qualifier specifier_qualifier_list
| type_qualifier
;

struct_declarator_list
: struct_declarator
| struct_declarator_list ',' struct_declarator
;

struct_declarator
: ':' constant_expression
| declarator ':' constant_expression
| declarator
;
```

```
enum_specifier
: ENUM '{' enumerator_list '}'
| ENUM '{' enumerator_list ',' '}'
| ENUM IDENTIFIER '{' enumerator_list '}'
| ENUM IDENTIFIER '{' enumerator_list ',' '}'
| ENUM IDENTIFIER
;

enumerator_list
: enumerator
| enumerator_list ',' enumerator
;

enumerator
: enumeration_constant '=' constant_expression
| enumeration_constant
;

atomic_type_specifier
: ATOMIC '(' type_name ')'
;

type_qualifier
: CONST
| RESTRICT
| VOLATILE
| ATOMIC
;

function_specifier
: INLINE
| NORETURN
;

alignment_specifier
: ALIGNAS '(' type_name ')'
| ALIGNAS '(' constant_expression ')'
;
```

```
declarator
: pointer direct_declarator
| direct_declarator
;

direct_declarator
: IDENTIFIER {crule->direct_declarator__IDENTIFIER();}
| '(' declarator ')'
| direct_declarator '[' ']'
| direct_declarator '[' '*' ']'
| direct_declarator '[' STATIC type_qualifier_list assignment_expression ']'
| direct_declarator '[' STATIC assignment_expression ']'
| direct_declarator '[' type_qualifier_list '*' ']'
| direct_declarator '[' type_qualifier_list STATIC assignment_expression ']'
| direct_declarator '[' type_qualifier_list assignment_expression ']'
| direct_declarator '[' type_qualifier_list ']'
| direct_declarator '[' assignment_expression ']'
| direct_declarator '(' parameter_type_list ')'
| direct_declarator '(' ')'
| direct_declarator '(' identifier_list ')'
;

pointer
: '*' type_qualifier_list pointer
| '*' type_qualifier_list
| '*' pointer
| '*'
;

type_qualifier_list
: type_qualifier
| type_qualifier_list type_qualifier
;


parameter_type_list
: parameter_list ',' ELLIPSIS
| parameter_list
```

```
;

parameter_list
: parameter_declaration
| parameter_list ',' parameter_declaration
;

parameter_declaration
: declaration_specifiers declarator
| declaration_specifiers abstract_declarator
| declaration_specifiers
;

identifier_list
: IDENTIFIER
| identifier_list ',' IDENTIFIER
;

type_name
: specifier_qualifier_list abstract_declarator
| specifier_qualifier_list
;

abstract_declarator
: pointer direct_abstract_declarator
| pointer
| direct_abstract_declarator
;

direct_abstract_declarator
: '(' abstract_declarator ')'
| '[' ']'
| '[' '*' ']'
| '[' STATIC type_qualifier_list assignment_expression ']'
| '[' STATIC assignment_expression ']'
| '[' type_qualifier_list STATIC assignment_expression ']'
| '[' type_qualifier_list assignment_expression ']'
| '[' type_qualifier_list ']'
| '[' assignment_expression ']'
```

```
| direct_abstract_declarator '[' ']'
| direct_abstract_declarator '[' '*' ']'
| direct_abstract_declarator '[' STATIC type_qualifier_list assignment_expression ']'
| direct_abstract_declarator '[' STATIC assignment_expression ']'
| direct_abstract_declarator '[' type_qualifier_list assignment_expression ']'
| direct_abstract_declarator '[' type_qualifier_list STATIC assignment_expression ']'
| direct_abstract_declarator '[' type_qualifier_list ']'
| direct_abstract_declarator '[' assignment_expression ']'
| '(' ')'
| '(' parameter_type_list ')'
| direct_abstract_declarator '(' ')'
| direct_abstract_declarator '(' parameter_type_list ')'
;

initializer
: '{' initializer_list '}'
| '{' initializer_list ',' '}'
| assignment_expression
;

initializer_list
: designation initializer
| initializer
| initializer_list ',' designation initializer
| initializer_list ',' initializer
;

designation
: designator_list '='
;

designator_list
: designator
| designator_list designator
;

designator
: '[' constant_expression ']'
| '.' IDENTIFIER
```

```
;

static_assert_declaration
: STATIC_ASSERT '(' constant_expression ',' STRING_LITERAL ')' ';'
;

statement
: labeled_statement
| compound_statement
| expression_statement
| selection_statement
| iteration_statement
| jump_statement
;

labeled_statement
: IDENTIFIER ':' statement
| CASE constant_expression ':' statement
| DEFAULT ':' statement
;

compound_statement
: '{' '}'
| '{'  block_item_list '}'
;

block_item_list
: block_item
| block_item_list block_item
;

block_item
: declaration
| statement
;

expression_statement
: ';'
| expression ';'
```

```
;

selection_statement
: IF '(' expression ')' statement ELSE statement
| IF '(' expression ')' statement
| SWITCH '(' expression ')' statement
;

iteration_statement
: WHILE '(' expression ')' statement
| DO statement WHILE '(' expression ')' ';'
| FOR '(' expression_statement expression_statement ')' statement
| FOR '(' expression_statement expression_statement expression ')' statement
| FOR '(' declaration expression_statement ')' statement
| FOR '(' declaration expression_statement expression ')' statement
;

jump_statement
: GOTO IDENTIFIER ';'
| CONTINUE ';'
| BREAK ';'
| RETURN ';'
| RETURN expression ';'
;

translation_unit
: external_declaration
| translation_unit external_declaration
;

external_declaration
: function_definition
| declaration
;

function_definition
: declaration_specifiers declarator declaration_list compound_statement
| declaration_specifiers declarator compound_statement
;
```

```
declaration_list
: declaration
| declaration_list declaration
;

%%
#include <stdio.h>

void yyerror(const char *s)
{
fflush(stdout);
fprintf(stderr, "*** %s\n", s);
}
```