EasyChair Preprint

№ 2147

CWE Pattern Recognition Algorithm in Any-Language Source Code

Sergiu Zaharia

December 12, 2019

CWE Pattern Recognition Algorithm in Any-Language Source Code*

Sergiu Zaharia†
Security Center of Excellence
BearingPoint / Bucharest, Romania
sergiu.zaharia@bearingpoint.com

ABSTRACT

Source code became one of the backbones for business and personal processes, with significant growth rate. As applications are one of the most used attack surfaces against individuals and organizations from all sectors, their intrinsic vulnerability arising from the supporting source code must be reduced by design. Currently there are technology providers and open communities which provide Static Analysis Security Testing (SAST) solutions, able to detect vulnerabilities in code written in the most used programming languages and development frameworks.

The proposed solution consists of a Code Analysis Module that can identify vulnerability patterns in source code written in languages with less coverage, including code developed in languages which have not been previously learned by the solution. The ability of understanding and transforming unknown programming languages to the Intermediate Representation, which is then analyzed by a common machine learning algorithm for vulnerability patterns, is core idea for this research project.

CCS CONCEPTS

• Security and privacy / Software and application security

• Security and privacy / Software security engineering

KEYWORDS

Static Analysis, Software Vulnerabilities, Application Security

1. CONTEXT

Source code became one of the backbones for business and personal processes, with significant growth rate. GitHub, the development platform used by 28 million developers, declared that more than 2.9 trillion lines of code have been committed in 2017 alone [1]. As applications are one of the most used attack surfaces against individuals and organizations from all sectors in the last years, their intrinsic vulnerability arising from the supporting source code has to be reduced by design. Overall, there is a strong need for solutions being able to scan source code automatically and identify code level security vulnerabilities early in the software development phase. Currently, SAST solutions cover only Top 30 most used programming languages and development frameworks. The remaining programming languages (e.g. D, R languages) are not secured as result of the gap in both supporting technology and security experts. This situation opens a huge attack surface for hackers willing to compromise applications and consequently, organizations' or individuals' security. Today, an amount of 995 technical vulnerabilities - specific to different programming languages or common to all types of source code - is maintained by MITRE [3] as Common Weakness Enumeration (CWE) items.

2. PROPOSED SOLUTION

We recommend the Code Analysis Module which identifies CWE patterns in source code written in both popular programming languages or in languages with less coverage, including languages which have not been learned by the solution. From a functional perspective, this consists of the following two blocks:

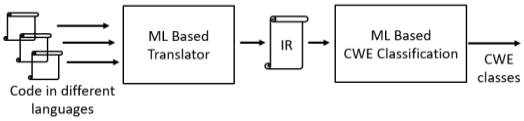

Figure 1: Code Analysis Module – Functional Blocks

The **ML Based Translator** is built on a Language-Agnostic scanner which transforms any language into an Intermediate Representation (IR) using similarities of lexical tokens within programming languages. Languages like C, Java and those inheriting them have quite similar keywords used by different but close grammars. This property can leverage the transformation of various languages in a common IR, using NLP-aware algorithms to choose the best IR keyword ($r_k^{(IR)}$, k=1..n, in Figure 2).

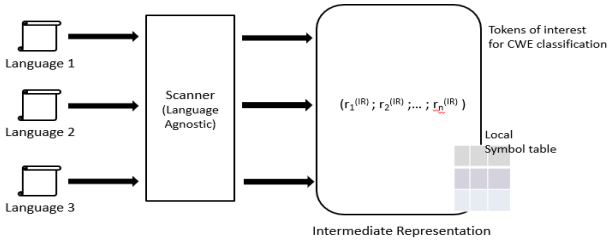

Figure 2: Transformation of programming languages in IR

For example, a snippet of CWE23 vulnerable code:

```
recvResult = recv(connectSocket, (char *)(data + dataLen), sizeof(char) * (FILENAME_MAX - dataLen - 1), 0); pFile = FOPEN(data, "wb+");
```

is represented in IR format as below, where keyword are tokens not yet attributed to CWEs, which preserve program's context, and data flow is maintained through the relative positioning of tokens in the CWE classification input vector. The IR is practically an ordered set of CWE relevant tokens and generic identifiers, built using a specially designed Machine Learning algorithm.

*Research topic under the PhD Program in University POLITEHNICA of Bucharest;
† Correspondence to: sergiu.zaharia@bearingpoint.com

IR for code snippet	IR for CWE classification algorithm
CWE_TOKENS['0'] = 'recv'	nr_cwe_tokens['0'] = '2'
CWE_TOKENS['1'] = 'FOPEN'	nr_cwe_tokens['1'] = '1'
CWE_TOKENS['2'] = '0'	nr_cwe_tokens['2'] = '0'
CWE_TOKENS['3'] = '0'	nr_cwe_tokens['3'] = '0'
CWE_KEYWORDS['0'] = 'char'	nr_cwe_keywords['0'] = '11'
CWE_KEYWORDS['1'] = 'sizeof'	nr_cwe_keywords['1'] = '13'
CWE_KEYWORDS['2'] = 'char'	nr_cwe_keywords['2'] = '11'
CWE_KEYWORDS['3'] = '0'	nr_cwe_keywords['3'] = '0'
CWE_KEYWORDS['4'] = '0'	nr_cwe_keywords['4'] = '0'
CWE_KEYWORDS['5'] = '0'	nr_cwe_keywords['5'] = '0'
IDENTIFIERS['0'] = 'recvResult'	generic_identifiers['0'] = '1'
IDENTIFIERS['1'] = 'connectSocket'	generic_identifiers['1'] = '2'
IDENTIFIERS['2'] = **'data'**	generic_identifiers['2'] = **'3'**
IDENTIFIERS['3'] = ''	generic_identifiers['3'] = '4'
IDENTIFIERS['4'] = 'dataLen'	generic_identifiers['4'] = '5'
IDENTIFIERS['5'] = 'FILENAME_MAX'	generic_identifiers['5'] = '6'
IDENTIFIERS['6'] = 'dataLen'	generic_identifiers['6'] = '5'
IDENTIFIERS['7'] = 'pFile'	generic_identifiers['7'] = '7'
IDENTIFIERS['8'] = **'data'**	generic_identifiers['8'] = **'3'**
IDENTIFIERS['9'] = '"wb'	generic_identifiers['9'] = '8'
IDENTIFIERS['10'] = '"'	generic_identifiers['10'] = '9'
IDENTIFIERS['11'] = '0'	generic_identifiers['11'] = '0'

Generic identifiers are abstract representations of real identifiers, whose position in the source code is maintained very loosely via an empirical symbol table (only generic name and relative positions are maintained). The ability of understanding and transforming unknown programming languages to IR - based on lexical similarities between programming languages - is core to this research project.

The **ML Based CWE Classification** block identifies CWE patterns in the IR, using machine learning algorithms for classification of non-linear patterns (e.g. SVM). The algorithm should be able to identify CWE classes for each source code snippet, using an "one versus all" approach. Code snippets may have more than one CWE vulnerability, consequently, the classifier may identify more than one class per each code snippet.

3. APPROACH

Research is planned in three main phases, as defined below,

Phase I: to design and build the training set for the CWE Classifier, starting from popular programming languages. The activities consist of identifying source code known as being vulnerable and the CWE-pattern relevant code snippets; designing the pre-processing algorithm applied to IR data sets, for later use in ML Based CWE classification algorithm; and finally, building the training data set for one programming language (e.g. C) and one vulnerability (e.g. CWE 23). We use NIST Juliet Test Suite [2] with vulnerable C, Java, C# and PHP source code. For C/C++, the repository consists of 8.67 million lines of code covering 118 CWEs. As today, the potential structure of IR has been designed considering the relevant tokens for specific CWEs, generalization of identifiers and relative positioning of tokens and identifiers, as a light data flow remanence.

Phase II: to identify the best model for the machine learning algorithm used for CWE pattern identification, using the training data sets from Phase I, and to enrich the solution with one more class (CWE pattern) and one more language (Java code snippets).

Phase III: to design and demonstrate the core concept of identifying CWE pattern in any-language source code. Includes Language Agnostic Scanner design for C and Java code translation to IR, adding new programming languages to adjust the algorithm, training the ML Based Translator to correctly represent the source code snippets in the IR format, and adjusting the accuracy of CWE pattern classification using the input resulted from the ML Based Translator, for C, Java and new languages.

4. RELATED WORK

Studies for source code splitting in smaller pieces like code snippets, logically mapped to vulnerabilities or code clone [4] do focus on specific languages, the Language-Agnostic Scanner not being in general addressed by previous work. One similar approach [5] considers deep learning for source code vulnerability detection. The authors use "code gadgets" to represent programs in a granular way, then vectorized as input to deep learning. The authors declare solution's limitations to C/C++ programs and to vulnerabilities dealing only with library/API calls.

Other studies [6][7][9][10][14][12] propose static analysis methods strongly related to one or two programming languages, even when the concept may be replicated for different languages. The drawback comes from the cumulated time and from the required expertise in both the new language scanner to be implemented and the static analysis concept itself defined in the respective study.

As an exception, ReDeBug [11] identifies latent security vulnerabilities in programs "written in different languages", as result of "a lightweight syntax-based code clone detection system" but limited to languages used in OS distributions. A different concept is implemented using signal processing techniques [13], which maintains the method language-independent, with the limitation that security vulnerabilities' localization is not realized.

ACKNOWLEDGMENTS

The author acknowledges Prof. Dr. Ștefan Trausan-Matu and Associate Professor, Dr. Traian Rebedea for their support and creative ideas provided under this PhD exercise.

REFERENCES

[1] https://github.com/ten, April 2018
[2] https://samate.nist.gov/SARD/testsuite.php
[3] https://cwe.mitre.org/
[4] Pham, Nam Hoai, *Detection of recurring software vulnerabilities* (2010). Graduate Theses and Dissertations, 11590, https://lib.dr.iastate.edu/etd/11590
[5] *VulDeePecker: A Deep Learning-Based System for Vulnerability Detection*, Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, 5 Jan 2018
[6] *Chucky: Exposing Missing Checks in Source Code for Vulnerability Discovery*, Nov. 2013, https://user.informatik.uni-goettingen.de/~krieck/docs/2013-ccs.pdf
[7] *Understanding Bag-of-Words Model: A Statistical Framework,* Yin Zhang, Rong Jin, Zhi-Hua Zhou
[8] *Generating robust parsers using island grammars*, IEEE2001, academia.edu/31982245/Generating_robust_parsers_using_island_grammars
[9] *Automated software vulnerability detection with machine learning*, J. Harer, L.Y. Kim, R.L. Russell, O. Ozdemir, L.R. Kosta, K. Rangamani, Lei H. Hamilton, Gabriel I. Centeno, Jonathan R. Key, Paul M. Ellingwood, Marc W. McConley, Jeffrey M. Opper, Peter Chin, Tomo Lazovich, 14 February 2018.
[10] *Automatic Inference of Search Patterns for Taint-Style Vulnerabilities*, F.Yamaguchi, A.Maier, H. Gascon, K. Rieck, Univ. of G¨ottingen, Germany
[11] J. Jang, A. Agrawal, and D. Brumley, *ReDeBug: Finding unpatched code clones in entire OS distributions,* in Proceedings of the 33th IEEE Symposium on Security and Privacy. IEEE, 2012, pp. 48–62.
[12] *SourcererCC: Scaling Code Clone Detection to Big Code*, H. Sajnani, V. Saini. J. Svajlenkoy, C. K. Royy, C.V. Lopes, 2015, USA
[13] *The use of machine learning with signal- and NLP processing of source code to fingerprint, detect, and classify vulnerabilities and weaknesses with MARFCAT*, Serguei A. Mokhov, Nov 2011, https://arxiv.org/pdf/1010.2511.pdfConference
[14] R.L. Russell, L. Kim, L.H. Hamilton, T. Lazovich, J.A. Harer, O. Ozdemir, P.M. Ellingwood, M.W. McConley, *Automated Vulnerability Detection in Source Code Using Deep Representation Learning*