On the Effectiveness of Deep Vulnerability Detectors to Simple Stupid Bug Detection

Jiayi Hua, Haoyu Wang
School of Computer Science, Beijing University of Posts and Telecommunications, Beijing, China

Abstract—**Recent studies have shown the promising direction of deep learning based bug detection, which relieves human experts from the tedious and subjective task of manually summarizing features. Simple one-statement bugs (i.e., SStuBs), which occur relatively often in Java projects, cannot be well spotted by existing static analysis tools. In this paper, we make effort to empirically analyze whether deep learning based techniques could be used to detecting SStuBs. We have re-implemented two state-of-the-art techniques in approximately 3,000 lines of code and adopted them to detecting Java SStuBs. Experiments on large-scale datasets suggest that although deep vulnerability detectors can achieve much better results than existing static analyzers, the SStuBs cannot be well flagged when comparing with traditional complex vulnerabilities. We further look in detail on the per bug category basis, observing that deep learning based methods perform better when detecting some specific types of bugs (e.g., "Same Function Change Caller"), which have strong data flow and control flow semantic. Our observations could offer implications on the automated detection and repair of SStuBs.**

I. Introduction

Bug detection and program repair are indispensable in software maintenance. Detecting and fixing bugs at the early stage of software development cycle will reduce software maintenance cost. In order to alleviate manual effort of locating and repairing bugs, many tools and techniques are proposed to detect bugs automatically, e.g., SpotBugs [1], PMD [2] and CheckStyle [3]. However, these traditional static analysis tools need experts to define specific detection rules for different types of bugs in advance, which is still labour-intensive and time-consuming, and may incur high false positive/negative rates. To deal with this limitation, some researchers turn their attention to deep learning based techniques. Recent studies have shown that deep learning can boost the performance of detecting data-flow-related vulnerabilities [4], control-flow-related vulnerabilities [5] and a wide range of vulnerabilities [6, 7], comparing to well-known conventional static detectors. For example, VulDeePecker [4] claims to achieve an F1-measure of over 90% when detecting buffer errors, which is much better than other static pattern based analyzers, including Flawfinder [8], RATS [9] and Checkmarx [10].

Simple one-statement bugs, or the so-called simple stupid bugs (SStuBs), are bugs that appear on a single statement and the corresponding fix is within that statement. Prior work [11] showed that SStuBs are relatively common in Java projects with a frequency of about one bug per 1,600-2,500 lines of code. They also provided a dataset, ManySStuBs4J, which has 153,652 SStuBs fix changes mined from 1,000 popular open-source Java projects in GitHub. They also classified fixes into 16 bug templates, such as "change identifier used", "change caller in function call" and "overload method more args". Since deep learning based bug detection methods show high performance on traditional complex vulnerabilities, the question arises *whether these methods can be used to detect SStuBs and achieve promising results.*

This Work. In this paper, we seek to empirically analyze the performance of deep learning based bug detection techniques on locating SStuBs. Specifically, we make a huge effort to re-implement two state-of-the-art techniques, VulDeePecker [4] and SySeVR [6] for analyzing Java source code. Note that these two techniques are originally designed for C/C++ code, and VulDeePecker [4] is not open source. We implemented these two detectors in approximately 3,000 lines of code to demonstrate their performance on Java SStuBs bugs. VulDeePecker [4] considers the data dependency of program and performs program slicing based on key library/API function calls. It then assembles the program slices obtained into code gadgets for training and detecting. SySeVR [6] extracts both data dependency and control dependency information extending from vulnerability syntax characteristics for model training and detecting. These two deep learning based techniques are considered in this paper because they perform excellently in bug detection tasks and they have a finer granularity (i.e., at program slice level) when pinpointing bugs, which is more practical in real world scenarios.

To compare the effectiveness of these two approaches on detecting both complex Java code vulnerabilities and SStuBs, we first built benchmarks for two representative vulnerabilities (i.e., CWE-22 and CWE-79), and then applied the two deep learning based methods to them. Both methods can achieve a detection accuracy over 90%, which suggests that our re-implementation is correct and traditional complex vulnerabilities like CWE-22 and CWE-79 can be well flagged by these two deep learning based methods. As a comparison, we next applied these two deep learning based techniques to the ManySStuBs4J dataset, where the detection accuracy is only around 70% for VulDeePecker detector and 66% for SySeVR detector. Although these two approaches can achieve much better results than existing static analysis tools, the results showed that SStuBs cannot be well flagged comparing to traditional complex vulnerabilities. We further analyze the results for the 16 types of bugs, finding that the effectiveness of deep vulnerability detectors vary among different types of SStuBs. The detecting accuracy of some types of SStuBs is obviously higher than other bug types. This is mainly because

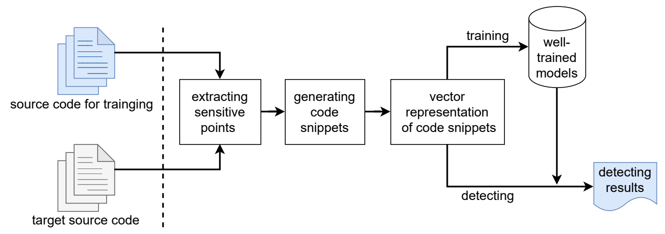


Fig. 1. Overview of the deep learning based bug detection.

some kinds of bugs only involve minor changes, which have little relationship with data flow and control flow information. Our findings suggest that different kinds of approaches should be combined together for better detecting SStuBs. We have released our crafted benchmark and experiment results to the research community at:

https://doi.org/10.5281/zenodo.4609689

II. Deep Vulnerability Detectors

We first introduce the overview of the methodology we have implemented to detecting SStuBs. Then we illustrate the process step by step with a real example.

A. Overview

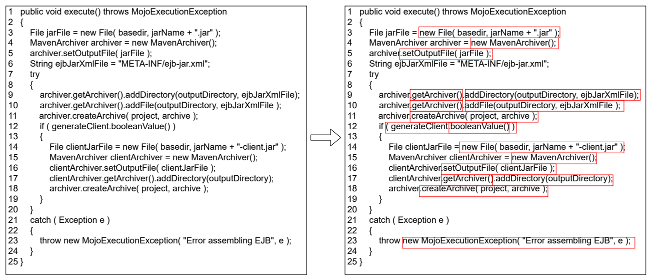
As shown in Fig.1, deep vulnerability detectors in general consist of two phases, *training phase* and *detecting phase*. The inputs to training phase are the source code with ground truth information (i.e., vulnerable and correct). After feature extraction, the source code is represented in vectors and used to train bug detection models. In detecting phase, unknown source code is transferred into vectors through the same steps as training phase. The output of detecting phase is “correct” or “vulnerable” for each code snippet.

B. Training phase

Training phase consists of four main steps, including 1) locating sensitive points, 2) code snippets generation and labeling, 3) vector representation and 4) model training.

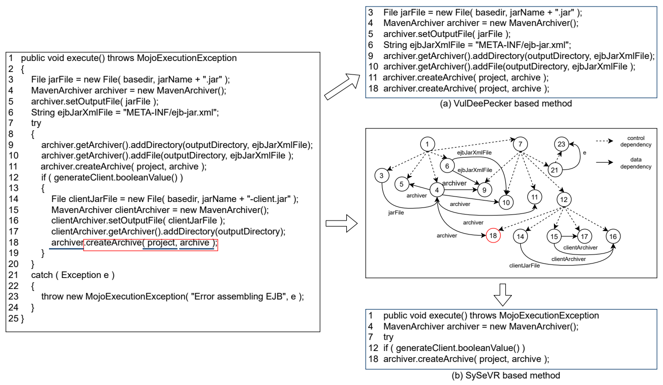
1) Locating Sensitive Points: Sensitive point is the syntax characteristics where most simple stupid bugs manifest, which is similar as “key point” mentioned in VulDeePecker [4] and “SyVCs” defined in SySeVR [6]. Here we choose the flowing syntax characteristics as sensitive points: object construction, method invocation, expression statement, conditional statement and loop statement. In this step, we first create abstract syntax trees (ASTs) of each source code files and then extract all sensitive points. Fig. 2 shows an example of a simple SStuB. The developer wants to build `ejb` client in line 18 but build `ejb` twice by error. The sensitive points we extracted from the code snippet are highlighted by boxes.

2) Code Snippets Generation and Labeling: A code snippet consists of a number of semantically related lines of code. As aforementioned, we have implemented two different detectors to generate code snippets, the VulDeePecker detector and SySeVR detector. (1) *VulDeePecker detector*. For each sensitive point, we trace the backward data flow of corresponding



```
1  public void execute() throws MojoExecutionException
2  {
3      File jarFile = new File( basedir, jarName + ".jar" );
4      MavenArchiver archiver = new MavenArchiver();
5      archiver.setOutputFile( jarFile );
6      String ejbJarXmlFile = "META-INF/ejb-jar.xml";
7      try
8      {
9          archiver.getArchiver().addDirectory(outputDirectory, ejbJarXmlFile);
10         archiver.getArchiver().addFile(outputDirectory, ejbJarXmlFile );
11         archiver.createArchive( project, archive );
12         if ( generateClient.booleanValue() )
13         {
14             File clientJarFile = new File( basedir, jarName + "-client.jar" );
15             MavenArchiver clientArchiver = new MavenArchiver();
16             clientArchiver.setOutputFile( clientJarFile );
17             clientArchiver.getArchiver().addDirectory(outputDirectory);
18             archiver.createArchive( project, archive );
19         }
20     }
21     catch ( Exception e )
22     {
23         throw new MojoExecutionException( "Error assembling EJB", e );
24     }
25 }
```

Fig. 2. An example of locating sensitive points.



```
1  public void execute() throws MojoExecutionException
2  {
3      File jarFile = new File( basedir, jarName + ".jar" );
4      MavenArchiver archiver = new MavenArchiver();
5      archiver.setOutputFile( jarFile );
6      String ejbJarXmlFile = "META-INF/ejb-jar.xml";
7      try
8      {
9          archiver.getArchiver().addDirectory(outputDirectory, ejbJarXmlFile);
10         archiver.getArchiver().addFile(outputDirectory, ejbJarXmlFile );
11         archiver.createArchive( project, archive );
12         if ( generateClient.booleanValue() )
13         {
14             File clientJarFile = new File( basedir, jarName + "-client.jar" );
15             MavenArchiver clientArchiver = new MavenArchiver();
16             clientArchiver.setOutputFile( clientJarFile );
17             clientArchiver.getArchiver().addDirectory(outputDirectory);
18             archiver.createArchive( project, archive );
19         }
20     }
21     catch ( Exception e )
22     {
23         throw new MojoExecutionException( "Error assembling EJB", e );
24     }
25 }
```

```
3  File jarFile = new File( basedir, jarName + ".jar" );
4  MavenArchiver archiver = new MavenArchiver();
5  archiver.setOutputFile( jarFile );
6  String ejbJarXmlFile = "META-INF/ejb-jar.xml";
9  archiver.getArchiver().addDirectory(outputDirectory, ejbJarXmlFile);
10 archiver.getArchiver().addFile(outputDirectory, ejbJarXmlFile );
11 archiver.createArchive( project, archive );
18 archiver.createArchive( project, archive );
```

(a) VulDeePecker based method

```
1  public void execute() throws MojoExecutionException
4  MavenArchiver archiver = new MavenArchiver();
7  try
12 if ( generateClient.booleanValue() )
18 archiver.createArchive( project, archive );
```

(b) SySeVR based method

Fig. 3. An example of code snippets generation.

identifiers (e.g., operand, arguments and caller of a method), and assemble the lines of code obtained into code snippets. As shown in Fig. 3, method invocation in line 18 is a sensitive point, and “archiver”, “project” and “archive” are three corresponding identifiers. We get the data flow slices of the three identifiers and assemble them according to the order of the statements’ appearance in code. The final code snippet is shown in Fig.3 (a). (2) *SySeVR detector*. For each sensitive point, we first generate program dependency graph (PDG) for each method, then generate the inter-procedural backward and forward slice, finally assemble the corresponding code of lines into code snippets. For example, in Fig. 3, sensitive point in line 18 has data dependency on line 4 and control dependency on line 12, line 4 and line 12 further have control dependency on line 1 and line 7, separately. Therefore the code snippet consists of line 1,4,7,12,18, as shown in Fig. 3 (b).

3) Vector Representation: We use the word2vec [12] to transform code tokens into vectors. A code snippet will be divided into a sequence of tokens and then transformed into integers using a well-trained word2vec model. Since deep learning models usually take equal-length vectors as input, the vector representation of code snippets need some adjustment. Let l denotes the length of vectors that we should input to the models. For vectors that are shorter than l, we pad zeros in the beginning of the vector. For vectors that are longer than l, we delete the begin part of the vector.

4) Model Training: The bug detection task can be formulated as a binary classification problem. We select Bidirectional Long Short-Term Memory (BLSTM) as detecting model because it can catch the information of both earlier statements and later statements in the program which may affect sensitive points [4]. The input of the training process is the vectors of length l and ground truth. Our model consists of an embedding layer, a BLSTM layer, a dense layer and a softmax layer. For each bug type, we randomly choose 80% code snippets as training dataset, 20% code snippets for evaluation.

C. Detecting phase

Similar to the training phase, given a target source code, we first extract sensitive points, then generate code snippets for each sensitive point. Next, we transform code snippets into vectors and input them to the trained BLSTM model. The output of the model is "0" (the code snippet does not have SStuB) or "1" (the code snippet has SStuB).

III. Experiments and Results

A. Study Design

1) Research Questions: Our evaluation is driven by the following research questions (RQs):

RQ1 *What is the performance of deep learning based methods on detecting traditional complex Java vulnerabilities and SStuBs?*

RQ2 *Do the deep learning based methods achieve consistent performance on different types of SStuBs?*

2) Datasets: To compare the effectiveness of these two approaches on detecting both complex vulnerabilities and SStuBs, we use three large-scale datasets, including CWE-22, CWE-79 and ManySStuBs4J [11] for evaluation. First, for the traditional complex vulnerabilities, We make effort to craft a benchmark dataset of 3,776 samples for CWE-22 Path Traversal and a dataset of 4,827 samples for CWE-79 XSS (Cross-Site Scripting) from SARD [13] and OWASP [14]. We choose CWE-22 and CWE-79 mainly because they are common vulnerabilities and have more available samples. In each item of the datasets, the buggy or fixing line number, bug type, and the path of the source codes are provided. Following previous work[4, 6], we choose the invocation of file reading and writing methods as sensitive points for CWE-22 and invocation of method that sending information to client as sensitive points for CWE-79. Furthermore, we evaluate the performance of detecting SStuBs basing on the ManySStuBs4J dataset [11]. In each item of the dataset, it provides the line in which the bug exists in the buggy version of the file, the hash of the commit fixing the bug and the hash of the last commit containing the bug. We harvest related source code files according to the commit hashes.

For both datasets, we deem the code snippets generated from the source code lines before fixing as "1" (has bug), after fixing as "0" (no bug). To ensure the correctness of samples, we only saved code snippets generated from sensitive points of which the line numbers are "bug line num" or "fix line num". For CWE-22 and CWE-79, we got over 4,400 and 5,700 code snippets, respectively. The proportion of positive and negative samples is around 1:2. For ManySStuBs4J, We finally got 61,667 code snippets for VulDeePecker based method and 68,768 code snippets for SySeVR based method with

TABLE I
Evaluation Results on Traditional Complex Vulnerabilities VS. SStuBs

dataset	metrics	VulDeePecker based	SySeVR based
CWE-22	FPR	6%	3%
	P	91%	94%
	FNR	2%	12%
	ACC	95%	94%
CWE-79	FPR	4%	3%
	P	90%	94%
	FNR	7%	1%
	ACC	95%	98%
ManySStuBs4J	FPR	31%	33%
	P	70%	66%
	FNR	28%	38%
	ACC	71%	65%

a proportion of around 1:1 between positive and negative samples. The test sets (roughly 20% of the dataset) consist of 12,338 code snippets for VulDeePecker based method and 13,761 code snippets for SySeVR based method.

3) Model Training: Overall, we have trained six different models on the three datasets. For each dataset, we have trained two models for the two deep learning based techniques. The length of input vectors l is set to 50. The hidden size, dropout and recurrent dropout of BLSTM layer are set to 64, 0.5 and 0.5, respectively. The binary crossentropy loss and ADAMAX with default parameters are used for training. The batch size is 64 and the number of epochs is 50.

4) Metrics: We use four widely used metrics including accuracy (ACC), false positive rate (FPR), false negative rate (FNR), and precision (P) to evaluate the performance of bug detection. Let TP be the number of samples with bugs that are detected correctly, FP be the number of samples without bugs while are detected as vulnerable, TN be the number of clean samples that are detected correctly and FN be the number of clean samples that are detected as vulnerable. The ACC measures the correctness of all detected samples and can be denote as $ACC = \frac{TP+TN}{TP+FP+TN+FN}$. The FPR means proportion of false-positive samples in the total samples that are not vulnerable and can be calculate by $FPR = \frac{FP}{FP+TN}$. $FNR = \frac{FN}{TP+FN}$, means the proportion of false-negative samples in the total samples that are vulnerable. P measures the correctness of detected vulnerable samples and $P = \frac{TP}{TP+FN}$.

B. RQ1: Traditional complex vulnerabilities VS. SStuBs

Performance on Complex Vulnerabilities. Table I shows the overall results. Obviously, both methods can achieve a detection accuracy of 95% with relatively low FPR and FNR. This result is inline with previous studies on C/C++ vulnerabilities [4, 6], which suggests that our re-implementation of these approaches in Java is accurate, and these two deep vulnerability detectors are able to well flag traditional complex vulnerabilities like CWE-22 and CWE-79.

Performance on SStuBs The result is shown in Tab.I. The VulDeePecker based method achieves a P of 70% and ACC of 71%, which is obviously lower than the performance of detecting traditional vulnerabilities. The SySeVR based method has 66% precision detecting SStuBs, and incurs a FPR of 33% and FNR of 38%. Previous work also measured the proportion of bugs in ManySStuBs4J that can be identified by popular static analysis tools such as SpotBugs [1], Error Prone [15] and Infer [16]. The overall bug detection rate of all three bug detectors together on their studied bugs is only 4.5% [17]. In another work, researches find that SpotBugs could only locate about 12% of SStuBs while also reporting more than 200 million possible false positives. The results show that deep-learning based method may not perform as well as traditional complex vulnerabilities in detecting simple bugs like SStuBs, but can outperform existing static analysis tools. Moreover, the VulDeePecker based method performs slightly better than SySeVR based method, which suggests that data flow information is more sensitive than control-flow information to detecting SStuBs.

C. RQ2: Performance on different types of SStuBs

We further look in detail on a per-category basis, which is shown in Tab. II (The bug types that can be better detected are shown in bold). Note that bug types "DELETE THROWS EXCEPTION" and "ADD THROWS EXCEPTION" are not shown in the table for the sample sizes are small and the sensitive points we defined cannot cover all syntax characteristics of these two bug types. Interestingly, we find that the effectiveness of our detecting methods vary among different types of SStuBs. Both methods are better at detecting "CHANGE OPERAND", "CHANGE IDENTIFIER", "CHANGE CALLER IN FUNCTION CALL", "DIFFERENT METHOD SAME ARGS", "MORE SPECIFIC IF" and "LESS SPECIFIC IF" bugs, which can reach a precision of roughly 80% with FNR less than 20%. This is because the sensitive point related identifiers (e.g, method caller) are changed before and after fixing, thus leading more differences in code snippets. We also notice that similar bugs may appear many times in one project, and there are some patterns in bug repair (e.g, using LinkedHashMap instead of HashMap), thus leading to better detecting performance. The performance on type "OVERLOAD METHOD MORE ARGS" and "OVERLOAD METHOD DELETED ARGS" are not as good as our anticipation. We look into the dataset and find that many arguments added or deleted are string, "null" value, Boolean value and number that are hard to handle by our methods. Also, bug types, such as "CHANGE OPERATOR" and "CHANGE NUMERAL" encounter similar problem. The rest kinds of bugs, "SWAP ARGUMENTS", cannot be well detected mainly because the code snippets before and after fixing are almost same, for the sensitive point related identifiers would not be changed by the order of occurrence.

IV. Related Work

A number of static analysis tools and research works have been proposed to detect software vulnerabilities. These tools and research works can be divided into two main categories. The first category is traditional static analyzers which detecting vulnerabilities based on predefined patterns, such as SpotBugs [1] and PMD [2]. But these traditional tools often

TABLE II
Performance on Detecting Different Types of SStuBs

Bug Type	**VulDeePecker based**				**SySeVR based**			
	FPR	P	FNR	ACC	FPR	P	FNR	ACC
CHANGE_OPERATOR	67%	36%	66%	34%	55%	35%	74%	35%
CHANGE_OPERAND	14%	87%	9%	88%	20%	80%	24%	78%
CHANGE_IDENTIFIER	25%	75%	22%	77%	28%	72%	33%	70%
CHANGE_MODIFIER	68%	12%	89%	22%	61%	19%	78%	32%
CHANGE_NUMERAL	58%	43%	53%	44%	53%	46%	59%	44%
CHANGE_CALLER_IN_FUNCTION_CALL	10%	89%	17%	87%	18%	80%	19%	81%
CHANGE_UNARY_OPERATOR	53%	50%	43%	52%	51%	50%	51%	49%
OVERLOAD_METHOD_MORE_ARGS	46%	58%	36%	59%	47%	50%	49%	52%
OVERLOAD_METHOD_DELETED_ARGS	58%	52%	48%	48%	47%	51%	57%	48%
DIFFERENT_METHOD_SAME_ARGS	19%	81%	15%	83%	20%	79%	23%	78%
MORE_SPECIFIC_IF	20%	78%	25%	77%	24%	77%	32%	72%
LESS_SPECIFIC_IF	14%	86%	19%	84%	28%	73%	25%	74%
SWAP_ARGUMENTS	54%	51%	38%	54%	36%	56%	56%	54%
SWAP_BOOLEAN_LITERAL	86%	24%	70%	22%	67%	27%	76%	29%

incur high false positive rate or false negative rate. The second category is machine learning based approaches. Some approaches detecting vulnerabilities according to code similarity, which obtaining abstract representations of code fragments and comparing the similarity between pairs of the representations [18, 19, 20]. There are also many approaches to detecting well-defined vulnerabilities using machine learning techniques. For example, Yan *et al.* [21] introduced a static use-After-free detector that bridges the gap between typestate and pointer analyses by a Support Vector Machine. Li *et al.* [4] designed VulDeePecker, embedding codes using data flow information to detecting resource management errors and buffer overflows. Moreover, some researches also pay attention to simple bugs like simple one-statement bugs. Pradel *et al.* [22] presented DeepBugs, a learning approach to name-based bug detection. They focused on three kinds of bugs, including accidentally swapped function arguments, incorrect binary operators, and incorrect operands in binary operations.

V. Conclusion and Discussion

In this paper, we empirically analyzed the effectiveness of deep learning based bug detection techniques on locating SStuBs. The experimental results on ManySStubs4J show that the effectiveness of detecting some types of SStuBs is obviously better than others. One main reason why other kinds of bugs cannot be well flagged is that those bugs only involve tiny changes on operator, Boolean value, string and so on, which are hard to trace corresponding semantic information by the implemented methods that rely on mainly control-flow and data-flow information. To better detect other types of SStuBs, it may be helpful to analyze code intent or function from source codes and annotations. Moreover, at present we just directly choose all aforementioned syntax characteristics as sensitive points, which probably introduce some useless information. Refining the sensitive points, e.g, identifying what kinds of functions are more likely to incur SStuBs, might be a useful way for improvement. Our findings suggest that different kinds of approaches should be combined together for better detecting SStuBs, while our study offers practical implications on this direction.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (grant numbers 62072046 and 61702045). Haoyu Wang is the Corresponding author.

References

[1] "spotbugs," https://spotbugs.github.io/.

[2] "pmd," https://pmd.github.io/.

[3] "checkstyle," http://checkstyle.sourceforge.io/.

[4] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *Proceedings 2018 Network and Distributed System Security Symposium(NDSS)*, 2018. [Online]. Available: http://dx.doi.org/10.14722/ndss.2018.23158

[5] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui, "Static detection of control-flow-related vulnerabilities using graph embedding," in *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2019, pp. 41–50.

[6] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *arXiv preprint arXiv:1807.06756*, 2018.

[7] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2021.

[8] "Flawfinder," https://dwheeler.com/flawfinder/.

[9] "Rats," https://code.google.com/archive/p/rough-auditing-tool-for-security/.

[10] "Checkmarx," https://www.checkmarx.com/, 2017.

[11] R.-M. Karampatsis and C. Sutton, "How often do single-statement bugs occur? the manysstubs4j dataset," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2020.

[12] "word2vec," http://radimrehurek.com/gensim/models/word2vec.html.

[13] "Software assurance reference dataset," https://samate.nist.gov/SARD/index.php, 2017.

[14] "Owasp benchmark," https://owasp.org/www-project-benchmark/.

[15] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, "Building useful program analysis tools using an extensible java compiler," in *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation(SCAM)*. USA: IEEE Computer Society, 2012, p. 14–23. [Online]. Available: https://doi.org/10.1109/SCAM.2012.28

[16] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NASA Formal Methods*. Springer International Publishing, 2015, pp. 3–11.

[17] A. Habib and M. Pradel, "How many of all bugs do we find? a study of static bug detectors," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering(ASE)*. New York, NY, USA: Association for Computing Machinery, 2018, p. 317–328. [Online]. Available: https://doi.org/10.1145/3238147.3238213

[18] J. Jang, D. Brumley, and A. Agrawal, "Redebug: Finding unpatched code clones in entire os distributions," in *2012 IEEE Symposium on Security and Privacy(SP)*. IEEE Computer Society, may 2012, pp. 48–62. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP.2012.13

[19] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 595–614.

[20] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering(ICSE)*. Association for Computing Machinery, 2016, p. 1157–1168. [Online]. Available: https://doi.org/10.1145/2884781.2884877

[21] H. Yan, Y. Sui, S. Chen, and J. Xue, "Machine-learning-guided typestate analysis for static use-after-free detection," in *Proceedings of the 33rd Annual Computer Security Applications Conference(ACSAC)*. Association for Computing Machinery, 2017, p. 42–54. [Online]. Available: https://doi.org/10.1145/3134600.3134620

[22] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276517