Detecting Integer Overflow Errors in Java Source Code via Machine Learning

Yu Luo
Computer Science Electrical Engineering
University of Missouri-Kansas City
Kansas City, USA
ylzqn@mail.umkc.edu

Weifeng Xu
School of Criminal Justice
The University of Baltimore
Baltimore, USA
wxu@ubalt.edu

Dianxiang Xu
Computer Science Electrical Engineering
University of Missouri-Kansas City
Kansas City, USA
dxu@umkc.edu

***Abstract*—Integer overflow is a common cause of software failure and security vulnerability. Existing approaches to detecting integer overflow errors rely on traditional static code analysis and dynamic testing. This paper presents a novel machine learning-based approach that predicts integer overflow errors by treating source code as text. It exploits text classifiers to determine whether each method in a given Java program contains an integer overflow error. As the training data is essential, we have constructed a comprehensive dataset to accounts for (a) integer overflow errors of all integer types and operations in Java (i.e., positive samples); (b) various programming techniques for preventing integer overflow errors (i.e., negative samples); and (c) malicious scenarios that may mislead text classifiers (i.e., adversarial samples). We have trained three classifiers, BERT, fastText, and NBSVM, that represent different text embedding techniques. BERT, as a representative deep-learning transformer, has achieved the highest performance scores and remained robust even when tested with the adversarial samples.**

***Index Terms*—integer overflow, machine learning, static code analysis, text classification, BERT**

I. Introduction

Unanticipated arithmetic overflow is a common cause of software failure and security vulnerability. This paper presents a machine-learning approach for detecting integer overflow errors through automated classification of Java source code. It assigns "positive" (existence of integer overflow) and "negative" (absence of integer overflow) tags to each method in the given source code.

The contributions of this paper are twofold. First, this paper is the first to exploit machine learning-based text classification to detect integer overflow errors. We have evaluated and compared three state-of-the-art text embedding techniques represented by BERT, fastText, and NBSVM. BERT (Bidirectional Encoder Representation from Transformer) is a recent NLP language model from Google AI Language [1]. fastText is an efficient text classifier created by Facebook's AI Research lab [2]. NBSVM is a text classifier that integrates Support Vector Machines (SVM) with Naïve Bayes (NB) features [10]. They deal with input sentences (i.e., Java methods in this paper) in different ways. BERT considers bi-directional contexts of words. fastText treats a sentence as a bag of n-grams where word order is observed. NBSVM transforms sentences in terms of word frequency. The results show that BERT outperforms fastText and NBSVM. Second, this paper offers a comprehensive dataset that accounts for integer overflow errors of all integer types and operations in Java programs (i.e., positive samples), various programming techniques for preventing integer overflow errors (negative samples), and malicious scenarios that may be used to trick text classifiers (i.e., adversarial samples).

The remainder of this paper is organized as follows. Section II introduces integer overflow errors in Java programs. Section III presents the overall approach; Section IV introduces the baseline dataset and experiment; Section V presents the complete dataset; Section VI evaluates and compares the three classifiers; Section VII reviews related work; Section VIII concludes this paper.

This work was supported in part by US NSF under grant 1820685.

II. Integer Overflow in Java Programs

The integer data types in Java include $byte$, $short$, int, $long$, and $char$ (16-bit unsigned). An integer overflow occurs when an integer operation evaluates to a value that is either greater than the maximum or less than the minimum representable value, i.e., out of the range of the underlying integer type. In this case, Java's built-in integer operators silently wrap the result, which leads to an incorrect computation and unanticipated outcome. Overflow errors are typically introduced by the use of a wrong integer type or inappropriate assumption about the operands' ranges.

Integer overflow happens to binary operations and unary operations because the ranges of each integer type are not symmetric. For an integer type other than $char$, the minimum value's negation is one more than the maximum value. Therefore, unary negation overflows when applied to the minimum value. Even the $java.lang.math.abs()$ method can overflow if used to obtain the absolute value of a minimum number.

Not all integer operators are relevant to integer overflow errors. The operators subject to integer overflow include +, -, *, /, ++, –, +=, -=, *=, /=, unary -. The other operators, such as %, %=, $<$, $>$, >=, <=, ==, !=, and unary +, are usually overflow-free. They may indirectly contribute to the occurrence of integer overflow in another expression. Some integer operators are overloaded with other data types. For example, the + operator for string concatenation does not involve integer overflow. However, a machine learning algorithm may incorrectly treat it as an integer operator.

Effective software development should prevent potential integer overflow errors from the production code before it is tested or verified. Systematic prevention requires careful program design and good coding practices. The primary methods for Java programming are precondition test, built-in safe methods ($Math.*Exact()$ in Java 8), upcasting, and BigInteger [5].

III. Detection of Integer Overflows

Given the source code of a Java program, we first filter out the methods without integer operations because they are always free from integer overflows. For each method with integer operations, we aim to classify it either positive (i.e., the existence of integer overflow) or negative (i.e., absence of integer overflow). To do so, we convert it to a text string, like a natural language paragraph. As text classifiers ignore special symbols, including integer operators, we replace them with predefined words (e.g., SYMZPLUS for '+').

Words (e.g., class names, method names, variable names) in Java source code are different from those in natural language texts. A word not contained in the classifier's built-in dictionary will be broken into multiple tokens. For example, the tokenization of 'fsReader' may result in four tokens: 'f', '##sr', '##ead' and '##er', where '##' indicates this token is connected to the previous token. The tokenizer checks whether 'fsReader' is in the dictionary. If not, the last letter 'r' is removed and the tokenizer checks whether 'fsReade' is in the dictionary until it finds 'f' is in the dictionary, and 'f' is extracted as the first token. The tokenizer repeats the above process until all four tokens are extracted.

Text classifiers typically limit the input to 512 tokens for balancing performance and memory consumption. When the input is too long, it is not easy to learn the relationship from the first word to the last. It will also cause a memory overload. In this work, the limit of 512 tokens has caused no problem. No method of the real-world Java projects in Section VII exceeds the limit. If a method under evaluation has more than 512 tokens, we break it into multiple inputs. It is positive if one of the inputs is classified as positive. Section IV will discuss how to deal with lengthy methods with known integer overflow errors (i.e., positive samples for training purposes) because we need to include all lines of the flawed code in the samples.

Any text classifier can be adopted in our approach. The main research issues are creating a comprehensive dataset and finding a high-performing classifier. This paper focuses on three state-of-the-art text classifiers, BERT [1], fastText [2], and NBSVM [10]. They represent different text-embedding techniques for dealing with input sentences: BERT considers bi-directional contexts of words. fastText treats a sentence as a bag of n-grams where word order is observed. NBSVM transforms input sentences in terms of word frequency.

IV. Baseline Dataset and Experiment

This work builds upon the IARPA STONESOUP3.0 dataset, part of the NIST SARD (Software Assurance Reference Dataset) suites [7] for testing static analysis tools with seeded security flaws. As demonstrated below, the STONESOUP's integer overflow samples are very limited. We refer to it as the baseline dataset.

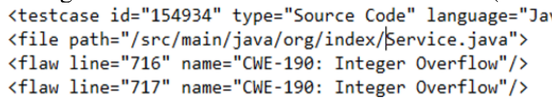
A. Baseline Dataset

Table I lists the Java applications in the STONESOUP dataset. There are 2,711 seeded vulnerabilities, including 93 integer overflow errors. The dataset accounts for 46 CWE (Common Weakness Enumeration) vulnerability types.

Table I
Java Programs in the STONESOUP Dataset

Subject	KLOC	Files	Vul.	Int. Overflow
Coffee MUD	54	4,475	478	17
Elastic Search	36	4,836	478	20
Apache Jena	41	10,700	476	14
Apache JMeter	11	1,921	160	6
Apache Lucene	45	4,190	480	14
Apache POI	33	7,916	479	16
JTree	1	123	160	6
Total	221	34,161	2,711	93

The dataset provides an XML file that contains the information on each vulnerability, including the test case id, the source code file with the flaw, and the lines of the code where the vulnerability is located. The following is an example where the integer overflow code consists of two lines (716 and 717):



```
<testcase id="154934" type="Source Code" language="Java">
<file path="/src/main/java/org/index/Service.java">
<flaw line="716" name="CWE-190: Integer Overflow"/>
<flaw line="717" name="CWE-190: Integer Overflow"/>
```

We treat each method with an integer overflow flaw as a positive sample. If it does not exceed 512 tokens, we include its entire source code in the sample. If it has more than 512 tokens, we ensure that all flaw code lines are included in the sample. To do so, we first extract the entire code block of the integer overflow error according to the XML file. If the code block has more than 512 tokens, we take the first 512 tokens for the positive sample; otherwise, we find the enclosing code block with the maximum token size where the integer overflow code is centered. If there is no more text before (or after) the flawed code, we keep adding text on the other side where the text is still available until the maximum is reached.

A negative sample is a method with integer operations but free from integer overflow errors. We exclude those methods that have no integer operations because they will never have integer overflows. For each negative method in the STONESOUP dataset with more than 512 tokens, we break its source code into multiple negative samples in sequential order. Thus, we treat lengthy positive and negative samples differently. However, no underfitting or overfitting was found in our experiments.

The Java programs in the STONESOUP dataset share many classes and methods. We have excluded duplicate code when converting the negative samples. In brief, the baseline dataset consists of 93 positive and 5,032 negative samples.

B. Baseline Experiment

The baseline experiment aimed to train the classifiers with the baseline dataset, evaluate their performance, and analyze the baseline dataset's limitations. It is implemented in Python Ktrain and Tensorflow and performed on Google Colab Pro with Tesla P100-PCIE GPU and 27.4 G high RAM. We build the models using ktrain v0.25.x. We train BERT to fine-tune related parameters and add a softmax layer for prediction. For fastText and NBSVM, we use the default training parameters. We set the learning rate to 2e-5 and epoch to 40 for all models.

We applied 10-fold cross-validation to the baseline dataset – 90% for training and 10% for testing. The initial epoch is 40. If the accuracy and loss values are satisfactory in the 40th epoch, the training terminates; otherwise, it will continue.

Table II
RESULT OF BASELINE EXPERIMENT

Model	Accuracy	Precision	Recall	F1	AUC
BERT	1.0000	1.0000	1.0000	1.0000	1.0000
fastText	0.9924	0.9962	0.8000	0.8731	1.0000
NBSVM	1.0000	1.0000	1.0000	1.0000	1.0000

Table II presents the experiment results. Both BERT and NBSVM have achieved a perfect score on all performance indicators, i.e., accuracy, precision, recall, F1, and AUC. AUC stands for the Area Under the ROC (receiver operating characteristic) Curve, a performance measure of classification models. fastText's scores are also very high.

Although the prediction results are correct, they are biased because certain words overexpose the trained model. Many words with positive correlations are not the right features. They have weighted out or even ignored other important features. A thorough review of all 93 positive samples reveals a set of common words that appear in every positive sample, such as 'trigger,' 'tracepointmessage,' 'tracer,' 'before,' and 'after.' Obviously, such an artificial pattern was introduced into the source code when the vulnerabilities are seeded into the original Java programs. These words do not appear in negative examples. It appears that BERT can quickly learn the features that accurately separate positive samples from negative ones. The same variable names used in all 93 positive samples, such as 'stonesoup_checked_value' and 'stonesoup_value,' also mislead the model to consider critical features.

C. Limitations of the Baseline Dataset

The integer overflow errors (and other vulnerabilities) in the STONESOUP dataset were created by deliberate design with easily observable features. The artificial patterns are useful for tracking and understanding the inserted flaws. As they can be sorted out accurately by text classifiers, the baseline dataset is meaningful for evaluating a vulnerability detection technique only if the artificial patterns do not contribute to the prediction result. We address this issue by creating negative samples with the same patterns of positive examples.

Another limitation is that the integer overflow samples have only covered three addition operators (+, +=, ++) of one integer type (i.e., short). No negative sample accounts for the prevention of integer overflows. As discussed in Section II, various integer types and operators are subject to integer overflows, and there are several methods for preventing integer overflow errors. Therefore, the training dataset must cover all of them to build effective prediction models.

Moreover, the high-performance scores in Table II do not necessarily indicate the classifiers are good at predicting integer overflow errors in real-world Java programs. To validate this hypothesis, we tested the classifiers with 35 positive and 35 negative samples. The positive samples are collected from the bug history of OpenJML, apache-tomcat-6.0.32, and ical4j-develop and created according to various integer operators in Section II. For each positive sample, a negative sample was created using the prevent techniques in Section II.

Table III
RESULTS OF BASELINE EXPERIMENT I ON DIVERSE SAMPLES

Model	Accuracy	Precision	Recall	F1	AUC
BERT	0.3857	0.2177	0.3857	0.2784	0.4727
fastText	0.4857	0.2464	0.4857	0.3269	0.0008
NBSVM	0.4000	0.2222	0.4000	0.2857	0.6751

Table III presents the testing results. All classifiers trained with the baseline dataset are much worse than a random guess. It is because the baseline dataset does not represent the typical integer overflows in real-world applications. This paper aims at an effective classifier with a comprehensive dataset.

V. BUILDING THE DATASET

High-quality datasets are essential to effective machine learning. This section extends the baseline dataset to account for all types of integer overflow errors (normal positive samples), prevention methods (normal negative samples), and various malicious code that may mislead the classifiers (i.e., adversarial samples). Inspired by contrastive learning, we create negative samples by applying a prevention technique to fix the error in each positive sample if feasible. Therefore, our dataset consists of many pairs of contrastive samples. Unlike contrastive learning that only uses contrastive samples, our dataset includes the 5,032 negative samples in the baseline dataset. They represent widely applied Java programs of various integer types and operations without overflow errors. A comparative experiment will be presented in Section VI-C.

A. Normal Positive Samples

Normal positive samples are created as follows:

- Refactor the 93 STONESOUP samples by changing variable names, modifying arithmetic expressions, and adding more comments (called STONESOUP refactorings).
- Collect integer overflow samples from textbooks, websites, and bug histories of open-source projects.

- Insert overflow code segments into negative samples.
- Write new code to cover all integer types and operations that may cause overflows, as described in Section II.

For the last two methods, we maintain a set of method/variable names, a set of overflow-free code segments in addition to the sets of integer types, operators subject to integer overflows, method modifiers, and return types. We also strive to avoid using similar variable names. Table IV shows the distributions of positive examples classified by integer operators. The total number is 757.

Table IV
NORMAL POSITIVE SAMPLES

Category	#Positive Samples
STONESOUP Refactorings	93
+ (+, +=, ++)	165
- (-, -=, –)	155
* (*, *=)	160
/ (/, /=)	50
unary -	30
absolute value	30
other operators	74
Total	757

B. Normal Negative Samples

We created 702 negative samples from the existing positive samples using the prevention techniques to fix each overflow error if feasible. It will reduce the bias of certain syntactic features because they will appear in both positive and negative samples. It was inspired by contrastive learning that allows training models to learn the distinctiveness of positive and negative features. The distinctiveness is achieved by pairing a positive with one or more negatives.

C. Adversarial Samples

We created a total of 211 adversarial samples (100 positives and 111 negatives) after the classifiers have been trained with the normal samples (i.e., Dataset I in the next section). From the normal samples, the classifiers have identified critical features that contribute to their prediction. For example, many negative samples share words, such as "ArithmeticException", "BigInteger", and "Math", representing the prevention of integer overflow errors. Example words that contribute to positive samples are "stonesoup_checked_value", "tracepoint-VariableShort", and "trigger-point".

An adversarial positive sample is a flawed method with the features that usually appear in negative samples. An adversarial negative sample is an overflow-free method with specific features that usually contribute to positive samples. Training with adversarial samples will allow the classifiers to distinguish between normal and adversarial cases. Although adversarial samples do not represent real-world software, they are essential for measuring machine-learning approaches.

The adversarial samples are created as follows:

- Refactor the existing positive and negative samples by using new variables named after the frequent words (phrases) or replacing original variable names with the top frequent words.
- Insert print statements with top frequent words to both positive and negative samples. For example, 'System.out.println("ArithmeticException is not a key word.");' can be inserted into a positive sample.
- For each positive sample with multiple integer overflows, fix one and keep the others unchanged to create an adversarial positive sample.

VI. EMPIRICAL STUDIES

The complete dataset includes the baseline dataset and the new normal and adversarial samples. It is referred to as Dataset II in Table V. To investigate the impacts of adversarial samples, we refer to the combination of the baseline and the normal samples as Dataset I (i.e., with no adversarial samples).

Table V
DATASETS FOR EMPIRICAL STUDIES

Dataset Name	Postives	Negatives	Total
Baseline dataset	93	5,032	5,125
Dataset I	850	5,827	6,677
Dataset II	950	5,938	6,888

Our studies aim at the following research questions.

- **RQ1**: How well do the classifiers perform on the dataset without the adversarial samples? Are they robust when tested with adversarial samples?
- **RQ2**: How well do the classifiers perform on the dataset with adversarial samples? Are they robust when tested with adversarial samples?

A. Training without Adversarial Samples (Dataset I)

Table VI shows the results of training the classifiers with Dataset I without the adversarial samples. Compared to the baseline experiment in Table II, the new normal and negative samples have slightly reduced the scores. All of the classifiers have performed well, although perfect scores no longer exist. BERT is the best per each performance indicator (accuracy, precision, recall, F1, and AUC). Its scores are all above 98.7%.

Table VI
TRAINING WITHOUT ADVERSARIAL SAMPLES (DATASET I)

Method	Accuracy	Precision	Recall	F1	AUC
BERT	0.9956	0.9871	0.9922	0.9897	0.9955
fastText	0.9648	0.8968	0.9537	0.9224	0.9941
NBSVM	0.9677	0.9005	0.9658	0.9296	0.9924

We further tested the classifiers with 30 positive and 25 negative adversarial samples. Table VII shows the results. All classifiers have suffered a significant performance reduction. The scores range from 50-74%. These scores indicate that the adversarial samples can fool them. BERT has outperformed fastText and NBSVM.

Table VII
Testing the Classifiers with Adversarial Samples (a)

Method	Accuracy	Precision	Recall	F1	AUC
BERT	0.7091	0.7393	0.7233	0.7067	0.6947
fastText	0.6727	0.6917	0.6533	0.6464	0.7947
NBSVM	0.5636	0.5560	0.5533	0.5516	0.5053

B. Training with Adversarial Samples (Dataset II)

Table VIII shows the results of training the classifiers with Dataset II, excluding the above 55 adversarial examples for comparison purposes. The resultant models will be applied to real-world projects in the next section. With the adversarial samples, BERT has slightly improved the performance, with all scores above 99%. fastText and NBSVM have slightly decreased their performance. BERT is still the best per each performance indicator.

Table VIII
Result of Training with Adversarial Samples (Dataset II)

Method	Accuracy	Precision	Recall	F1	AUC
BERT	0.9959	0.9939	0.9903	0.9921	0.9968
fastText	0.9375	0.8603	0.9302	0.8899	0.9826
NBSVM	0.9470	0.8763	0.9468	0.9063	0.9767

We further tested the classifiers with the aforementioned 55 adversarial samples. Table IX shows the results. BERT is robust as its performance scores are all above 97%. fastText and NBSVM still performed poorly even when they have been trained with adversarial samples.

Table IX
Testing the Classifiers with Adversarial Samples (b)

Method	Accuracy	Precision	Recall	F1	AUC
BERT	0.9818	0.9908	0.9833	0.9817	0.9733
fastText	0.6909	0.7221	0.6700	0.6623	0.8360
NBSVM	0.6909	0.7422	0.6667	0.6538	0.6227

C. Summary

To summarize, the main findings are as follows:

- The three classifiers can achieve high performance scores (all over 90%) for normal positive and negative samples.
- fastText and NBSVM can be fooled by adversarial samples no matter whether trained with adversarial samples.
- BERT can be fooled by adversarial samples if not trained with such samples. However, it performs very well if trained with adversarial samples.
- BERT, as a representative of the new generation text-embedding technique for NLP, has always outperformed fastText and NBSVM.

VII. Related Work

Scandariato et al. [8] proposed a vulnerability prediction model for Java projects through text-mining source code and used Naïve Bayes and Random Forest to predict vulnerability. VulDeePecker [4] aims to detect buffer error and resource management error in C/C++ code. It applies Bidirectional Long Short-Term Memory (BLSTM) to API function calls and forward/backward program slices. uVulDeePecker [11] extends VulDeePecker by dealing with 40 vulnerability types. Code attention and its extraction method are used to help pinpoint vulnerability types.

Li et al. [3] labeled each function as sensitive or nonsensitive, vectorized it with the one-hot encoding method, and built the prediction model with a Dense layer, multiple BLSTM layers, and an output layer. Russell et al. [6] explored both CNNs and RNNs for feature extraction from the embedded source representations. A random forest classifier is used to determine if the C/C++ source code contains five types of vulnerabilities. Based on graph learning, FUNDED [9] uses word2vec network to generate node embedding in AST and update each node by analyzing nine types of edges.

Unlike the above work, our approach focuses on detecting integer overflow errors via text classification.

VIII. Conclusions

We have presented the approach for detecting integer overflow errors in Java code via text classification. The evaluations with a comprehensive dataset and real-world applications demonstrate that BERT as a representative deep-learning transformer is a viable solution. It has achieved very high performance scores and found many errors in real-world projects.

References

[1] J. Devlin, M. Chang, K. Lee, and K. Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2018. arXiv: 1810.04805 [cs.CL].

[2] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov. *Bag of Tricks for Efficient Text Classification*. 2016. arXiv: 1607.01759 [cs.CL].

[3] R. Li, C. Feng, X. Zhang, and C. Tang. "A Lightweight Assisted Vulnerability Discovery Method Using Deep Neural Networks". In: *IEEE Access* 7 (2019), pp. 80079–80092.

[4] Z. Li, D. Zou, S. Xu, X. Ou, and Y. Zhong. "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection". In: *Network and Distributed System Security Symposium*. Feb. 2018.

[5] E. Mertikas. *NUM00-J. Detect or prevent integer overflow*. https://wiki.sei.cmu.edu/confluence/display/java/NUM00-J.+Detect+or+prevent+integer+overflow. 2018.

[6] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. "Automated vulnerability detection in source code using deep representation learning". In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE. 2018, pp. 757–762.

[7] SARD. *NIST Software Assurance Reference Dataset Project*. https://samate.nist.gov/SRD/testsuite.php. 2019.

[8] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. "Predicting vulnerable software components via text mining". In: *IEEE Transactions on Software Engineering* 40.10 (2014), pp. 993–1006.

[9] H. Wang, G. Ye, Z. Tang, S.H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang. "Combining Graph-based Learning with Automated Data Collection for Code Vulnerability Detection". In: *IEEE TIFS* 16 (2020), pp. 1943–1958.

[10] S. Wang and C. Manning. "Baselines and bigrams: Simple, good sentiment and topic classification". In: *50th Annual Meeting of the Association for Computational Linguistics, ACL 2012 - Proceedings of the Conference*. 2012, pp. 90–94.

[11] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin. "μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection". In: *IEEE Transactions on Dependable and Secure Computing* (2019).