International Journal on Artificial Intelligence Tools
© World Scientific Publishing Company

# Predicting Integer Overflow Errors via Supervised Learning

Yu Luo
*Computer Science Electrical Engineering*
*University of Missouri-Kansas City, Kansas City, USA*
*ylzqn@umsystem.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*

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

An integer overflow error occurs when an integer operation in computer software evaluates a value out of the integer range. It can lead to a fatal system failure. The existing approaches to detecting integer overflow errors rely on data/control-flow analysis of the code or execution of the code with test cases. This paper presents a supervised learning approach to predicting whether each method in a given Java program has an integer overflow error by treating the source code as text. Built upon real-world programs, our Java dataset covers all integer data types and operations in Java, the methods for preventing integer overflow errors, and adversarial samples. We have evaluated six classification models, BERT, DistilBERT, codeBERT, Code2Vec, fastText, and NBSVM. They represent different text embedding techniques for dealing with source code. The experiment results show that BERT and its variants have outperformed other models. We have applied the resultant BERT model to eleven real-world projects, including JDK13.0 and ten top-ranked GitHub projects, and revealed 181 integer overflow errors. In addition, we have evaluated the classification models with a C/C++ dataset. The result is similar to that of the Java dataset.

*Keywords*: integer overflow; machine learning; static code analysis; text classification; BERT.

## 1. Introduction

Unanticipated arithmetic overflow is a common cause of software failure and security vulnerability. It occurs when an arithmetic operation results in a value outside the range of the pre-defined or assumed integer type. Consider the arithmetic mean of two integer numbers, which is calculated by adding the two numbers and then dividing by two. An integer overflow happens when the sum is too large to be represented. Sample consequences of arithmetic overflow errors include the possible power loss in Boeing 787 reported in 2015 [25], and the lawsuit of the prize ticket of $42,949,672.76 printed by a Casino machine at Resorts World Casino in 2016 although the stated maximum payout was $10,000 [14]. Vulnerabilities due to integer overflows have been reported widely in popular software products, such as OpenOffice, Adobe Flash Player, Adobe Reader, RealPlayer, QuickTime Player, and Microsoft Linker [44].

While the concept of integer overflow seems straightforward, and it is not difficult to identify integer operations that potentially overflow, it can be hard to automatically determine "which potential overflows are true errors and which are intended by the programmer" [20]. Beyond industry tools that provide a partial solution to the problem, many research publications fall into two broad categories: static analysis and dynamic testing. They usually rely on code semantics (e.g., data flow and control flow) or functional specification. As a unique static analysis technique, machine learning-based detection of code defects and vulnerabilities has recently gained increasing attention. The existing machine learning methods, however, have not yet explicitly targeted integer overflow. Other related work on static analysis and dynamic testing has focused on C/C++ programs.

This paper aims at the prediction of integer overflow errors in Java source code, although the proposed approach applies to other languages. In Java, various integer types and operators are subject to integer overflow. There are also different methods for preventing integer overflow. Existing static code analysis tools such as SpotBugs (evolved from FindBugs) [40], Coverity Scan [5], and PMD [30] cannot reveal integer overflow errors. In principle, model checkers such as Java Pathfinder [27] may deal with overflow errors. However, they require the formalization and specification of overflow-related properties.

This paper presents a supervised learning approach for predicting integer overflow errors through automated classification of Java source code. Text classification is a fundamental task in Natural Language Processing (NLP) with broad applications such as spam detection, topic labeling, and sentiment analysis. It is the process of assigning tags or categories to natural language text according to its content. In this paper, the text classification approach aims to assign "positive" (existence of integer overflow) and "negative" (absence of integer overflow) tags to each method in the given source code. The dataset is built upon real-world programs, covering all integer data types and operations in Java, the methods for preventing integer overflow errors, and adversarial samples. We have evaluated six models, BERT, DistilBERT, CodeBERT, Code2Vec, fastText and NBSVM, representing different text embedding techniques for dealing with Java source code. BERT [6] considers bi-directional contexts of words. It is pretrained with natural language text. DistilBERT [34] is a lighter and faster version of BERT trained with knowledge distillation technology. Based on BERT, CodeBERT [8] is trained with both natural language text and programming language code. Code2Vec [1] is a machine learning model of source code learned from the abstract syntax trees of Java source code. fastText [12] treats a sentence as a bag of n-grams where word order is observed. NBSVM [43] transforms sentences in terms of word frequency. Our experiment results show that BERT has outperformed other models. Applying the resultant BERT model to eleven real-world projects has revealed many integer overflow errors.

The remainder of this paper is organized as follows. Section 2 introduces various forms of integer overflow errors in Java programs and the main methods for preventing integer overflows. Section 3 presents the framework for predicting integer overflow errors via text classification; Section 4 presents the dataset and experiment results; Section 5 applies the resultant models to real-world Java projects; Section 6 reviews related work; Section 7 concludes this paper.

## 2. Integer Overflow in Java Programs

In this section, we first describe various integer overflow errors in Java programs and then present the methods for preventing such errors. Programs with integer overflow errors are called positive samples, whereas programs that have prevented potential integer overflows are negative samples. In addition to using existing programs for machine learning purposes, positive and negative samples can be created according to the ideas in subsections 2.1 and 2.2, respectively. This paper does not consider programs without integer operations, which can be easily determined. Thus, all negative samples involve integer operations without overflows.

### 2.1. *Integer Overflow Errors*

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. Table 1 presents the list of integer data types in Java. Note that *char* is a 16-bit unsigned integer type. Its values represent UTF-16 code units to which integer operations apply.

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.

Another common mistake is the incorrect evaluation order of multiple operators in a compound expression. The order of these operators may not matter in the

Table 1. Integer Data Types in Java.

| Table | Size | Inclusive Range |
|---|---|---|
| byte | 1 byte | -128 .. 127 |
| short | 2 bytes | -32,768 .. 32,767 |
| int | 4 bytes | -2,147,483,648 .. 2,147,483,647 |
| long | 8 bytes | - 9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807 |
| char | 2 bytes | 0 .. 65,535 |

mathematical sense. However, it is important in computer programs because overflow can occur to an intermediate calculation. Consider a mathematical formula, $y = x * 2/5$, which is independent of whether the multiplication is performed before or after the division. Integer overflow happens to the statement $int\ y = x * 2/5$; when the *int* variable $x$ is greater than $Integer.MAX\_Value/2$. Similarly, the order of addition and subtraction operators is also important. Incorrect order may lead to overflow or underflow.

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.

## 2.2. *Debugging and Prevention of Integer Overflow*

Some integer overflow errors, once detected, can be fixed by simply changing the expressions where the overflow occurs. For example, $int\ y = x*2/5$; may be rewritten as $int\ y = x/5 * 2$; For other cases, however, bug fixes may apply to statements or expressions that do not exhibit overflow behavior themselves but cause an overflow elsewhere.

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. In the following, we describe the primary methods for Java programming: precondition test, built-in safe methods ($Math. * Exact()$ in Java 8), upcasting, and BigInteger [20].

The precondition test is to ensure safe calculation by checking the operands of each arithmetic operator. The following method, $safeAdd$, performs the addition only when overflow will not happen. It will throw an exception when $right$ is greater than 0 and $left$ is greater than $MAX\_VALUE - right$, or when $right$ is not greater than 0 and $left$ is less than $MIN\_VALUE - right$.

The Java 8 release introduced several safe arithmetic methods in the $Math$ class,

```
static final int safeAdd(int left, int right) {
    if (right > 0 ? left > Integer.MAX_VALUE - right
                  : left < Integer.MIN_VALUE -right) {
        throw new ArithmeticException("Integer overflow");
    }
    return left + right;
}
```

such as *addExact*, *subtractExact*, *negateExact* and *multiplyExact*. They either return a mathematically correct value or throw *ArithmeticException*. To avoid integer overflow, for example, the expression $oldAcc + (newVal * scale)$ can be coded as $Math.addExact(oldAcc, Math.multiplyExact(newVal, scale))$. The $Math$ class does not provide any methods for safe division or absolute value, though.

The upcasting method uses the next larger primitive integer type to store the operands and perform the calculation to ensure the overflow does not happen. Each intermediate result stored in the larger primitive integer type in the calculation can compare with the range of the original smaller type. If it is within the range, the result is downcast to the original smaller type before assigned to a variable of the smaller type; otherwise, it throws an exception. In this case, the range check must be performed after each arithmetic operation. Larger expressions without per-operation bounds checking can overflow the larger type.

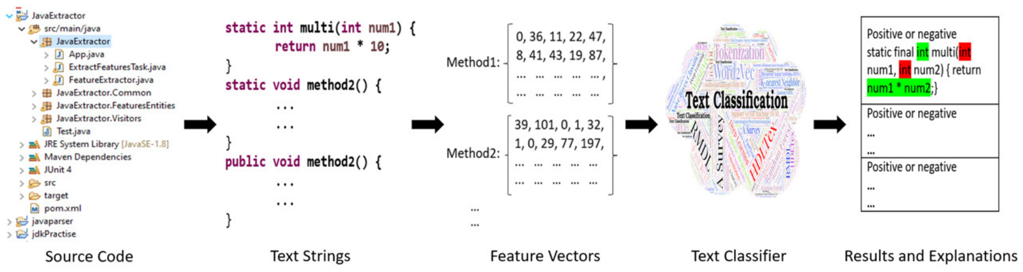
BigInteger is the standard arbitrary-precision integer type provided by the Java standard libraries. The BigInteger methods convert the operands into BigInteger objects and perform all arithmetic operations using overflow-free methods. A single range check is needed before converting the result to the original smaller type.

## 3. Prediction of Integer Overflow Errors

### 3.1. *The Framework*

Figure 1 illustrates the framework for predicting integer overflow errors via automated text classification. Given the source code of a Java program with a number of classes and methods, the goal is to classify each method either positive (i.e., the existence of integer overflow) or negative (i.e., absence of integer overflow). In this approach, we first convert each method into a text string, like a natural language sentence, and transform the text into the internal representation (e.g., feature vector) of a classifier. The classifier assigns a tag of either positive or negative to the text and provides an explanation of what features have contributed to the prediction. In the visualized result, each word with green (or red) background has a positive (or negative) correlation to the result. The correlation weight is indicated by the level of greenness or redness. In general, many words may be correlated to a prediction result. However, when only a few words are highlighted in green for a positive prediction, debugging may focus on these locations.

When the text string of a Java method is transformed for text classification, non-word symbols including integer operators are typically ignored. To address this



```
static int multi(int num1) {
        return num1 * 10;
}
static void method2() {
        ...
        ...
}
public void method2() {
        ...
        ...
}
```

```
Method1: [0, 36, 11, 22, 47,
          8, 41, 43, 19, 87,
          ... ... ... ... ...,
          ... ... ... ... ...]
Method2: [39, 101, 0, 1, 32,
          1, 0, 29, 77, 197,
          ... ... ... ... ...
          ... ... ... ... ...]
...
...
```

```
Positive or negative
static final int multi(int num1, int num2) { return num1 * num2;}
Positive or negative
...
...
Positive or negative
...
...
```

Figure 1. The framework for predicting integer overflow errors

issue, we substitute these symbols for predefined words because they are critical to integer overflows. Table 2 shows the list of symbols and their substitutions. It also indicates whether each symbol is related to integer overflow. "(" and ")" may involve overflow because they affect how an expression is evaluated.

## 3.2. *Classification Models*

While any classification models can be adopted in the above framework, this paper focuses on six models that represent different text embedding techniques for dealing with input sentences (i.e., Java methods). BERT, DistilBERT and codeBERT consider bi-directional contexts of words. Code2Vec extracts path-contexts from the abstract syntax trees of source code. fastText treats a sentence as a bag of n-grams where word order is observed. NBSVM transforms sentences in terms of word frequency.

### 3.2.1. *BERT*

BERT [6] is a recent NLP language model from Google AI Language. Previous models often suffer from polysemy or memory loss. BERT resolves these symptoms using an attention mechanism to enable contextual learning and long sentence learning. Attention is, to some extent, motivated by how we pay visual attention to different regions of an image or correlate words in one sentence. For example, the word "key" would have the same context-free representation in "a lock key" and "the key to this problem." An attention mechanism enables contextual learning by computing a representation of each word, e.g., word embedding, based on the left and right context of the word "key" in the sentence. The attention mechanism does not discriminate against these words that are positioned a distance away from the word that machines try to interpret, and thus, it solves the memory loss issue caused by the vanished gradient descendent problem.

BERT is designed to pre-train deep bidirectional representations from unlabeled texts, such as Wikipedia, by jointly conditioning on both left and right context in all layers. The pre-trained BERT model can be fine-tuned with just one additional output layer to create new state-of-the-art models for a wide range of NLP tasks.

Table 2. Substitutions of Symbols.

| Symbol | Overflow? | Substitution |
|---|---|---|
| + | Yes | SYMZPLUS |
| - | Yes | SYMZMINU |
| * | Yes | SYMZMUL |
| / | Yes | SYMZDIV |
| ++ | Yes | SYMZDOUBPLUS |
| – | Yes | SYMZDOUBMINU |
| += | Yes | SYMZPLUS SYMZEQAL |
| -= | Yes | SYMZMINU SYMZEQAL |
| *= | Yes | SYMZMUL SYMZEQAL |
| /= | Yes | SYMZDIV SYMZEQAL |
| ( | Yes | SYMZLETPARE |
| ) | Yes | SYMZRITPARE |
| unary - | Yes | SYMZMINU |
| unary + | No | SYMZPLUS |
| % | No | SYMZPERC |
| = | No | SYMZEQAL |
| %= | No | SYMZPERC SYMZEQAL |
| <<= | No | SYMZLESS SYMZLESS SYMZEQAL |
| >>= | No | SYMZGREAT SYMZGREAT SYMZEQAL |
| &= | No | SYMZAND SYMZEQAL |
| \| = | No | SYMZVERT SYMZEQAL |
| $^=$ | No | SYMZCARET SYMZEQAL |
| << | No | SYMZLESS SYMZLESS |
| >> | No | SYMZGREAT SYMZGREAT |
| & | No | SYMZAND |
| \ | No | SYMZSLASH |
| ˆ | No | SYMZCARET |
| ˜ | No | SYMZTILDE |
| ! | No | SYMZTEXCLA |
| < | No | SYMZLESS |
| > | No | SYMZGREAT |
| >= | No | SYMZGREAT SYMZEQAL |
| <= | No | SYMZLESS SYMZEQAL |
| == | No | SYMZEQAL SYMZEQAL |
| != | No | SYMZTEXCLA SYMZEQAL |

This paper uses a forwarding neural network as the additional output layer. We obtain the BERT-based classifier through fine-tuning (re-learning) and validation. The fine-tuning process takes a set of vulnerable and non-vulnerable Java methods (i.e., positive and negative samples) to fine-tune BERT with the forwarding neural network. The validation process produces a list of contextual word embeddings for each word in a given validation sample. The forwarding neural network reduces the semantics to a vector and further converts the vector to a vulnerability probability using a softmax function.

### 3.2.2. *DistilBERT*

DistilBert [34] is a lighter and faster model trained with knowledge distillation technology on the basis of BERT-base, which involves three steps: 1) Use the original BERT-base as the teacher network; 2) the number of network layers is halved on the basis of BERT-base; 3) train a student network on the soft label and the hidden layer parameters of the teacher network. Compared to BERT, the size of DistilBert is reduced by 40%, and the inference speed is increased by 60%. The performance is only reduced by about 3%.

### 3.2.3. *CodeBERT*

Based on BERT, CodeBERT [8] is trained with a hybrid objective function, including the masked language modeling (MLM) and replaced token detection (RTD) pre-training task. RTD corrupts the input by substituting some of the input tokens and then trains a discriminator to predict whether the generator sample covers each token in the corrupted input. Different from BERT, CodeBERT uses both natural language and programming language as input in the pre-training phase, which is a combination of two segments and a special delimiter, i.e. $[CLS], w_1, w_2, ..., w_n, [SEP], c_1, c_2, ..., c_m, [EOS]$. One of the snippets is natural language text, and the other is code written in some programming language. The outputs of CodeBERT include a contextual vector representation of each token (for both natural language and code) and an aggregated vector representation of sequence.

### 3.2.4. *Code2Vec*

Code2Vec [1] is a network structure that can embed code and develop a continuously distributed vector representation for the code. Model inputs are code snippets and their corresponding semantic features, such as method names. In the process of embedding, the model captures semantic information between names, and method names can be compared and inferred. First, distinct pathways are extracted from the abstract syntax tree corresponding to the code fragment, and the tuple created by the values of all nodes on the path is called path-context. Then, the nodes in each path-context are concatenated into a vector representing the path-context. Finally, all path-contexts merge together through an attention model and generate a final vector to capture the features of method name.

### 3.2.5. *fastText*

fastText [12], created by Facebook's AI Research (FAIR) lab, uses a simple architecture to achieve high speed on training and testing tasks while retaining accuracy comparable to some deep learning models. fastText is similar to the CBOW model in word2vec. CBOW aims to predict words, whereas fastText focuses on predicting labels. The architecture of fastText consists of three layers: input layer, hidden layer, and output layer.

- **Input layer**: Each input sentence is treated as a bag of n-grams, rather than a bag of words. N-gram divides the sentence into several n-window-size fragments and composes a contiguous sequence to ensure word order is considered as a significant feature. There are two ways to implement n-gram: n-word and n-char. For the sentence 'How are you?', 2-word breaks it into a contiguous sequence of "how are, are you", whereas 3-char converts it to "<ho, how, owa, war, are, rey, eyo, you, ou>", where '<' and '>' are counted as boundary symbols. Although word order is considered, there are much more fragments need to be embedded. Hence, fastText stores all fragments into hash buckets, and fragments in the same bucket share the same embedding vector, which can effectively reduce duplicate encoding. For a word that has not appeared in training samples, it still can be encoded into a vector based on the existing n-char fragments.
- **Hidden layer**: This is a process of learning weight matrix through back-propagation learning. The initial weight matrix is randomly generated. When the input vector of a training sample comes through, the loss can be calculated based on the expected and actual values. Then, the weight matrix is updated by reducing the gradient of loss and weight matrix. After multiple rounds of training, the weight matrix can be optimal.
- **Output layer**: A hierarchical softmax is used to calculate the probability of each label. It can reduce the amount of calculation, especially for multi-label tasks. The hierarchical softmax is implemented through Huffman tree. All vectors and labels are used as leaf notes to draw the Huffman tree. High frequency notes are close to the root, and low frequency notes are far from the root. The label in the path with the highest probability is the output of the model.

### 3.2.6. *NBSVM*

NBSVM [43] integrates Support Vector Machines (SVM) with Naïve Bayes (NB) features, two traditional linear text classifiers. NBSVM is a variant of SVM by using the ratio of logarithm and count from NB as eigenvalues. Unlike BERT and fastText that have specialized algorithms to predict words by context, NBSVM uses word frequency to convert sentences into a vector matrix. All words from the dataset of an NLP task are extracted and made into a dictionary based on word frequency. A complete sentence is then represented by a vector matrix, where each word is encoded as a vector. Term Frequency–Inverse Document Frequency (TF-IDF) is used to balance word frequency and its importance to avoid overstating the influence of words with high-frequency. By measuring the word frequency and universality (quantity of samples with related words), TF-IDF can weigh each word, reducing the influence of words with high frequency and low universality.

## 4. Empirical Studies

### 4.1. *The Java Dataset*

Our Java dataset is based on IARPA STONESOUP3.0, part of the NIST SARD (Software Assurance Reference Dataset) suites [35] for testing static analysis tools with seeded security flaws. The seven Java projects in IARPA STONESOUP3.0 contain 2,711 seeded vulnerabilities, including 93 integer overflow errors. As described in our preliminary study [19], IARPA STONESOUP3.0 has several limitations: (a) The 93 positive samples have covered only three addition operators (+, +=, ++) of one integer type (i.e., short). (b) No negative sample reflects how integer overflow errors are mitigated. (c) Although the classifiers have accurately learned the syntactic features of the STONESOUP programs, the resultant prediction models were worse than a random guess when applied to real-world programs with integer overflow errors. In short, IARPA STONESOUP3.0 is not suitable for evaluating machine learning based approaches to vulnerability prediction.

To address the above limitations, we have created 857 positive samples and 813 negative samples. The positive samples account for all types of integer overflow errors discussed in Section 2.1. The negative samples cover all methods for preventing integer overflow errors described in Section 2.2. Inspired by contrastive learning, we created the negative samples by applying a prevention technique to fix the error in each positive sample if feasible. Thus, our dataset consists of many pairs of contrastive samples.

Unlike contrastive learning that only uses contrastive samples, however, our dataset also includes many negative samples that represent widely applied Java programs of various integer types and operations without overflow errors. They originate from IARPA STONESOUP3.0. Among the new samples, 100 are adversarial positives that are incorrect programs with the key features that usually appear in negative samples; 111 are adversarial negatives that are correct programs with the key features that usually appear in positive samples. Adversarial samples are essential for machine-learning-based prediction models, which could easily be fooled by such malicious code if trained only with normal code. In brief, our dataset consists of 6,888 samples, including 950 positive and 5,938 negative samples.

In the following, we describe how the new samples are created.

#### 4.1.1. *Normal Positive Samples*

We used several methods to create normal positive samples. First, we refactored the 93 STONESOUP samples by changing variable names, modifying arithmetic expressions, and adding more comments (called STONESOUP refactorings). Second, we collected samples from textbooks, websites, and bug histories of open-source projects. Based on these samples, we further crafted positive samples by writing new code segments to cover all integer types and operations that may cause overflows and by inserting overflow code segments into negative samples. To do so, 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.

Algorithm 1 creates a positive sample by inserting a flawed code segment into an existing negative sample. Given a set of method/variable names $N$, a set of integer data types $T$, a set of integer operators $O$, and a negative sample $S_n$, we first reorganize $S_n$ according to the positions of ‘;’, ‘{’ and ‘}’ to ensure the code segment will not be inserted in the middle of a statement. Next, we select an operator $o$ from $O$, a type $t$ from $T$, a line number $l$ from range $[2, L-1]$, where $L$ is the number of lines of $S_n$. Then we choose (a) a name $n$ from $N$ if the operator is ‘++’, ‘–’, ‘unary –‘ or absolute value, (b) two names $n$, $n_1$ from $N$ if the operator is ‘+=’, ‘-=’, ‘*=’, or ‘/=’, or (c) three names $n$, $n_1$, $n_2$ from $N$ for other operators. The selected names and integer type $t$ are used to declare variables in the method parameters (only the last two if there are three names). Finally, we compose an integer overflow code segment with the selected names and operator $o$ and insert it into line $l$ of $S_n$ to obtain a positive sample $S_p$.

**Algorithm 1:** Create a positive sample by inserting a flawed code segment to a negative sample

```
Input: a set of method/variable names $N$, a set of integer data types $T$, a
       set of integer operators $O$, a negative sample $S_n$
Output: a positive sample $S_p$
1  L = #lines of $S_n$
2  Select a random operator o from O, a random type t from T, a random line
     number l from [2, L-1]
3  if o = '++', '- -', 'unary –' or absolute value then
4      Select a random name n from N
5      t, n $\rightarrow$ parameter p
6      o, n $\rightarrow$ flawed code segment c
7  else if o = '+=', '-=', '*=', or '/=' then
8      Select two random names $n$, $n_1$ from N
9      t, $n$, $n_1$ $\rightarrow$ p
10     o, $n$, $n_1$ $\rightarrow$ c
11 else
12     Select three random names $n$, $n_1$, $n_2$ from N
13     t, $n_1$, $n_2$ $\rightarrow$ p
14     o, $n$, $n_1$, $n_2$ $\rightarrow$ c
15 $S_p$ $\leftarrow$ insert p to parameter position of $S_n$ and c to line l of $S_n$
16 Return $S_p$
```

Algorithm 2 creates a new Java method from scratch. Given a set of method/-variable names $N$, a set of method modifiers $M$, a set of method return types $R$, a set of integer operators $O$, and a set of overflow-free code segments, we first select a method name $n_m$ from $N$, a method modifier $m$ from $M$ and a method return type $r$ from $R$ to compose an empty method template $S_n$. Then, $num$ (a random number from 0 to 3) code segments are selected from $C$ and inserted into $S_n$. Then we use $N$ (besides $n_m$), $r$, $O$ and $S_n$ as the input of Algorithm 1 to obtain a positive sample $S_p$. Finally, if the return type is not 'void', the first variable name, 'return', and ';' form a return statement as the last line (before '}').

```
Algorithm 2: Generate a new positive sample
  Input: a set of method/variable names $N$, a set of method modifiers $M$, a
         set of method return types $R$, a set of integer operators $O$, a set of
         overflow-free code segments $C$
  Output: a positive sample $S_p$
1 Select a random method name $n_m$ from N, a random method modifier m
    from M, a random return type r from R
2 $n_m$, m and r compose an empty method $\rightarrow S_p$
3 $num$ = a random integer $\in$ [0,3]
4 Select num random code segments from $C$ and insert them into $S_p$
5 $S_p$ = Algorithm 1(N\$n_m$, r, O, $S_p$)
6 if $r \neq$ 'void' then
7     insert a return statement into the last line of $S_p$
8 Return $S_p$
```

Table 3 shows the distributions of positive examples classified by integer operators.

Table 3. Normal Positive Samples.

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

### 4.1.2. *Normal Negative Samples*

The creation of normal negative samples was inspired by contrastive learning, which 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. Specifically, we created negative samples from the existing positive samples using the prevention techniques to fix each overflow error if feasible. There are 55 positive samples to which the prevention techniques do not apply because concrete values cause the overflows. The above approach can reduce the bias of syntactic features because such features appear in both positive and negative samples.

Table 4 shows the distributions of negative samples classified by prevention techniques. Each sample in the category of "other operators" involves an operator that indirectly leads to integer overflow and can be fixed by changing the operator.

Table 4. Negative Samples.

| Prevention Techniques | Negative Samples |
|---|---|
| Precondition Test | 173 |
| Upcasting | 166 |
| BigInteger | 166 |
| Math.*Exact | 123 |
| Other Operators | 74 |
| Total | 702 |

### 4.1.3. *Adversarial Samples*

An adversarial positive sample is a flawed Java 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. We created 100 adversarial positives and 111 adversarial negatives after the classifiers have been trained with the normal samples. 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", "tracepointVariableShort", and "trigger-point". 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.

We used the following methods to create the adversarial samples:

- 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.

### 4.2. *Experiment Results of the Java Dataset*

The experiments were performed on Google Colab Pro with a Tesla V100 GPU and 27.4G RAM, using ktrain v.0.28.2, transformers v.4.10.3, and scikit-learn v.0.23.2. For BERT, we used the BERT-Base(uncased) model initially embedding each sample and train BERT with our dataset to fine-tune the parameters. We added a softmax layer to get a value between 0 to 1 for prediction. For code2vec, distilBERT, codeBERT, fastText and NBSVM, we used the default training parameters. We set the learning rate to 2e-5 and epoch to 40 for all models. If the accuracy and loss values are satisfactory in the 40th epoch, the training terminates; otherwise, it will continue.

We compare the classifiers with the following performance metrics:

- **Accuracy**: Accuracy is the ratio of correctly predicted samples to the total number of test samples. It can reflect the general performance on both positive and negative samples
- **Precision**: Precision indicates that among all samples a model predicts as positive, how many of them are true positive. It can reflect the accuracy of prediction on these samples. For samples predicted as positive, there are two possible outcomes: a positive sample predicted positive (i.e., true positive - TP) and a negative sample predicted positive (i.e., false positive - FP).
$$Precision = \frac{TP}{TP + FP}$$
- **Recall**: Recall indicates that among all positive samples in our dataset, how many of them are predicted as positive. A positive sample can be predicted as positive (i.e., true positive - TP) or negative (i.e., false negative - FN).
$$Recall = \frac{TP}{TP + FN}$$
- **F1 Score**: F1 score is the harmonic mean of precision and recall. It ranges from 0 to 1. A high F1 score indicates a good performance on positive samples.
$$F1\ score = 2 * \frac{precision * recall}{precision + recall}$$
- **Area Under the Curve**: AUC is the area under ROC (receiver operating characteristic curve). A high AUC means that the classifier has a high probability to make a correct prediction.

We applied 10-fold cross-validation to evaluate the classifiers, i.e., 90% of the samples for training and 10% for testing in each fold on Java dataset (except 30 positive and 25 negative adversarial samples reserved for the purposes of comparative studies) Table 5 shows the results on Java dataset. BERT outperforms the other five classifiers in all performance measures. It has correctly identified the vast majority of the integer overflow errors, with a minimum score of 99% for accuracy, precision, recall, F1 score, and AUC. A high precision can reduce the number of false positive samples, which saves time for verification. A high recall indicates that there are less missed integer overflow errors. CodeBERT gives the second-best performance. Although codeBERT can handle both natural language and programming language, integer overflow errors usually occur in an expression. The RTD in codeBERT doesn't help model understand the contextual features of an expression. DistilBERT, as a smaller and faster variant of BERT, performs 1% lower than BERT dose. Code2Vec can be an effective approach, however, it represents source code on AST paths for method name prediction. fastText and NBSVM give low performance, while others have an average score of over 90%.

Table 5. Experiment Results of the Java Dataset.

| Model | Accuracy | Precision | Recall | F1 | AUC |
|---|---|---|---|---|---|
| fastText | 0.9375 | 0.8603 | 0.9302 | 0.8899 | 0.9826 |
| NBSVM | 0.9470 | 0.8763 | 0.9468 | 0.9063 | 0.9767 |
| Code2Vec | 0.9597 | 0.9017 | 0.9681 | 0.9336 | 0.9881 |
| DistilBERT | 0.9878 | 0.9783 | 0.9868 | 0.9819 | 0.9914 |
| CodeBERT | 0.9890 | 0.9836 | 0.9897 | 0.9862 | 0.9927 |
| BERT | 0.9959 | 0.9939 | 0.9903 | 0.9921 | 0.9968 |

#### 4.2.1. *The Impacts of Adversarial Samples*

To evaluate the impacts of the adversarial samples, we have also trained the classifiers without the adversarial samples and then applied the resultant prediction models to the reserved 30 positive and 25 negative adversarial samples. Figure 2 compares the results of BERT prediction models trained with and without the adversarial samples. When BERT is not trained with the adversarial samples, its performance of predicting the 55 adversarial samples is about 70%. If trained with the adversarial samples, however, the performance has increased to 98%.

Figure 3 compares the results of fastText prediction models trained with and without the adversarial samples. Training with the adversarial samples only slightly affects the prediction. In both cases, the performance scores are not satisfactory.

Figure 4 compares the results of NBSVM prediction models trained with and without the adversarial samples. The adversarial samples have a notable impact (e.g., increasing the performance from 60% to 70%). In both cases, however, the performance scores are not satisfactory.

In brief, fastText and NBSVM can be fooled by adversarial samples no matter whether trained with the adversarial samples. While BERT can be fooled by adversarial samples if not trained with such samples, it can perform very well if trained with the adversarial samples.

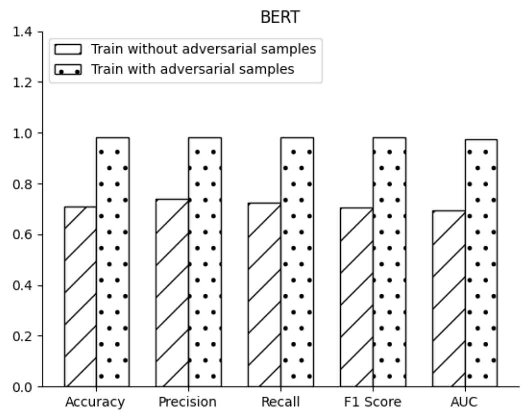


Figure 2. BERT Prediction Models Trained with and without the Adversarial Samples

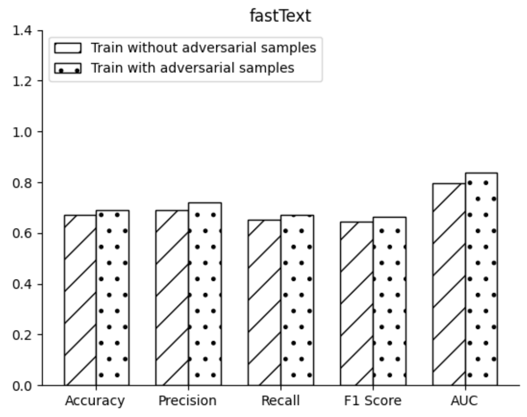


Figure 3. fastText Prediction Models Trained with and without the Adversarial Samples

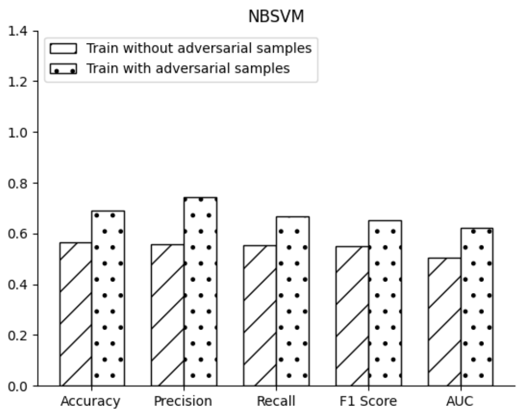


Figure 4. NBSVM Prediction Models Trained with and without the Adversarial Samples

### 4.2.2. *Comparison with Contrastive Learning*

To evaluate the performance of contrastive learning, we trained the classifiers with the original 93 positive samples in IARPA STONESOUP3.0, normal positive samples, and the contrastive negative samples created from the normal positive samples (excluding the negative samples of the Java programs in IARPA STONESOUP3.0). The results are shown in Figures 5, 6, and 7, respectively. All classifiers have significantly dropped their performance. It indicates that the negative samples from real-world Java programs in IARPA STONESOUP3.0 are very useful. The samples created through the idea of contrastive learning alone are inadequate.

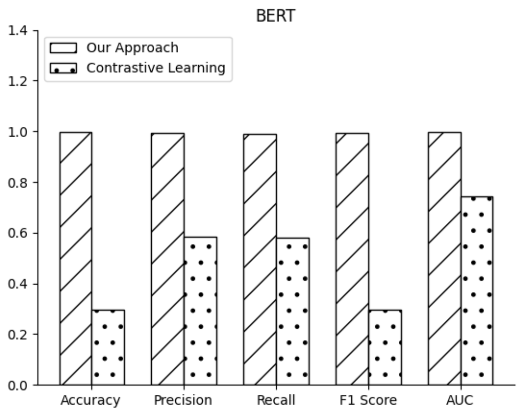


Figure 5. Comparison with Contrastive Learning for BERT

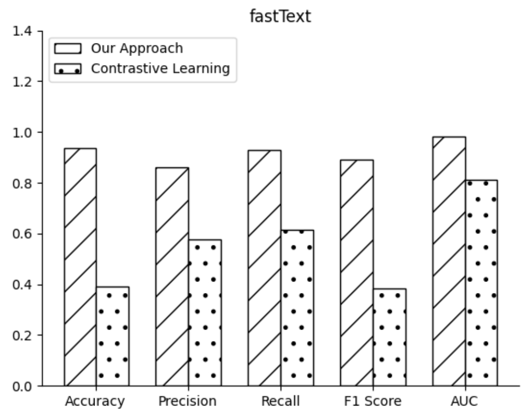


Figure 6. Comparison with Contrastive Learning for fastText

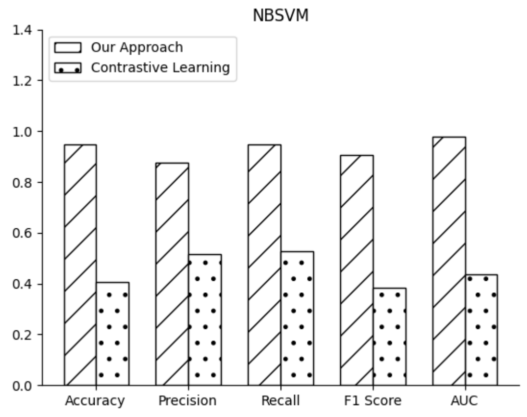


Figure 7. Comparison with Contrastive Learning for NBSVM

### 4.3. *The Experiment of a C/C++ Dataset*

We have applied our approach to the C/C++ programs in the NIST SARD dataset [35]. They are listed in Table 6. There are 3,751 positive and 92,798 negative samples.

Table 7 presents the experiment results. Code2Vec does not apply to C/C++ because it is built only from Java source code. The performance of each classifier on the C/C++ dataset is similar to that of the Java dataset. BERT has identified

Table 6. C/C++ Programs in the SARD Dataset

| Subjects | Version | KLOC | Files | Integer Overflow |
|---|---|---|---|---|
| Apache Subversion | 1.8.3 | 968 | 1,728 | 12 |
| FFmpeg | 1.2.2 | 615 | 3,478 | 10 |
| Gimp | 2.8.8 | 736 | 6,117 | 12 |
| GNU Grep | 2.14 | 77 | 918 | 3 |
| Juliet | 1.3 | 7,935 | 15,476 | 3,687 |
| OpenSSL | 1.0.1e | 361 | 2,203 | 5 |
| PostgreSQL | 9.2.4 | 650 | 5,458 | 11 |
| Tree | 1.7.0 | 80 | 412 | 4 |
| Wireshark | 1.10.2 | 2,334 | 5,109 | 7 |
| Total | - | 13,756 | 40,899 | 3,751 |

most of the C/C++ positive samples with an average score of 99% for all metrics. CodeBERT and distilBERT have slightly lower scores than BERT. fastText and NBSVM are not as good as the others.

Table 7. Experiment Results of the C/C++ Dataset

| Model | Accuracy | Precision | Recall | F1 | AUC |
|---|---|---|---|---|---|
| fastText | 0.8995 | 0.8516 | 0.9131 | 0.8799 | 0.9633 |
| NBSVM | 0.9010 | 0.8633 | 0.9168 | 0.8801 | 0.9713 |
| DistilBERT | 0.9789 | 0.9709 | 0.9777 | 0.9743 | 0.9891 |
| CodeBERT | 0.9793 | 0.9748 | 0.9802 | 0.9787 | 0.9905 |
| BERT | 0.9863 | 0.9842 | 0.9827 | 0.9830 | 0.9931 |

## 4.4. *Comparison with Related Work*

Although this paper is the only work dedicated to integer overflow errors, several other studies on code vulnerability detection have used integer overflow samples, including DEEPBUGS [32], VUDDY [13], DEVIGN [53], VulDeePecker [17], $\mu$VulDeePecker [54] and FUNDED [42]. They primarily target C/C++ programs. Figure 8 shows the results of comparison for integer overflow errors. VUDDY and DEEPBUGS have low accuracy scores. VUDDY focuses on detecting code clones with hash functions, whereas DEEPBUGS is built on a feed-forward neural network. DEVIGN, built on GNN through source code AST, has a accuracy score of 75%. VulDeePecker and $\mu$VulDeePecker utilize BLSTM to detect multiple types of vulnerabilities. Their accuracy scores are about 87%. FUNDED applies multiple graph representations with different types of edges on GNN. Its accuracy is 91%. Although DEVIGN and FUNDED build graph representations of source code to capture structure features, integer overflow errors most commonly occur in an expression (a sequential structure). A model for contextual and long sentence learning, like BERT and DistilBERT, would perform better on the detection of integer overflow errors. As such, BERT, CodeBERT, and DistilBERT have outperformed other methods.

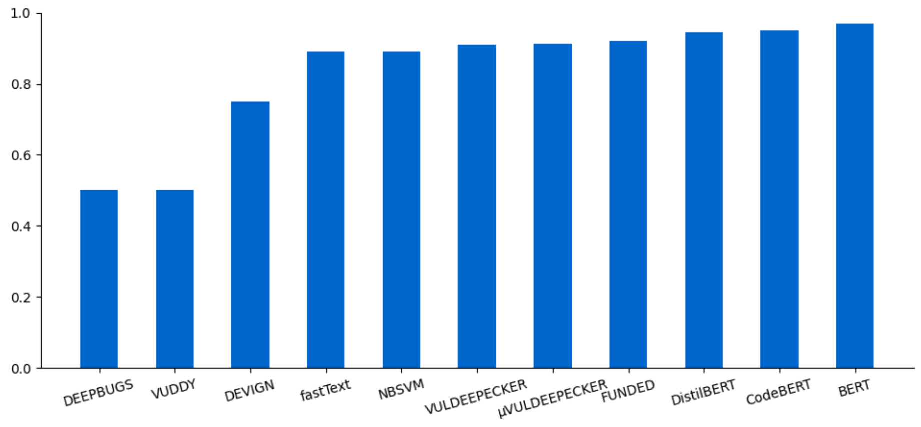


Figure 8. Comparison of Prediction Models of Integer Overflow Errors

## 5. Applications

We have applied the resultant models in Table 5 to 11 real-world projects listed in Table 8. The sizes range from 32 KLOC (thousand lines of code) to 3.2 MLOC (million lines of code). Besides JDK 13.0, the ten applications are the most popular GitHub Java projects ranked by GitHub stars. The eleven projects have a total of 5.714 MLOC, 34,995 classes, and 363,034 methods. 15.59% of the methods contain integer operations. These methods are the samples used for prediction. The total number of samples is 56,606, where 20,727 are from JDK13.0.

We started by applying all classifiers to two medium projects RxJava-3.x and dubbo-master. They have 4,689 and 3,334 methods with integer operations, respectively. Table 9 shows the results. We cannot use the performance metrics (accuracy, recall, F1, and AUC) in the previous sections because we do not know whether each method has an integer overflow error. We have to verify each reported positive manually. BERT predicted five positive methods in RxJava-3.x. Our code review determined that three of them are true positives. fastText reported 72 positives, and only two are true positives. NBSVM reported nothing. For dubbo-master, 6 of 11 reported by BERT are true positives. fastText reported 209 positives, and only four are true positives. Among the six positives reported by NBSVM, 4 are true positives. Overall, BERT is the best. fastText and NBSVM do not seem to work well for real-world programs.

In the following, we focus on the application of BERT. Table 10 shows the prediction results of the BERT model. The eleven projects are sorted by KLOC

Table 8. Sizes of the Applications.

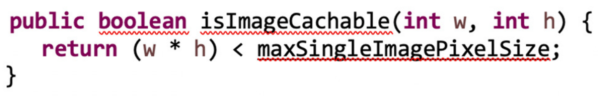
| App # | Name | KLOC | Classes | Methods |
|---|---|---|---|---|
| 1 | retrofit-master | 32 | 235 | 1,916 |
| 2 | MPAndroidChart | 34 | 223 | 2,117 |
| 3 | okhttp-master | 40 | 153 | 2,827 |
| 4 | zxing-master | 60 | 484 | 2,554 |
| 5 | Mall | 68 | 514 | 13,626 |
| 6 | dubbo-master | 237 | 2,232 | 17,517 |
| 7 | RxJava-3.x | 390 | 1,838 | 22,275 |
| 8 | spring-boot-master | 476 | 4,966 | 33,266 |
| 9 | dbeaver-devel | 506 | 4,152 | 32,058 |
| 10 | guava-master | 670 | 2,833 | 50,085 |
| 11 | JDK13.0 | 3,201 | 17,365 | 184,793 |
|  | Total | 5,714 | 34,995 | 363,034 |

Table 9. Sizes of the Applications.

| **Program** | **Samples** | **Model** | **#Positives** | **#True Positives** |
|---|---|---|---|---|
|  |  | BERT | 5 | 3 |
| RxJava-3.x | 4,689 | fastText | 73 | 2 |
|  |  | NBSVM | 0 | 0 |
|  |  | BERT | 11 | 6 |
| dubbo-master | 3,334 | fastText | 209 | 4 |
|  |  | NBSVM | 6 | 4 |

in ascending order in Table 8. However, the number of methods containing integer operations varies from project to project. The sample size is not propositional to the KLOC number. BERT has reported a total of 529 positives. 434 are from JDK13.0 production code (excluding 196 positives in the test code). They account for 2.09% of the samples. For the other ten projects, only 0.26% of the samples were predicted positive. It indicates that JDK13.0 involves much more integer operations than others. zxing-master is the second-highest – 1.1% of the samples were reported positive.

After reviewing the reports, we determined that 181 are true positives. We further classify them into public methods and non-public methods (private, protected, and default). Usually, the former is riskier because invocations to these public methods will more likely cause failures. For example, the following public method, `isImageCachable`, of class `ImageCache` is of the true positives in JDK 13.0.



```
public boolean isImageCachable(int w, int h) {
   return (w * h) < maxSingleImagePixelSize;
}
```

Table 10. BERT's Prediction Results.

| No. | Samples | Positives | %Positive | True Positives | Public Methods |
|---|---|---|---|---|---|
| 1 | 769 | 0 | 0 | 0 | 0 |
| 2 | 514 | 2 | 0.4 | 1 | 0 |
| 3 | 1,665 | 1 | 0.06 | 1 | 0 |
| 4 | 1,099 | 12 | 1.1 | 5 | 1 |
| 5 | 547 | 0 | 0 | 0 | 0 |
| 6 | 3,334 | 11 | 0.33 | 6 | 4 |
| 7 | 4,689 | 5 | 0.1 | 3 | 1 |
| 8 | 10,616 | 3 | 0.03 | 0 | 0 |
| 9 | 5,935 | 16 | 0.23 | 10 | 8 |
| 10 | 6,711 | 45 | 0.67 | 18 | 9 |
| 11 | 20,727 | 434 | 2.09 | 137 | 45 |
| Total | 56,606 | 529 | 0.93 | 181 | 68 |

It checks if an image is cacheable. When the product of `w` and `h` is greater than the maximum `int` value, `w * h` results in a negative integer. `isImageCachable` will return true, which is wrong.

BERT reported no positives in Mall and retrofit-master. None of the three positives reported for spring-boot-master is true positive. One of the two positives in MPAndroidChart is a true positive, but in a non-public method. The true positive in okhttp-master is in a non-public method. Overall, five of the ten applications have no integer overflow errors found in their public methods, including Mall, MPAndroidChart, okhttp-master, retrofit-master, and spring-boot-master.

Among the 181 true positives, 137 are in JDK13.0, and 45 of them are public methods; guava-master has 18 true positives, and 9 are public methods;dbeaver-devel has 10 true positives, and eight are public methods; dubbo-master has six true positives, and four are public methods; Five true positives are found in zxing-master, and only one is in a public method.

To summarize, BERT revealed many errors in real-world applications. Although the overall precision 46.3% (except for JDK13.0) is far from perfect, 181 true positives were found from 363,034 methods in 5.714 MLOC. This would be very beneficial to the developers because no additional work beyond the source code is required. In comparison, to reveal the above errors with model checkers such as Java Pathfinder [27], the specifications of overflow-related properties for all relevant code can be a daunting task. We have also evaluated several static code analyzers publicly available for Java programs, including Spotbugs [40], PMD [30], and Coverity Scan [5]. None of them can detect the overflow errors in our dataset.

## 6. Related Work

The related work falls into three categories: detecting integer overflow errors, detecting code vulnerability via machine learning, and machine learning-based source code representation. The first has focused on binary code or C/C++ source code, such as SIFT [18], TAP [38], SoupInt [44], Indio [52], IntEQ [41], Diode [39], IntFlow [31], IntFinder [2], IOC [7]. The second has not yet targeted integer overflow errors. The third builds source code representation through machine learning for other tasks, such as code understanding and code clone detection.

## 6.1. *Detection of Integer Overflow Errors*

### 6.1.1. *Binary Code Analysis.*

IntScope [45] can detect integer overflow errors in binary code through static analysis. It translates x86 binaries into an intermediate representation for symbolic execution. It uses taint analysis to determine bounds of tainted data and locates sensitive points that probably contain integer overflows. The sensitive points need to be confirmed manually. UQBTng [48] detects integer overflows in Win 32 binaries by de-compiling the binaries into C code and checking the properties through C.assert statements. Zhang et al. [50] combines static analysis and dynamic analysis. The static analysis translates the binary code into intermediate representations such as control flow graph and call graph. It extracts all sensitive code related to integer as input to the dynamic analysis. The dynamic analysis further uses taint analysis to construct the relative paths and explore all symbolic execution to check integer overflow errors. Brick [3] and SmartFuzz [23] are dynamic analysis tools based on Valgrind [28], a dynamic binary analysis framework. Brick works on compiled executables, and SmartFuzz needs to generate inputs. SAGE [9] keeps track of execution traces of binary code and generate new inputs through symbolic execution.

### 6.1.2. *Static Analysis of C/C++ Source Code*

Sample methods for integer overflow detection include ARCHER [4], PREfast [21], and Prefix [26]. PREfast checks integer overflows during runtime. It only works on a small section of code and returns a log recording code defect. Prefix can work on a large range of source code and use a ranking mechanism to detect integer overflow. Laguna and Martin [15] detected integer overflow errors in large-scale parallel applications. They identify integer variables from a large scale and run them at a small scale to detect whether overflow occurs on the original scale.

## 6.2. *Detection of Code Vulnerability via Machine Learning*

Scandariato et al. [36] proposed a vulnerability prediction model for Java projects through text-mining source code. The source code is tokenized into monograms and integrated with frequency to generate feature vectors. They used Naïve Bayes and Random Forest to predict vulnerability. Yamaguchi et al. [49] extended NLP techniques towards vulnerability assessment by using vulnerability extrapolation. This approach extracts API symbols, embeds API symbols in a vector space, and determines API usage patterns using machine learning. Shabtai et al. [37] used principal component analysis on the abstract syntax tree of source code to detect malicious code. Similarly, Mokhov et al. [22] used multiple algorithms from WEKA [10] and the abstract syntax tree as features to build prediction models. Hovsepyan et al. [11], and Pang et al. [29] used SVM on a bag-of-words (BOW) and n-grams representation of simple tokenization of Java source code to predict vulnerable code. VUDDY [13] utilize a robust parser of C/C++ and a hash function to detect vulnerable code clones.

VulDeePecker [17] 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. A forward slice corresponds to the statements affected by the argument in question, whereas a backward slice corresponds to the statements that can affect the argument in question. uVulDeePecker [54] extends VulDeePecker by dealing with 40 vulnerability types. Code attention and its extraction method are used to help pinpoint vulnerability types.

Li et al. [16] 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. [33] 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, including buffer overflow and NULL pointer dereference. Based on the graph learning, DEVIGN [53] generates a graph representation of code snippets based on their AST, control flow graph (CFG) and data flow graph (DFG). It builds a vulnerability detector through graph neural network (GNN). FUNDED [42] uses word2vec network to generate node embedding in AST and update each node by analyzing nine types of edges. It has achieved an average accuracy of 91% on integer overflow.

Unlike the above work, our approach focuses on detecting integer overflow errors via text classification. Our dataset has covered all integer types and operators in Java, methods for preventing integer overflows, and adversarial samples.

## 6.3. *Machine Learning-Based Source Code Representation*

White et al. [47] applied source code abstract syntax tree (AST) through Recursive Neural Network (RNN) to learn lexical and syntactic features, which has a accuracy score of 93% on code clone detection. Mou et al. [24] designed a tree-based convolutional neural network (TBCNN) and add a convolution kernel on program AST to capture structural information. CDLH [46] is a functional clone detection tool that transfer source code through an AST-based LSTM and a hash function to hash code. DeepBugs represents code via word2vec for detecting name-based bugs. DeepBugs [32] learned code representation by building a word2vec network to detect name-based bugs. Zhang et al. [51] split an entire program AST into some small statement trees, use a bidirectional RNN model to leverage small statements, and generate a vector representation of source code.

Code2Vec [1] first decompose the code into its corresponding set of paths in the AST, then use a neural network to learn the representation of each path and learn how to integrate the representations of all paths. CodeBERT [8] is built on a multi-layer transformer. It can handle both natural and programming languages. It captures the semantic connections between natural and programming languages and outputs a general representation that broadly supports NL-PL understanding tasks and generation tasks.

## 7. Conclusions

We have presented the supervised learning approach to the prediction of integer overflow errors in Java source code and evaluated six models with a comprehensive dataset built from real-world Java programs and an existing C/C++ dataset. The models represent different text embedding techniques for dealing with source code. The Java dataset covers all integer data types and operations in Java, the methods for preventing integer overflow errors, and adversarial samples. The experiment results have demonstrated that BERT as a representative deep-learning transformer has outperformed other models. It can reveal integer overflow errors in real-world Java programs.

Automated detection of software vulnerabilities in source code is a major challenge because there are many types of vulnerabilities. For instance, the National Vulnerability Database (nvd.nist.gov) has identified 839 vulnerability types, called Common Weakness Enumerations (CWEs). However, there is no existing Java vulnerability dataset suitable for machine learning-based vulnerability prediction. This paper offers a unique dataset for the systematic study of integer overflow errors in Java programs. Our future work will expand the approach to other types of code vulnerabilities and discover effective classifiers for identifying multiple types of errors.

## 8. Acknowledgment

This work was supported in part by US National Science Foundation (NSF) under grant 1820685.

## References

[1] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. "code2vec: Learning distributed representations of code". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29.

[2] P. Chen, H. Han, Y. Wang, X. Shen, X. Yin, B. Mao, and L. Xie. "IntFinder: Automatically detecting integer bugs in x86 binary program". In: *International Conference on Information and Communications Security*. Springer. 2009, pp. 336–345.

[3] P. Chen, Y. Wang, Z. Xin, B. Mao, and L. Xie. "Brick: A binary tool for run-time detecting and locating integer-based vulnerability". In: *2009 International Conference on Availability, Reliability and Security*. IEEE. 2009, pp. 208–215.

[4] R. Chinchani, A. Iyer, B. Jayaraman, and S Upadhyaya. "ARCHERR: Runtime Environment Driven Program Safety". In: *Proceedings of the European Symposium on Research in Computer Security.* 2004, pp. 385–406.

[5] *Coverity Scan.* `https://scan.coverity.com`.

[6] 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]`.

[7] W. Dietz, P. Li, J. Regehr, and V. Adve. "Understanding Integer Overflow in C/C++". In: *ACM Transactions on Software Engineering & Methodology* 25.1 (2015), pp. 1–29.

[8] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. "Codebert: A pre-trained model for programming and natural languages". In: *arXiv preprint arXiv:2002.08155* (2020).

[9] P. Godefroid, M. Levin, and D. Molnar. "Automated whitebox fuzz testing". In: *Network and Distributed System Security Symposium.* 2008.

[10] G. Holmes, A. Donkin, and I.H. Witten. "Weka: A machine learning workbench". In: *Proceedings of ANZIIS'94-Australian New Zealnd Intelligent Information Systems Conference.* IEEE. 1994, pp. 357–361.

[11] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden. "Software vulnerability prediction using text analysis techniques". In: *Proceedings of the 4th international workshop on Security measurements and metrics.* 2012, pp. 7–10.

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

[13] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. "Vuddy: A scalable approach for vulnerable code clone discovery". In: *2017 IEEE Symposium on Security and Privacy (SP).* IEEE. 2017, pp. 595–614.

[14] D. Kravets. *Sorry ma'am you didn't win $43M – there was a slot machine 'malfunction'.* Ed. by Ars Technica. June 2017. URL: `https://arstechnica.com/tech-policy/2017/06/sorry-maam-you-didnt-win-43m-there-was-a-slot-machine-malfunction/`.

[15] I. Laguna and M. Schulz. "Pinpointing scale-dependent integer overflow bugs in large-scale parallel applications". In: *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE. 2016, pp. 216–227.

[16] 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.

[17] 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.

[18] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard. "Sound input filter generation for integer overflow errors". In: *Proceedings of the Symposium on Principles of Programming Languages (POPL2014).* 2014.

[19] Y. Luo, W. Xu, and D. Xu. "Detecting Integer Overflow Errors in Java Source Code via Machine Learning". In: *The 33rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI2021).* 2021.

[20] 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.

[21] Microsoft. *PREfast Analysis Tool.* `https://msdn.microsoft.com/en-us/library/ms933794.aspx`. 2012.

[22] S. A Mokhov, J. Paquet, and M. Debbabi. "MARFCAT: Fast code analysis for defects and vulnerabilities". In: *2015 IEEE 1st International Workshop on Software Analytics*. IEEE. 2015, pp. 35–38.

[23] D. Molnar, X. Li, and D.A. Wagner. "Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs". In: *USENIX Security Symposium*. Vol. 9. 2009, pp. 67–82.

[24] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. "Convolutional neural networks over tree structures for programming language processing". In: *Thirtieth AAAI conference on artificial intelligence*. 2016.

[25] J. Mouawad. *F.A.A. Orders Fix for Possible Power Loss in Boeing 787*. Ed. by New York Times. Apr. 2015. URL: `https://www.nytimes.com/2015/05/01/business/faa-orders-fix-for-possible-power-loss-in-boeing-787.html?_r=0`.

[26] Y. Moy, N. Bjørner, and D. Sielaff. "Modular bug-finding for integer overflows in the large: Sound, efficient, bit-precise static analysis". In: *Tech. Rep. MSR-TR-2009–57* (2009).

[27] NASA. *Java Pathfinder*. `https://github.com/javapathfinder/jpf-core`. 2005.

[28] N. Nethercote and J. Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100.

[29] Y. Pang, X. Xue, and A. Namin. "Predicting vulnerable software components through n-gram analysis and statistical feature selection". In: *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE. 2015, pp. 543–548.

[30] *PMD*. `https://github.com/pmd/pmd`.

[31] M. Pomonis, T. Petsios, K. Jee, M. Polychronakis, and A.D. Keromytis. "IntFlow: improving the accuracy of arithmetic error detection using information flow tracking". In: *Proceedings of the 30th Annual Computer Security Applications Conference(ACSAC 2014)*. 2014, pp. 416–425.

[32] M. Pradel and K. Sen. "Deepbugs: A learning approach to name-based bug detection". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–25.

[33] 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.

[34] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". In: *arXiv preprint arXiv:1910.01108* (2019).

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

[36] 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.

[37] A. Shabtai, R. Moskovitch, Y. Elovici, and C. Glezer. "Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey". In: *information security technical report* 14.1 (2009), pp. 16–29.

[38] S. Sidiroglou-douskos, E. Lahtinen, and M. Rinard. *Automatic discovery and patching of buffer and integer overflow errors*. Tech. rep. 2015.

[39] S. Sidiroglou-Douskos, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and M. Rinard. "Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement". In: *Proceedings of the Twentieth International Confer-*

*ence on Architectural Support for Programming Languages and Operating Systems.* 2015, pp. 473–486.

[40] *Spotbugs.* `https://github.com/spotbugs/spotbugs`.

[41] H. Sun, X. Zhang, Y. Zheng, and Q. Zeng. “Inteq: recognizing benign integer overflows via equivalence checking across multiple precisions”. In: *Proceedings of the 38th International Conference on Software Engineering.* 2016, pp. 1051–1062.

[42] 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 Transactions on Information Forensics and Security* 16 (2020), pp. 1943–1958.

[43] 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.

[44] T. Wang, C. Song, and W. Lee. “Diagnosis and Emergency Patch Generation for Integer Overflow Exploits”. In: *Detection of Intrusions & Malware & Vulnerability Assessment (DIMVA 2014).* 2014, pp. 255–275.

[45] T. Wang, T. Wei, Z. Lin, and W. Zou. “IntScope: Automatically Detecting Integer Overflow Vulnerability In X86 Binary Using Symbolic Execution”. In: *Network & Distributed System Security Symposium (NDSS2009).* 2009.

[46] H. Wei and M. Li. “Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code.” In: *IJCAI.* 2017, pp. 3034–3040.

[47] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. “Deep learning code fragments for code clone detection”. In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE. 2016, pp. 87–98.

[48] R. Wojtczuk. “UQBTng: a tool capable of automatically nding integer overows in Win32 binaries”. In: *Proceedings of Chaos Communication Congress* (2005).

[49] F. Yamaguchi, F. Lindner, and K. Rieck. “Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning”. In: *Proceedings of the 5th USENIX conference on Offensive technologies.* 2011, pp. 13–13.

[50] B. Zhang, C. Feng, B. Wu, and C. Tang. “Detecting integer overflow in Windows binary executables based on symbolic execution”. In: *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD).* 2016, pp. 385–390.

[51] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. “A novel neural source code representation based on abstract syntax tree”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* IEEE. 2019, pp. 783–794.

[52] Y. Zhang, X. Sun, Y. Deng, L. Cheng, S. Zeng, Y. Fu, and D. Feng. *Improving Accuracy of Static Integer Overflow Detection in Binary.* Springer International Publishing, 2015, pp. 247–269.

[53] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks”. In: *Advances in neural information processing systems* 32 (2019).

[54] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin. “$\mu$VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection”. In: *IEEE Transactions on Dependable and Secure Computing* (2019).