Detecting Runtime Exceptions by Deep Code Representation Learning with Attention-Based Graph Neural Networks

Rongfan Li, Bihuan Chen, Fengyi Zhang, Chao Sun, Xin Peng
School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China

***Abstract*—Uncaught runtime exceptions have been recognized as one of the commonest root causes of real-life exception bugs in Java applications. However, existing runtime exception detection techniques rely on symbolic execution or random testing, which may suffer the scalability or coverage problem. Rule-based bug detectors (e.g., SpotBugs) provide limited rule support for runtime exceptions. Inspired by the recent successes in applying deep learning to bug detection, we propose a deep learning-based technique, named DREX, to identify not only the types of runtime exceptions that a method might signal but also the statement scopes that might signal the detected runtime exceptions. It is realized by graph-based code representation learning with (i) a lightweight analysis to construct a joint graph of CFG, DFG and AST for each method without requiring a build environment so as to comprehensively characterize statement syntax and semantics and (ii) an attention-based graph neural network to learn statement embeddings in order to distinguish different types of potentially signaled runtime exceptions with interpretability. Our evaluation on 54,255 methods with caught runtime exceptions and 54,255 methods without caught runtime exceptions from 5,996 GitHub Java projects has indicated that DREX improves baseline approaches by up to 18.2% in exact accuracy and 41.6% in F1-score. DREX detects 20 new uncaught runtime exceptions in 13 real-life projects, 7 of them have been fixed, while none of them is detected by rule-based bug detectors (i.e., SpotBugs and PMD).**

***Index Terms*—runtime exceptions, deep learning**

I. INTRODUCTION

Exceptions are abnormal or unexpected events that occur at runtime and disrupt the normal flow of a program's execution, and hurt the program's robustness [21, 32]. To facilitate the development of robust programs against exceptions, most modern programming languages provide built-in exception handling mechanisms to signal, propagate and handle exceptions [31, 43]. Exception handling mechanisms are implemented differently across different programming languages. In particular, the exception handling mechanism implemented by Java distinguishes three types of exceptions, i.e., checked exceptions, runtime exceptions and errors; and runtime exceptions and errors are collectively known as unchecked exceptions. Checked exceptions potentially raised in a method are required to be either locally handled in method body, or explicitly declared on method signature. The Java compiler will signal a compilation error if this requirement is not satisfied. Unchecked exceptions are not subject to this requirement. Thus, uncaught checked exceptions will be detected at compile time and will not occur at runtime, whereas uncaught unchecked exceptions will not be detected at compile time but will occur at runtime.

Previous empirical studies on exception handling have recognized uncaught exceptions as one of the most common root causes of real-life exception bugs in Java applications [6, 13, 17, 18, 25, 72]. Similar empirical evidences have also been reported in Android applications which reuse Java's exception handling mechanism [16, 47, 67]. In fact, such uncaught exceptions are all unchecked exceptions, which can cause serious consequences such as denial-of-service, especially for server applications that should provide long-running services. Hence, it is important to automatically detect runtime exceptions so that developers can fix the bug or catch the exception without crashing the application. As errors are often caused by external problems that are difficult to detect, we do not consider errors.

Some attention has been paid to unchecked exception detection. Kadar et al. [46] use symbolic execution to systematically explore execution paths of each method to detect runtime exceptions, and Csallner et al. [22] and Wu et al. [96] use random testing on methods to heuristically detect runtime exceptions. The former may suffer the scalability problem due to the limitation of symbolic execution [5], while the latter might incur high time overhead and suffer the coverage problem [74]. On the other hand, rule-based bug detectors (e.g., FindBugs [41], PMD [19], Google's Error Prone [2] and Facebook's Infer [40]) are very efficient, but provide limited rule support for runtime exceptions [37].

Inspired by recent successes in deep learning-based bug detection [15, 38, 53, 55, 69, 88, 101, 106], we propose a deep learning-based technique, named DREX, to efficiently and effectively detect runtime exceptions via automatically learned semantic features of runtime exceptions. The goal of DREX is to detect not only the types of runtime exceptions that a method may signal, but also the statement scopes that might signal the detected runtime exceptions. To achieve this goal, the existing deep learning-based bug detection techniques are limited for two main reasons. First, except for [53, 106], they represent source code as a flat sequence of tokens or AST (abstract syntax tree) nodes, which limits the capability to learn semantic features with respect to data and control dependencies. However, such dependencies are important for runtime exception detection. Second, except for [101], they all work at a coarse-grained level by detecting whether a file or method is buggy. As a result, their code representation learning is designed to embed files or methods into vectors, which is not applicable to our fine-grained statement-level detection and also lacks interpretability.

To collect sufficient data for enabling deep learning-based runtime exception detection, we extract and investigate *try-catch* blocks in 5,996 GitHub Java projects which were created after 2013 and have more than 20 stars. Of the 5,745,434 non-test methods in these projects, 54,255 (0.9%) methods contain 63,513 runtime exception *try-catch* blocks, covering 1,008 runtime exception types, mostly inherited from JDK's 83 runtime exception types to add customized exception information. We trace these project-customized runtime exception types back to the JDK's runtime exception types where they are inherited, and obtain the top eight frequently-caught runtime exception types and regard the other types as *RuntimeException*. Thus, we can formulate runtime exception detection as a multi-class classification problem, and regard the statements in *try* blocks as the statement scopes that may signal runtime exceptions.

To overcome the limitations of existing deep learning-based bug detectors, we propose a graph-based code representation to combine CFG (control flow graph), DFG (data flow graph) and AST of a method into a joint graph so that both semantic dependencies among statements and syntactic structures of statements are comprehensively characterized. To facilitate the construction of joint graphs and enable learning from "big code", we propose a lightweight analysis to construct CFG and DFG with type information at the statement level without requiring a build environment. Based on this graph representation, we propose an attention-based graph neural network by combining GGNN (gated graph neural network) [54] and GAN (graph attention network) [83]. In our code representation learning, GGNN allows our model to embed each node into three vectors by propagating and aggregating information of neighboring nodes according to different edge types (i.e., CFG edge, DFG edge, and AST edge). GAN allows our model to weigh the importance of neighboring nodes in runtime exception detection, and models the interpretability. Based on the weighed vector of each statement node, we adopt a single-layer neural network to detect the runtime exception types that the statement may potentially signal. Our code representation learning advances the state-of-the-art by providing fine-grained statement embedding based on weighed syntactic and semantic information.

We have conducted a set of experiments to evaluate the effectiveness and efficiency of the proposed approach on a data set with 54,255 methods with caught runtime exceptions and 54,255 methods without caught runtime exceptions. First, we compare DREX with several baseline approaches. The result demonstrates that DREX achieves an exact accuracy of 60.0% and an F1-score of 78.4%, while improving the baselines by up to 18.2% and 41.6%; and GAN and type information used in DREX bring the highest gain, i.e., 16.9% in exact accuracy and 7.3% in F1-score, and 8.3% in exact accuracy and 10.8% in F1-score, respectively. Second, we measure the time overhead of DREX. The result indicates that our approach takes 16.0 hours to train the model, and 2.71 seconds to construct the graph and 0.26 seconds to identify runtime exceptions for each method. Finally, we manually analyze 50 methods with uncaught runtime exceptions detected by DREX, successfully write test cases to trigger 20 uncaught runtime exceptions in 13 real-life projects,

```
1  public boolean equals(Object other) {
2      try {
3          if (other == null) {
4              return false;
5          }
6          return d.equals(((BigReal) other).d);
7      } catch (ClassCastException cce) {
8          return false;
9      }
10 }
```

Fig. 1: A Motivating Example of *ClassCastException*

and 7 of them have been confirmed and fixed by developers. None of the 20 uncaught runtime exceptions are detected by rule-based bug detectors (i.e., SpotBugs and PMD).

In summary, this work makes the following contributions.

- We combine CFG, DFG and AST into a joint graph to comprehensively characterize both statement syntax and statement semantics, and develop a lightweight analysis to construct the graph without requiring a build environment.
- We propose an attention-based graph neural network by combining GGNN and GAN to detect the type and statement scope of uncaught runtime exceptions with interpretability.
- We conduct experiments to show the improvement of DREX over baseline approaches and the capability in detecting 20 new uncaught runtime exceptions in real-life projects.

II. PRELIMINARIES, MOTIVATIONS AND DATA COLLECTION

In this section, we explain the exception handling mechanism in Java, illustrate motivating examples on runtime exceptions, and present our data collection of runtime exceptions.

A. Exception Handling Mechanism in Java

Java has three basic categories of exceptions: checked exceptions (i.e., *Exception*), runtime exceptions (i.e., *RuntimeException*), and errors (i.e., *Error*). All JDK's exceptions are inherited from the three exceptions. Developers have the flexibility to extend any of the JDK's exceptions. Runtime exceptions and errors are also known as unchecked exceptions. We focus on runtime exceptions as checked exceptions will not occur at runtime and errors are caused by external problems.

Exceptions are handled by *try-catch* constructs. Specifically, in the block *try* $\{S\}$ *catch* $(E\ e)$ $\{H\}$, the *try* block guards a set of statements S from occurrences of exceptions flowing out of the block. S is also called the statement scope of an exception. This scope has a direct impact on the robustness of a program, and if scopes are not properly defined in each program location, the possible actions in the associated handlers are significantly constrained [43]. The *catch* block defines a handler for an exception instance e of the type E (and sub-types of E) with a set of statements H (i.e., actions to handle the caught exception).

B. Motivating Examples on Runtime Exceptions

Figure 1 presents an example of *ClassCastException* in the method *equals* of the project *Apache Commons Math*. This method determines whether the current object of type *BigReal* equals to the passed argument *other* of type *Object*. At Line 6, it first casts *other* to type *BigReal*, and then checks whether the member variable d of the current object equals to the member

```
1  public static FSA loadFSA(File file, Query query) throws IOException {
2      log(utilsLogger, query, "Loading FSA file");
3      String filePath = null;
4      try {
5          filePath = file.getAbsolutePath();
6      } catch (SecurityException e1) {
7          error(utilsLogger, query, "No read access for the FSA file");
8          throw new IOException("No read access for the FSA file");
9      }
10     FSA fsa = loadFSA(filePath, query);
11     return fsa;
12 }
```

Fig. 2: A Motivating Example of *SecurityException*

variable *d* of the casted *other*. Due to the polymorphism mechanism in Java, this method can accept an argument of any type, leading to a *ClassCastException* if *other* is not an instance of *BigReal*. Therefore, a *try-catch* block is added to handle the buggy cast expression. From this example, we can observe that a comprehensive semantic understanding of a method is needed to detect runtime exceptions; e.g., the parameter *other* is used (i.e., data flow information) in the cast expression (i.e., syntax information) without any type validation in the path (i.e., control flow information) to the cast expression. Therefore, existing deep learning-based bug detectors (e.g., [15, 38, 55, 69, 88]) that only rely on source tokens or AST nodes can have a low accuracy. We can also observe from this example that a large amount of semantic information can be extracted among statements in a method, which contributes to runtime exceptions differently and needs to be explicitly weighed to distinguish and interpret the statements that signal or do not signal runtime exceptions. Therefore, existing deep learning-based bug detectors (e.g., [53, 106]) that use CFG and DFG but fail to weight their importance can have a low recall.

Figure 2 illustrates an example of *SecurityException* in the method *loadFSA* of the project *Vespa*. This method loads the FSA file by calling an overloaded method *loadFSA* at Line 10, which might throw an *IOException* that is not handled but declared at the method signature at Line 1. To prepare the first argument for calling the overloaded method, it calls the method *getAbsolutePath* on the argument *file* of type *File*. However, this method invocation may throw a *SecurityException* if a required system property value (e.g., read permission) cannot be accessed, because the file indicated by the argument *file* can be located anywhere in the system. Similar observations from the previous example can be made from this example, and we can further observe that runtime exceptions from third-party library APIs can be easily missed since developers could be unfamiliar with third-party library APIs and runtime exceptions in third-party library APIs are often not documented [16, 47]. As such runtime exceptions can be deeply hidden, it is non-trivial for approaches based on symbolic execution or random testing (e.g., [22, 46]) to trigger them.

Motivated by these observations, we propose a deep learning-based technique to detect both types and scopes of runtime exceptions, which is built upon (i) a graph-based code representation that combines CFG, DFG and AST to comprehensively capture code syntax and semantics (see Section III-B) and (ii) an attention-based graph neural network that learns fine-grained statement embedding based on weighed syntactic and semantic information (see Section III-C).

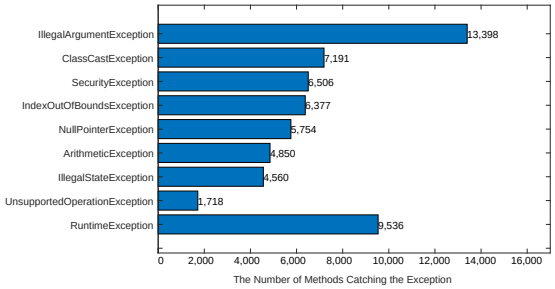


Fig. 3: Frequently Caught Runtime Exception Types

C. Data Collection of Runtime Exceptions

To enable our deep learning-based runtime exception detector, we need a large number of code samples that signal runtime exceptions and code samples that do not. It is straightforward to collect code samples that signal runtime exceptions because open-source projects use *try-catch* blocks to handle potential runtime exceptions; i.e., we can parse the *try-catch* blocks in open-source projects to collect methods with runtime exceptions. To this end, we choose the GitHub Java projects that were created after 2013 and have more than 20 stars. These criteria are designed to balance the sample size and the sample quality, which restrict our selection to 5,996 projects. We use JavaParser [76] to parse runtime exception *try-catch* blocks in each non-test method of these projects. Test methods are excluded to only focus on *try-catch* behaviors in production methods. 5,745,434 non-test methods are analyzed, and 54,255 (0.9%) methods contain a total of 63,513 runtime exception *try-catch* blocks. Only 80 of these runtime exception *try-catch* blocks catch more than one runtime exception type.

All these blocks cover 1,008 runtime exception types. JDK has a total of 83 runtime exception types, 21 of which are caught in 60,928 *try-catch* blocks in 51,932 methods. The other 987 runtime exception types are inherited from JDK's runtime exception types to add customized exception information, and they are caught in 2,665 *try-catch* blocks in 2,439 methods. Actually, project-customized runtime exception types have the same nature of the JDK's runtime exception types where they are inherited. Therefore, to collect sufficient samples for each runtime exception type, we trace project-customized runtime exception types back to JDK's runtime exception types where they are inherited. Finally, we obtain eight frequently-caught runtime exception types, as reported in Figure 3. The most frequently caught runtime exception type is *IllegalArgumentException*, which is caught in 13,398 methods. It indicates that an illegal or inappropriate argument has been passed to a method call. For the other less frequently caught runtime exception types in 17.6% of methods, we treat them as *RuntimeException*.

Then, for each of the 54,255 methods that signal runtime exceptions, we randomly select from its residing file a method that does not signal runtime exceptions. Thus, we collect 54,255 methods as the samples that do not signal runtime exceptions.

III. Methodology

In this section, we first present an overview of DREX, and then introduce each step of DREX in detail.

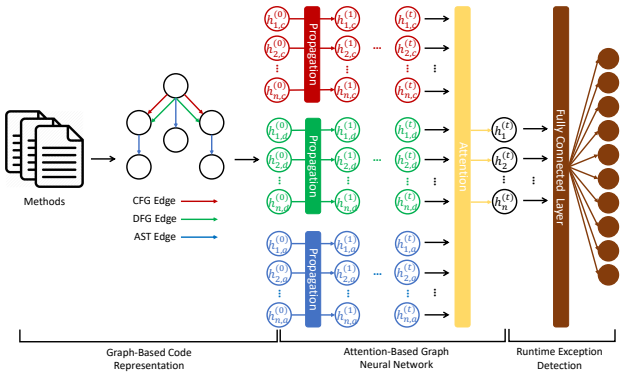


Fig. 4: Approach Overview of DREX

A. Approach Overview

We formulate runtime exception detection as a multi-class (i.e., ten classes, nine classes for the nine runtime exception types in Figure 3 and one class for no runtime exception) classification problem at the statement level. The classifier is learned from our method samples that signal and do not signal runtime exceptions from a large corpus of open-source projects (see Section II-C). We regard the statements in runtime exception *try* blocks as the statement scopes that may signal runtime exceptions. Figure 4 presents an overview of our approach, which consists of the following three main steps.

- **Graph-Based Code Representation.** For each method from an open-source project, we use our lightweight analysis to construct the CFG (control flow graph) and DFG (data flow graph) at the statement level without the need of a build environment, and combine the CFG, DFG and AST (abstract syntax tree) into a joint graph to comprehensively characterize the semantic dependencies among statements and the syntactic structures of statements. Such semantic and syntactic knowledge can be used to distinguish whether a statement signals specific types of runtime exceptions.
- **Attention-Based Graph Neural Network.** With our graph representation, we adopt a 3-layer attention-based graph neural network[1] to learn statement embedding with weighed semantic and syntactic knowledge. In each layer, we first adopt GGNN (gated graph neural network) [54] to embed each node into three vectors through propagating and aggregating information of neighboring nodes following three edge types (i.e., CFG, DFG and AST edges), and then adopt GAN (graph attention network) [83] to weigh the importance of neighboring nodes in exception detection, generate a vector representation for each node, and provide model interpretability.
- **Runtime Exception Detection.** With the weighed vector of each statement node, we adopt a single-layer neural network as the classifier to decide whether each statement signals runtime exceptions and what runtime exception types it signals.

B. Graph-Based Code Representation

As motivated in Section II-B, a comprehensive semantic understanding of a method is needed to determine whether the method signals runtime exceptions. However, the extraction of such semantic information often relies on heavyweight program analysis techniques which often require a build environment, greatly hindering the data collection. Therefore, we propose a graph-based code representation that combines CFG, DFG and AST and is constructed in a lightweight way at the statement level. The graph $\mathcal{G}$ of each method has two types of nodes (i.e., statement nodes $\mathcal{V}_s$ and AST nodes $\mathcal{V}_a$) and three types of edges (i.e., CFG edges $\mathcal{E}_c$, DFG edges $\mathcal{E}_d$, and AST edges $\mathcal{E}_a$). Formally, $\mathcal{G}$ is denoted as a tuple $\langle \mathcal{V}, \mathcal{E} \rangle$, where $\mathcal{V} = \mathcal{V}_s \cup \mathcal{V}_a$ denotes nodes, and $\mathcal{E} = \mathcal{E}_c \cup \mathcal{E}_d \cup \mathcal{E}_a$ denotes edges. The graph is constructed in the following steps. Notice that here we only present the high-level procedure to construct the graph. A detailed implementation has been open-sourced.

1) CFG Construction: As DREX is intended to detect runtime exceptions at the statement level, we construct CFG at the statement level. To this end, we adopt JavaParser [76] to visit each statement by walking through the AST. During the walk, we construct a node for each statement and determine whether the statement belongs to four groups of control flow statements breaking up the execution flow, i.e., decision-making statements (i.e., *IfStmt*, *SwitchStmt* and *SwitchEntryStmt*), looping statements (i.e., *WhileStmt*, *ForStmt*, *ForeachStmt* and *DoStmt*), branching statements (i.e., *BreakStmt*, *ContinueStmt*, *ReturnStmt* and *LabeledStmt*), and exceptional statements (i.e., *TryStmt*). If yes, we construct an edge between the statement and each possible control flow target; otherwise, we construct an edge between the statement and the next statement. There are two exceptions in this process: (i) we skip the CFG construction for the whole method when a statement has parser errors (i.e., *UnparsableStmt* in JavaParser); and (ii) we exclude from the constructed CFG the exceptional statements that catch runtime exceptions in order to remove the explicit indicator of runtime exceptions. After this CFG construction, we construct $\mathcal{V}_s$ and $\mathcal{E}_c$. For example, the four green nodes in Figure 5 represent the statements at Line 1, 3, 5 and 6 in Figure 1. The three red edges represent the control flow edges among statements, and the *if* statement at Line 3 leads to two control flows to Line 4 and Line 6, respectively.

2) AST Construction: For each statement node in the constructed CFG, we use JavaParser to generate its AST subtree, and link the statement node to the root node of the AST subtree. In this way, we combine CFG and AST, and construct $\mathcal{V}_a$ and $\mathcal{E}_a$. For example, as shown in Figure 5, the black nodes represent AST nodes, the blue edges denote AST edges, and each statement node is connected via AST edge to its AST subtree.

3) DFG Construction: Based on the constructed CFG, we determine how data flows along the control flows. Compared with the data flow analysis on intermediate representation of the SSA (static single assignment) form in Soot [81], our data flow analysis at the statement level differs in the variable reaching definition and use analysis, challenged by the various forms of statements. To address this challenge, we maintain a reaching definition set to record the variables that are reachable to the current statement node as well as the statement nodes that define the variables, and define the specific reaching definition and use analysis for each type of statements and expressions

[1]For simplicity, we only show a 1-layer in Figure 4.

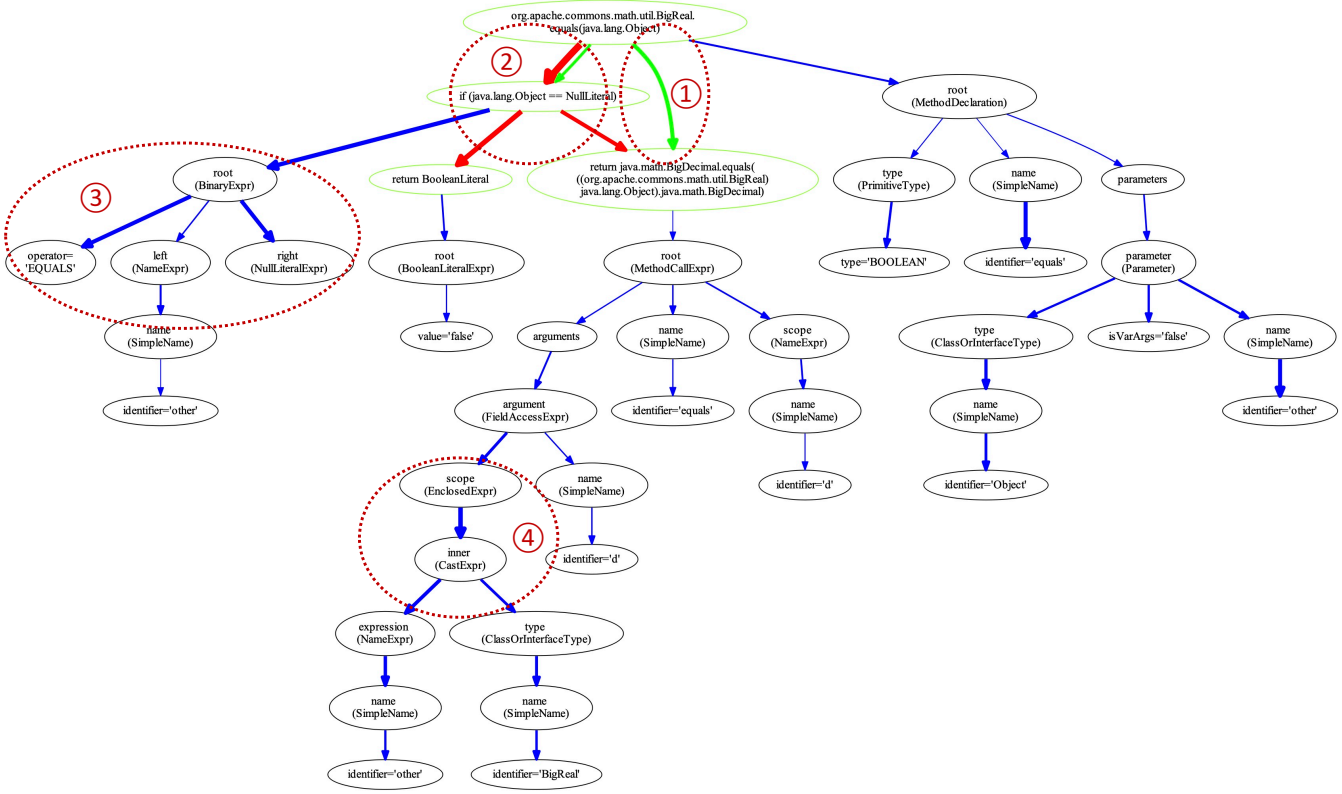


Fig. 5: The Graph Representation of the Method in Figure 1 with Highly Weighted Edges

(i.e., the composing elements of statements) based on its syntactic structure. Basically, for a method declaration, we directly put its parameters into the reaching definition set. Then, for each statement or expression type, we actually know the composing elements that may define or use a variable, and parse the composing element to search for *NameExpr* to determine the variable. If the variable is defined, we update the reaching definition set; and if the variable is used, we search from the reaching definition set the statements that define the variable, and construct DFG edges to those statements. In this way, we construct $\mathcal{V}_d$. For example, as shown in Figure 5, the statement node corresponding to Line 1 in Figure 1 is a *MethodDeclaration*, and we put its parameter *other* into the reaching definition set. The statement node corresponding to Line 3 contains a *BinaryExpr*, and we know that its *left* and *right* expression may use a variable. Then, we parse its *left* expression which is a *NameExpr* of *other*, and thus we construct a DFG edge between the statement nodes corresponding to Line 1 and Line 3, as visualized by the green edge. The DFG edge between Line 1 and Line 6 is similarly constructed.

4) Type Extraction: As type information carries important semantics (e.g., type information is important for *ClassCastException*), we extract type information for literals, variables and classes used in each method. To this end, for each class in a project, we first collect the fully qualified names of its visible classes, i.e., classes in the same package, its imported classes[2] and classes in the package *java.lang*. Then, we determine the fully qualified name of the type of its member variables by matching with its visible classes. Finally, for each method in the class, similar to DFG construction, we parse the expressions in each statement based on the specific syntactic structure. Specifically, if a variable declaration is encountered, we record and replace the type with its fully qualified name by matching with the visible classes and remove the variable name; if a variable reference is encountered, we replace the variable name with the fully qualified name of its type; if a class name is encountered, we replace it with its fully qualified name by matching with the visible classes; and if a literal is encountered, we replace it with the corresponding literal type. For example, as shown in the statement nodes in Figure 5, we add the fully qualified name *org.apache.commons.math.util.BigReal* to the method name, replace the parameter *Object other* with its fully qualified name of the type *java.lang.Object*; and we replace the conditional expression *other == null* with *java.lang.Object == NullLiteral*.

5) Node Embedding Initialization: After constructing the graph of each method, we regard the content in each node as a doc, and apply Doc2Vec [49] over all methods to initialize the node embedding $\boldsymbol{x}_v$ of each node $v \in \mathcal{V}$ for each method.

C. Attention-Based Graph Neural Network

With our graph representation $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ for each method, we develop a 3-layer attention-based graph neural network to learn statement embedding based on recent advances in graph neural networks [97, 105]. Each layer consists of two modules.

1) Propagation Module: As each node in $\mathcal{G}$ is connected to neighboring nodes by CFG, DFG and AST edges, we adopt

[2]We exclude the class (and its enclosing methods) if it has wildcard imports because we cannot resolve types. However, wildcard imports are not used in all our 5,996 projects potentially because of automated supports in IDEs.

GGNN (gated graph neural network) [54] to embed each node into one vector by propagating and aggregating information of neighboring nodes following one edge type. As a result, three vectors are generated for each node following three edge types. Specifically, the propagation works as follows. At time step 1, we initialize the node state vector $\boldsymbol{h}_{v,p}^{(1)}$ for each node v as its initial node embedding $\boldsymbol{x}_v$ (i.e., Eq. 1), where $p \in \{c, d, a\}$ denotes an edge type (i.e., c, d and a respectively denotes CFG, DFG and AST edge). Then, at each time step t, information is propagated and aggregated among neighboring nodes along edges of a specific type independently; i.e., a new state vector $\boldsymbol{f}_{v,p}^{(t)}$ of each node v is generated via aggregating information of its neighboring nodes following edges of type p (i.e., Eq. 2), where $\boldsymbol{A}_{v,p}$ is v's sub-matrix of the graph adjacent matrix $\boldsymbol{A}$ with respect to edge type p, $[;]$ denotes vector concatenation, $\boldsymbol{b}$ denotes learnable parameters, and $\top$ denotes matrix transpose. Then, a gated recurrent unit (GRU) [14] is used to integrate information from v's state vector at the previous time step and v's new state vector at the current time step (i.e., Eq. 3). At the last time step T, the state vector $\boldsymbol{h}_{v,p}^{(T)}$ becomes the node v's final representation $\boldsymbol{h}_{v,p}$ with respect to edge type p.

$$\boldsymbol{h}_{v,p}^{(1)} = \boldsymbol{x}_v \tag{1}$$

$$\boldsymbol{f}_{v,p}^{(t)} = \boldsymbol{A}_{v,p}^{\top}[\boldsymbol{h}_{1,p}^{(t-1)\top}; ...; \boldsymbol{h}_{|\mathcal{V}|,p}^{(t-1)\top}]^{\top} + \boldsymbol{b} \tag{2}$$

$$\boldsymbol{h}_{v,p}^{(t)} = GRU(\boldsymbol{h}_{v,p}^{(t-1)}, \boldsymbol{f}_{v,p}^{(t)}) \tag{3}$$

2) Attention Module: As motivated in Section II-B, a large amount of semantic information can be extracted among statements, which contributes to runtime exceptions differently and needs to be explicitly weighed to distinguish and interpret whether the statements signal runtime exceptions. Therefore, we use GAN (graph attention network) [83] to weigh the importance of neighboring nodes in runtime exception detection following all types of edges, and generate a vector representation for each node with interpretability. Our attention module is a single-layer feedforward neural network. For each node v, it computes the importance (or weight) $\boldsymbol{\alpha}_{vu}^{p}$ of each neighboring node u following edges of a specific type p (i.e., Eq. 4), where $LeakyReLU$ is the activation function, $\mathcal{V}_p(v)$ is v's neighboring nodes connected via edges of type p, and $\boldsymbol{W}$ is learnable parameters. The final node representation $\boldsymbol{h}_v$ is generated via a weighted combination of its neighboring nodes along all types of edges after applying a nonlinearity (i.e., Eq. 5), where σ denotes the nonlinear $sigmoid$ function.

$$\boldsymbol{\alpha}_{vu}^{p} = \frac{exp(LeakyReLU(\boldsymbol{W}[\boldsymbol{h}_{v,p}; \boldsymbol{h}_{u,p}]))}{\sum_{p\in\{c,d,a\}}(\sum_{j\in\mathcal{V}_p(v)} exp(LeakyReLU(\boldsymbol{W}[\boldsymbol{h}_{v,p}; \boldsymbol{h}_{j,p}])))} \tag{4}$$

$$\boldsymbol{h}_v = \sigma(\sum_{p\in\{c,d,a\}} \sigma(\sum_{u\in\mathcal{V}_p(v)} \boldsymbol{\alpha}_{vu}^{p} \boldsymbol{W}^{p} \boldsymbol{h}_{u,p})) \tag{5}$$

Despite the black-box nature of neural networks, our model provides partial interpretability thanks to the attention module; i.e., the computed weights actually allow us to visualize on our constructed graph the importance of various edges to the detected runtime exception. For example, Figure 5 highlights the highly weighted edges, where the width of each highly-weighted edge is proportional to the weight learned by our model. Our model successfully identifies the *ClassCastException* at Line 6 in Figure 1. It can be seen that highly-weighted edges clearly interpret how our model identifies the runtime exception: ① reveals the data flow that the parameter *other* is used at Line 6, ② reveals the control flow to Line 6 and ③ reveals that there is only a null checking (but not type checking) in this control low, and ④ reveals the cast expression at Line 6.

D. Runtime Exception Detection

We formulate runtime exception detection as a multi-class classification problem. Given the representation of a statement node $\boldsymbol{h}_v$ ($v \in \mathcal{V}_s$), its detected runtime exceptions $\tilde{\boldsymbol{y}}_v$ is computed by a classifier realized by a fully connected layer with a $sigmoid$ activation function (i.e., Eq. 6), where $\tilde{\boldsymbol{y}}_v$ is a 10-dimension vector with the first nine dimensions corresponding to the probability of signaling the nine exception types in Figure 3 and the last dimension corresponding to the probability of not signaling any runtime exception, $\boldsymbol{W}$ is learnable parameters, and σ denotes the $sigmoid$ activation function. During training, the loss function is defined as the averaged cross-entropy over all statement nodes (i.e., Eq. 7), where $\boldsymbol{y}_v$ denotes the ground truth detection value for statement node v. During prediction, we regard a statement node as signaling a type of runtime exception if its predicted probability is larger than 0.5. In that sense, it is possible that a statement node may signal multiple types of runtime exceptions.

$$\tilde{\boldsymbol{y}}_v = \sigma(\boldsymbol{W}\boldsymbol{h}_v) \tag{6}$$

$$loss = \frac{-\sum_{v=1}^{|\mathcal{V}_s|}(\boldsymbol{y}_v * log(\tilde{\boldsymbol{y}}_v) + (\boldsymbol{1} - \boldsymbol{y}_v) * log(\boldsymbol{1} - \tilde{\boldsymbol{y}}_v))}{|\mathcal{V}_s|} \tag{7}$$

IV. Evaluation

We have implemented the proposed approach in 12.8K lines of Python and Java code, using PyTorch for deep learning and JavaParser for graph construction. We have released the source code of our approach at https://drex-drex.github.io/ together with the dataset used in our evaluation.

A. Evaluation Setup

To evaluate the effectiveness and efficiency of our approach, we conducted a set of experiments using the dataset collected in Section II-C. We split the dataset into training, validation and testing dataset by 7:1:2, while ensuring the same splitting ratio across various runtime exception types.

Research Questions. Our evaluation is designed to answer the following research questions:

- **RQ1:** What is the effectiveness of DREX in detecting runtime exceptions?
- **RQ2:** What is the contribution of each graph component in DREX to the achieved effectiveness?
- **RQ3:** What is the efficiency of DREX in detecting runtime exceptions?
- **RQ4:** What is the capability of DREX in detecting new uncaught runtime exceptions?

Evaluation Metrics. We adopted four metrics as the indicator of the effectiveness of runtime exception detection. The first metric is *exact accuracy*, i.e., the ratio of methods whose

caught runtime exception types and scopes are all exactly correct. For example, given a method whose third line signals a *NullPointerException*, only a prediction of a *NullPointerException* at the third line is considered as exactly correct. The other three metrics are *precision*, *recall* and *F1-score* at the statement level. For example, for a statement that signals a *NullPointerException*, a prediction of *NullPointerException* and *ArithmeticException* has full recall but low precision.

Training Configuration. We used the Doc2Vec model to convert each node in our graph into a 120-dimensional vector, and then fed the entire graph into our attention-based graph neural network module for three iterations (i.e., 3-layer). Our model is trained via Adam optimizer with the learning rate set to 0.001. Our model was trained on a server machine with a Nvidia GTX 2080Ti GPU and a 3.60GHZ CPU with 16 cores and 128G memory. The training last 16.0 hours with 10 epochs.

B. Detection Effectiveness Evaluation (RQ1 and RQ2)

We compared DREX with the following approaches to evaluate the effectiveness of DREX as well as the contribution of each graph component in DREX to the achieved effectiveness.

- NexGen [101]: It represents code as a sequence of tokens. It uses Bi-LSTM and attention mechanism to obtain token embedding and statement embedding, and uses a binary classifier to predict whether a statement throws an exception. We adapted it to support multi-class classification. It is a representative of the existing code representation learning approaches that regard code as a sequence of tokens [15, 38, 55, 55, 69].
- DREX (CFG): We only use CFG as the graph representation of a method in DREX. It can be seen as a representative of the approaches that represent code as CFG [23].
- DREX (CFG+DFG): We combine CFG and DFG as the graph representation in DREX. It can represent the approaches that treat code as CFG and DFG [8, 104].
- DREX (CFG+DFG+AST) w/o GAN: We remove GAN (i.e., attention module) and assign equal weights to neighboring nodes in DREX. It can be regarded as a representative of the approaches that integrate CFG, CFG and AST [53, 106].
- DREX (CFG+DFG+AST) w/o type: We directly use source code without type information for each statement node.

It is worth mentioning that we cannot directly compare DREX with the existing approaches except for [101] as they do not provide statement-level embedding and detection. Instead, we use the above approaches as representatives of the existing approaches although they are not exactly the same.

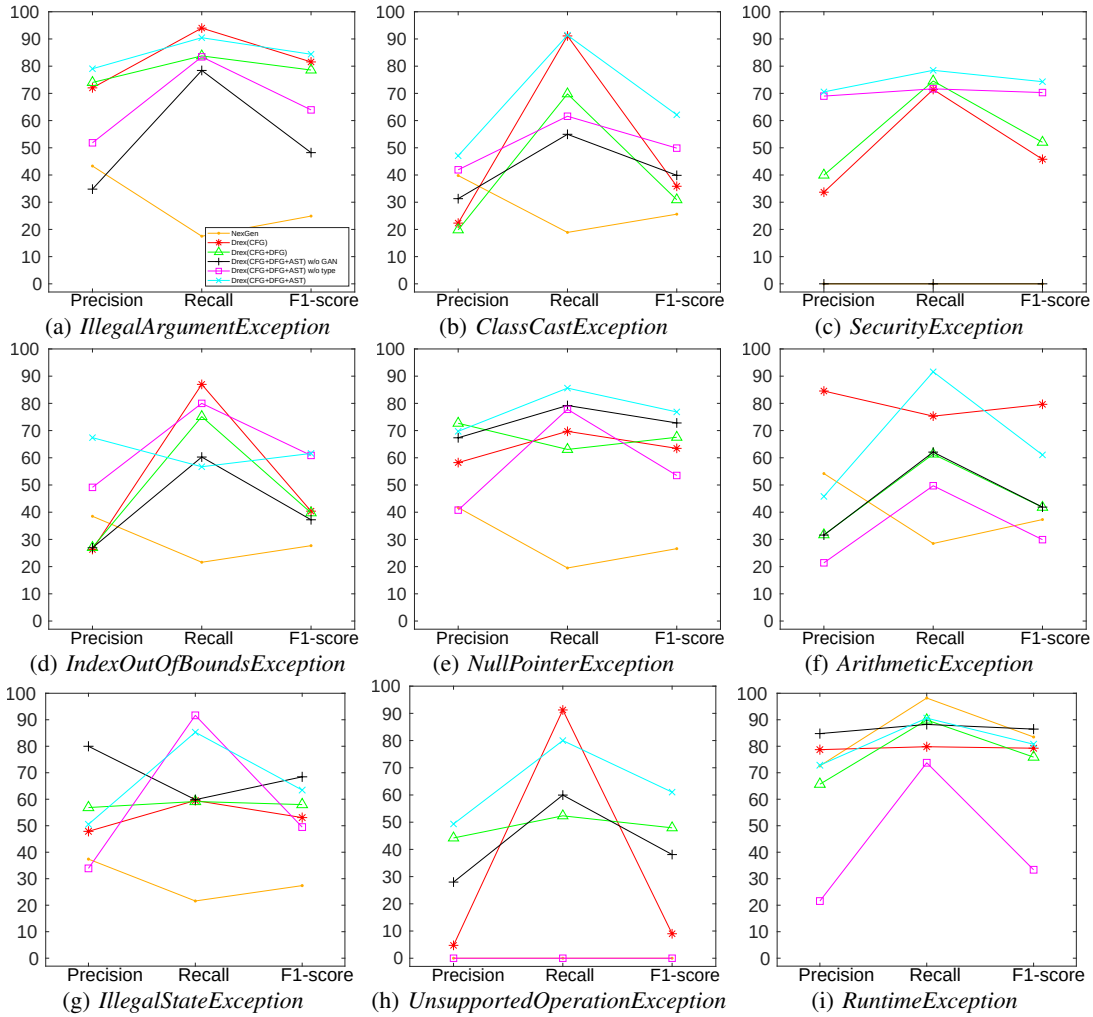
Table I reports the exact accuracy, precision, recall and F1-score of these approaches, where the last row presents the result of DREX in full configuration. Overall, DREX outperformed all these five approaches in all the four metrics, and achieved an exact accuracy of 60.0% and an F1-score of 78.4%. DREX significantly outperformed NexGen by 18.2%, 42.1%, 40.7% and 41.6% in exact accuracy, precision, recall and F1-score. This result indicates that code semantics would be difficult to capture by only considering source code tokens. When we only used CFG, DREX had a degradation of 7.4%, 2.2%, 3.8% and 3.1% in exact accuracy, precision, recall and F1-score, while still outperforming NexGen. When we integrated CFG with DFG, the degradation of DREX in all the four metrics decreased. These results demonstrate that all the three code representations (i.e., CFG, DFG and AST) contribute to the semantic understanding for effective runtime exception detection. Moreover, after we removed GAN, DREX suffered a significant degradation of 16.9% and 11.4% in exact accuracy and recall since the importance of different semantics is not distinguished, making the model have weak capability in distinguish various runtime exception types. When we did not leverage type information, DREX had a significant degradation in all the four metrics. This indicates that type information is important semantics for runtime exception detection.

TABLE I: Effectiveness Comparisons across Approaches

Approach	Exact	Precision	Recall	F1
NexGen	41.8	42.3	32.5	36.8
DREX (CFG)	52.6	82.2	69.4	75.3
DREX (CFG+DFG)	55.7	83.0	72.9	77.6
DREX (CFG+DFG+AST) w/o GAN	43.1	83.5	61.8	71.1
DREX (CFG+DFG+AST) w/o type	51.7	76.0	60.9	67.6
DREX (CFG+DFG+AST)	**60.0**	**84.4**	**73.2**	**78.4**

Moreover, we report the precision, recall and F1-score of these approaches for each of the nine runtime exception types in Figure 6, where the legend is only shown in Figure 6a and omitted in others for clarity. DREX outperformed NexGen in precision, recall and F1-score for all the nine runtime exception types, except for the precision for *ArithmeticException* and recall and F1-score for *RuntimeException*. This clearly demonstrates the advantages of combining CFG, DFG and AST over source code tokens in a comprehensive understanding of code semantics. When we removed AST, or AST and DFG from our graph representation, DREX had a degradation in F1-score for all runtime exception types, except for *ArithmeticException*, while having a slight improvement in precision (resp. recall) but a significant degradation in recall (resp. precision) for some types of runtime exceptions (e.g., *UnsupportedOperationException*). This indicates that a combination of CFG, DFG and AST is needed to achieve the best overall detection effectiveness with balanced precision and recall. Further, GAN (i.e., attention module) had a significant contribution to the detection of all the nine runtime exception types, except for *IllegalStateException* and *RuntimeExceptions*. This is potentially due to their diverse characteristics that are difficult to learn as *IllegalStateException*, denoting that a method has been invoked at an illegal or inappropriate time, could be caused by various reasons, and *RuntimeException* represents all the other less frequent runtime exceptions (see Section II-C). Type information contributed to detect all the nine runtime exception types, except for *IndexOutOfBoundsException* as it is mostly related to specific types (e.g., arrays) and constant values.

DREX outperformed baselines by up to 18.2% in exact accuracy and 41.6% in F1-score. Each code representation contributed to the understanding of code semantics. Both attention module and type information significantly contributed to the achieved effectiveness of DREX.



(a) *IllegalArgumentException* (b) *ClassCastException* (c) *SecurityException*

(d) *IndexOutOfBoundsException* (e) *NullPointerException* (f) *ArithmeticException*

(g) *IllegalStateException* (h) *UnsupportedOperationException* (i) *RuntimeException*

Fig. 6: Precision, Recall and F1-score for Runtime Exception Types (Two Approaches overlap in (c) and (h))

C. Efficiency Evaluation (RQ3)

We measured the efficiency of DREX in terms of training time and detection time. Detection time consists of the time to construct the graph and the time to detect the runtime exceptions using our trained model. As model training can be seen as a one-time job and can be done off-line, detection time is more important. Our model training took around 16.0 hours. On average, the graph construction and runtime exception detection respectively took 2.71 seconds and 0.26 seconds for a method. Thus, about 3 seconds are needed to detect runtime exceptions in a method, which is acceptable for real-world applications.

> Our model training took 16.0 hours. Our end-to-end runtime exception detection took around 3 seconds for each method, which is acceptable for real-world applications.

D. Detected Uncaught Runtime Exceptions (RQ4)

To evaluate the capability of DREX in detecting uncaught runtime exceptions, we randomly selected from our testing dataset 50 methods with uncaught runtime exceptions reported by DREX, and manually analyzed the reported uncaught runtime exceptions at reported statements. We successfully wrote test cases to trigger 20 uncaught runtime exceptions in 20 of the methods, spanning over 13 projects. For the remaining uncaught runtime exceptions, we either failed to write test cases to trigger runtime exceptions due to the complicated encapsulation, or failed to be sure about the intended behavior or business logic of a particular method. Thus, we conservatively considered all the remaining uncaught runtime exceptions as false positives.

Of the 20 uncaught runtime exceptions, 7 have been confirmed and fixed by developers after we submitted the issue. Moreover, we ran SpotBugs (i.e., the successor of FindBugs [41]) and PMD [19] to check whether rule-based bug detectors can detect the 20 uncaught runtime exceptions. It turns out that they can find none of them. These results demonstrate the capability of DREX in detecting uncaught runtime exceptions, which is not covered by rule-based bug detectors.

> DREX detected 20 new uncaught runtime exceptions in 13 real-world projects, and 7 of them have been fixed, while none of them is detected by rule-based bug detectors.

E. Qualitative Study

Apart from the quantitative evaluation in the previous sections, we analyzed several interesting case studies to compre-

```
1  public Set<Member> getMethodUsage(Method method) {
2      return getMembersFromDescriptors(store.get(MemberUsageScanner.class, name(
           method)));
3  }
```

Fig. 7: An Example of Detected Uncaught Runtime Exception

```
1  public void setReference(Record reference) {
2      try {
3          this.key = reference.getField(0, IntValue.class).getValue();
4      } catch (NullPointerException npex) {
5          throw new NullKeyFieldException();
6      }
7  }
8
9  @SuppressWarnings("unchecked")
10 public <T extends Value> T getField(int fieldNum, T target) {
11     // range check
12     if (fieldNum < 0 || fieldNum >= this.numFields) {
13         throw new IndexOutOfBoundsException();
14     }
15     if (target == null) {
16         throw new NullPointerException("The target object may not be null");
17     }
18     // get offset and check for null
19     final int offset = this.offsets[fieldNum];
20     if (offset == NULL_INDICATOR_OFFSET) {
21         return null;
22     }
23     ...
24 }
```

Fig. 8: An Example of Incorrect Detection

```
1  private String shortPath(String path) {
2      try {
3          if (path != null) {
4              if (path.length() == 0) {
5                  return "xDrip Default";
6              }
7              Ringtone ringtone = RingtoneManager.getRingtone(mContext, Uri.parse(
                   path));
8              if (ringtone != null) {
9                  return ringtone.getTitle(mContext);
10             } else {
11                 String[] segments = path.split("/");
12                 if (segments.length > 1) {
13                     return segments[segments.length - 1];
14                 }
15             }
16         }
17         return "";
18     } catch (SecurityException e) {
19         // need external storage permission?
20         checkStoragePermissions("Need permission to access audio files");
21         return "";
22     }
23 }
```

Fig. 9: An Example of Incorrect Detection

hensively illustrate the capability of DREX.

Figure 7 presents one of the 20 uncaught runtime exceptions detected by DREX (see Section IV-D) in the project *Reflections*, which has more than 3.2K stars and over 2.5 million downloads per month from Maven Central. This method tries to obtain all usages of a given *method* through reflection. DREX detected a *RuntimeException*. As *Reflections* claims to support Java 8 at its homepage and lambda expression is one of the most important features introduced in Java 8, we wrote a test case that used the given *method* in a lambda expression, and successfully triggered a *ReflectionsException* inherited from *RuntimeException*. This issue was also experienced by other developers.

Figure 8 and 9 presents two case studies of incorrect detection from Section IV-B. The statement at Line 3 in Figure 8 signals a *NullPointerException* thrown by the invoked method *getFiled* declared at Line 10–24. Although *getFiled* may throw an *IndexOutOfBoundsException* if the parameter *fieldNum* is not in a range, the statement at Line 3 will not signal an *IndexOutOfBoundsException* as it passes 0 to *fieldNum*. However, DREX failed to detect *NullPointerException* potentially because DREX applies intra-procedural analysis and thus fails to extract the semantic knowledge in the called methods. Instead, DREX detected *IndexOutOfBoundsException* because DREX replaces 0 with its type information. However, as shown in Section IV-B, type information significantly contributed to other exception types. We believe such trade-offs are reasonable. This example also motivates the challenge in distinguishing different types of runtime exceptions.

The method in Figure 9 signals a *SecurityException* due to the call to *parse* at Line 7. In fact, this *parse* method opens a file that the program does not have permission to read. However, DREX cannot learn such semantics as it uses intra-procedural analysis. Instead, DREX links its semantics to format parsing or string parsing that often throws *IllegalArgumentException*. As a result, DREX detected an *IllegalArgumentException* for the first three *if* statements. This example indicates that if developers use uncommon words to naming some operations, it will mislead deep learning-based detectors.

F. Discussion

Threats. We relied on manual analysis to analyze detected uncaught runtime exceptions in Section IV-D. To reduce the threat, three of the authors first separately conducted the manual analysis, and then organized a group discussion to review the uncertain cases and conflict cases to reach a consensus. As the three authors are not the developers of the projects where DREX detected uncaught runtime exceptions, we took a conservative strategy (i.e., only when we can write a test case to trigger the uncaught runtime exception) to confirm true cases.

Limitations. Our lightweight analysis is an intra-procedural analysis, and it might hider the semantic understanding of called methods (as shown by the examples in Section IV-E) and negatively impact the accuracy of DREX. One remedy is to encode call graph knowledge into our graph representation; i.e., we can first construct a call graph, and then encode the learned semantic of each method at their callsites. Moreover, for third-party library API calls, we can also use their documentation to enhance their semantic understanding. However, the problem is how to extract such information in a lightweight way to enable learning from "big code".

V. RELATED WORK

Empirical Studies on Exception Handling. Studies have been conducted to understand and characterize exception handling patterns [11, 48, 64, 73], exception handling bugs [6, 13, 17, 18, 24, 25, 25, 72], exception stack traces [16, 28, 47, 67], and mechanism design of exception handling [29, 31, 58, 60, 82, 95]. These studies characterize the common existence of uncaught runtime exceptions, which inspire our work to automatically detect uncaught runtime exceptions.

Exception Detection, Validation and Policies. Several approaches have been proposed to detect uncaught exceptions using static and dynamic analysis techniques. Jo et al. [45] proposed an inter-procedural analysis of Java projects based on set-based analysis to identify uncaught checked exceptions that should be more specific than the already thrown or caught exceptions and to identify unnecessary checked exceptions. Our approach is complementary to this work by detecting uncaught

runtime exceptions. Kadar et al. [46] also detected runtime exceptions, and used symbolic execution on each method to determine branches that throw runtime exceptions. However, this work might not scale to large-scale Java projects due to the well-known scalability issues in symbolic execution [5]. Csallner et al. [22] detected runtime exceptions by automatically generating and executing random tests for each public method in a Java project. Similarly, Wu et al. [96] dynamically extracted system services in Android systems and fuzzed the services so as to expose runtime exceptions. Zhang et al. [103] proposed a dynamic analysis approach to amplify a given set of tests by mocking external resources in each test and exhaustively explore their exceptional behavior space. However, these approaches might incur high time overhead due to executions of the large number of generated tests. Besides, Sinha et al. [75] and Jiang et al. [44] attempted to identify the faulty statement that caused the runtime exception, which could be served as the next complementary step of runtime exception detection.

Zhang et al. [101] proposed a deep learning approach to localize which statement will throw an exception and to generate the catch block. This work focuses on all types of exceptions, while ours focuses on runtime exceptions. In fact, there is no need to use deep learning to localize checked exceptions as they can be accurately detected by modern IDEs. Moreover, this work treats code as token sequences, which may fail to learn semantics.

Another line of work is to validate the correctness and completeness of exception handling code [78, 91, 92, 93, 99]. These approaches aim to validate the business logic of exception handling code (e.g., a resource is not released), while DREX is to identify uncaught runtime exceptions. Domain-specific languages have been proposed to allow software architects and developers to explicitly specify exception handling policies [1, 7, 30, 62] that govern the use of exceptions. These approaches rely on developers to manually specify the exception handling policies, while DREX does not require such human intervention.

Code Representation Learning. Recent advance and success in deep learning has attracted an increased interest in applying deep learning techniques to learn code representations from a massive code corpus for various programming language and software engineering tasks [3, 10, 27, 98], e.g., program property prediction (e.g., [4, 100]), bug prediction and localization (e.g., [51, 71]), bug fixing (e.g., [36, 80]), code generation (e.g., [12, 77]), code completion (e.g., [70, 94]), code search (e.g., [33, 84]), code classification (e.g., [85, 102]), code migration (e.g., [35, 65]), code clone detection (e.g., [79, 102]), API usage search (e.g., [34, 52]), comment generation (e.g., [42, 50]), and commit message generation (e.g., [20, 59]). Except for [8, 23, 39, 79, 85, 86, 104], to the best of our knowledge, all these approaches represent code at the level of tokens and AST, often failing to bridge the gap between code syntax and semantics. As a result, such syntax-level code representations have limited potentiality to capture semantics of uncaught runtime exceptions.

To characterize code semantics, some works [8, 23, 104] represented code as CFG and DFG, and some works [39, 85, 86] represented code as symbolic or concrete execution traces. While execution traces can capture deep and precise program semantics, they are more difficult to obtain than AST, CFG and DFG, which limits their scalability. Tufano et al. [79] empirically demonstrated that different code representations (i.e., tokens, AST, CFG and bytecode) were complementary to each other in the task of clone detection. Following this direction, Wan et al. [84] learned representations of tokens, AST and CFG, and leveraged multi-modal attention to combine these representations for code search. Different from these approaches, we represent code as a joint graph that builds connections among AST, CFG and DFG and combines properties of AST, CFG and DFG, and used attention-based graph neural networks to learn the important semantics of runtime exceptions.

Static Bug Detection. Some pattern-based techniques rely on manually-crafted rules; and typical detectors include FindBugs (and its successor, SpotBugs) [41], PMD [19], Google's Error Prone [2], and Facebook's Infer [40]. They are efficient, but they usually require great manual efforts to create rules for new bug types. Differently, some pattern-based techniques have the capability to automatically mine implicit API usage rules from a large corpus of source code, and identify violations of the mined rules as potential API misuse bugs [9, 26, 56, 57, 61, 63, 66, 68, 87, 89, 90]. They extract rules from API usages without distinguishing whether they are buggy or not, and thus they cannot distinguish buggy API usage from uncommon API usage, leading to high false positives. Differently, we learn semantic features of exceptions from the code with runtime exceptions and the code without runtime exceptions.

Some initial attempts have been recently made to use deep learning techniques to find bugs. Specifically, one line of work is focused on specific bugs [15, 55, 55, 69]. They rely on artificially created and manually labeled training data, and represent code as tokens. In contrast to these approaches, we use training data automatically collected from real-world projects, and use a graph-based code representation to better characterize semantics of runtime exceptions. Another line of work is focused on general-purpose bug detection. Wang et al. [88] and Habib et al. [38] encoded code as AST nodes or tokens, failing to bridge the gap between syntax and semantics. To capture bug semantics, Li et al. [53] and Zhou et al. [106] tried to combine AST, CFG and DFG, but we further extract type information and use attention mechanisms to improve the effectiveness and model the interpretability. More importantly, all these general-purpose bug detection approaches work at a coarse-grained level by predicting whether a source file or method is buggy or not, while ours works at a fine-grained level by predicting which statements have which kinds of runtime exception bugs.

VI. CONCLUSIONS

In this paper, we propose a deep learning-based technique, named DREX, to detect both the types and statement scopes of runtime exceptions that a method may signal in Java projects. Our extensive evaluation has demonstrated promising results of DREX over baseline approaches.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China (Grant No. 61802067). Bihuan Chen is the corresponding author of this paper.

REFERENCES

[1] J. Abrantes and R. Coelho, "Specifying and dynamically monitoring the exception handling policy," in *SEKE*, 2015, pp. 370–374.
[2] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, "Building useful program analysis tools using an extensible java compiler," in *SCAM*, 2012, pp. 14–23.
[3] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys*, vol. 51, no. 4, p. 81, 2018.
[4] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," in *OOPSLA*, 2019, p. 40.
[5] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Survey*, vol. 51, no. 3, pp. 50:1–50:39, 2018.
[6] E. A. Barbosa, A. Garcia, and S. D. J. Barbosa, "Categorizing faults in exception handling: A study of open source projects," in *SBES*, 2014, pp. 11–20.
[7] E. A. Barbosa, A. Garcia, M. P. Robillard, and B. Jakobus, "Enforcing exception handling policies with a domain-specific language," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 559–584, 2016.
[8] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: a learnable representation of code semantics," in *NIPS*, 2018, pp. 3585–3597.
[9] P. Bian, B. Liang, W. Shi, J. Huang, and Y. Cai, "Nar-miner: Discovering negative association rules from code for bug detection," in *ESEC/FSE*, 2018, p. 411–422.
[10] P. Bielik, V. Raychev, and M. Vechev, "Programming with" big code": Lessons, techniques and applications," in *SNAPL*, 2015, pp. 1–10.
[11] J. Bloch, *Effective Java*. Pearson Education India, 2016.
[12] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu, "From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation," in *ICSE*, 2018, pp. 665–676.
[13] H. Chen, W. Dou, Y. Jiang, and F. Qin, "Understanding exception-related bugs in large-scale cloud systems," in *ASE*, 2019, pp. 339–351.
[14] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder–decoder for statistical machine translation," in *EMNLP*, 2014, pp. 1724–1734.
[15] M.-J. Choi, S. Jeong, H. Oh, and J. Choo, "End-to-end prediction of buffer overruns from raw source code via neural memory networks," in *IJCAI*, 2017, p. 1546–1553.
[16] R. Coelho, L. Almeida, G. Gousios, and A. van Deursen, "Unveiling exception handling bug hazards in android based on github and google code issues," in *MSR*, 2015, pp. 134–145.
[17] R. Coelho, A. Rashid, A. Garcia, F. Ferrari, N. Cacho, U. Kulesza, A. von Staa, and C. Lucena, "Assessing the impact of aspects on exception flows: An exploratory study," in *ECOOP*, 2008, pp. 207–234.
[18] R. Coelho, A. Rashid, A. von Staa, J. Noble, U. Kulesza, and C. Lucena, "A catalogue of bug patterns for exception handling in aspect-oriented programs," in *PLoP*, 2008, pp. 1–13.
[19] T. Copeland, *PMD Applied*. Centennial Books Arexandria, Va, USA, 2005.
[20] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *SCAM*, 2014, pp. 275–284.
[21] F. Cristian, "Exception handling and software fault tolerance," *IEEE Transactions on Computers*, vol. 31, no. 6, pp. 531—-540, 1982.
[22] C. Csallner and Y. Smaragdakis, "Jcrasher: An automatic robustness tester for java," *Software: Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.
[23] D. DeFreez, A. V. Thakur, and C. Rubio-González, "Path-based function embedding and its application to error-handling specification mining," in *ESEC/FSE*, 2018, pp. 423–433.
[24] F. Ebert and F. Castor, "A study on developers' perceptions about exception handling bugs," in *ICSM*, 2013, pp. 448–451.
[25] F. Ebert, F. Castor, and A. Serebrenik, "An exploratory study on exception handling bugs in java programs," *Journal of Systems and Software*, vol. 106, pp. 82–101, 2015.
[26] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *SOSP*, 2001, pp. 57–72.
[27] M. D. Ernst, "Natural language is a programming language: Applying natural language processing to software development," in *SNAPL*, 2017, pp. 1–14.
[28] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *ICSE*, 2018, pp. 408–419.
[29] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, and C. M. F. Rubira, "Exceptions and aspects: The devil is in the details," in *FSE*, 2006, pp. 152–162.
[30] J. L. M. Filho, L. Rocha, R. Andrade, and R. Britto, "Preventing erosion in exception handling design using static-architecture conformance checking," in *ECSA*, 2017, pp. 67–83.
[31] A. F. Garcia, C. M. Rubira, A. Romanovsky, and J. Xu, "A comparative study of exception handling mechanisms for building dependable object-oriented software," *Journal of Systems and Software*, vol. 59, no. 2, pp. 197–222, 2001.
[32] J. B. Goodenough, "Exception handling: Issues and a proposed notation," *Communications of the ACM*, vol. 18, no. 12, pp. 683—-696, 1975.
[33] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *ICSE*, 2018, pp. 933–944.
[34] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *FSE*, 2016, pp. 631–642.
[35] ——, "Deepam: Migrate apis with multi-modal sequence to sequence learning," in *AAAI*, 2017, pp. 3675–3681.
[36] R. Gupta, A. Kanade, and S. Shevade, "Deep reinforcement learning for syntactic error repair in student programs," in *AAAI*, 2019, pp. 930–937.
[37] A. Habib and M. Pradel, "How many of all bugs do we find? a study of static bug detectors," in *ASE*, 2018, p. 317–328.
[38] ——, "Neural bug finding: A study of opportunities and challenges," *arXiv*, vol. abs/1906.00307, 2019.
[39] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps, "Code vectors: Understanding programs through embedded abstracted symbolic traces," in *ESEC/FSE*, 2018, pp. 163–174.
[40] P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NFM*, 2015, pp. 3–11.
[41] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *OOPSLA*, 2004, pp. 92–106.
[42] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred API knowledge," in *AAAI*, 2018, pp. 2269–2275.
[43] B. Jakobus, E. A. Barbosa, A. Garcia, and C. J. P. De Lucena, "Contrasting exception handling code across languages: An experience report involving 50 open source projects," in *ISSRE*, 2015, pp. 183–193.
[44] S. Jiang, H. Zhang, Q. Wang, and Y. Zhang, "A debugging approach for java runtime exceptions based on program slicing and stack traces," in *QSIC*, 2010, pp. 393–398.
[45] J.-W. Jo, B.-M. Chang, K. Yi, and K.-M. Choe, "An uncaught exception analysis for java," *Journal of Systems and Software*, vol. 72, no. 1, pp. 59–69, 2004.
[46] I. Kádár, P. Hegedüs, and R. Ferenc, "Runtime exception detection in java programs using symbolic execution," *Acta Cybernetica*, vol. 21, no. 3, p. 331–352, 2014.
[47] M. Kechagia and D. Spinellis, "Undocumented and unchecked: Exceptions that spell trouble," in *MSR*, 2014, p. 312–315.
[48] M. B. Kery, C. Le Goues, and B. A. Myers, "Examining programmer practices for locally handling exceptions," in *MSR*, 2016, p. 484–487.
[49] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *ICML*, 2014, pp. 1188–1196.
[50] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *ICSE*, 2019, pp. 795–806.
[51] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization," in *ISSTA*, 2019, pp. 169–180.
[52] X. Li, H. Jiang, Y. Kamei, and X. Chen, "Bridging semantic gaps between natural languages and apis with word embedding," *IEEE Transactions on Software Engineering*, 2018, to appear.
[53] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, "Improving bug

detection via context-based code representation learning and attention-based neural networks," in *OOPSLA*, 2019, pp. 162:1–162:30.
[54] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in *ICLR*, 2016.
[55] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *NDSS*, 2018.
[56] Z. Li and Y. Zhou, "Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *ESEC/FSE*, 2005, pp. 306–315.
[57] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Cai, "Antminer: Mining more bugs by reducing noise interference," in *ICSE*, 2016, pp. 333–344.
[58] M. Lippert and C. V. Lopes, "A study on exception detection and handling using aspect-oriented programming," in *ICSE*, 2000, p. 418–427.
[59] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," in *ACL*, 2017, pp. 287–292.
[60] R. Miller and A. Tripathi, "Issues with exception handling in object-oriented systems," in *ECOOP*, 1997, pp. 85–103.
[61] M. Monperrus, M. Bruch, and M. Mezini, "Detecting missing method calls in object-oriented software," in *ECOOP*, 2010, pp. 2–25.
[62] T. Montenegro, H. Melo, R. Coelho, and E. Barbosa, "Improving developers awareness of the exception handling policy," in *SANER*, 2018, pp. 413–422.
[63] V. Murali, S. Chaudhuri, and C. Jermaine, "Bayesian specification learning for finding api usage errors," in *ESEC/FSE*, 2017, pp. 151–162.
[64] S. Nakshatri, M. Hegde, and S. Thandra, "Analysis of exception handling patterns in java projects: An empirical study," in *MSR*, 2016, p. 500–503.
[65] A. T. Nguyen, T. D. Nguyen, H. D. Phan, and T. N. Nguyen, "A deep neural network language model with contexts for source code," in *SANER*, 2018, pp. 323–334.
[66] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *ESEC/FSE*, 2009, pp. 383–392.
[67] J. Oliveira, D. Borges, T. Silva, N. Cacho, and F. Castor, "Do android developers neglect error handling? a maintenance-centric study on the relationship between android abstractions and uncaught exceptions," *Journal of Systems and Software*, vol. 136, pp. 1–18, 2018.
[68] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking api protocol conformance with mined multi-object specifications," in *ICSE*, 2012, p. 925–935.
[69] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," in *OOPSLA*, 2018, pp. 147:1–147:25.
[70] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *PLDI pages=419–428, year=2014,*.
[71] E. L. Seidel, H. Sibghat, K. Chaudhuri, W. Weimer, and R. Jhala, "Learning to blame: Localizing novice type errors with data-driven diagnosis," in *OOPSLA*, 2017, pp. 60:1–60:27.
[72] D. Sena, R. Coelho, U. Kulesza, and R. Bonifácio, "Understanding the exception handling strategies of java libraries: An empirical study," in *MSR*, 2016, p. 212–222.
[73] H. Shah, C. Gorg, and M. J. Harrold, "Understanding exception handling: Viewpoints of novices and experts," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 150–161, 2010.
[74] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges," in *ASE*, 2015, pp. 201–211.
[75] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold, "Fault localization and repair for java runtime exceptions," in *ISSTA*, 2009, pp. 153–164.
[76] N. Smith, D. van Bruggen, and F. Tomassetti, "Javaparser: Visited," *Leanpub, oct. de*, 2017.
[77] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation," 2020, to appear.
[78] S. Thummalapenta and T. Xie, "Mining exception-handling rules as
[80] ——, "An empirical investigation into learning bug-fixing patches in sequence association rules," in *ICSE*, 2009, pp. 496–506.
[79] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *MSR*, 2018, pp. 542–553.
the wild via neural machine translation." in *ASE*, 2018, pp. 832–837.
[81] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON*, 1999, pp. 13–.
[82] M. Van Dooren and E. Steegmans, "Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations," in *OOPSLA*, 2005, pp. 455–471.
[83] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," in *ICLR*, 2018.
[84] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. Yu, "Multi-modal attention network learning for semantic source code retrieval," in *ASE*, 2019, pp. 13–25.
[85] K. Wang, "Learning scalable and precise representation of program semantics," *arXiv*, vol. abs/:1905.05251, 2019.
[86] K. Wang, R. Singh, and Z. Su, "Dynamic neural program embeddings for program repair," in *ICLR*, 2018.
[87] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: Bug detection with n-gram language models," in *ASE*, 2016, pp. 708–719.
[88] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *ICSE*, 2016, pp. 297–308.
[89] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," in *ASE*, 2009, pp. 295–306.
[90] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *ESEC/FSE*, 2007, pp. 35–44.
[91] W. Weimer and G. C. Necula, "Finding and preventing run-time error handling mistakes," in *OOPSLA*, 2004, pp. 419–431.
[92] ——, "Mining temporal specifications for error detection," in *TACAS*, 2005, pp. 461–476.
[93] ——, "Exceptional situations and program reliability," *ACM Transactions on Programming Languages and Systems*, vol. 30, no. 2, pp. 1–51, 2008.
[94] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *MSR*, 2015, pp. 334–345.
[95] R. J. Wirfs-Brock, "Toward exception-handling best practices and patterns," *IEEE Software*, vol. 23, no. 5, p. 11–13, 2006.
[96] J. Wu, S. Liu, S. Ji, M. Yang, T. Luo, Y. Wu, and Y. Wang, "Exception beyond exception: Crashing android system by trapping in "uncaughtexception"," in *ICSE-SEIP*, 2017, pp. 283–292.
[97] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.
[98] E. Yahav, "Programming with "big code"," in *APLAS*, 2015, pp. 3–8.
[99] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *OSDI*, 2014, pp. 249–265.
[100] F. Zhang, B. Chen, R. Li, and X. Peng, "A hybrid code representation learning approach for predicting method names," *Journal of Systems and Software*, vol. 180, p. 111011, 2021.
[101] J. Zhang, X. Wang, H. Zhang, H. Sun, Y. Pu, and X. Liu, "Learning to handle exceptions," in *ASE*, 2020, pp. 29–41.
[102] 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 *ICSE*, 2019, pp. 783–794.
[103] P. Zhang and S. Elbaum, "Amplifying tests to validate exception handling code," in *ICSE*, 2012, pp. 595–605.
[104] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *ESEC/FSE*, 2018, pp. 141–151.
[105] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *arXiv preprint arXiv:1812.08434*, 2018.
[106] 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 *NIPS*, 2019, pp. 10 197–10 207.