How About Bug-Triggering Paths? - Understanding and Characterizing Learning-Based Vulnerability Detectors

Xiao Cheng, Xu Nie, Ningke Li, Haoyu Wang, *Member, IEEE*, Zheng Zheng, *Senior Member, IEEE*, and Yulei Sui

Abstract—Machine learning and its promising branch deep learning have proven to be effective in a wide range of application domains. Recently, several efforts have shown success in applying deep learning techniques for automatic vulnerability discovery, as alternatives to traditional static bug detection. In principle, these learning-based approaches are built on top of classification models using supervised learning. Depending on the different granularities to detect vulnerabilities, these approaches rely on learning models which are typically trained with well-labeled source code to predict whether a program method, a program slice, or a particular code line contains a vulnerability or not. The effectiveness of these models is normally evaluated against conventional metrics including precision, recall and F1 score. In this paper, we show that despite yielding promising numbers, the above evaluation strategy can be insufficient and even misleading when evaluating the effectiveness of current learning-based approaches. This is because the underlying learning models only produce the classification results or report individual/isolated program statements, but are unable to pinpoint bug-triggering paths, which is an effective way for bug fixing and the main aim of static bug detection. Our key insight is that a program method or statement can only be stated as vulnerable in the context of a bug-triggering path. In this work, we systematically study the gap between recent learning-based approaches and conventional static bug detectors in terms of fine-grained metrics called BTP metrics using bug-triggering paths. We then characterize and compare the quality of the prediction results of existing learning-based detectors under different granularities. Finally, our comprehensive empirical study reveals several key issues and challenges in developing classification models to pinpoint bug-triggering paths and calls for more advanced learning-based bug detection techniques.

Index Terms—Software vulnerabilities, machine learning, bug-triggering paths, empirical study

1 Introduction

Nowadays software vulnerabilities play a major role in many cyberspace security issues and are increasing at an unprecedented pace. It is challenging yet important to locate and fix these emerging problems in a timely manner during the early stage of the software development cycle, in order to save costs for later software maintenance. Traditional static bug detectors rely heavily on user-defined specifications to discover different types of vulnerabilities, which makes them labor-intensive [1]. The recent success of machine learning techniques has opened up new opportunities to develop effective vulnerability detection techniques without the need of manually defining detection patterns. The existing approaches typically extract unstructured and/or structural code features to train a classification model that captures the correlation between vulnerable code fragments and their (extracted) features [1], [2], [3], [4], [5], [6], [7], [8], [9]. Once the model is trained, it can be used to classify an unseen code fragment as safe or vulnerable.

The underlying classification model of these approaches is typically trained by using pre-labeled source code under different granularities, in order to predict whether a method or a particular line of an input program contains vulnerabilities or not. The effectiveness of a model is measured using standard metrics, e.g., precision, recall and F1 score [10]. However, these standard metrics can be biased when evaluating existing learning-based approaches, because their underlying learning models aim to produce coarse-grained classification results rather than comprehending the semantics of vulnerabilities, whereas a typical static analyzer (e.g., Clang Static Analyzer [11] and Infer [12]) often explains how a bug is generated and triggered by reporting its *bug-triggering path* which contains a chain of program points leading to a bug-triggering point.

Later, several efforts have been made to conduct fine-grained vulnerability detection by learning models over individual statements [13] or using advanced (post-)training techniques (e.g., edge-masking [14], attention mechanism [15] and mutual information maximization [16]) to harvest important statements of the program contributing to the classification results [16], [17], [18]. Unfortunately, the gap still exists

- *Xiao Cheng and Yulei Sui are with the University of Technology Sydney, Ultimo, NSW 2007, Australia. E-mail: xiao.cheng@student.uts.edu.au, yulei.sui@uts.edu.au.*
- *Xu Nie is with the Beijing University of Posts and Telecommunications, Beijing 100876, China. E-mail: ives-nx@bupt.edu.cn.*
- *Ningke Li and Haoyu Wang are with the Huazhong University of Science and Technology, Wuhan, Hubei 430074, China. E-mail: lnk_01@126.com, haoyuwang@hust.edu.cn.*
- *Zheng Zheng is with the Beihang University, Beijing 100191, China. E-mail: zhengz@buaa.edu.cn.*

Manuscript received 13 December 2021; revised 7 July 2022; accepted 16 July 2022. Date of publication 19 July 2022; date of current version 14 March 2024. This work was supported by Australian Research under Grants DP200101328 and DP210101348.
(Corresponding author: Xiao Cheng.)
Digital Object Identifier no. 10.1109/TDSC.2022.3192419

```
libswscale/swscale.c:3191:24: invalidation part of the trace
                        starts when calling `av_malloc` here
3186.   SwsFilter *sws_getDefaultFilter(...
...
3190.   {
3191.>      SwsFilter *filter = av_malloc(sizeof(SwsFilter));
```

```
libavutil/mem.c:56:9: assigned is the null pointer
54.          /* let's disallow possible ambiguous cases */
55.          if(size > (INT_MAX-16) )
56.  >          return NULL;
```

```
libswscale/swscale.c:3197:9: invalid access occurs here
3196.       } else {
3197.>          filter->lumH = sws_getIdentityVec();
3198.           filter->lumV = sws_getIdentityVec();
3199.       }
```

Fig. 1. An example of the detection result by INFER.

between precise static analyzers and these recent fine-grained learning approaches, which report individual statements as vulnerable without pointing out their bug-triggering path/context. This type of report is not only imprecise but also introduce false information for later bug fixing, because these statements can be non-vulnerable under safe program paths. The prior statement-level works are trained and evaluated based on vulnerability patches without leveraging pre-labeled program path information. Despite the training datasets, the code representation learning of these works does not distinguish program paths, which are opaque to backend neural networks. Therefore, the resulting interpretations (e.g., reported vulnerable code lines) may contain disconnected statements which are insufficient or incomplete with regard to the context and semantics of the vulnerability.

In contrast, traditional static analysis techniques (e.g., CLANG STATIC ANALYZER [11], FLAWFINDER [19], INFER [12], ITS4 [20], CHECKMARX [21] and SVF [22], [23]), which approximate the runtime behavior of a program, can provide more informative detection results, e.g., how the bug is generated and triggered. One of their major aims is to pinpoint bug-triggering paths so that developers can quickly locate and fix the reported vulnerabilities. For example, INFER [12], an abduction based industrial-strength static bug detector, can reason about the interprocedural bug trace of the vulnerability by symbolically executing the code and tracing the sequence of statements from the bug-originating location leading to the bug-triggering point using bi-abductive inference [24].

Fig. 1 shows the bug report of a real-world security vulnerability found by INFER. The vulnerable code is extracted from the open-source project, *FFmpeg*[1]. This bug report reveals a null pointer dereference vulnerability triggered on Line 3197 of File "libswscale/swscale.c". INFERnot only provides the bug type information but also highlights the bug-triggering path of this vulnerability: on Line 3191 of File "libswscale/swscale.c", a pointer "filter" is pointed to a memory during the call to "av_malloc()" and dereference on Line 3197, but this pointer could be null according to Line 56 of File "libavutil/mem.c". This can trigger a null dereference and crash the program. Note that the bug-triggering path is produced based on the bi-abductive inference process of INFER. Simply conducting backward slicing from the null dereference point (e.g., Line 3197) to obtain its bug-triggering path is insufficient because the slicing can imprecisely bring in multiple program paths (containing more than 20 lines of code) which include both bug-triggering and safe ones.

Developers can understand how a bug is generated and triggered more easily by following the bug-triggering paths provided by a precise static analyzer like INFER. However, existing learning-based bug detectors only conservatively predict whether particular program methods or code lines contain a vulnerability or not. For example, the learning-based approach REVEAL [25] only reports method "sws_getDefaultFilter" (57 lines of code) as vulnerable. SYSEVR [3], a recent approach that predicts vulnerabilities at the code line level, reports a buggy program lines set of more than 20 lines, while INFERhighlights the key statements related to the vulnerability with only 3 lines. Regarding VELVET [13], ICVH [16], IVDETECT [17] and VULDEELOCATOR [18], the size of the interpretation (statements) can be configured and set manually to a maximum of the entire program method/slice. VELVETand ICVHneed to report the entire method "sws_getDefaultFilter" in order to cover the bug-triggering path. IVDETECT's interpretation of subgraphs in a program dependence graph fails to pinpoint the bug-triggering point Line 3197 until setting the size of the graph to more than 20 nodes (lines). VULDEELOCATORalso takes up to 15 lines to identify the bug-triggering path. We also find that their interpretation contains disconnected code lines without listing their execution order.

Misleadingly, the conventional metrics used to evaluate the learning models can achieve a promising result, because a bug is assumed to be soundly and precisely captured if their classification results cover a pre-labeled program method or line. This evaluation strategy is insufficient and has biases since the reports by the learning-based approaches are challenging to interpret and fail to locate the bug-triggering paths for developers. It is clear that there is a gap between learning-based and traditional bug detectors regarding the quality of their detection results.

This paper aims to conduct a comprehensive empirical study of the gap between the state-of-the-art learning-based approaches and traditional static bug detection techniques in terms of pinpointing bug-triggering paths. To address the evaluation biases, we first summarize and characterize the detection results of eleven recent learning-based bug detection approaches, and then propose fine-grained metrics called BTP metrics using bug-triggering paths to evaluate each vulnerability reported by the learning-based approaches. We also provide a unified toolkit consisting of these eleven approaches (non-publicly available ones are re-implemented by strictly following their methodologies). We conduct a comprehensive and fair comparison using BTP metrics on a large-scale dataset comprising well-labeled programs extracted from real-world mainstream projects [26]. Finally, based on our empirical study and findings, we propose several suggestions and insights to guide future research and help develop better learning-based bug detection approaches.

Our major contributions are as follows:

- We propose quantitative and fine-grained evaluation metrics (BTP metrics) using bug-triggering paths, to

1. https://github.com/FFmpeg/FFmpeg

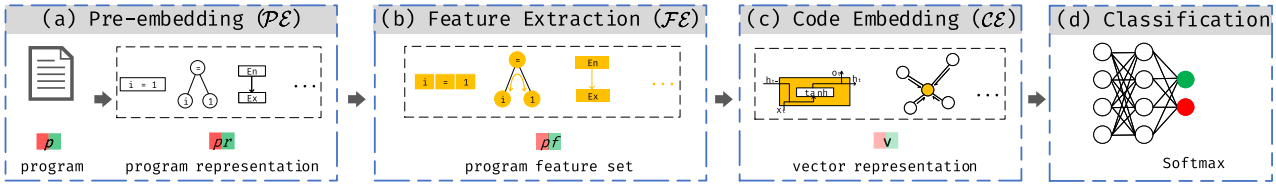

Fig. 2. General training phase of the learning-based vulnerability detection process.

complement traditional evaluation metrics (precision/recall/F1 score).

- We conduct a comprehensive empirical study and comparison on eleven state-of-the-art learning-based bug detectors in terms of the proposed BTP metrics, and analyze the gaps between learning-based solutions and precise static analyzers.
- We perform a comparison between our BTP metrics and traditional evaluation metrics. The experimental results demonstrate the effectiveness of our evaluation methodology.
- Our study reveals several key issues and challenges in developing classification models to pinpoint bug-triggering paths and call for more advanced learning-based bug detection techniques.
- We provide a unified toolkit consisting of eleven recent learning-based approaches (the non-publicly available ones are re-implemented by strictly following their methodologies) available at: https://github.com/tdsc2022-artifact/artifact

2 General Detection Process Formulation

In this section, we formalize the general detection process of current learning-based static vulnerability detection techniques. There are typically two phases (i.e., training and detecting) for these approaches built on top of classification models. Note that, based on the classification model, some approaches (e.g., [13], [16], [17], [18]) further infer important program feature sets contributing to the results and report fine-grained interpretation (e.g., at the statement-level).

2.1 Training Phase

As shown in Fig. 2, this phase consists of the following four steps:

(a) Pre-embedding ($\mathcal{PE}$). The initial step involves using program analysis techniques to generate program representations (i.e., lexical tokens, abstract syntax tree (AST), control flow graph (CFG), data flow graph (DDG) and program dependence graph (PDG)), which is elaborated in Section 3.1. Formally, given a program p, we define the operation as $\mathcal{PE} : p \rightarrow pr$, where pr represents the resulting program representation after pre-embedding.

(b) Feature Extraction ($\mathcal{FE}$). Based on a program representation pr, the features of the program using different heuristic methodologies are extracted to preserve program semantics, formulated as $\mathcal{FE} : pr \rightarrow pf$, where pf represents a set of program features after feature extraction.

(c) Code Embedding ($\mathcal{CE}$). In this step, a learning strategy (e.g., via RNN and GNN) is used to automatically embed program features in the latent embedding space. The program feature set (pf) is first associated with vulnerability information and then transformed into a compact vector. This process is formally expressed as $\mathcal{CE} : pf \rightarrow \mathbf{v}$. Here $\mathbf{v}$ means the vector representation of pf.

(d) Classification. The detection is based on a classification model to predict whether a method/line is vulnerable or not. Generally, one or more linear layers (e.g., MLP [40]) are used for the final prediction model. For the last layer, the predicted distribution of the model $q(lb_i)$ is normally computed using a softmax function, i.e., the dot product between $\mathbf{v}$ and the vector representation $\mathbf{lb}_i$ of each label $lb_i \in lb$.

$$q(lb_i) = \frac{\exp(\mathbf{v} \cdot \mathbf{lb}_i)}{\sum_{lb_j \in lb} \exp(\mathbf{v} \cdot \mathbf{lb}_j)} \tag{1}$$

2.2 Detecting Phase

A well-trained model obtained from the training phase is used to classify the embedding vectors generated from target input programs through $\mathcal{PE}$, $\mathcal{FE}$ and $\mathcal{CE}$. Finally, an unseen sample can be predicted as vulnerable or safe by the model.

3 Characterizing Existing Learning-Based Bug Detectors

To have an in-depth understanding of existing learning-based bug detectors, in this section, we characterize and discuss the internal mechanisms of eleven learning-based approaches by instantiating each of their training phase as generalized in Section 2. We then compare and contrast these approaches using an example.

Table 1 introduces and compares these learning-based approaches. The main notations used to formulate the learning-based approaches are listed in Table 2. Note that, to make a broader comparison, we also include the techniques for code classification (Code2Seq [6]) because code classification tasks can be easily adopted for vulnerability detection. These approaches can be divided into three categories:

- *method-level*: Token Embedding [41], Code2Seq [6], VGDetector [9] and Reveal [25]
- *slice-level*: VulDeePecker [1], SySeVR [3], DeepWukong [36]
- *statement-level*: VELVET [13], ICVH [16], IVDetect [17] and VulDeeLocator [18]

Method-level and slice-level approaches train and perform coarse-grained prediction. They report a program method/slice as vulnerable if it overlaps with the bug-triggering paths, i.e., containing at least one statement on the paths. Existing statement-level approaches report a set of statements as vulnerable or safe, which may be incomplete and disconnected, because their code embedding does not include path information. Hence these approaches are still insufficient when pinpointing bug-triggering paths.

Next, we detail and instantiate each of the four training steps generalized in Fig. 2 using the eleven approaches.

TABLE 1
Learning-Based Detectors

Approaches (year)	Pre-embedding Tool	Feature Extraction		Training and Detecting	
		Feature	Method	Granularity	Model
TOKEN EMBEDDING [2] (2015)	ASTMINER [31]	Lexical tokens	Tokenization	Program method	CNN
CODE2SEQ [6] (2019)	ASTMINER [31]	AST	Path Sampling	Program method	BLSTM&Attention
VGDETECTOR [9] (2019)	JOERN [32]	CFG	Graph Embedding	Program method	DOC2VEC [33]&GCN
REVEAL [24] (2020)	JOERN [32]	CPG	Graph Embedding	Program method	WORD2VEC [34]&GGNN
VULDEEPECKER [1] (2018)	JOERN [32]	DDG	Slicing	Code lines	WORD2VEC [34]&BLSTM
SYSEVR [3] (2021)	JOERN [32]	PDG	Slicing	Code lines	WORD2VEC [34]&BGRU
DEEPWUKONG [35] (2021)	JOERN [32]	PDG	Slicing& Graph Embedding	Code lines	DOC2VEC [33]&GCN
VULDEELOCATOR [18] (2020)	dg [36]	PDG	Slicing	Code lines	Attention&WORD2VEC [34]& BGRU&BLSTM
IVDETECT [17] (2021)	JOERN [32]	PDG	Graph Embedding	Code lines	GloVe [37]&BGRU&Attention& GCN&GNNExplainer [14]
VELVET [13] (2022)	JOERN [32]	CPG	Ensemble Learning [38]	Code lines	WORD2VEC [34]&Attention [15] GGNN&Ensemble Learning [38]
ICVH [16] (2021)	ASTMINER [31]	Lexical Tokens	Tokenization& MIM	Code lines	BLSTM

CNN: Convolutional Neural Network [27], BLSTM: Bidirectional Long Short-Term Memory [28], BGRU: Bidirectional Gated Recurrent Unit [29], GCN: Graph Convolutional Network [30], GGNN: Gated Graph Sequence Neural Network [31], MIM: Mutual Information Maximization [16].

3.1 Pre-Embedding ($\mathcal{PE}$)

The program representation pr extracted from a code fragment can be one of the following:

- *Lexical tokens* are code tokens with identified meanings of a program including identifiers (names specified by developers), keywords, separators (punctuation and delimiters), and operators (pre-defined symbols carrying various operations).
- *AST* (Abstract Syntax Tree) is an abstract syntactic representation of source code using a tree structure, where each leaf node denotes a lexical token occurring in the source code and non-terminal node denotes an abstract construct. It contains more structural and content-related details compared to lexical tokens.
- *CFG* (Control Flow Graph) is a graph representation of all execution paths that might be traversed during a program's execution.

TABLE 2
Main Notations Used in the Paper

Notation	Description
m	Program method
tk	Program token
$N_{tk}(\cdot)$	The number of program tokens
$P_a(\cdot,\cdot)$	The path on the AST between two nodes
st	Statement tree of AST
$N_{st}(\cdot)$	The number of statement trees
S_{BB}	Basic block set
V_p, E_p	Node and Edge set of PDG
V_a	Node set of an AST
E_a, E_c, E_d, E_n	Edges of AST, CFG, DDG and NCS
vc_f	Library/API Function Call
vc	Vulnerability syntax characteristic (vc_f, Array/Pointer Usage, or Arithmetic Expression)
$SL_d(vc)$	Data-dependence slicing starting from vc
$SL_p(vc)$	Program-dependence slicing starting from vc
$SL_p^{IR}(vc)$	Intermediate Representation program-dependence slicing starting from vc
$CA(\cdot)$	Extracting Code attentions

- *DDG* (Data Dependence Graph) captures the def-use data dependence through its edges for each program variable.
- *PDG* (Program Dependence Graph) uses its edges to represent data dependencies or control dependencies, which are computed through the DDG and the CFG of a program.
- *CPG* (Code Property Graph) is a graph combing the information of AST, CFG and PDG.

Note that, for the approaches we study, *Lexical tokens*, *AST*, *CFG* and *CPG* are used in *method-level* representations. *DDG*, *PDG* and *CPG* are used for *slice-level* representations. *Lexical tokens*, *PDG* and *CPG* are used for *statement-level* approaches.

3.2 Feature Extraction ($\mathcal{FE}$)

3.2.1 Method-Level Approaches

TOKEN EMBEDDINGtreats each method m as an individual sample and generates a lexical token sequence $tk_1 \ldots tk_{N_{tk}(m)}$ for each method m to represent its program feature set: $pf = \{tk_i\}_{1 \le i \le N_{tk}(m)}$.

CODE2SEQ uses path-context ($\langle tk_i, P_a(tk_i, tk_j), tk_j \rangle$) as extracted features for each program method and splits code tokens into subtokens (e.g., decomposing "ArrayList" into "Array" and "List") to obtain the feature set pf:

$$pf = \left\{ \left\langle \lfloor tk_i \rfloor, \lfloor P_a(tk_i, tk_j) \rfloor, \lfloor tk_j \rfloor \right\rangle \right\}_{1 \le i,j \le N_{tk}(m), i \ne j} \quad (2)$$

Here $\lfloor \cdot \rfloor$ means the decomposing operation.

VGDETECTOR uses CFGs to embed the execution order of a program. Each node on a CFG is a sequential basic block and each edge represents the execution order between basic blocks. Therefore, we have $pf = (S_{BB}, E_c)$.

REVEAL uses the code property graph consisting of AST, CFG and PDG information: $pf = (V_a, E_a, E_p, E_c)$.

3.2.2 Slice-Level Approaches

VULDEEPECKER first locates bug-related APIs and then extracts the data-flow-related program slices based on the

signatures of these APIs to generate code slices named code gadget by assembling API-related program slices: $pf = SL_d(vc_f)$.

SySeVR first generates the PDG of a program. Each node represents a program statement and each edge represents a data- or control-dependence relation between two statements. It then generates code slices, also called SeVC, to produce the feature set $pf = SL_p(vc)$.

DeepWukong generates a subgraph of PDG, also called XFG, by slicing on PDG starting from vc, to produce the feature set $pf = (V_p^-, E_p^-)$, where $V_p^- \subseteq V_p$ comes from $SL_p(vc)$ and $E_p^- \subseteq E_p$ contains the edges connecting the nodes in V_p^- on PDG.

3.2.3 Statement-Level Approaches

VulDeeLocator conducts control and data-dependency slicing on the Intermediate Representation (IR) of the source code starting from vc, and generates IR code slices (iSeVCs) as the feature set $pf = SL_p^{IR}(vc)$.

IVDetect extracts the program feature set based on the program dependence graph: $pf = (V_p, E_p)$.

VELVET uses the code property graph as its program feature set: $pf = (V_a, E_a, E_p, E_c)$.

ICVHtokenizes the program to $tk_1 \ldots tk_{N_{tk}(m)}$ as its program feature set: $pf = \{tk_i\}_{1 \le i \le N_{tk}(m)}$.

3.3 Code Embedding ($\mathcal{CE}$)

After obtaining the program features pf, the code embedding phase is to produce a vector representation of the collected features, so that the distributed vector representation is used for model training to capture the correlation between features and labeled vulnerabilities.

3.3.1 Method-Level Approaches

Token Embedding first uses a word embedding layer to embed a set of program tokens $tk_1, tk_2, ..., tk_{|pf|}$ to vectors $\mathbf{v}_{tk_1}, \mathbf{v}_{tk_2}, ..., \mathbf{v}_{tk_{|pf|}}$, which are then fed into a CNN model [27]. Formally, given the i^{th} fixed-length window of $k+1$ tokens $tk_i, tk_i+1, ..., tk_{i+k}$, a convolution filter $conv(\cdot)$ is applied to the concatenation of these token vectors, to generate $\mathbf{r}_i = conv([\mathbf{v}_i, \mathbf{v}_{i+1}, ..., \mathbf{v}_{i+k}])$. After this, all $\mathbf{r}_i$ are concatenated to produce the vector representation of the method $\mathbf{v} = [\mathbf{r}_i]_i$.

Code2Seq first uses a $BLSTM$ model [28] to encode the AST path $P_a(tk_i, tk_j)$ in each path-context in pf. The state transition rules of BLSTM are:

$$\begin{aligned}
\mathbf{r}_t &= \sigma(\mathbf{W}_r \cdot \mathbf{v}_t + \mathbf{U}_r \cdot \mathbf{h}_{t-1} + \mathbf{b}_r) \\
\mathbf{z}_t &= \sigma(\mathbf{W}_z \cdot \mathbf{v}_t + \mathbf{U}_z \cdot \mathbf{h}_{t-1} + \mathbf{b}_z) \\
\mathbf{h}'_t &= tanh(\mathbf{W}_h \cdot \mathbf{v}_t + \mathbf{r}_t \odot (\mathbf{U}_h \cdot \mathbf{h}_{t-1}) + \mathbf{b}_h) \\
\mathbf{h}_t &= (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \mathbf{h}'_t
\end{aligned} \tag{3}$$

where $\mathbf{r}_t$ and $\mathbf{z}_t$ represent the reset gate, which can control the influence of the previous state, and the update gate, which combines the previous and current state, respectively. The current state $\mathbf{h}_t$ is determined by $\mathbf{h}'_t$, which is the candidate state, and previous state $\mathbf{h}_{t-1}$ through linear interpolation. $\mathbf{W}_r; \mathbf{W}_z; \mathbf{W}_h; \mathbf{U}_r; \mathbf{U}_z; \mathbf{U}_h$ are the weight matrices and $\mathbf{b}_r; \mathbf{b}_z; \mathbf{b}_h$ are bias terms. The last hidden state is used to represent the AST path. The vector representation of the terminal tokens in the path-context is the sum of all its sub-tokens' word embedding vectors. After this, a fully-connected layer is applied to combine the three parts in the path-context. The attention mechanism is then used to produce the code vector $\mathbf{v} = attn(\{\mathbf{c}'_i\}_{1 \le i \le |pf|})$. The attention mechanism $attn$ is formally expressed as:

$$\alpha_i = \frac{exp(\mathbf{c}'_\mathbf{i} \cdot \mathbf{a})}{\sum_{j=1}^{N} exp(\mathbf{c}'_\mathbf{j} \cdot \mathbf{a})} \qquad \mathbf{v} = \sum_{i=1}^{|pf|} \alpha_i \cdot \mathbf{c}'_\mathbf{i} \tag{4}$$

where $\mathbf{a}$ is a learnable global attention vector.

VGDetector applies GCN [30] to update the feature vector $\mathbf{v}_b$ of each node (basic block). It then uses a global pooling layer $pool$ to aggregate the vector of each basic block to produce $\mathbf{v} = pool(GCN(pf(S_{BB}, E_c)))$. GCN is formally expressed as:

$$\mathbf{F}^{(l)} = \sigma\left(\widetilde{\mathbf{D}}^{-\frac{1}{2}}\widetilde{\mathbf{A}}\widetilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{F}^{(l-1)}\mathbf{W}^{(l)}\right) \tag{5}$$

where $\mathbf{F}^{(l)} = \{\mathbf{f}^{(l)}(0), \mathbf{f}^{(l)}(1), ..., \mathbf{f}^{(l)}(|S_{BB}|)\}$ means the matrix of activations in the l^{th} layer of the neural network. For the first layer, $\mathbf{F}^{(0)} = \{\mathbf{f}^{(0)}(0), \mathbf{f}^{(0)}(1), ..., \mathbf{f}^{(0)}(|S_{BB}|)\}$ is the initial S_{BB} feature matrix, which is generated by Doc2Vec. $\widetilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N$ represents the adjacency matrix of the graph (S_{BB}, E_c) with added self-connections where $\mathbf{I}_N$ is the identity matrix. $\widetilde{\mathbf{D}}_{ii} = \sum_j \widetilde{\mathbf{A}}_{ij}$ and $\mathbf{W}^{(l)}$ is the trainable weight matrix for the l^{th} layer. The pooling layer $pool$ is formulated as follows:

$$\mathbf{v} = \left[\frac{1}{|S_{BB}|}\sum_{i=1}^{|S_{BB}|}\mathbf{f}(i), max_{j=1}^{|S_{BB}|}\mathbf{f}(j)\right] \tag{6}$$

Reveal leverages Word2Vecembedding to encode the code fragment of each vertex and one-hot encoding to encode vertex type information. These vectors are concatenated and fed into Gated Graph Recurrent Layers, to produce a vector for each vertex $\mathbf{v}_i$. The propagation BGRU are expressed as:

$$\begin{aligned}
\overrightarrow{\mathbf{v}_t} &= \overrightarrow{GRU}(\mathbf{v}_{n_t}) \\
\overleftarrow{\mathbf{v}_t} &= \overleftarrow{GRU}(\mathbf{v}_{n_t}) \\
\mathbf{v}_t &= [\overrightarrow{\mathbf{v}_t}, \overleftarrow{\mathbf{v}_t}]
\end{aligned} \tag{7}$$

where GRU is the gated recurrent unit. Finally, a simple element-wise summation is used to aggregate all the vectors: $\mathbf{v} = \sum_{i=1}^{|V_a|} \mathbf{v}_i$.

3.3.2 Slice-Level Approaches

VulDeePecker first uses Word2Vecto encode the tokens of each code gadget to vectors, which are then passed to a $BLSTM$ (Equation (3)). The last hidden state is used to represent the code gadget as $\mathbf{v} = BLSTM(\{\mathbf{v}_{tk_i}\}_{1 \le i \le N_{tk}(pf)})$.

SySeVR first uses Word2Vecto encode the tokens in program slices (called SeVC) to vectors, which are then passed to a $BGRU$ (Equation (7)). The last hidden state is used to represent the SeVC as $\mathbf{v} = BGRU(\{\mathbf{v}_{tk_i}\}_{1 \le i \le N_{tk}(pf)})$.

DeepWukong first uses Doc2Vecto transform the tokens in each node (statement) into their vector representations, which are then fed into a GCN (Equation (5)) and pooling layer (Equation (6)), to produce the XFG vector $\mathbf{v}$.

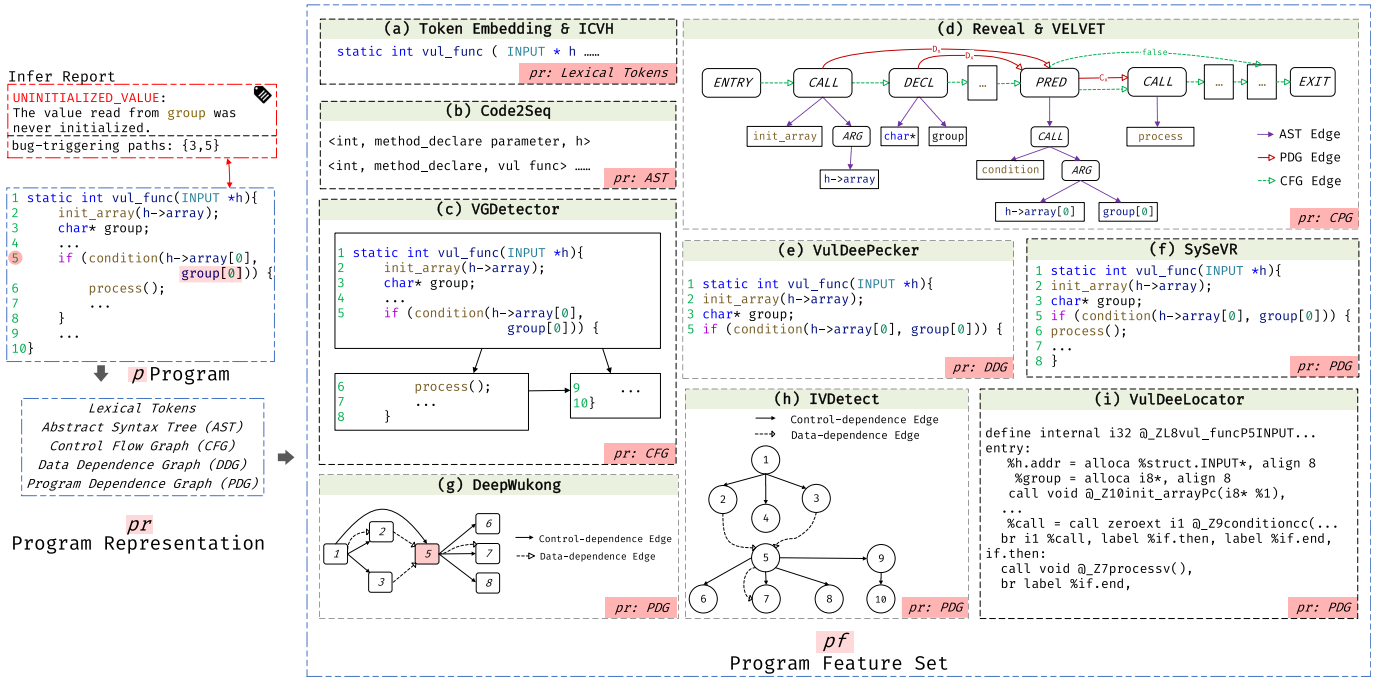

Fig. 3. A real-world security flaw from *FFmpeg* (a multimedia player), which illustrates and compares the feature extraction process of the bug detectors.

3.3.3 Statement-Level Approaches

VulDeeLocator first uses Word2Vecto transform iSeVCs into vectors and feed them into a $BGRU$ (Equation (7)). It then applies a k-max (selecting the top-k values) and average pooling layer to aggregate the hidden states and to produce a vector representation of the iSeVC: $\mathbf{v} = ave(max_k(BGRU(\{\mathbf{v}_{tk_i}\}_{1 \le i \le N_{tk}(pf)})))$.

IVDetect first builds the vector for each statement using a hybrid representation: subtokens, AST subtree, variable names/types and program dependency context. The feature of subtokens, variable names/types and program dependency context are modeled with GloVe [38] and GRU. AST subtree is modeled with a tree LSTM [42]. These vectors are merged using BGRU (Equation (7)) and the attention mechanism (Equation (4)) to produce a statement vector $\mathbf{v}_p$. After this, a GCN layer (Equation (5)) is used to update the statement vector by propagating a message between V_s and a CNN is used to aggregate all the statement vectors into $\mathbf{v}$.

VELVET first uses Word2Vecto produce the initial vector representations of the code tokens in each node of the CPG and use GGNN (same as Reveal) and transformer [15] separately to learn the vector representations $\mathbf{v}$ of each node.

ICVH uses the $BLSTM$ model (Equation (3)) to encode the lexical tokens of a program to produce $\mathbf{v} = BLSTM(tk_1, ..., tk_{|pf|})$.

Once the vector representation $\mathbf{v}$ of the code is available, then classification training can be done via Equation (1) (Section 2). Based on the probability distribution $q(lb)$ of the code, VELVETproduces ensemble scores by averaging the vulnerability scores learned from GGNN and transformer respectively, and select the statements with high ensemble scores as the vulnerability locations. ICVHuses another BLSTM neural network to compute the probability distribution of each statement and train the statement selection parameters θ by maximizing the mutual information of $q(lb)$ and $p_\theta(S|m)$, where S denotes the selected important statements. IVDetectperforms vulnerability interpretation by producing a subgraph of the original PDG using the edge masking technique in GNNExplainer [14]. VulDeeLocatorselects important statements based on the value learned from its k-max layer.

4 A Running Example

Let us take a look at an example to compare and contrast each of the aforementioned approaches, and then discuss the gaps between learning-based techniques and traditional static analyzers in terms of bug finding and reporting.

Fig. 3 illustrates the inputs and outputs of the $\mathcal{FE}$ step of each learning-based detector. The buggy code fragment p (shown on the left) is simplified and extracted from a real-world multimedia player[2]. Different program representations pr of program p are first generated in the $\mathcal{PE}$ stage (Section 3.1). Based on these representations, the $\mathcal{FE}$ stage (Section 3.2) produces program feature sets pf for different learning-based bug detectors, as shown on the right side. After this, each pf is transformed to a compact vector $\mathbf{v}$ through $\mathcal{CE}$ (Section 3.3) for classification training (Section 2).

4.1 Bug-Triggering Paths

The code content and bug report from Inferare shown on the left. On Line 5 (marked in red), this program attempts to read the first element from the array object "group". However, "group" was never initialized before this statement. As reported by Infer, this will trigger an uninitialized value vulnerability. Inferalso precisely provides the bug-triggering paths containing 2 lines, where Line 3 declares "group" and Line 5 triggers the bug.

2. https://www.ffmpeg.org/

4.2 Program Feature Set

TOKEN EMBEDDINGand ICVH(Fig. 3(a)) tokenize method `vul_func` into its lexical tokens (e.g., "static", "int", "vul_func") and treats all these sequential tokens as pf. CODE2SEQ(Fig. 3(b)) extracts the paths on the AST and uses a set of triplets (e.g., "⟨int, [method_declare, parameter], h⟩") to represent pf. Note that "[method_declare, parameter]" stands for $\lfloor P_a(\text{"int"},\text{"h"})\rfloor$. VGDETECTOR(Fig. 3(c)) constructs the CFG of the program method. REVEALand VELVET(Fig. 3(d)) build a code property graph comprising of AST CFG and PDG edges as its pf. VULDEEPECKER(Fig. 3(e)) conducts the data-flow slicing by starting from a Library/API Function Call (i.e., Line 5 in our example). The approach then traverses backward to get the definition or declaration of "h", "h→ array" and "group" on Lines 1, 2 and 3. SYSEVR(Fig. 3(f)) extends the statements with control-dependence-related statements (Lines 6, 7 and 8). DEEPWUKONG(Fig. 3(g)) constructs a subgraph of PDG, where each node represents the control or data-related statement (Lines 1, 2, 3, 5, 6, 7, 8) and each edge represents a control or data-dependence relation between statements. IVDETECT (Fig. 3(h)) utilizes the PDG as pf and conducts the edge-masking technique to find its representative nodes contributing to the prediction. VULDEELOCATOR(Fig. 3(i)) extracts the IR slice starting from Line 5 and uses k-max pooling to find important statements.

4.3 Gaps in Terms of Bug-Triggering Paths

To understand the gap, we assume every learning-based approach can always achieve perfect prediction results under the traditional metrics. Let us compare their best possible predicted classification results with the ground truth.

Method-level detectors (TOKEN EMBEDDING, CODE2SEQ, VGDETECTOR, REVEAL) report the whole 10-line program method as vulnerable (Lines 1-10), compared to only 2 lines for bug-triggering paths (Lines 3, 5). This means that programmers have to examine the entire method to figure out which statements are relevant to the bug. It becomes even worse if a bug happens across multiple program methods.

Slice-level detectors VULDEEPECKER, SYSEVRand DEEPWUKONGunfortunately report all the control- and/or data-related statements in relation to the API call (Line 5): VULDEEPECKERreports 4 lines (Lines 1, 2, 3, 5) and SYSEVRand DEEPWUKONGreport 7 lines (Lines 1, 2, 3, 5, 6, 7, 8). However, they all fail to analyze the fine-grained bug-triggering paths (Lines 3, 5). The program slices used for training are either too generic (i.e., dependence information of all variables/statements) or unaware of bug-triggering paths (i.e., slicing from arbitrary API calls [1]). This not only makes training unnecessarily costly, but also makes the prediction hard to precisely capture even a simple bug-triggering path (e.g., caused by the uninitialized variable "group").

For statement-level detectors ICVH, VELVET, IVDETECTand VULDEELOCATOR, the size of the interpretation (statements) can be specified by users. Theoretically, they can predict and report any program statement as vulnerable. However, the underlying models are unaware of bug-triggering paths. After training the models, when setting the size of the interpretation to 4 statements, ICVHand VELVETboth fail to report Line 3 and Line 5. IVDETECTcan distinguish the bug-triggering point (Line 5) but fails to report Line 3. VULDEELOCATORcan cover the intact bug-triggering path but fails to identify Line 3 when changing the interpretation size to 2 statements. The above observations confirm that there is still a substantial gap between the most sophisticated learning-based approaches and a precise static bug detector.

5 BTP METRICS AND EVALUATION DESIGN

In this section, we discuss the design of a new methodology to quantitatively evaluate the gap in the bug finding capability between traditional bug detectors and learning-based ones in terms of bug-triggering paths. We first introduce the BTP metrics, a simple yet effective concept using bug-triggering paths. Next, we describe our toolkit implementation and the dataset.

5.1 Bug-Triggering Paths

The bug-triggering paths consist of a number of program statements describing the offending execution trace of a bug. Starting from the program point where the bug is originated, a bug-triggering path includes the statements that (1) reside in the execution paths towards the location where the error is triggered and (2) contain variables that are aliased with the variables in the bug-originating and triggering points. Note that a bug-triggering path can go through multiple files and methods because of the inter-procedural nature of many types of bugs. Bug-triggering paths are essential for pinpointing and understanding the triggering logic of vulnerabilities so that developers can follow to fix the bugs more quickly.

5.2 BTP Metrics

Table 3 gives an overview of six existing common evaluation metrics used in the previous studies related with learning-based vulnerability detection. We do not discuss the ranking-based metrics (e.g., mean first ranking [17]) because they are not generalizable and can only be applied to the approaches which yield a probability for each statement. However, all the statements in a bug-triggering path should be treated as equally important.

Existing Metrics and Limitations. Some existing evaluation metrics (precision, recall, F1 score) used by current learning-based approaches only assess the bug finding capability at coarse-grained method/slice levels without understanding the semantic of a vulnerability. A data sample is assumed to be correctly predicted as long as the prediction is consistent with the pre-labeled method/slice. Recent approaches [16], [17], [18] propose fine-grained evaluation metrics based on the correctness of statement-level prediction. Statement accuracy (SA) [17] counts a correct and precise detection as long as one labeled vulnerable statement is reported without considering the proportion of correctly predicted statements. Vulnerability coverage proportion (VCP) [16] considers the ratio of overlapped statements (correctly predicted statements) over true vulnerable statements, while Jaccard index (JI) [18] computes the ratio over the union of true vulnerable statements and reported vulnerable statements. However, these two metrics ignore the prediction correctness of data samples and directly compute over program statements. The influence of small samples' statement-level prediction performance on

TABLE 3
Existing Evaluation Metrics for Evaluating Learning-Based Vulnerability Detection

Metric	Evaluation Granularity	Formula	Description
Precision [10]	Data Sample	$P = \frac{TP}{TP+FP}$	Correctness of detected vulnerable samples
Recall [10]	Data Sample	$R = \frac{TP}{TP+FN}$	Proportion of detected vulnerable samples in all vulnerable samples
F1 Score [10]	Data Sample	$F1 = \frac{2*P*R}{P+R}$	Overall measurement considering both precision and recall
Statement Accuracy [17]	Statement	$SA = \frac{TI}{TP+FP}$	Correctness of reported interpretations
Vulnerability Coverage Proportion [16]	Statement	$VCP = \frac{TVCS}{LVCS}$	Proportion of correctly detected vulnerable statements in labeled vulnerable statements
Jaccard Index [18]	Statement	$JI = \frac{TVCS}{LVCS+FVCS}$	Proportion of correctly detected vulnerable statements in all vulnerable statements

TP denotes the number of vulnerable programs predicted as vulnerable. FP denotes the number of safe program samples predicted as vulnerable. FN is the number of vulnerable samples predicted as safe. TI denotes the number of correct interpretations (at least one important statement on the path is reported). TVCS, FVCS and LVCS denote the number of correctly detected, falsely detected and labeled vulnerable statements, respectively.

the macro evaluation of the datasets can be greatly affected by the large samples containing a substantial number of vulnerable statements.

BTP Metrics. An ideal quantitative metric should be able to reflect the fine-grained bug-triggering paths when the detector finds a buggy sample, in order to address evaluation biases. The program statements of a bug-triggering path are the key to understanding and characterizing the bug-triggering logic because they can reveal the behavior and context of the vulnerability. It would be perfect if the reported statements only cover the statements in that bug-triggering path without additional program statements. Therefore, we design our BTP metrics based on the degree of overlap between the detected statements and the statements on the bug-triggering paths, which is formally defined as follows:

$$\begin{aligned} BTP\_P &= |S_d \cap S_p|/|S_d| \ s.t.\ S_d \neq \varnothing \\ BTP\_R &= |S_d \cap S_p|/|S_p| \ s.t.\ S_p \neq \varnothing \\ BTP\_IoU &= |S_d \cap S_p|/|S_d \cup S_p| \ s.t.\ S_p \cup S_d \neq \varnothing \end{aligned} \tag{8}$$

where BTP_P, BTP_R and BTP_IoU represent BTP's improved precision, recall and IoU (intersection over union) [43] respectively by considering bug-triggering paths. S_d denotes the set of statements reported by detectors. S_p denotes the set of statements in the bug-triggering paths labeled in the dataset. $|\cdot|$ denotes the size of a set. Note that, BTP_P is computed over the reported vulnerable samples, i.e., $|S_d| \neq 0$. BTP_R is computed over the labeled vulnerable samples, i.e., $|S_p| \neq 0$. BTP_IoU is computed over the reported or labeled vulnerable samples, i.e., $|S_p \cup S_d| \neq 0$. Let us revisit the example in Fig. 3, where the bug-triggering path comprises Lines 3 and 5. The detected statements of Lines 3 and 5 will be a perfect match (100% BTP precision/recall/IoU), while only reporting Line 5 has 100% BTP precision, but 50% BTP recall. In contrast, SySeVR(Lines 1, 2, 3, 4, 5, 6, 7) yields 100% BTP recall, but only 29% BTP precision.

The bug-originating and triggering points are essential to locate the bug-triggering paths. We also define a relaxed and coarse-grained BTP metric called BTP's accuracy, BTP_A, which is the ratio between the detected bug-originating and triggering points and all the labeled bug-originating and triggering points:

$$BTP\_A = \begin{cases} |S_d \cap S_{OT}|/|S_{OT}| & \text{if } S_p \neq \varnothing \\ 0 & \text{else if } S_d \neq \varnothing \end{cases} \tag{9}$$

where S_{OT} stands for the labeled bug-originating and triggering points (statements). Note that BTP_A is also computed over the reported or labeled vulnerable samples, i.e., $|S_p \cup S_d| \neq 0$.

The BTP metrics for all bug samples are the arithmetic average of BTP metrics for each bug. The BTP precision/recall/IoU for all bug samples in a dataset are calculated as follows:

$$\begin{aligned} BTP\_P_{avg} &= AVG(BTP\_P_i) \\ BTP\_R_{avg} &= AVG(BTP\_R_i) \\ BTP\_IoU_{avg} &= AVG(BTP\_IoU_i) \\ BTP\_A_{avg} &= AVG(BTP\_A_i) \end{aligned} \tag{10}$$

where $AVG(\cdot)$ denotes the arithmetic average of all data samples. As such, the BTP metrics have equal influence on the evaluation results in terms of each data sample, including large and small samples on the entire dataset.

5.3 Toolkit

Since the implementations of VGDetectorand VulDeePeckerare not publicly available, we have re-implemented them by strictly following their methods elaborated in the original papers. We have tested our re-implementation on the dataset in the original paper and Table 4 shows that our framework can achieve similar results in their original paper. We have also adapted the other nine open-source tools Token Embedding, Code2Seq, Reveal, IVDetect, VulDeeLocator, SySeVR, ICVH, VELVETand DeepWukongintoour unified toolkit. Given a target program, our toolkit extracts code representations (e.g., AST, CFG, DDG and PDG) and feeds them to the downstream detectors for a fair comparison. We use ASTMiner [32] as the lexer and parser for Token Embedding, ICVHand Code2Seq. We use joern [33], which is also used by SySeVR, IVDetectand Reveal, to dump AST, CFG and PDG

TABLE 4
A Comparison of Evaluation Results of Our Re-Implementations and the Original Paper

Vul Category	Approaches	Precision	Recall	F1 score
CWE119	**VulDeePecker**	**0.93**	**0.86**	**0.89**
	Original Paper	0.92	0.82	0.87
CWE399	**VulDeePecker**	**0.91**	**0.94**	**0.92**
	Original Paper	0.95	0.95	0.95
CWE691	**VGDetector**	**0.74**	**0.86**	**0.80**
	Original Paper	0.70	0.85	0.77
CWE840	**VGDetector**	**0.75**	**0.91**	**0.82**
	Original Paper	0.71	0.89	0.79
CWE438	**VGDetector**	**0.75**	**0.90**	**0.82**
	Original Paper	0.70	0.90	0.79

to support program dependence analysis for VGDetector, VulDeePecker, SySeVR, DeepWukong, Revealand IVDetect. We use dg [37] to generate the IR slice for VulDeeLocator. We use PyTorch Lightning [49], a lightweight PyTorch wrapper for high-performance AI research, to build neural network models. We use SMOTE [50] to deal with the data imbalance problem during the training process.

5.4 Dataset

The scope of our study is to evaluate learning-based approaches for static and source-code-based bug detection. Therefore, this study does not cover binary code analysis [18], [51] or dynamic analysis [52]. Table 5 presents an overview of eight datasets which have been used in the previous literature to evaluate software vulnerability detection. The valid dataset should satisfy the following requirements:

1) It should be built upon real-world projects rather than synthetic or conceived samples, which are not as diverse and complex as real-world vulnerabilities.
2) It should be labeled with reliable methods and have a sufficient number of high-quality samples for model training.
3) It should be well-labeled at the program statement level because some of the learning-based approaches report vulnerabilities based on program lines.
4) It should provide links to the original source code repositories, in order to be able to soundly produce program feature sets for each approach.
5) It should provide detailed bug type information so that a developer can investigate the results of different vulnerabilities.

In this study, we use D2A [26], the most recent and comprehensive dataset for our evaluation because it satisfies all of the aforementioned requirements. Unlike the synthetic datasets (Juliet [44], Choi et al. [46] and S-babi [45] in Table 5) which contain tiny code fragments generated by predefined patterns, D2A is built upon real-world large-scale projects (including `OpenSSL`, `FFmpeg`, `libav`, `httpd`, `NGINX` and `libtiff`) and contains 1,270,139 high-quality vulnerabilities (Column 2 of Table 6).

Unlike the datasets with coarse-grained labels at the program method level (Draper [47], Devign [8], CDG [1] and Fan et al. [48]), the samples in D2A are well-labeled at statement-level with differential analysis, which runs before and after each bug-introducing commit from 349,373,753 issues using industrial-strength static analyzers [26]. Some approaches label the modified/removed lines of bug-fixing commits as vulnerable [1], [8], [48]. However, the labeled lines may not be vulnerable when putting them in other (safe) program context. Also, these lines may not be bug-originating or triggering points, making it hard for manually bug tracing. In comparison, each vulnerability in D2A is labeled at the statement (line) level with clearly annotated bug-triggering paths related to the vulnerability, which is produced by the industrial-strength static analyzer Infer. Each vulnerability is also labeled with its corresponding type, with all types summarized in Table 7. The labeled vulnerabilities and bug-triggering paths might not be perfect and may contain mislabeled samples. In this study, we trust the labeling results in D2A since its labeled samples and their bug-triggering paths are generated by differential analysis to greatly reduce false positives and have also been reviewed by domain experts [26]. In addition, previous studies have shown that the neural network models of learning-based approaches are robust against some mislabeled samples [1], [53]. Given these vulnerabilities associated with fine-grained and comprehensive labels, it will be more effective for a fair comparison between various learning-based detectors at different granularity levels.

We divide D2A's samples into vulnerable programs and safe ones. A program is treated as vulnerable if it contains

TABLE 5
Popular Datasets for Evaluating Learning-Based Vulnerability Detection

Dataset	Sample Type	Labelling Granularity	Vulnerability Type	Link to Source code	Labelling Method
Juliet [43]	Synthetic	Statement	✓	-	Predefined bug pattern
S-Babi [44]	Synthetic	Statement	✓	-	Predefined bug pattern
Choi et al. [45]	Synthetic	Statement	✓	-	Predefined bug pattern
Draper [46]	Synthetic+Real-world	Method	✓	✗	Static analysis
Devign [8]	Real-world	Method	✗	✗	Commit message+Code diff
CDG [1]	Real-world	Method	✗	✗	NVD + Code diff
Fan et al. [47]	Real-world	Method/Statement	✓	✓	CVE + Code diff
D2A [25]	Real-world	Statement	✓	✓	Differential analysis

Beneficial characters of datasets are highlighted. D2A has high-quality labels from real-world projects with sufficient bug information.

TABLE 6
Label Distribution of Vulnerable and Safe Samples

Category	# Program method			# Program slices (code gadget)			# Program slices (SeVC, XFG, iSeVC)		
	Total	Vulnerable	Safe	Total	Vulnerable	Safe	Total	Vulnerable	Safe
BUFFER_OVERRUN	358,073	5,560	352,513	805,557	13,066	792,491	350,404	6,247	344,157
INTEGER_OVERFLOW	815,898	11,125	804,773	3,540,640	67,001	3,473,639	10,636,731	5,206,271	5,430,460
NULL_DEREFERENCE	28,959	641	28,318	103,281	691	102,590	185,906	1,329	184,577
DEAD_STORE	4,171	96	4,075	3,665	193	3,472	7,052	306	6,746
DIVIDE_BY_ZERO	1,139	12	1,127	2,701	35	2,666	3,656	34	3,622
MEMORY_LEAK	9,324	38	9,286	4,507	18	4,489	15,943	57	15,886
RESOURCE_LEAK	198	3	195	98	5	93	348	6	342
UNINITIALIZED_VALUE	10,185	240	9,945	14,431	752	13,679	20,685	1,246	19,439
USE_AFTER_FREE	735	19	716	364	13	351	653	19	634
Total	1,270,139	17,734	1,252,405	4,475,244	81,774	4,393,470	6,017,263	114,261	5,903,002

at least one statement in the bug-triggering paths labeled by D2A, and safe otherwise. We remove the duplicate samples, which could affect the performance metrics. In the end, from the D2A dataset, we collect 17,734 vulnerable and 1,252,405 safe programs. Our evaluation is conducted on a server running Ubuntu Linux with NVIDIA GeForce GTX GM200 GPU and Intel(R) Xeon(R) CPU E5-2603 v4 with 1.70GHz and 64GB memory.

6 Evaluation

Our evaluation aims to answer the following questions:

- *RQ1* What is the gap between different learning-based bug detectors and conventional static bug detectors under the BTP metrics?
- *RQ2* What are the differences between the existing metrics and BTP metrics?
- *RQ3* What are the results for different vulnerability types using the BTP metrics?

6.1 RQ1: Gap Between Learning-Based Bug Detectors and the Static Analyzer Under the BTP Metrics

As described in Section 3, the underlying prediction models of *method-level* and *slice-level* approaches aim to classify whether a program method or program slices (a set of statements) is vulnerable or safe. Given the well-labeled D2A dataset, we first want to estimate the ideal prediction results for both *method-level* and *slice-level* learning-based approaches, to understand the gap between the best possible results of learning-based detectors and the ground truths confirmed and produced by static analyzers under the BTP metrics. In addition, we also evaluate the actual performance of *method-level*, *slice-level* and *statement-level* approaches with regard to bug-triggering paths when training and applying the detection models. Finally, for *statement-level* approaches, we compare their performance on locating bug-triggering paths using the datasets labeled with bug-triggering paths (i.e., D2A) and the corresponding program patches, respectively.

TABLE 7
Vulnerability Types in Our Dataset [26]

Type	Description
BUFFER_OVERRUN	Out-of-boundary read/write of a buffer
INTEGER_OVERFLOW	Exceeding increment of an integer value
NULL_DEREFERENCE	Dereferencing a null pointer
DEAD_STORE	Never use an assigned value
DIVIDE_BY_ZERO	Divide a value by zero
MEMORY_LEAK	Never release an allocated memory
RESOURCE_LEAK	Never close a resource after usage
UNINITIALIZED_VALUE	Read a value before initialized
USE_AFTER_FREE	Use a freed memory

Column 2 of Table 6 gives a total number of 1,270,139 program methods to be predicted by Token Embedding, Code2Seq, VGDetectorand Reveal. Columns 5 gives 4,475,244 program slices generated by VulDeePeckerfrom all the samples in the D2A dataset. Column 8 shows 6,017,263 program slices extracted by SySeVRand DeepWukong. Columns 3 and 4 give the number of vulnerable and safe methods using ideal classification, where a method is classified vulnerable if it contains a buggy statement, otherwise it is treated as safe given the ground-truth labels in D2A. Columns 6 and 7 list the number of vulnerable and safe code gadgets using ideal classification for VulDeePecker, where a code gadget is classified as vulnerable if it contains a buggy statement and safe otherwise [1], similarly for Columns 9 and 10 by SySeVRand DeepWukong. The above ideal classification (with 100% precision and 100% recall measured by the traditional metrics in Table 3) provides the upper limit or the best possible prediction results can be produced by the current method-level (Token Embedding, Code2Seq, VGDetectorand Reveal) and slice-level (VulDeePecker, SySeVRand DeepWukong) learning-based approaches.

Statement-level approaches ICVH, VELVET, IVDetectand VulDeeLocatorcan theoretically predict and report any program statement as vulnerable. However, the underlying learning model is still unaware of bug-triggering paths. They cannot precisely pinpoint the bug-triggering paths because both of their classification and interpretation models do not distinguish program paths. In addition to the ideal prediction results, we compare the performance of different learning-based bug detectors in terms of BTP metrics by training a detection model to understand their ability to pinpoint bug-triggering paths in real-world scenarios. In order to train a detection model, we randomly split the dataset shown in Table 6 into 80%, 10% and 10% for training, validation and testing respectively. For each learning-based bug detector, a detection model is first trained and tuned using the training and validation dataset and applied

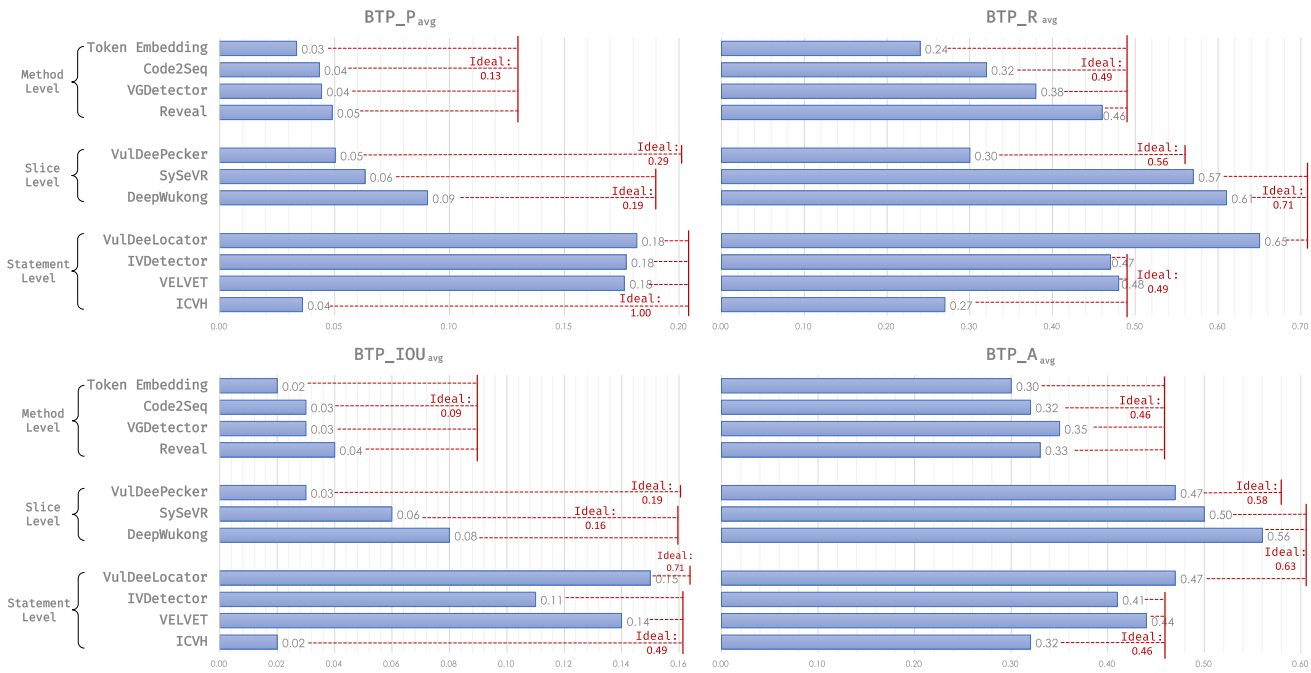

Fig. 4. A comparison of the BTP metrics of the state-of-the-art learning-based bug detectors.

to the testing dataset for BTP metrics evaluation. To compare the performance on the datasets using different statement-level labeling, we build another dataset based on the code changes of the Github commits in relation with the ground truth of bug-triggering paths from the D2A dataset.

Result. As shown in Fig. 4, method-level approaches achieve a lower ideal BTP recall/precision/IoU/accuracy than slice-level approaches. The ideal BTP recall is below 50% at 49%, 31% lower than SySeVRand 13% lower than VulDeePecker. Method-level approaches also record a significantly lower ideal BTP precision at merely 13%, compared to SySeVRat 19% and VulDeePeckerat 29%. Among the slice- and method-level approaches, the highest ideal BTP IoU is achieved by VulDeePecker(19%), which is more than the method-level approaches (9%) and SySeVR(16%). SySeVRobserves the best ideal BTP accuracy at 63%, 9% higher than VulDeePeckerand 37% higher than method-level approaches. For statement-level approaches, all the four approaches can ideally achieve a 100% BTP precision. However, the ideal BTP recall and IoU for IVDetect, ICVHand VELVETis 49%, compared to 71% for VulDeeLocator. Fig. 4 also presents the actual performance of these approaches under BTP metrics and the gap with the ideal results. Overall, they are not comparable to their upper limits with a decline of 72%, 26%, 72% and 32% BTP precision, recall, IoU and accuracy on average. Method-level approaches have a lower BTP precision/recall/IoU than their slice-level counterparts, which is consistent with the ideal result in Fig. 4. Of the method-level approaches, Revealachieves the best performance with a BTP IoU of 4%, double the lowest BTP IoU from Token Embeddingat only 2%. Revealis also slightly better than the AST-based approach Code2Seq. With regard to the slice-level approaches, it is interesting to find that VulDeePecker, which has the highest ideal BTP IoU, records the lowest BTP IoU at merely 3%. As for SySeVRand DeepWukong, DeepWukongis more superior to the other two approaches with a 9% BTP precision and 8% BTP IoU, compared to an average of 7% BTP precision and 6% BTP IoU for the other two approaches. Regarding statement-level approaches, the highest BTP IoU comes from VulDeeLocatorreaching 15%, while VELVETalso records a relatively high BTP IoU capped at 14%. The BTP precision of these two approaches is markedly higher than the rest at about 18%, nearly double the rate of DeepWukongand more than five times the rate of Token Embedding. VulDeeLocatoralso observes the highest BTP recall capped at 65%, approaching the upper limit around 71%. VELVETreports a BTP recall at 48% which is close to its upper limit. The BTP IoU for ICVHis the lowest among its statement-level counterparts at merely 2%, which is even lower than some of the method-level approaches. IVDetectand VulDeeLocatorrecord identical BTP metrics when training with the datasets labeled with bug-triggering paths and the corresponding program patches. However, ICVHand VELVETobserve a poorer performance in terms of BTP metrics when training with program patches with the BTP IoU dropping by nearly 50%.

Analysis. Method-level approaches have the lowest BTP precision because they report the whole buggy program method, which contains a large proportion of statements not related to the bug-triggering paths. Taking the code fragment in Fig. 3 as an example, the number of irrelevant statements (Lines 1, 2, 4, 6, 7, 8, 9 and 10) is four times as high as the size of the bug-triggering paths (Lines 3, 5). The ideal BTP precision here is merely 20%. Note that the BTP precision can be even worse for large and complex buggy program method in real-world software, making it difficult to understand the bug triggering paths and adversely affecting productivity for bug fixing and software maintenance. Of the method-level approaches, Revealachieves the best performance because it uses a more precise program representation (CPG) hence a more robust prediction model with less false positives/negatives is produced. Code2Seqthough is proven to be effective on code classification or summarization tasks, they are still insufficient for vulnerability detection.

VulDeePeckerhas a higher ideal BTP precision than SySeVRbecause it can ideally preserve the bug-triggering paths more precisely without including control-related statements. For instance, the buggy slices (SeVC) from Fig. 3 includes the whole program statements (Lines 6, 7, 8) controlled by Line 5 (library/API call) through forward control-dependence slicing. These statements are not relevant to the bug so they do not appear in the bug-triggering paths, making the SeVC less precise. In fact, due to the control-dependence slicing, the average number of statements of buggy SeVC is 12, compared to only 7 for code gadgets in VulDeePecker, which only conducts data-dependence slicing. As such, SySeVRintroduced more noisy statements in this case, making its ideal BTP precision of SySeVRlower than VulDeePecker. The average number of statements of the method-level approaches is 62, significantly larger than that of SySeVRand VulDeePecker. This partly implies that slice-level approaches have better BTP precision than the method-level approaches, because the slicing operation of SySeVRand VulDeePeckercan exclude some statements not relevant to the bug-triggering paths. Although VulDeePeckerhas the highest ideal BTP precision, it still includes lots of false positive statements (i.e., $S_d \setminus (S_d \cap S_p)$). By looking into the buggy code gadget generated by VulDeePecker, we find the main reason that leads to the imprecision is the distracted value flows of objects other than the vulnerable objects. For instance, in Fig. 3, the vulnerable object is the array "group", which was not initialized in Statement 3 but is indexed in Statement 5. The array object "h $\rightarrow$ array", which is properly initialized in Statement 2, is not vulnerable, so its value flows are not in the bug-triggering paths. However, these statements are included by the reported code gadget of VulDeePeckerthrough backward data-dependence slicing from the API call at Line 5. This is caused by the imperfection of VulDeePecker's slicing method which starts from all the arguments of arbitrary API calls. However, some of the arguments of an API can be irrelevant to a bug. Moreover, a bug (e.g., buffer overflows) may not be relevant to any API call.

VulDeePeckeris inferior to SySeVRbased on the model prediction results under the BTP metrics. This is because multiple real-world vulnerabilities (e.g., memory leak) are affected by the control-dependency information. Excluding control-related statements though can produce a thin and precise slice, it also fails to preserve important statements in the program representation, e.g., the branch conditions that determine the execution order of programs; hence adversely affecting model training and introducing more false positives/negatives. The lower ideal BTP recall of VulDeePecker-also indicates that it misses a proportion of buggy statements in the bug-triggering paths. DeepWukongand SySeVRextract samples under the same granularity (control- and data-related statements) thus having the same ideal BTP metrics; however, after training a prediction model, DeepWukongoutperforms SySeVRbecause it incorporates more comprehensive structural dependencies of the program so can train a more effective prediction model.

Statement-level approaches VulDeeLocator, VELVETand IVDetectachieve a significantly higher BTP precision because they use a comprehensive program representation with interpretable AI techniques to comprehend the prediction result. As such, the well-trained prediction model is more effective under the traditional metrics [10], and the interpretable AI ensures that the reported vulnerable statements contain less bug-unrelated statements. VulDeeLocatoroutperforms the other approaches because it considers the interprocedural data-flow, while the others utilize intraprocedural feature and cannot recall the vulnerable statements in the interprocedural bug-triggering paths. VELVETand IVDetectperform better than ICVHbecause of their more effective classification models. This demonstrates that the BTP metrics can reflect the effectiveness of both the classification and interpretation models.

VELVETand ICVHperform better in terms of pinpointing bug-triggering paths when training on the ground truth of bug-triggering paths because they can learn the individual statements information on the paths by training a node (statement) classification model supervised by the pre-labeled statements on the bug-triggering paths. In comparison, VulDeeLocatorand IVDetectare not affected by statement-level labeling because they do not leverage/learn the pre-labeled statements information during model training; rather, they report statements based on the interpretation model (attention mechanism [15] and GNNExplainer [14]) by identifying important statements contributing to the coarse-grained classification result on a program method/slice.

ANSWER to RQ1

There exists a significant gap between learning-based bug detectors and traditional static analyzers (85% in BTP IoU on average). VulDeeLocator, VELVETand IVDetect-greatly outperform the other approaches under BTP metrics. VulDeePeckerdetects the bug-triggering paths more precisely (3% more than average) than SySeVRand DeepWukong. ICVHand method-level approaches report the worst performance.

6.2 RQ2: Differences Between Existing Metrics and BTP Metrics

In this research question, we aim to compare our BTP metrics with the existing evaluation metrics described in Table 3 by evaluating the learning-based vulnerability detection approaches using these metrics. For fair comparison, we use the same experimental settings in RQ1 to train a prediction model: we randomly split the dataset into 80%, 10% and 10% for training, evaluation and testing, and apply each trained and tuned model to the testing dataset for evaluation using the metrics in Table 3.

Result. Table 8 presents the results of the learning-based detectors under the existing metrics. Revealoutperforms the other method-level approaches with an F1 score reaching 61%. Its SA is also the highest among method-level approaches at 65%. This comparison result is similar to BTP metrics. However, the VCP and JI of method-level approaches are rather close to each other with relatively low numbers. The best performance among slice-level approaches comes from DeepWukongwhich achieves an F1 score at 79% (8% and 14% higher than SySeVRand VulDeePeckerrespectively). Similarly, the differences between slice-level approaches under VCP and JI are also negligible (with less than 3% difference). Statement-level VulDeeLocatorrecords a better performance

TABLE 8
A Comparison of Existing Metrics of the State-of-the-Art Learning-Based Bug Detectors

Approaches	Precision	Recall	F1 score	SA	VCP	JI
Token Embedding	0.63	0.49	0.55	0.59	0.006	0.002
Code2Seq	0.63	0.53	0.57	0.60	0.006	0.003
VGDetector	0.64	0.56	0.60	0.61	0.007	0.003
Reveal	**0.66**	**0.57**	**0.61**	**0.65**	**0.007**	**0.004**
VulDeePecker	0.76	0.63	0.69	0.72	0.010	0.005
SySeVR	0.77	0.68	0.73	0.73	0.013	0.005
DeepWukong	**0.84**	**0.74**	**0.79**	**0.81**	**0.014**	**0.006**
VulDeeLocator	**0.83**	**0.74**	**0.78**	**0.79**	0.015	**0.007**
IVDetect	0.67	0.71	0.69	0.64	**0.016**	0.005
VELVET	0.74	0.58	0.65	0.71	0.012	0.004
ICVH	0.62	0.55	0.58	0.58	0.007	0.003

SA stands for statement accuracy, VCP denotes vulnerability coverage proportion and JI denotes Jaccard index.

than its statement-level counterparts with F1 score and SA capped at 78% and 79% respectively. Note that ICVHreports the lowest numbers. For example, its F1 score is only 58%, which is 5% less than Reveal, the method-level approach. The other three statement-level approaches also observe similar VCP and JI. It is worth noting that statement-level VELVETand method-level Revealonly see a difference of 7% under F1 score while their gap in terms of BTP IoU is 250%. Slice-level DeepWukongeven outperforms statement-level VulDeeLocatorin terms of F1 score.

Analysis. The traditional metrics (e.g., precision, recall and F1 score) are mainly designed for the evaluation of the classification model. The correct prediction of data samples with different granularities is treated as equivalent performance under these metrics, even though a more fine-grained prediction (e.g., statement) can better help practitioners to locate the vulnerability than coarse-grained prediction (e.g., method). For example, method-level Revealshows a close F1 score to statement-level VELVETbut VELVETis clearly more useful than Revealwhen locating buggy statements. In comparison, the BTP metrics can obviously show the gap between the approaches with different granularities. For example, BTP metrics exhibit explicit different numbers between VELVETand Revealalthough they have similar results under precision, recall and F1 score. Moreover, the BTP metrics can be used to evaluate the approaches with the same granularity. Taking method-level approaches as an example, the evaluation results (Fig. 4) under our metrics can clearly reveal their performance differences caused by their different classification models. Their numbers are also consistent with traditional metrics.

Statement accuracy (SA) is designed for statement-level evaluation but is still coarse-grained because a perfect prediction is counted as long as one statement on the bug-triggering paths is correctly captured. As a result, the SA of all the approaches is relatively high and there is no big gap between statement-level approaches and its coarse-grained counterparts, e.g., VELVETand Reveal. Therefore, it is important to consider all the statements on the bug-triggering paths when analyzing the performance of the target detection models, which is the principle of the BTP metrics. For example, the BTP recall considers the proportion of statements correctly identified on the bug-triggering paths, while the BTP precision considers the proportion of the reported statements on bug-triggering paths over all the reported statements.

Vulnerability coverage proportion (VCP) only evaluates the ratio of all the discovered statements. It mainly focuses on evaluating the ability of recalling vulnerable statements on the bug-triggering paths but fails to reflect the precision of locating statements, while BTP metrics reflect both aspects. Similarly, Jaccard index (JI) only evaluates the overall result of statement-level recall and precision but fails to reflect each aspect independently. In addition, both VCP and JI are computed based on all the statements of the datasets so the results are biased towards the numbers of large samples and can only reflect their performance. The gap between different approaches is not clear probably because their performance on large samples is identically not good. For instance, we assume that Code (a) has 3 lines of bug-triggering paths and the detector can predict exactly the same bug-triggering paths, while Code (b) has 30 lines of bug-triggering paths but the detector fails to predict one line on the paths. For the two code fragments, we get an average VCP of 9% and an average BTP recall of 50%. The VCP result is greatly affected by the large sample Code (b), while our metric offers a relatively unbiased result for each of the sample.

ANSWER to RQ2

The BTP metrics provide a more unbiased, fine-grained and explicit methodology to evaluate and compare general learning models with different granularities, while precision, recall and F1 score are biased with different granularities, statement accuracy is coarse-grained and vulnerability coverage proportion and Jaccard index are not clear.

6.3 RQ3: Evaluation for Different Vulnerability Types

To understand the results of different vulnerability types under our BTP metrics, we apply our BTP metrics for each type of vulnerability separately. Only the vulnerable samples as shown in Table 6 are evaluated to understand and compare the best possible prediction results of different types in terms of bug-triggering paths. Fig. 5 shows a heatmap depicting the results of each type of vulnerability.

Result. The highest rate for BTP recall goes to UNINITIALIZED_VALUE and DEAD_STORE (both around 95% for

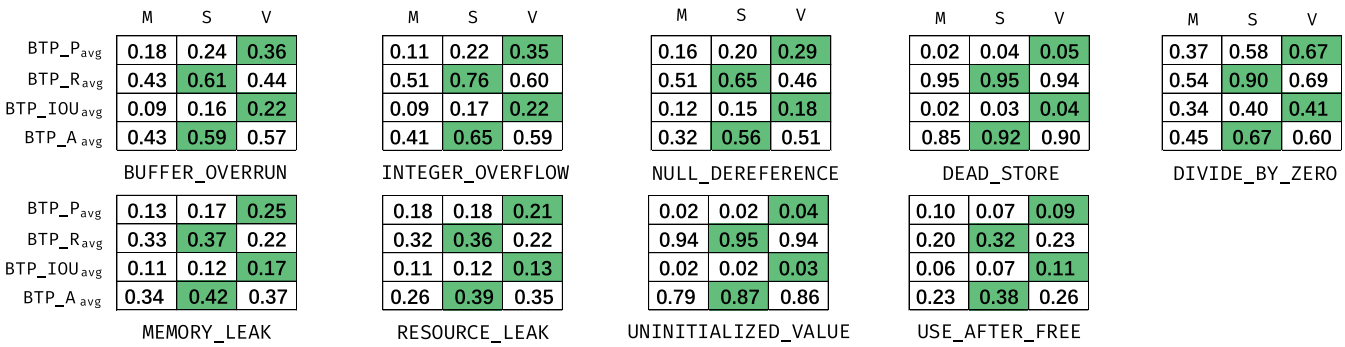

Fig. 5. A comparison of the BTP metrics of the state-of-the-art approaches across vulnerabilities. M, S, V stand for Method-level approaches, SySeVR/DeepWukongand VulDeePecker, respectively. The nine subfigures represent the nine types of vulnerabilities. For a given vulnerability, each cell represents the performance of the approach (x-axis) under the corresponding metric (y-axis). The best results are marked with dark color.

method- and slice-level approaches). The trend of BTP accuracy is similar with BTP recall: UNINITIALIZED_VALUE and DEAD_STORE observe the highest BTP accuracy for all the approaches. However, the BTP precision of UNINITIALIZED_VALUE and DEAD_STORE are the lowest of all the vulnerability categories. For instance, the BTP precision of the method-level approaches for UNINITIALIZED_VALUE and DEAD_STORE are both around 2%, while for VulDeePecker, the BTP precision is also the lowest at only 2% and 4%, respectively. By comparison, DIVIDE_BY_ZERO observes the highest BTP precision at 37% for the method-level approaches, 58% for SySeVRand 67% for VulDeePecker. The BTP IoU on DIVIDE_BY_ZERO reaches 34% for method-level approaches, 40% for SySeVRand 41% for VulDeePecker, which is higher than any other vulnerability type. BUFFER_OVERRUN, INTEGER_OVERFLOW have a close BTP IoU at about 10% for method-level detectors, 16% for SySeVRand 22% for VulDeePeckeron average. The BTP IoU for MEMORY_LEAK, RESOURCE_LEAK and USE_AFTER_FREE is lower. The BTP IoU by VulDeePeckerare merely 17%, 13% and 11%, respectively.

Analysis. The reason for high BTP recall and accuracy under DEAD_STORE and UNINITIALIZED_VALUE is that these two types of bugs typically occur across multiple program methods. As for the low BTP precision, there are two reasons behind this. (1) First, these two vulnerability types normally have a small number of bug-triggering paths (9 on average compared to 50 for the other types), making $|S_p|$ and $|S_p \cap S_d|$ small. This is caused by their vulnerability behaviors. The DEAD_STORE vulnerability occurs when an object is assigned a value but is never used after, while UNINITIALIZED_VALUE is triggered when an object is used but is never initialized before; therefore the bug-triggering paths do not contain the initialization or the usage statements. (2) Second, these vulnerabilities are usually related to the value flow of a standalone object, but VulDeePeckerand SySeVRinclude all the value-flow statements related to the arguments of any library/API calls, many of which are irrelevant to the vulnerability, thus introducing redundant statements S_d with a reduced BTP precision.

Regarding DIVIDE_BY_ZERO, the BTP precision is the highest because the bug-triggering logic of this bug is less complex, i.e., a value of zero is divided. VulDeePeckerand SySeVRcan precisely capture the value flow of the denominator as in the bug-triggering paths, making $S_d \cap S_p$ close to S_d and S_p. BUFFER_OVERRUN and INTEGER_OVERFLOW are normally caused by the collective effect of a number of objects (e.g., copying a memory to a buffer of a smaller size). Their bug-triggering paths are often large in size because they include all the vulnerable data flows of these objects. On the other hand, the proportion of bug-related arguments in the library/API calls is likely to be larger than standalone-object vulnerabilities like MEMORY_LEAK and USE_AFTER_FREE, so the S_d under BUFFER_OVERRUN and INTEGER_OVERFLOW is closer to the bug-triggering paths with a chance of having fewer bug-unrelated arguments, resulting in a relatively higher BTP precision.

ANSWER to RQ3

UNINITIALIZED_VALUE and DEAD_STORE report the lowest BTP Metrics (3% for BTP IoU) while DIVIDE_BY_ZERO observes the best performance (38% for BTP IoU). This is caused by the length of bug-triggering paths and the number of relevant values. Other vulnerabilities like BUFFER_OVERRUN report an intermediate BTP IoU for 16%.

7 Discussion

Compared to the method-level approaches which predict whether a program method is vulnerable or not, the statement-level approaches using program slices as training features, though closer to an understanding of bug-triggering paths, are yet imprecise. Unfortunately, the gap still exists between the most sophisticated learning-based approach and a precise static analyzer as shown in our evaluation. In this section, we share the following insights in the hope of improving future research on learning-based vulnerability detection.

(1) Customized Feature Learning. Existing efforts utilize a unified method to pinpoint all types of vulnerabilities. However, different types of vulnerabilities may have different bug semantics and bug triggering patterns, thus requiring different learning strategies. For the vulnerabilities which are highly related to value flows like USE_AFTER_FREE and RESOURCE_LEAK, it is interesting to investigate whether introducing precise interprocedural data-flow analysis into code embedding can avoid bug-irrelevant control/data-dependence. The DIVIDE_BY_ZERO vulnerability is caused by dividing by zero, so the backward value flow of the denominator on the arithmetic expression should be taken into consideration. In addition, the value flow of the

denominator will be highlighted if detected. For high-level vulnerabilities like Business Logic Errors (CWE840) [54], it is better to include more in-depth semantic features which are possibly assisted by user annotations. Overall, it would be helpful to design customized feature engineering methods to tackle different vulnerability categories in order to boost the efficiency and precision of the detectors.

(2) Fine-Grained and On-Demand Feature Extraction. We believe that the current feature extraction methods are still shallow. The approaches either simply leverage the graph structure (e.g., ASTs and CFGs of a program [5], [7], [8]) or extract program slices relevant only to library/API calls [1]. However, existing precise static analyses are yet to be used for more fine-grained feature extraction, e.g., context-sensitive interprocedural analysis, handling recursive data structures, abstract interpretation and path-sensitive analysis. For example, a more precise code embedding is introduced in [55] to boost code classification and summarization tasks by leveraging the precise context-sensitive interprocedural value-flow analysis. Moreover, balancing efficiency and precision when using advanced static analysis for learning-based approaches is also an interesting and impactful research direction.

(3) Informative Bug Reports. Static bug detectors outperform their learning-based counterparts in terms of the interpretability of the bug report, because they can provide more detailed bug information including bug location, bug-triggering paths and a readable explanation of how the bug is triggered, while it is difficult to explain the detection or interpretation result of a learning-based bug detector due to the black-box nature of deep learning classification models. As such, it is an interesting research direction to combine the advantages of static and learning-based detectors, and to develop techniques to produce more informative bug reports based on the prediction/interpretation result of the deep-learning model by leveraging the knowledge bases from traditional static analysis techniques. For example, it is beneficial to use static analysis to generate precise path information and incorporate the information into code embedding to boost the performance of locating bug-triggering paths for statement-level vulnerability detection. In turn, learning-based vulnerability detection can save the efforts of crafting specifications for traditional static analysis to identify vulnerable paths among all the program paths.

8 Related Work

Static Vulnerability Detection. There are a number of traditional static analysis frameworks (e.g., Clang Static Analyzer [11], Flawfinder [19], Infer [12], ITS4 [20], Checkmarx [21] and SVF [22]) which aim to statically analyze the runtime behavior of source code and detect vulnerabilities in a wide variety of software systems. There are also many approaches [23], [56], [57], [58], [59], [60], [61], [62], [63], [64], [65], [66], which seek to detect specific vulnerabilities like memory errors or divide-by-zero bugs through conventional static analysis methodologies (e.g., sparse data-flow analysis, abstract interpretation, symbolic execution and incorrectness logic).

Machine-Learning-Based Vulnerability Detection. Recently, there are several studies in successfuly applying machine learning techniques in automated software vulnerability detection. Neuhaus et al. [67] use support vector machines (SVM) to detect vulnerabilities from Red Hat packages. Grieco et al. [68] utilise the static and dynamic features of source code to detect memory corruption. DeepBugs [69] proposes to represent a program as a text vector to detect name-based bugs. VulDeePecker [1] uses data-flow and BLSTM to detect resource management errors and buffer overflows. VGDetector [9] uses CFGs and the graph convolutional network to detect control-flow-related vulnerabilities. SySeVR [3] and μVulDeePecker [4] combine both control and data flow using different recurrent neural networks (RNNs) to detect various types of vulnerabilities. DeepWukong [36] uses structural control and data-dependence to pinpoint vulnerabilities. Devign [8] uses Gated Graph Sequence Neural Networks (GGNNs) [31] and a composition of AST, CFG, DDG and natural code sequence (NCS) edges to detect vulnerabilities in program method, while Reveal [25] proposes to leverage code property graph and GGNNs. ICVH [16] proposes to highlight code statements based on the mutual information maximization of source code and code statement probability distribution. VELVET [13] uses ensemble learning to select important statements based on the scores learned from GGNN and transformer. IVDetect [17] first uses PDG and GCN to detect vulnerabilities at coarse-grained level and interpret a subgraph of PDG using edge-masking [14]. VulDeeLocator [18] utilizes IR-based control and data flow to detect vulnerabilities, and k-max pooling to pinpoint fine-grained vulnerable elements in the program.

Code Embedding. Code embedding aims to produce low-dimensional vector representations of source code to enable the application of advanced learning techniques to various code analysis tasks. White et al. [41] stream software tokens to embed source code for code suggestion. Wang et al. [70] use token vectors extracted from Abstract Syntax Trees (ASTs) for software defect prediction. Astnn [7] embeds an input program by extracting a sequence of subtrees of each AST of the program for code classification and code clone detection. Code2Vec [5] and Code2Seq [6] conduct code embedding by encoding a bag of AST paths in the latent space for method name prediction and code summarization. Flow2Vec [55] embeds the interprocedural alias-aware value-flow-graph of a program for code classification and summarization. Kharkar et al. [71] proposes a transformer-based learning approach to reduce false alarms in static analyzers.

9 Conclusion

In this article, we propose quantitative and fine-grained evaluation metrics called BTP metrics by leveraging bug-triggering paths, to understand and characterize learning-based vulnerability detection approaches, thus complementing traditional evaluation metrics. We conduct a comprehensive comparison on existing learning-based bug detectors, which perform classification on program method or slice, or statements without pinpointing the vulnerable paths. We have evaluated state-of-the-art learning-based approaches in terms of BTP metrics under different vulnerability categories. Our empirical study carefully analyzes the gap between learning-based bug detectors and traditional static analyzers. Finally, our evaluation reveals several key issues and challenges in developing classification models to pinpoint bug-triggering paths and calls for more advanced learning-based bug detection techniques.

REFERENCES

[1] Z. Li et al., "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[2] R. L. Russell et al., "Automated vulnerability detection in source code using deep representation learning," in *Proc. IEEE 17th Int. Conf. Mach. Learn. Appl.*, 2018, pp. 757–762.

[3] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," *IEEE Trans. Dependable Secure Comput.*vol. 19, no. 4, pp. 2244–2258, Jul./Aug. 2022.

[4] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 5, pp. 2224–2236, Sep./Oct. 2021.

[5] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *ACM POPL*, vol. 3, 2019, Art. no. 40.

[6] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *Proc. Int. Conf. Learn. Representations*, 2019.

[7] 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 *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 783–794.

[8] 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 *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 10197–10207.

[9] X. Cheng et al., "Static detection of control-flow-related vulnerabilities using graph embedding," in *Proc. 24th Int. Conf. Eng. Complex Comput. Syst.*, 2019, pp. 41–50.

[10] K. M. Ting, *Precision and Recall*, Berlin, Germany: Springer, 2010, pp. 781–781.

[11] Apple Inc, "Clang static analyzer," 2021. [Online]. Available: https://clang-analyzer.llvm.org/scan-build.html

[12] Facebook, "Infer," 2021. [Online]. Available: https://fbinfer.com/

[13] Y. Ding et al., "Velvet: A novel ensemble learning approach to automatically locate vulnerable statements," 2021, *arXiv:2112.10893*.

[14] Z. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "Gnnexplainer: Generating explanations for graph neural networks," in *Proc. Adv. Neural Informat. Process. Syst.*, 2019, pp. 9244–9255. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/d80b7040b773199015de6d3b4293c8ff-Paper.pdf

[15] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Informat. Process. Syst.*, 2017, pp. 5998–6008. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[16] V. Nguyen, T. Le, O. De Vel, P. Montague, J. Grundy, and D. Phung, "Information-theoretic source code vulnerability highlighting," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, 2021, pp. 1–8.

[17] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, pp. 292–303.

[18] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "VulDeeLocator: A deep learning-based fine-grained vulnerability detector," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 4, pp. 2821–2837, Jul./Aug. 2022.

[19] D. A. Wheeler, "Flawfinder," 2021. [Online]. Available: https://dwheeler.com/flawfinder/

[20] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, "Its4: A static vulnerability scanner for c and c++ code," in *Proc. IEEE 16th Annu. Comput. Secur. Appl. Conf.*, 2000, pp. 257–267.

[21] Israel, "Checkmarx," 2021. [Online]. Available: https://www.checkmarx.com/

[22] Y. Sui and J. Xue, "SVF: Interprocedural static value-flow analysis in LLVM," in *Proc. 25th Int. Conf. Compiler Construction*, 2016, pp. 265–266.

[23] Y. Sui, D. Ye, and J. Xue, "Detecting memory leaks statically with full-sparse value-flow analysis," *IEEE Trans. Softw. Eng.*, vol. 40, no. 2, pp. 107–122, Feb. 2014.

[24] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," *J. ACM*, vol. 58, no. 6, Dec. 2011.

[25] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" 2020, *arXiv:2009.07235*.

[26] Y. Zheng et al., "D2A: A dataset built for ai-based vulnerability detection methods using differential analysis," in *Proc. ACM/IEEE 43rd Int. Conf. Softw. Eng.: Softw. Eng. Pract.*, 2021, pp. 111–120.

[27] Y. Kim, "Convolutional neural networks for sentence classification," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2014, pp. 1746–1751. [Online]. Available: https://www.aclweb.org/anthology/D14–1181

[28] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional LSTM and other neural network architectures," *Neural Netw.*, vol. 18, no. 5, pp. 602–610, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608005001206

[29] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder–decoder approaches," in *Proc. 8th Workshop Syntax, Semantics Struct. Statist. Transl.*, 2014, pp. 103–111. [Online]. Available: https://www.aclweb.org/anthology/W14–4012

[30] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.

[31] Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow, "Gated graph sequence neural networks," in *Proc. Int. Conf. Learn. Representations*, 2016. [Online]. Available: https://www.microsoft.com/en-us/research/publication/gated-graph-sequence-neural-networks/

[32] V. Kovalenko, E. Bogomolov, T. Bryksin, and A. Bacchelli, "Pathminer: A library for mining of path-based representations of code," in *Proc. IEEE 16th Int. Conf. Mining Softw. Repositories*, 2019, pp. 13–17.

[33] Fabian, "joern," 2021. [Online]. Available: https://github.com/ShiftLeftSecurity/joern/

[34] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. Int. Conf. Mach. Learn.*, 2014, pp. 1188–1196. [Online]. Available: http://dblp.uni-trier.de/db/conf/icml/icml2014.html#LeM14

[35] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. 26th Int. Conf. Neural Informat. Process. Syst.*, 2013, pp. 3111–3119. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999792.2999959

[36] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "DeepWukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, pp. 1–33, 2021.

[37] Marek Chalupa, "dg," 2016. [Online]. Available: https://github.com/mchalupa/dg

[38] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2014, pp. 1532–1543. [Online]. Available: https://aclanthology.org/D14–1162

[39] Z.-H. Zhou, *Ensemble Learning*, Berlin, Germany: Springer, 2009, pp. 270–273.

[40] H. Ramchoun, M. A. J. Idrissi, Y. Ghanou, and M. Ettaouil, "Multilayer perceptron: Architecture optimization and training with mixed activation functions," in *Proc. 2nd Int. Conf. Big Data, Cloud Appl.*, 2017, pp. 1–6.

[41] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proc. IEEE 12th Work. Conf. Mining Softw. Repositories*, 2015, pp. 334–345. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820559

[42] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," in *Proc. 53rd Annu. Meeting Assoc. Comput. Linguistics 7th Int. Joint Conf. Natural Lang. Process.*, 2015, pp. 1556–1566. [Online]. Available: https://aclanthology.org/P15–1150

[43] Jaccard, "Jaccard index," 2021. [Online]. Available: https://en.wikipedia.org/wiki/Jaccard_index

[44] American Information Technology Laboratory, "Software assurance reference dataset," 2017. [Online]. Available: https://samate.nist.gov/SARD/index.php

[45] C. D. Sestili, W. S. Snavely, and N. M. VanHoudnos, "Towards security defect prediction with AI," 2018, *arXiv:1808.0989*.

[46] 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 *Proc. 26th Int. Joint Conf. Artif. Intell.*, 2017, pp. 1546–1553.

[47] R. L. Russell et al., "Automated vulnerability detection in source code using deep representation learning," 2018, *arXiv.1807.04320*.

[48] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A c/c++ code vulnerability dataset with code changes and CVE summaries," in *Proc. 17th Int. Conf. Mining Softw. Repositories*, 2020, pp. 508–512.
[49] W. Falcon et al., "Pytorch lightning," 2021. [Online]. Available GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning
[50] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *J. Artif. Intell. Res.*, vol. 16, pp. 321–357, 2002.
[51] J.-F. Tian, W. Xing, and Z. Li, "Bvdetector: A program slice-based binary code vulnerability intelligent detection system," *Inf. Softw. Technol.*, vol. 123, 2020, Art. no. 106289.
[52] G. J. Duck and R. H. C. Yap, "Effectivesan: Type and memory error detection using dynamically typed c/c++," *SIGPLAN Not.*, vol. 53, no. 4, pp. 181–195, Jun. 2018.
[53] D. Rolnick, A. Veit, S. Belongie, and N. Shavit, "Deep learning is robust to massive label noise," 2017, *arXiv:1705.10694*.
[54] MITRE, "Common Weakness enumeration," 2021. [Online]. Available: https://cwe.mitre.org/
[55] Y. Sui, X. Cheng, G. Zhang, and H. Wang, "Flow2vec: Value-flow-based precise code embedding," *Proc. ACM Prog. Lang.*, vol. 4, no. OOPSLA, pp. 1–27, Nov. 2020.
[56] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 499–510.
[57] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *Proc. IEEE Symp. Secur. Privacy*, 2015, pp. 797–812.
[58] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic discovery and quantification of information leaks," in *Proc. IEEE 30th Symp. Secur. Privacy*, 2009, pp. 141–153.
[59] H. Yan, Y. Sui, S. Chen, and J. Xue, "Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities," in *Proc. IEEE 40th Int. Conf. Softw. Eng.*, 2018, pp. 327–337.
[60] Y. Sui, S. Ye, J. Xue, and P.-C. Yew, "Spas: Scalable path-sensitive pointer analysis on full-sparse SSA," in *Programming Languages and Systems*, H. Yang, Ed., Berlin, Germany: Springer, 2011, pp. 155–171.
[61] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang, "Pinpoint: Fast and precise sparse value flow analysis for million lines of code," *SIGPLAN Not.*, vol. 53, no. 4, pp. 693–706, Jun. 2018.
[62] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 209–224.
[63] D. Kroening and M. Tautschnig, "CBMC–C bounded model checker," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds., Berlin, Germany: Springer, 2014, pp. 389–391.
[64] Y. Guo, J. Zhou, P. Yao, Q. Shi, and C. Zhang, "Precise divide-by-zero detection with affirmative evidence," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng.*, 2022, pp. 1718–1729.
[65] Q. Shi, P. Yao, R. Wu, and C. Zhang, "Path-sensitive sparse analysis without path conditions," in *Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Des. Implementation*, 2021, pp. 930–943.
[66] Q. L. Le, A. Raad, J. Villard, J. Berdine, D. Dreyer, and P. W. O'Hearn, "Finding real bugs in big programs with incorrectness logic," in *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, pp. 1–27, Apr. 2022.
[67] S. Neuhaus and T. Zimmermann, "The beauty and the beast: Vulnerabilities in red hat's packages," in *Proc. USENIX Annu. Tech. Conf.*, 2009, pp. 383–396.
[68] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy*, 2016, pp. 85–96.
[69] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proc. ACM Prog. Lang.*, vol. 2, no. OOPSLA, pp. 147:1–147:25, Oct. 2018.
[70] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. IEEE 38th Int. Conf. Softw. Eng.*, 2016, pp. 297–308.
[71] A. Kharkar et al., "Learning to reduce false positives in analytic bug detectors," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng.*, 2022, pp. 1307–1316.

Xiao Cheng is currently working toward the PhD degree with the School of Computer Science, Faculty of Engineering and Information Technology, University of Technology Sydney (UTS), Sydney, Australia. His research interests include program analysis and machine learning.

Xu Nie is currently working toward the postgraduate degree with the School of Computer Science, Beijing University of Posts and Telecommunications (BUPT), Beijing, China. His research interests include program analysis and machine learning.

Ningke Li is currently working toward the postgraduate degree with the School of Cyber Science and Engineering, Huazhong University of Science and Technology (HUST), Wuhan, China. Her research interests include program analysis and machine learning.

Haoyu Wang (Member, IEEE) received the PhD degree from Peking University, in 2016. He is currently a full professor with the School of Cyber Science and Engineering, Huazhong University of Science and Technology (HUST). His research interests lie at the intersection of mobile system, privacy and security, and program analysis.

Zheng Zheng (Senior Member, IEEE) received the PhD degree in computer software and theory from the Chinese Academy of Sciences, Beijing, China, in 2006. He is currently a full professor in control science and engineering with the School of Automation Science and Electrical Engineering, Beihang University, Beijing. In 2014, he was a research scholar with the Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA. His research interests include software dependability, unmanned aerial vehicle path planning, artificial intelligence applications, and software fault localization.

Yulei Sui is an associate professor with the School of Computer Science, University of Technology Sydney (UTS). He is broadly interested in the research field of software engineering and programming languages, particularly interested in static and dynamic program analysis for software bug detection and compiler optimizations.