Towards Reliable AI for Source Code Understanding

Sahil Suneja
IBM Research
Yorktown Heights, NY, USA
suneja@us.ibm.com

Yunhui Zheng
IBM Research
Yorktown Heights, NY, USA
zhengyu@us.ibm.com

Yufan Zhuang
IBM Research
Yorktown Heights, NY, USA
yufan.zhuang@ibm.com

Jim A. Laredo
IBM Research
Yorktown Heights, NY, USA
laredoj@us.ibm.com

Alessandro Morari
IBM Research
Yorktown Heights, NY, USA
amorari@us.ibm.com

ABSTRACT

Cloud maturity and popularity have resulted in Open source software (OSS) proliferation. And, in turn, managing OSS code quality has become critical in ensuring sustainable Cloud growth. On this front, AI modeling has gained popularity in source code understanding tasks, promoted by the ready availability of large open codebases. However, we have been observing certain peculiarities with these black-boxes, motivating a call for their reliability to be verified before offsetting traditional code analysis. In this work, we highlight and organize different reliability issues affecting AI-for-code into three stages of an AI pipeline- data collection, model training, and prediction analysis. We highlight the need for concerted efforts from the research community to ensure credibility, accountability, and traceability for AI-for-code. For each stage, we discuss unique opportunities afforded by the source code and software engineering setting to improve AI reliability.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**; • **Computing methodologies → Machine learning**.

KEYWORDS

machine learning, reliability, signal awareness, explainability

SoCC '21, November 1–4, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11…$15.00
https://doi.org/10.1145/3472883.3486995

ACM Reference Format:
Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim A. Laredo, and Alessandro Morari. 2021. Towards Reliable AI for Source Code Understanding. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3472883.3486995

1 INTRODUCTION

The maturity and acceptance of Cloud for application deployment have fundamentally altered how code is developed, managed, and distributed. Specifically, open-source software (OSS) existence has flourished, with reliance on OSS steadily gained momentum along with the rise of microservices paradigm, cloud-native CI/CD pipelines, serverless platforms, amongst other technologies. The relationship is symbiotic, with OSS also playing a significant role in powering today's Cloud ecosystem [21], in addition to serving as foundations of a great majority of applications and industries [29, 55]

Although OSS helps developers build applications faster, it also puts developers at risk of bringing in defects or even security vulnerabilities hidden in those OSS components. According to a recent report [50], 11% of the OSS components developers build into their applications are known to be vulnerable, with 38 vulnerabilities on average. In addition, the number of cyber attacks targeting OSS has surged by 430% and new 0-day vulnerabilities are exploited in the wild within 3 days of public disclosure. Therefore, ensuring code quality remains critical to infrastructures and applications atop OSS components in every industry.

Scanning the source code or testing the executions to detect vulnerabilities has been a domain traditionally dominated by static and dynamic program analysis techniques. Recently, the ready availability of large code bases to 'train' upon and the machine learning success in natural language understanding, have promoted the entry of AI into the source code analysis space. AI promises to understand the semantics of the code and alleviate the shortcomings of traditional code analysis approaches, for example the high false positives of static analyzers, and the lack of completeness of dynamic

analysis [35, 57, 62]. There has been a rapid proliferation of AI models for source code understanding, with each model surpassing its predecessor in terms of statistical model quality measures such as F1 and accuracy. As AI enters the CI/CD workflow, it will start affecting most cloud-native deployments where CI/CD provides fundamental quality assurance.

However, we observe certain peculiarities with the AI for code understanding ecosystem (AI-for-code), motivating a call for their reliability to be verified, before offsetting the similarly-scrutinized traditional code analysis toolchain. Subpar results, if any, of such quality checks can, for example, help decide whether the tremendous amount of resources which model training increasingly consumes [10, 31], are better utilized for other kinds of code analysis or testing, such as fuzzing the application longer to catch more bugs.

In this paper, using primarily a vulnerability analysis usecase, we highlight and categorize the reliability issues affecting AI-for-code, into three different stages of an AI pipeline- (i) data collection, (ii) model training, and (iii) prediction analysis. We call for efforts at each stage from the research community to ensure:

- Credibility of data collection correctness
- Accountability in ensuring task-relevant signal learning by the model
- Traceability in terms of analyzing behavioral trends across data affecting model performance

For each stage, we propose potential solutions, tied together with a common theme of utilizing proven software engineering (SE) and data-driven techniques to assist with improving AI reliability. We strongly believe that a concerted effort to acknowledge and resolve these reliability concerns will go a long way in better utilizing the potential of AI for source code understanding. And, the source code & SE setting offers unique opportunities to realize this goal.

2 BACKGROUND

Recent advancement in computing hardware has lead to a resurgence of deep learning. The end-to-end learning process starts from collecting a dataset, vectorizing it, feeding it to a neural network to fit a desired function atop the dataset, training the network's weights as per an objective function (e.g., error minimization), and finally evaluating the model's performance on unseen test data using statistical measures such as F1 and accuracy.

Various neural network architectures have been proposed for different domains and data formats. Some popular ones which have been employed for learning over source code include: (i) Convolutional neural networks (CNN), which learn on image inputs. They have been applied to source code learning by treating code as a photo [48], (ii) Recurrent neural networks (RNN), designed to learn over sequential inputs such as text and audio. These treat code as a linear sequence of tokens [39, 48]. (iii) Graph neural networks (GNN), which deal with graph-structured data such as social networks and molecular structures. These have been used to operate on graph representations of source code [36, 66].

In the software engineering domain, AI-for-code has been employed in source code understanding tasks such as defect detection [17, 26, 39, 48], code summarization [5, 24, 28, 33, 43], code completion [15, 25, 54], function and variable naming [3, 4, 7, 41], amongst others [23, 37, 58, 60]. While ever more sophisticated models are emerging and pushing the state of the art in AI-for-code rapidly, we notice certain reliability issues in the AI-for-code ecosystem, echoing some prevailing doubts regarding model quality [2, 6, 8, 11, 12, 45]. Our frustrating experiences with sophisticated AI-for-code models failing in real-world settings have made us skeptical of their published performance numbers. Besides general AI-for-SE challenges observed in previous works [23, 37, 58], we particularly work towards exposing reliability caveats and argue for sanity checks such as whether models are learning tasks related signals, which are usually not reflected by non-domain specific and signal-agnostic model performance metrics such as F1 scores.

We concur with the unfairness of the criticism of traditional SE code analysis techniques, to be replaced with signal-agnostic AI. But we do believe in the potential for AI to learn task-relevant signals with proper guidance. We believe the source code setting offers unique opportunities to leverage SE for tackling reliability problems, which exist in AI in general. That is why we look towards SE+AI collaborations rather than competition and work towards ascertaining at the least that the models learn task-relevant source code constructs. We believe the vision we are proposing in this paper is complementary to generic AI reliability solutions.

In this work, we highlight and organize various AI-for-code reliability issues into different stages of an AI pipeline, call for concerted efforts at each stage, and highlight unique opportunities afforded by the source code & SE setting to improve AI reliability.

3 THE RELIABILITY PROBLEM

In this Section, we describe the different reliability problems with AI-for-code, while organizing them within the three stages of an AI pipeline.

3.1 Data Collection

High-quality data serves as the backbone of AI because a learning-based model is only as good as the data it is trained upon. A well-labeled, low-noise dataset significantly improves the chances of a model performing well, especially in

the typical, controlled, train-test model evaluation setting. The data collection methodology also dictates how the model will perform 'in the wild'– when it is faced with real-world code samples, untouched by any data collection bias.

There are multiple avenues by which reliability concerns creep into the data collection stage, including:

(1) **Cross-split Leaks:** When the test split contains duplicates or near-duplicates of the training set's code samples, it can inflate the models' reported performance metrics significantly, sometimes up to 100% [2].

(2) **Correlation Capture:** The way in which a dataset is curated, can lead to certain code artifacts being present in samples belonging to one class, but missing in the other. When such data nuances are irrelevant to the task at hand, such as specific keywords or identifier names, this increases the chances of learning unnatural correlations to create its classifier. Such a model may show good F1 scores, but would struggle in real-world deployments.

(3) **Erroneous Labeling:** Labeling for synthetic datasets [49] is relatively easy and error-free because they are generated from predefined patterns, but they can't capture the complexity and diversity of real-world code. On the other hand, many real-world datasets are based on relatively straightforward GitHub commit-message analysis, which introduces a source of error in the labeling logic. For example, in the case of [66], if a commit is believed to fix bugs, all functions patched by the commit are labeled as being buggy, which isn't true in many cases. Others directly use static analysis verdicts to label the samples [48]. But given its limitation of generating False Positives, a subsequent manual validation, if any, is not scalable owing to the dataset size required for training. This leads to potentially incorrect labeling, significantly limiting the quality of the models trained over such datasets [34]

(4) **Scope Limitation:** A large majority of existing source code datasets is limited to function-level scope, and do not capture key inter-procedural information crucial for complete code analysis. Models trained on such narrow scope will, with a high probability, miss several cases for the code analysis tasks sensitive to inter-procedural flows, limiting their effectiveness in practice. The effect can be significant as shown in [45, 51], which attributes an extra 20% gains achievable by incorporating inter-procedural flows. Similarly, useful context, such as bug location, is lacking in several datasets, which can be quite useful for cross-verification of dataset as well as model quality (See Section 4.2).

There are obvious solutions to some of the aforementioned problems, such as detecting and removing duplicates, and normalizing code to remove identifier dependency. A sound data collection and pre-processing strategy would indeed filter these away, but this isn't always the case. Similarly, not all datasets suffer from these limitations [39, 64]. These examples merely serve to highlight the potential pitfalls in data collection methodology, and as a sanity-check call to ensure that this isn't truly the case before proceeding to model training. Furthermore, not all are fixable after-the-fact, requiring more careful data collection to begin with.

Summary: *While some dataset-bias issues can be resolved by smart pre-processing prior to model ingestion, the most critical elements require a meticulous code curation strategy to maximize chances for reliable modeling down the line.*

3.2 Model Training

AI-for-code models also seem to suffer from the usual low generality and robustness frailties of AI. But only a part of this can be attributed to the aforementioned pitfalls in the data collection stage. To highlight these issues, we now present some observations which raise model reliability concerns. Note that with the following analysis, our goal is not to perform any adversarial attacks on the model, or create experiment settings to discredit the models. We only use these observations to highlight the need for deeper inspection into 'what' the model is learning.

(1) **Low Robustness:** It is reasonable to expect that a good model, which is correctly picking up the correct task-relevant signals, show robustness in its prediction. However, we observe that even a 99 F1 model flips its prediction on *only slightly* syntactically-different code variants. An occurrence of this can be seen in Table 1. In fact, in a vulnerability detection setting, for even a simple dataset such as s-bAbI [49], this model doesn't even focus on the bug to give its prediction in almost 40% of the buggy samples. The problem exists across different models (CNN, RNN, GNN), and datasets (synthetic and real-world) [53]. This raises a question as to whether even such high F1 models can be trusted, especially in a security sensitive setting, or whether another metric is needed which captures a model's task-relevant learning ability.

(2) **Weak Generalization:** We observe that a well performing model, trained on one dataset performs poorly on others. This can be seen in Table 2, for a vulnerability detection setting. This is concerning because the datasets used for this experiment all target almost the same finite set of common vulnerability types including null pointer dereference, buffer overflow, use-after-free, amongst others. This raises a question about what the models are actually learning– actual vulnerability-relevant signals or merely dataset nuances.

Table 1: Bad Robustness: a 99 F1 GNN model flips on only slightly syntactically different function variants. Dataset from [49]. Model from [52].

	`int a = 99;`
	`int b = 57;`
	`char arr[69];`
	`if (b < a) {`
	`  b = 78;`
	`}`
	`arr[b] = 'X';`
		`int a = 99;`
		`int b = 57;`
		`char arr[69];`
		`if (b) {`
		`  b = 78;`
		`}`
		`arr[b] = 'X';`

Ground Truth	Buggy	Buggy
Prediction	Buggy	Non-buggy

(3) **Exponential growth for incremental benefits:** The rapid proliferation of AI-for-code models is leading to ever more sophisticated models for incremental accuracy improvements. Table 3 highlights this rate of growth of *cost* vs. performance for different popular models. Confirmation of model quality is necessary to justify such costs of improvement. Otherwise, this time budget is better spent on other software testing or code analysis techniques, such as fuzzing.

The goal of the analysis is not to create settings which break AI-for-code models, but rather to highlight their frailties so that steps can be taken to understand and resolve these shortcomings, as we shall discuss in the next section. For example, the robustness example can be treated as an 'adversarial attack' on the models [27, 59]. And similarly, there can be defenses such as training the model on multiple code variants. But this line of thought is complementary to our goal. Such occurrences only triggered our suspicion regarding what the models are learning, and whether the metrics correctly capture task relevance. Similarly, the generalization problem can be 'fixed' by first combining the datasets together, and only after that splitting them into train and test subsets. But again, that's not our objective. This would only mask the issue for a controlled train-test evaluation environment, rather than addressing it head-on as to whether the models are even learning task-relevant signals.

Summary: *The AI frailties apparent in AI-for-code models raise questions regarding model reliability. Specifically, whether the models are truly learning code structures, or mere task-irrelevant dataset nuances. And whether this aspect of the model quality is being captured by the usual statistical measures of model performance.*

3.3 Prediction Analysis

The black-box nature of AI modeling precludes easily deciphering its learned logic, unlike the explicit rules and path/flow analysis of static analyzers and the execution tracing of dynamic analysis. Attempts to explain black-box learning for AI-for-code models have been limited. For example, for attention-based models, the attention weights can be interpreted as importance scores of certain features of the data [20]. Similarly, gradient-based methods have been used to generate heatmaps for input tokens to highlight regions of the source code considered important by the model [48]. However, these saliency maps are not fully accurate and can be misleading [1].

Table 2: Low Generalization: a well-performing CNN model trained on one dataset performs poorly on others. All datasets target C/C++ code. Model from [48].

Dataset for Model Training	F1 on Self Test Set	F1 on Others' Test Set
Juliet [42]	90	28
VulDeePecker [39]	83	27
SySeVR [38]	90	39
Draper [48]	54	35

Table 3: Huge models delivering only incremental F1 improvements. Use case: vulnerability detection on the Draper dataset [48]. Baseline BiLSTM and GGNN models derived from [66], 3GNN from [67].

Model	# Parameters (millions)	GPU Time (mins / epoch)	F1
BiLSTM	1.3	28	49.3
GGNN	209	5120	50.8
3GNN	208	7680	54.3

Other than white-box explanations, an important aspect missing from current research is an analysis of the model's predictions from the dataset perspective, beyond the non-domain specific statistical measures of the model prediction accuracy. This includes analyzing the characteristics of the samples which the model predicted correctly versus those that it got wrong, along with the corresponding confidence levels. Such post-facto prediction analysis can help uncover the model logic, by using the common code characteristics across correctly (and incorrectly) predicted samples to derive what signals from the code the model may be picking up, and whether they are relevant to the task at hand. If the learning behavior is as per expectation, this can help assert confidence in the model's reliability.

4 DATA-DRIVEN AND SE-ASSISTED SOLUTIONS

We now present potential solutions to improve AI-for-code reliability. As in the previous Section, we categorize these

solutions as addressing the issues with the three– data collection, modeling, and post-facto analysis– stages of the AI pipeline. For each stage, we first present how some of the ideas from the broader AI community, such as vision and natural language processing (NLP), can be adapted to AI-for-code. We suggest potential data-driven solutions by borrowing proven techniques from the SE domain. When delving into the specifics, we occasionally use vulnerability analysis as an example setting. However, our data-driven vision is independent of the target source code understanding tasks, being applicable to other settings in general such as those explored in recent model and dataset probing works [2, 46, 53] including code summarization, method naming, and variable misuse detection, amongst others.

4.1 Data Credibility

An important takeaway from the Section 3.1's data collection reliability concerns is the need for high-quality real-world datasets with trustworthy labels and rich context associated with the code samples. Some datasets come close to satisfying these requirements. For example, the CDG dataset [39], in part, consists of real-world programs derived from the National Vulnerability Database (NVD). Its label quality is good since it is based upon confirmed bugs. Its code samples include richer inter-procedural context, but they are curated in the form of only subsets of program slices as opposed to valid sub-programs. Although this potentially removes noise, but limits the ability of the models to learn natural and valid code structures. Furthermore, the real-world (NVD) portion of the dataset is limited in size and by itself is insufficient for model training.

To satisfy the size requirements, certain techniques used in the vision and NLP AI counterparts can be borrowed, such as crowd-sourcing [18, 19, 40], To improve the credibility of the data gathered in such a noisy setting, the AskIt! [9] system is able to learn which question is best directed to which worker. In the AI-for-code ecosystem, this translates to directing program samples to better-suited software developers or security engineers for ground truth labelling. CrowdFill [44] is another system which used secondary validation on workers' responses, in terms of up/down voting from other workers. This translates to a collaborative ground truth labeling by software engineers in the AI-for-code domain.

However, since human labeling is an expensive and scare resource, especially in the more skilled SE domain, automatic labeling is a preferred avenue. One approach is to use the vast collection of OSS repositories as the raw data source, together with smart filtering atop traditional code analysis tools to automatically derive a reliable dataset. We have been curating a dataset, D2A [64], by combining differential analysis over Infer static analyzer outputs.

```
// FFMpeg commit f4730a58454283ef1141be4152b53a2b45e5a200
// Infer: BUFFER_OVERRUN_L1 @ libavcodec/vp9.c:3730:65
//        Array access: Offset: [3, 4], Size: 2
// Auto-labeler:  False Positive
// Manual Review: False Positive

  88 struct {
  97     unsigned tx32p[2][4];
 115 } counts;

3648 static void adapt_probs(VP9Context *s)
3649 {
3723     if (s->s.h.txfmmode == TX_SWITCHABLE) {
3724         for (i = 0; i < 2; i++) {
3725             unsigned ... *c32 = s->counts.tx32p[i];
3730             adapt_prob(..., ..., ... + c32[3], ..., ...);
3733         }
3734     }
3868 }
```

Figure 1: D2A Auto-labeler can improve the label quality by identifying issues that are very likely to be false positives. In this example, Infer incorrectly reported a buffer overflow triggered by `c32[3]` at line 3730 because it thinks the size of array `c32` is 2, while Auto-labeler concluded it is a false positive thanks to its differential analysis.

Because static analyzers are known to produce an excessive number of false positives, it's unreliable to directly use them to label samples [48]. Instead, we assume some commits are bug-fixing changes and run static analysis on the before and after versions. If some issues disappear, they are very likely to be true positives. Similarly, if an issue detected in the before-fix version appears again in a later version, they are very likely to be false positives because they are not fixed. For example, Figure 1 shows a false alarm that was correctly suppressed by our approach.

The input to our automated dataset generation pipeline is just a git repository. We first analyze all commit messages using an NLP model trained on the NVD database to identify commits that are likely to fix bugs. Then, for each commit, we run the static analysis on the before-fix and after-fix version pairs. After getting the static analysis results, they are consolidated and labeled by the auto-labeler that runs the differential analysis logic. We manually reviewed a set of samples and found our approach can significantly improve the label quality. Although D2A samples were captured by the static analyzer Infer, because of the better labels, the models trained on D2A may potentially achieve better results for some issues, especially the false positives suppressed by the differential analysis.

However, although D2A raises the bar of dataset reliability, there exists scope for improvement as detected via

```
// FFMpeg commit hash dfa988ee5ea704ba761d004f0c27e7acc1fb4251
// Infer: NULL_DEREFERENCE @ libavfilter/audio.c:167
//        pointer `outlink->out_buf` assigned @ 165 could be
          null and is dereferenced @ 167
// Auto-labeler:  False Positive
// Manual Review: True Positive

35 AVFilterBufferRef *ff_default_get_audio_buffer(...)
37 {
38     AVFilterBufferRef *samplesref = NULL;
51     samplesref = avfilter_get_audio_buffer_ref_from_arrays(...);
54     if (!samplesref)
55         goto fail;
59 fail:
63     return samplesref;
64 }

157 void ff_default_filter_samples(...)
158 {
159     AVFilterLink *outlink = NULL;
162     outlink = ...
164     if (outlink) {
165         outlink->out_buf = ff_default_get_audio_buffer(...);
167         outlink->out_buf->pts = ...;
172     }
175 }
```

Figure 2: D2A Auto-labeler incorrectly flagged a real bug as a false positive. This was because D2A didn't include the bug-fixing commit `c9c7bc4` in the analysis, as its commit message doesn't fit the profile learned from bug-fixing commit messages obtained from NVD. This can be fixed by improving the commit message NLP model, which in turn emphasizes the importance of the data credibility for the NLP model training.

```
// OpenSSL commit hash dfa988ee5ea704ba761d004f0c27e7acc1fb4251
// Infer: BUFFER_OVERRUN_L2 @ crypto/buffer/buf_str.c:98
//        Array access: Offset added: [0, 2147483646] Size: 96
// Auto-labeler:  True Positive
// Manual Review: False Positive

64 size_t BUF_strnlen(const char *str, size_t maxlen)
65 {
66     const char *p;
68     for (p = str; maxlen-- != 0 && *p != '\0'; ++p) ;
70     return p - str;
71 }

80 char *BUF_strndup(const char *str, size_t siz)
81 {
82     char *ret;
87     siz = BUF_strnlen(str, siz);
92     ret = OPENSSL_malloc(siz + 1);
93     if (ret == NULL) {
95         return NULL;
96     }
98     memcpy(ret, str, siz);
102 }
```

Figure 3: D2A Auto-labeler incorrectly flagged an issue as a true positive. Although its differential analysis can effectively suppress false positives, if a reported issue disappears in the after-commit version (due to some inconsistencies in the static analyzer) and is never reported in a later version analyzed, the Auto-labeler may incorrectly assume it's fixed by that commit. Including more versions in the analysis or further refining the differential analysis heuristics can correct this labeling issue.

manual validation of auto-labeler verdicts. This can be seen in Figure 2 and Figure 3. In particular, Figure 2 shows the case where the auto-labeler incorrectly labels a buggy sample as being non-buggy. This particular occurrence is attributed to a mismatch between a bug-fixing commit message versus the corresponding commit-message-profile learned by the NLP model used inside D2A. This example in turn emphasizes the importance of the data credibility for the commit-message-analysis NLP model training. Similarly, Figure 3 shows an example where the auto-labeler incorrectly labels a non-buggy case as buggy, highlighting the need of more refined differential analysis heuristics and an improved commit message analysis model.

A 'cheaper' alternative to curating a dataset from scratch is to augment an existing dataset. This can additionally mitigate its limitations and bias such as size or class imbalance. Statistical methods such as SMOTE [13] can generate synthetic samples derived from the learned class distributions. While statistical sampling methods have appealing features and can be extended onto sequential data such as human texts, but since they operate in vector space, the synthesized data is not typically valid in terms of syntax and semantics in a source code setting. Such augmentation can similarly be achieved by using software engineering techniques, while also maintaining code validity. Delta Debugging [63] combined with compiler validation offers one way of generating new samples, with the added advantage of noise reduction. We discuss this more in Section 4.2.

4.2 Model Accountability

The observations of Section 3.2 raise questions about how reliable the models really are, in terms of whether the models are picking up the real signals relevant to a code understanding task. We call this aspect of the model's quality 'signal awareness', which is different than correctness. A model's job is to learn a separator between the different classes in a dataset, say 'buggy' or 'healthy' in a vulnerability detection setting. And the model is free to choose the best differentiating features (signals) it can find from the samples. It can arrive at this separator by picking up non-representative signals to the task at hand, such as unexpected correlations between code samples and sample lengths, or certain programming constructs, which may happen to differ for buggy or healthy samples. This would still be 'correct' learning from the model's perspective, which may even lead to great-looking performance numbers by using the usual statistical

measures of model quality that don't capture the model's signal awareness. But, going by such metrics is a dangerous call, especially in a security-sensitive setting, because it is not clear if such a model has truly learned what makes a code snippet buggy.

This calls for accountability in ensuring that the models are learning the correct logic relevant to code analysis, to generate trust in models if they are to be put into the field in competition to, or alongside, traditional static and dynamic analyzers. This is along the dimension of the broader AI trustworthiness research, which includes efforts to ensure that the models will generalize on unseen data (robustness) [27, 56, 59], will not reveal people's identity from training data (privacy) [22, 32], and will be fair when making decisions (fairness) [14, 16].

The first step towards improving model accountability is measuring its signal awareness, which can then be used to guide model evolutions. To this end, we have developed a data-driven approach to uncover a model's ability to capture task-relevant signals. For this, we borrow a fault isolation technique from the Software Engineering domain called Delta Debugging [63]. We use it to extract the bare minimum excerpt from a source code input, which a model needs to arrive at and stick to its original prediction. And then verify whether the minimal snippet has the same task profile as the original code [53]. For the vulnerability detection setting, the model's reliance on incorrect signals can then be uncovered when a vulnerability in the original code is missing in the minimal snippet, both of which the model however predicts as being vulnerable. By using this approach to probe them for signal awareness, we show a sharp performance drop for multiple AI-for-code models (CNN, RNN, GNN) across multiple datasets (both- synthetic and real-wordl), with Recall dropping from high 90s to sub-60s.

With our data-driven SE-assisted approach we are able to highlight that the models are picking up a lot of noise, presumably dataset nuances, as opposed to capturing task-relevant signals. This ability to measure model signal awareness now enables systematic exploration towards improving model accountability in terms of ensuring that the models are learning the relevant code constructs. The usual white-box hyperparameter-tuning techniques can be employed with the enhanced goal of additionally improving model signal awareness, together with the usual statistical measures of model quality [30, 61]. Purely data-driven black-box approaches can also be utilized such as program simplification (e.g. via slicing), data segmentation, as well as augmentation. We have observed encouraging preliminary results using such data-driven approaches. For example, we are able to improve the signal awareness of models by over 30% by adding a notion of code complexity metrics into model training (difficulty: low), and by over 50% by using a program-simplification based augmentation approach similar to the aforementioned Delta Debugging technique (difficulty: high).

4.3 Prediction Traceability

Section3.3 highlighted an under-explored avenue in AI-for-code research with regards to post-facto prediction analysis, which can be a valuable resource towards uncovering the code analysis logic learned by models.

Analyzing mispredictions is a useful means towards model refinement. Statistical methods, such as Naive Bayes, linear and logistic regression, are helpful in identifying input features contributing to mispredictions. But the automatic feature extraction quality of deep learning models precludes such interpretability. To add some transparency into these black boxes, general AI explanation methods have been developed. For example, gradient-based methods [65] compute activation maps of gradients over inputs to highlight input regions most influencing the model's prediction. Approximation-based methods [47] use interpretable surrogate models to approximate the deep learning model's behavior, and then use the surrogate to derive the feature importance ranking for the input.

In addition to such white-box approaches, AI-for-code offers unique opportunities to use SE techniques for purely data-driven analytics. One approach is to analyze the code characteristics of the samples which the model predicted correctly versus those that it got wrong. By using source code-related metrics, such as lines of code, loop count, decision points, etc., to group samples by prediction accuracy, it can help uncover what aspects of code the model may be able to grasp, and where it may be facing difficulties. This can then provide insights into improving the model's learning to target the code characteristics common to mispredictions. Promoting task-relevant learning in this way in turn helps improve model accountability, as discussed in the last section. The same group-by-metric approach can be used to trace the evolution of the model across iterations. For example, our preliminary experiments suggest models facing difficulty capturing bigger and harder samples, which improves across augmentation iterations, even without any explicit notion of code complexity in model training.

5 CONCLUSION

With AI-for-code services gaining popularity in the Cloud, in this work, we categorized the reliability issues affecting these into the different stages of an AI pipeline: (i) data collection, (ii) model training, and (iii) prediction analysis. For each stage, we propose potential solutions tied to proven software engineering and data-driven techniques. We call for efforts from the research community to ensure achieving credibility, accountability, and traceability in the AI-for-code ecosystem.

REFERENCES

[1] Julius Adebayo, Justin Gilmer, Michael Muelly, Ian J. Goodfellow, Moritz Hardt, and Been Kim. 2018. Sanity Checks for Saliency Maps. (2018), 9525–9536. https://proceedings.neurips.cc/paper/2018/hash/294a8ed24b1ad22ec2e7efea049b8737-Abstract.html

[2] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Onward! 2019*. 143–153. https://doi.org/10.1145/3359591.3359735

[3] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *ESEC/FSE 2015*. ACM, 38–49. https://doi.org/10.1145/2786805.2786849

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *ICLR 2018*. https://openreview.net/forum?id=BJOFETxR-

[5] M. Allamanis, H. Peng, and C. Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *ICML 2016*, Vol. 48. 2091–2100. http://proceedings.mlr.press/v48/allamanis16.html

[6] Abdul Ali Bangash, Hareem Sahar, Abram Hindle, and Karim Ali. 2020. On the time-based conclusion stability of cross-project defect prediction models. *Empir. Softw. Eng.* 25, 6 (2020), 5047–5083. https://doi.org/10.1007/s10664-020-09878-9

[7] Rohan Bavishi, Michael Pradel, and Koushik Sen. 2018. Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts. (2018). http://arxiv.org/abs/1809.05193

[8] Pavol Bielik and Martin T. Vechev. 2020. Adversarial Robustness for Code. In *ICML 2020 (Proceedings of Machine Learning Research, Vol. 119)*. 896–907. http://proceedings.mlr.press/v119/bielik20a.html

[9] R. Boim, O. Greenshpan, T. Milo, S. Novgorodov, N. Polyzotis, and W. Tan. 2012. Asking the Right Questions in Crowd Data Sourcing. In *IEEE 28th International Conference on Data Engineering (ICDE 2012)*. https://doi.org/10.1109/ICDE.2012.122

[10] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. 2016. An Analysis of Deep Neural Network Models for Practical Applications. (2016). http://arxiv.org/abs/1605.07678

[11] Joymallya Chakraborty, Suvodeep Majumder, and Tim Menzies. 2021. Bias in machine learning software: why? how? what to do?. In *ESEC/FSE 2021*. 429–440. https://doi.org/10.1145/3468264.3468537

[12] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. 2021. Deep Learning based Vulnerability Detection: Are We There Yet. *IEEE Trans. Software Eng.* (2021). https://doi.org/10.1109/TSE.2021.3087402

[13] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. SMOTE: Synthetic Minority Over-sampling Technique. *J. Artif. Intell. Res.* 16 (2002). https://doi.org/10.1613/jair.953

[14] Alexandra Chouldechova. 2017. Fair Prediction with Disparate Impact: A Study of Bias in Recidivism Prediction Instruments. *Big Data* 5, 2 (2017), 153–163. https://doi.org/10.1089/big.2016.0047

[15] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Penta, and G. Bavota. 2021. An Empirical Study on the Usage of BERT Models for Code Completion. In *MSR 2021*. 108–119. https://doi.org/10.1109/MSR52588.2021.00024

[16] Sam Corbett-Davies, Emma Pierson, Avi Feller, Sharad Goel, and Aziz Huq. 2017. Algorithmic Decision Making and the Cost of Fairness. In *KDD 2017*. 797–806. https://doi.org/10.1145/3097983.3098095

[17] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2021. Automatic Feature Learning for Predicting Vulnerable Software Components. *IEEE Trans. Software Eng.* 47, 1 (2021), 67–85. https://doi.org/10.1109/TSE.2018.2881961

[18] Thomas Davidson, Debasmita Bhattacharya, and Ingmar Weber. 2019. Racial Bias in Hate Speech and Abusive Language Detection Datasets. (2019). http://arxiv.org/abs/1905.12516

[19] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *(CVPR 2009*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[20] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *ICLR 2020*. https://openreview.net/forum?id=SJeqs6EFvB

[21] Bill Doerrfeld. 2021. How Open Source Software Powers Digital Innovation. https://devops.com/how-open-source-software-powers-digital-innovation/.

[22] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *TCC 2006*. 265–284. https://doi.org/10.1007/11681878_14

[23] Fábio F. Ferreira, Luciana Lourdes Silva, and Marco Tulio Valente. 2021. Software engineering meets deep learning: a mapping study. In *SAC 2021*. ACM, 1542–1549. https://doi.org/10.1145/3412841.3442029

[24] Jaroslav M. Fowkes, Pankajan Chanthirasegaran, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. 2017. Autofolding for Source Code Summarization. *IEEE Trans. Software Eng.* 43, 12 (2017), 1095–1109. https://doi.org/10.1109/TSE.2017.2664836

[25] Github. 2021. GitHub Copilot- Your AI pair programmer. https://copilot.github.com/.

[26] Mojdeh Golagha, Alexander Pretschner, and Lionel C. Briand. 2020. Can We Predict the Quality of Spectrum-based Fault Localization?. In *ICST 2020*. 4–15. https://doi.org/10.1109/ICST46399.2020.00012

[27] S. Gowal, K. Dvijotham, R. Stanforth, R. Bunel, C. Qin, J. Uesato, R. Arandjelovic, T. Mann, and P. Kohli. 2018. On the Effectiveness of Interval Bound Propagation for Training Verifiably Robust Models. (2018). http://arxiv.org/abs/1810.12715

[28] David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. 2020. Code to Comment "Translation": Data, Metrics, Baselining & Evaluation. In *ASE 2020*. 746–757. https://doi.org/10.1145/3324884.3416546

[29] Red Hat. 2021. The State of Enterprise Open Source. https://www.redhat.com/en/enterprise-open-source-report/2021.

[30] Xin He, Kaiyong Zhao, and Xiaowen Chu. 2021. AutoML: A survey of the state-of-the-art. *Knowl. Based Syst.* 212 (2021). https://doi.org/10.1016/j.knosys.2020.106622

[31] Joel Hestness, Newsha Ardalani, and Gregory F. Diamos. 2019. Beyond human-level accuracy: computational challenges in deep learning. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019*. 1–14. https://doi.org/10.1145/3293883.3295710

[32] B. Hitaj, G. Ateniese, and F. Pérez-Cruz. 2017. Deep Models Under the GAN: Information Leakage from Collaborative Deep Learning. In *CCS 2017*. 603–618. https://doi.org/10.1145/3133956.3134012

[33] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *ACL 2016*. https://doi.org/10.18653/v1/p16-1195

[34] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. 2019. The importance of accounting for real-world labelling when predicting software vulnerabilities. In *ESEC/FSE 2019*. 695–705. https://doi.org/10.1145/3338906.3338941

[35] Ugur Koc, Parsa Saadatpanah, Jeffrey S. Foster, and Adam A. Porter. 2017. Learning a classifier for false positive error reports emitted by static code analysis tools. In *MAPL 2017*. 35–42. https://doi.org/10.1145/3088525.3088675

[36] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved Code Summarization via a Graph Neural Network. In *ICPC 2020*. 184–195. https://doi.org/10.1145/3387904.3389268

[37] Xiaochen Li, He Jiang, Zhilei Ren, Ge Li, and Jingxuan Zhang. 2018. Deep Learning in Software Engineering. *CoRR* abs/1805.04825 (2018). arXiv:1805.04825 http://arxiv.org/abs/1805.04825

[38] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2021). https://doi.org/10.1109/TDSC.2021.3051525

[39] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. In *Network and Distributed System Security Symposium, NDSS 2018*. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_03A-2_Li_paper.pdf

[40] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context. In *European Conference on Computer Vision, ECCV 2014*. 740–755. https://doi.org/10.1007/978-3-319-10602-1_48

[41] K. Liu, D. Kim, T. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Traon. 2019. Learning to spot and refactor inconsistent method names. In *ICSE 2019*. 1–12. https://doi.org/10.1109/ICSE.2019.00019

[42] NIST. 2017. Juliet Test Suite for C/C++ V1.3. https://samate.nist.gov/SRD/testsuite.php

[43] Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond J. Mooney. 2020. Learning to Update Natural Language Comments Based on Code Changes. In *ACL 2020*. 1853–1868. https://doi.org/10.18653/v1/2020.acl-main.168

[44] Hyunjung Park and Jennifer Widom. 2014. CrowdFill: collecting structured data from the crowd. In *International Conference on Management of Data, SIGMOD 2014*. https://doi.org/10.1145/2588555.2610503

[45] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2020. On the performance of method-level bug prediction: A negative result. *J. Syst. Softw.* 161 (2020). https://doi.org/10.1016/j.jss.2019.110493

[46] Md. Rafiqul Islam Rabin, Vincent J. Hellendoorn, and Mohammad Amin Alipour. 2021. Understanding neural code intelligence through program simplification. In *ESEC/FSE 2021*. 441–452. https://doi.org/10.1145/3468264.3468539

[47] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *KDD 2016*. 1135–1144. https://doi.org/10.1145/2939672.2939778

[48] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *ICMLA 2018*. https://doi.org/10.1109/ICMLA.2018.00120

[49] Carson D. Sestili, William S. Snavely, and Nathan M. VanHoudnos. 2018. Towards security defect prediction with AI. *CoRR* abs/1808.09897 (2018). http://arxiv.org/abs/1808.09897

[50] Sonatype. 2020. State of the Software Supply Chain Report. https://www.sonatype.com/resources/white-paper-state-of-the-software-supply-chain-2020.

[51] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2Vec: value-flow-based precise code embedding. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 233:1–233:27. https://doi.org/10.1145/3428301

[52] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim Laredo, and Alessandro Morari. 2020. Learning to map source code to software vulnerability using code-as-a-graph. (2020). https://arxiv.org/abs/2006.08614

[53] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim Alain Laredo, and Alessandro Morari. 2021. Probing model signal-awareness via prediction-preserving input minimization. In *ESEC/FSE 2021*. 945–955. https://doi.org/10.1145/3468264.3468545

[54] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. 2021. Fast and Memory-Efficient Neural Code Completion. In *MSR 2021*. 329–340. https://doi.org/10.1109/MSR52588.2021.00045

[55] Synopsys. 2020. Open Source Security and Risk Analysis Report. https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html.

[56] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations, ICLR 2014*. http://arxiv.org/abs/1312.6199

[57] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Y. Aravkin. 2014. ALETHEIA: Improving the Usability of Static Security Analysis. In *CCS 2014*. 762–774. https://doi.org/10.1145/2660267.2660339

[58] Cody Watson, Nathan Cooper, David Nader-Palacio, Kevin Moran, and Denys Poshyvanyk. 2020. A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research. (2020). https://arxiv.org/abs/2009.06520

[59] T. Weng, H. Zhang, H. Chen, Z. Song, C. Hsieh, L. Daniel, D. Boning, and I. Dhillon. 2018. Towards Fast Computation of Certified Robustness for ReLU Networks. In *ICML 2018*. 5273–5282. http://proceedings.mlr.press/v80/weng18a.html

[60] Yanming Yang, Xin Xia, David Lo, and John C. Grundy. 2020. A Survey on Deep Learning for Software Engineering. *CoRR* abs/2011.14597 (2020). arXiv:2011.14597 https://arxiv.org/abs/2011.14597

[61] Q. Yao, M. Wang, H. Escalante, I. Guyon, Y. Hu, Y. Li, W. Tu, Q. Yang, and Y. Yu. 2018. Taking Human out of Learning Applications: A Survey on Automated Machine Learning. (2018). http://arxiv.org/abs/1810.13306

[62] Ulas Yuksel and Hasan Sözer. 2013. Automated Classification of Static Code Analysis Alerts: A Case Study. In *ICSM 2013*. 532–535. https://doi.org/10.1109/ICSM.2013.89

[63] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498

[64] Y. Zheng, S. Pujar, B. Lewis, L. Buratti, E. Epstein, B. Yang, J. Laredo, A. Morari, and Z. Su. 2021. D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis. In *ICSE-SEIP 2021*. 111–120. https://doi.org/10.1109/ICSE-SEIP52600.2021.00020

[65] Bolei Zhou, Aditya Khosla, Àgata Lapedriza, Aude Oliva, and Antonio Torralba. 2016. Learning Deep Features for Discriminative Localization. In *CVPR 2016*. 2921–2929. https://doi.org/10.1109/CVPR.2016.319

[66] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *NeurIPS 2019*. 10197–10207. https://proceedings.neurips.cc/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html

[67] Yufan Zhuang, Sahil Suneja, Veronika Thost, Giacomo Domeniconi, Alessandro Morari, and Jim Laredo. 2021. Software Vulnerability Detection via Deep Learning over Disaggregated Code Graph Representation. (2021). https://arxiv.org/abs/2109.03341