Infrared: A Meta Bug Detector

CHI ZHANG, Nanjing University, China
YU WANG, Nanjing University, China
LINZHANG WANG, Nanjing University, China

The recent breakthroughs in deep learning methods have sparked a wave of interest in learning-based bug detectors. Compared to the traditional static analysis tools, these bug detectors are directly learned from data, thus, easier to create. On the other hand, they are difficult to train, requiring a large amount of data which is not readily available. In this paper, we propose a new approach, called meta bug detection, which offers three crucial advantages over existing learning-based bug detectors: bug-type generic (i.e., capable of catching the types of bugs that are totally unobserved during training), self-explainable (i.e., capable of explaining its own prediction without any external interpretability methods) and sample efficient (i.e., requiring substantially less training data than standard bug detectors). Our extensive evaluation shows our meta bug detector (MBD) is effective in catching a variety of bugs including null pointer dereference, array index out-of-bound, file handle leak, and even data races in concurrent programs; in the process MBD also significantly outperforms several noteworthy baselines including Facebook Infer, a prominent static analysis tool, and FICS, the latest anomaly detection method.

Additional Key Words and Phrases: Meta Bug Detection, Graph Neural Networks

1 INTRODUCTION

The recent breakthroughs in deep learning have created a new opportunity for static bug finding which have been dominated by formal methods [Clarke et al. 2000; Cousot and Cousot 1977; King 1975]. This new strand of bug detection work [Allamanis et al. 2018; Pradel and Sen 2018] is motivated by the observation that bug patterns exist [Hovemeyer and Pugh 2004], thus, models can be trained to capture them. Compared to the traditional symbolic-, logic-based approaches, this data-driven approach helps reduce the complexity of the creation of static bug finders, which now can be directly learned from data using various machine learning models that are publicly available. However, such learning-based bug detection suffers from their own weaknesses, one in particular, is that they require large datasets of known bugs for training, which can be difficult to assemble. Specifically, manually curated benchmarks [Just et al. 2014; Saha et al. 2018; Widyasari et al. 2020] are hard to scale, especially for deep neural networks which are notoriously data-hungry. On the other hand, automated methods [Chen et al. 2019; Dinella et al. 2020; Wang et al. 2020] that treat code commits as sources of bugs hurt the quality of training data. To start, code commits are not necessarily bug fixes (e.g., refactoring, optimization, etc.), even if they are, they may not be the correct, principled patches that address the root cause of bugs, instead, they can be hacks, work-arounds that developers are known to use to make code pass test cases. Therefore, how to accurately identify the context of a bug given the location of the fix remains a significant challenge. Recently, a new bug mining method [Madeiral et al. 2019; Tomassi et al. 2019a] has been proposed that relies on modern continuous-integration approaches, like Travis-CI. As the technology is still at its early stage, it has not grown datasets to the scale needed to train deep neural networks.
Finally, there are also works that resort to synthetic bugs for the sufficiency of training data [Ferenc et al. 2020, 2018], considering synthetic bugs are unlikely to fit to the distribution that describes the real bugs, this approach is also flawed.

In addition to the lack of large-scale, high-quality datasets, existing learning-based bug detectors have another significant limitation: they can only learn patterns specific to the type of bugs that are given as training data, in other words, in test-time they would simply surrender to a type of bugs that were unobserved during training.

**A Meta Bug Detector (MBD).** To address the weaknesses of the aforementioned prior works, this paper proposes a new approach to learning bug detectors, called *meta bug detection*. Although meta bug detector (MBD) is also a learning-based bug detector to its core, it is built upon a fundamentally different concept: instead of learning the patterns specific to each bug type, the approach taken by almost all existing bug detectors, MBD opts to learn how bugs make programs inconsistent with the norm independently of the bug types — hence the name *Meta*. Under this guiding principle, MBD offers many benefits.

First and foremost, it reduces the difficulty level of the learning task. This is because compared to the bug patterns per se, the manners in which buggy programs contradict correct programs are more obvious and far easier to learn. We illustrate this point using the programs in Figure 1. For the well-known null pointer dereference bugs, a standard bug detector would learn to recognize a program execution on which a pointer first gets assigned the value null and then dereferenced later. This is no easy task considering an accurate bug detector must precisely capture the pattern without raising a large number of false positives. Specifically for the bug in Figure 1a, there is no clear signal that indicates this pattern indeed exists in the program, however, if we contrast the buggy program to the correct code in Figure 1b and 1c, the inconsistency emerges, that is, in the correct code, dereferencing a pointer happens after the very pointer is checked against the null value whereas the two events happen in a reverse order in the buggy code. For this reason, the program in Figure 1a is ranked by MBD as one of the most probable anomalies to those that correctly handled potential null pointers. A direct benefit that MBD enjoys from performing an easier learning task is the substantial reduction of training data. Our experiment shows a few thousand programs suffice to train a MBD with decent performance. This is in fact a crucial advantage of MBD’s over the existing learning-based bug detectors especially considering that large-scale, high-quality, real-world defect datasets are not available. Given this constraint, MBD’s approach of learning inconsistency between buggy and correct programs promises a pathway forward.

Second, MBD is capable of catching the type of bugs it has never encountered during training, a property that existing learning-based bug detectors do not possess. Because, as we explained, those bug detectors can only deal with the bug types they are trained on and there are little to no commonalities across distinct bug types. We show, however, the commonalities indeed exist from MBD’s perspective and how MBD exploits this phenomenon to detect the unseen type of bugs in test-time. Take the three programs in Figure 1d-1f for example. From a standard bug detector’s perspective, detecting the pattern of file handle leak — a handle to a file is requested but not freed when it is no longer used — is nothing like detecting the pattern of null pointer dereference. However, not only does MBD detect the program in Figure 1d as a contradiction to those in Figure 1e and 1f but does so by learning exclusively from examples prepared for null pointer dereference like those in Figure 1a-1c. The key to MBD’s success is that MBD detects the manner in which buggy code contradict correct code in the file handle leak is similar to that in the null pointer dereference. Specifically, a statement is misplaced in the buggy code w.r.t. to where the statement is in the correct code. Regarding the file handle leak, it’s the stream closing operation which should have been placed in the `finally` clause rather than the `try` clause, whereas
public boolean equals(Object obj) {
    Match o = (Match) obj;
    if (o.left != null && o.right != null) {
        this.right = this.right.strip();
    }
    if (... && this.right != null) {
        return ... && this.right.equals(o.right);
    }
}

(a) A null pointer dereference.

protected void processMethod(Object bean,
    String beanName, Method method) {
    ApolloJsonValue apolloJsonValue =
    AnnotationUtils.getAnnotation(method,...);
    if (apolloJsonValue == null)
        return;
    String placeholder = apolloJsonValue.value();
}

(b) A correct example w.r.t. (a).

private void checkTypeLiteral(JCTree node,
    Symbol sym) {
    if (sym == null || sym.kind != Kinds.Kind.TYP)
        return;
    Path jarPath =
    getJarPath(sym.enclClass(),
    platformJars);
    if (jarPath != null) {
    ...
}

(c) Another correct example w.r.t. (a).

public InputStream openFile() {
    final InputStream tmp = zipFile.openFile();
    if (tmp != null) try {
        final ZipInputStream zis =
        new ZipInputStream(tmp);
        ZipEntry ze = zis.getNextEntry();
        zis.close();
    } catch (IOException e) { ... }
}

(d) A file handle leak.

protected void processMethod(Object bean,
    String beanName, Method method) {
    ApolloJsonValue apolloJsonValue =
    AnnotationUtils.getAnnotation(method,...);
    if (apolloJsonValue == null)
        return;
    String placeholder = apolloJsonValue.value();
}

(e) A correct example w.r.t. (d).

private void checkTypeLiteral(JCTree node,
    Symbol sym) {
    if (sym == null || sym.kind != Kinds.Kind.TYP)
        return;
    Path jarPath =
    getJarPath(sym.enclClass(),
    platformJars);
    if (jarPath != null) {
    ...
}

(f) Another correct example w.r.t. (d).

public void writeFile(File inputFile,
    String jarPath) throws IOException {
    FileInputStream fis =
    new FileInputStream(inputFile);
    try {
        JarEntry entry = new JarEntry(jarPath);
        entry.setTime(inputFile.lastModified());
        writeEntry(fis, entry);
    } catch (IOException ex) {...} finally {
    fis.close();
}

Fig. 1. MBD’s perspective in catching null pointer dereference and file handle leak. The bug in (a) is caused by the wrong order of null check and dereferencing. The bug in (d) is triggered when code in try clause throws an exception, in which case, catch clause immediately get executed causing zis.close() to be skipped, hence the leak.

for the null pointer dereference, the dereferencing should have occurred after the null check as we explained previously.

Third, MBD addresses a key issue for which existing learning-based bug detectors have long been lamented. That is, apart from the prediction results, they can not provide concrete feedback upon which developers can act. This shortcoming casts a doubt on the practical utility of learning-based bug detectors. On the contrary, MBD is capable of explaining its own predictions, in particular, it follows a standard practice from static analysis tools for reporting bugs to end users. That is, for a program predicted as buggy, MBD outputs a buggy trace — a program path along which the bug can be triggered — to illustrate the bug. Our insight is to use inconsistencies between code snippets, the reason that MBD considers programs to be buggy, to produce the buggy trace. By means of global attention mechanism (which we will present later), we first capture the part of the buggy program MBD deems to be inconsistent with the correct code, and then reduce it into a
public class Match {
    private final String left;
    private final String right;
    public Match(String left, String right) {
        this.left = left;
        this.right = right;
    }
    public boolean equals(Object obj) {
        Match o = (Match) obj;
        if (o == null) return false;
        if (o.left != null && o.right != null) {
            ...
            this.left = this.left.strip();
            ...
            this.right = this.right.strip();
            ...
        }
        ...
    }
    void Main() {
        Match X = new Match("left", null);
        Match Y = new Match("left", "right");
        ...
        X.equals(Y);
    }
}

Fig. 2. The trace format in which MBD explains its prediction (using Figure 1a as an example): (1) underlined statements are the key steps of the trace; (2) the number marks the order in which statements are executed; (3) the statement within the bounding box is the program point at which the bug occurs.

program trace. For example, Figure 2 shows the buggy trace for MBD’s prediction for the program in Figure 1a. In light of this new explainability notion, we redefine MBD’s output for a buggy program to be the buggy trace, which is considered to be correct only if the buggy trace triggers the bug. This way MBD will behave like a static analysis tool when it comes to bug reporting, ensuring its usefulness in practice.

We note that the key concept that underlies the design of MBD is not new. Prior works took this approach to detecting errors in system code [Engler et al. 2001], incorrect API usages [Li et al. 2021; Yun et al. 2016] and other specific types of bugs [Hangal and Lam 2002; Liang et al. 2016; Monperrus et al. 2010]. However, MBD surpasses these works in many ways. First, it is capable of catching the type of bugs it has not encountered during training. On the contrary, anomaly detection works cannot deal with the classes of anomaly that their design was not modeled after. Second, it does not require human experts to define rules or heuristics specific to the type of bugs it aims to catch. Third, it learns from both buggy and correct programs, enabling the machine learning model to accurately distinguish the two classes of examples. In contrast, anomaly detection approaches often infer information from existing code by learning only from correct programs, hence, they usually suffer from a high false positive rate.

We face two technical challenges to realize MBD: (1) developing a machine learning model that can accurately capture the semantic patterns that programs denote, and does so in a transparent manner to facilitate its explainability efforts; (2) formulating MBD’s learning objective to accomplish its mission of being a bug-type generic detector. To resolve the first challenge, we utilize Graph Neural Network (GNN) for its supreme performance achieved in a wide range of programming-language tasks. Furthermore, we propose a global attention mechanism in which GNN learns to focus on nodes that express the essential characteristics of the graph and de-emphasize the rest. Note that this method differs from the widely used graph attention mechanism (e.g., graph attention network [Veličković et al. 2018]) in which nodes are prioritized within a neighborhood as opposed
In the meta bug detection setting, our global attention mechanism has shown to significantly improve GNN’s learning precision. In addition, it also provides an insight into GNN’s inner workings, laying the foundation of the aforementioned MBD’s self-explainability. As for the second challenge, we first prepare a dataset that manifests the common ways that buggy code is inconsistent with correct code. Then, for each type of inconsistency, we collect buggy programs (in smaller quantities deemed as the anomalies) and correct programs (in larger quantities forming the main cluster) in a way that every buggy program is inconsistent with every correct program in the same manner. Obviously, directly fitting MBD to the specific patterns that buggy programs display will not make it generalize to bugs beyond this dataset, instead, we train MBD to recognize the patterns of inconsistency between buggy and correct code. Technically, we define a loss function based on triplet loss [Schultz and Joachims 2003] with which MBD learns to embed buggy programs further away from the main cluster of correct programs as anomalies for each type of the inconsistency. Because the inconsistencies from which MBD learns can also describe other bugs outside of this dataset, our training regime enables MBD to catch the type of bugs that are unobserved during training.

We have realized MBD as a new bug detector, Infrared, and extensively evaluated it. First, we show, on 99 bugs we extracted from existing datasets, Infrared is effective, accurately separating bugs from correct programs without being exposed to the types of bugs during training. We further evaluate the effectiveness of Infrared on Defects4J [Just et al. 2014], a well-established bug dataset. Regarding the baseline of this experiment, we pick the two most effective static analysis tools — Eradicate [Era 2022] and Infer [Calcagno et al. 2011] — according to the results of an extensive study recently conducted by Tomassi and Rubio-González [2021]. We also include FICS [Ahmadi et al. 2021], arguably the state-of-the-art anomaly detection work. When trained on 1091 programs, Infrared outperforms all baselines by a comfortable margin. Finally, we evaluate Infrared in detecting data race, the most reliable indicator of errors in concurrency. We choose data race because we believe it presents a bigger challenge than the previous tasks due to the unique characteristics that concurrent programs exhibit. Remarkably, using the same 1091 training instances without a single concurrent program, Infrared beats out RacerD [Blackshear et al. 2018], one of the most recent advances in static race detection.

**Contributions.** We make the following main contributions:

- We introduce the concept of MBD for training bug type-generic, self-explainable, and sample efficient bug detectors.
- We define a loss function with which MBD learns to recognize how buggy code are inconsistent with the correct code in a bug type-generic fashion.
- We propose global graph attention which not only improves GNN’s learning precision but also provides an insight into GNN’s inner workings, effectively moving GNN a step closer to being an explainable model.
- We report our extensive evaluation of Infrared, a realization of MBD, on three benchmarks. Results show Infrared is effective, consistently outperforming several prominent baselines.

**Paper Organization.** The rest of the paper is structured as follows. Section 2 illustrates MBD’s workflow. Section 3 gives a detailed presentation on MBD’s learning approach. Next, we describe our extensive evaluation of Infrared, an implementation of MBD (Section 4). Finally, we survey related work (Section 5) and conclude (Section 6).

## 2 OVERVIEW

In this section, we present a high-level workflow of MBD, in particular, we introduce MBD’s components for both training and test.
Chi Zhang, Yu Wang, and Linzhang Wang

Machine Learning Model
③  Graph Neural Network
①  Training Data Collector
④  Global Attention
②  Program Representation
⑥  Relational Embedding
⑤  Loss Function

$L(r_a, r_p, r_n) = \max(0, \text{margin} + \text{dis}(r_a, r_p) - \text{dis}(r_a, r_n))$

(a) Training.

Test Codebase
⑦  Test Data Collector
Trained GNN
⑧  Prediction Interpreter
Output

(b) Test.

Fig. 3. MBD’s workflow.

① Training Data Collector Curating a quality dataset lays the foundation for effective MBD. The primary criteria we define for the dataset is the inclusion of a diverse set of inconsistent code (e.g., missing or misplacing checks, using wrong variables/operators/APIs, etc.). By inconsistency code, we mean a set of programs containing a few bugs and far more correct programs, among them every bug is inconsistent with every correct program.\(^1\) We first attempt to use existing tools [Ahmadi et al. 2021; Yun et al. 2016] on inconsistency code detection to collect data. Specifically, we set them up to run on older versions of several open-source projects that are known to be buggy. After a manual inspection, we find the results of those tools are exceedingly noisy as existing tools routinely miss the known bugs; even for the few they manage to catch, the respective correct code do not show any inconsistencies.

Next, we attempt to construct our dataset based on existing datasets of bugs. Although those datasets include detailed information for a bug (e.g., location of the bug, patches of the bug, the codebase the bug is in, etc.), they do not provide a compact code snippet to describe the bug, instead, they specify the version of the codebase in which the bug is found. Considering machine learning models are most effective when given accurate data to learn, we must identify from the whole program (i.e., the entire codebase) only the code related to the bug. In addition, finding the corresponding correct programs after the (more precise representation of the) bug is obtained is another challenge we must overcome. We build a fully automated pipeline to resolve both

\(^1\)we include far more correct programs than bugs to reflect the ratio between buggy and correct programs in any mature codebase in the real-world.

Proc. ACM Program. Lang., Vol. 1, No. CONF, Article 1. Publication date: January 2018.
challenges. First, given the version of a codebase that contains a bug, we compute a slice [Weiser 1981] from the whole program using the statement where the bug occurs (which is provided by existing datasets) as the criterion. Since our slice contains all the dependencies related to the bug, it is a safe representation of the bug for our inconsistent code. Next, we patch an extracted bug slice to obtain its fixed version, which we use to search for correct programs that are inconsistent with the bug. The reason we can not directly take the fixed version as the correct program is to prevent a distribution discrepancy between training and test because bug detectors will never be asked to distinguish bugs from their fixed versions in test-time. For convenience, the search of the correct code happens within the codebase of the bug, in particular, we collect syntactically similar programs (based on the tree-edit distance between abstract syntax trees) to the fixed version of the bug. This is a reasonable design choice because the semantic inconsistency already exists between the bug and its fixed version, thus, programs that are syntactically similar to the fixed version are likely to maintain the semantic inconsistency. Finally, for efficiency purposes, we aggregate inconsistent code that share a large number of correct programs, a strong indicator that those inconsistent code express the same kind of inconsistency. Inevitably, our automated pipeline would induce noise into our dataset as not every correct program is guaranteed to be inconsistent with every bug in the inconsistent code. However, this is acceptable because MBD will not get to work with perfectly inconsistent code during test either. If MBD is pressured to learn to be tolerant on data with some noise, it will better cope with the noisy data it sees in the wild.

2. Program Representation (Section 3.2). We represent programs using program dependence graph (PDG), a principled, systematic graph representation, which makes explicit both the data and control dependence for each operation in a program. Because PDG does not preserve the execution order among statements that are control dependent on the same statement. We introduce a new type of edges to link those statements in the order they are executed. To refine the granularity of our graph representation, we encode each statement — a node in a standard PDG — with their abstract syntax trees (AST). Accordingly, we make the following two modifications to our graph. First, we reset the control dependence edges and execution order edges to connect the root nodes of statements’ AST. Second, we push down the data dependence edges to connect the leaf nodes of statements’ AST. Figure 4 depicts the graph representation of the program in Figure 1d.

3. Graph Neural Network. For performance consideration, we choose graph neural network (GNN) [Gori et al. 2005] as the underlying machine learning model for MBD, specifically, the category of GNNs that is powered by message-passing mechanism [Gilmer et al. 2017]. At a high-level, the goal of GNN is to learn node embeddings based on the graph structure. Such node embeddings are numeric vectors that represent the state of nodes. Technically, GNN approaches this problem in a step-wise manner. In each step, each node first sends messages to its neighboring nodes, then aggregates the messages it received from the neighboring nodes to compute its representation for the next step. Below we use GGNN’s [Li et al. 2016] approach as an illustrative example. Equation 1 computes a message $m_n$ for node $n$ through $f(\cdot)$ (e.g., a linear function) on the representations of its neighboring nodes $\Omega_n$. Next, a GRU takes $m_n$ and $h_n$ — the current representation of node $n$ — to compute the new state $h_n'$ (Equation 2).

$$m_n = \sum_{u \in \Omega_n} f(h_u)$$  (1)  
$$h_n' = GRU(m_n, h_n)$$  (2)

4. Global Graph Attention (Section 3.3). Despite GNN’s cutting-edge performance, they can still suffer from precision issues, especially when dealing with large graphs. This is mainly because of the high cost of its underlying message-passing procedure. Ideally, every node should pass messages directly or indirectly to every reachable node in the graph to allow sufficient information
exchange. However, such an expensive propagation is hard to scale to large graphs without incurring a significant precision loss. To overcome this challenge, we propose a global graph attention mechanism that prioritizes nodes in the message-passing procedure. Our intuition is that if GNN can focus on nodes that are essential to expressing the characteristics of graphs, GNN would become more precise in capturing patterns from the graph data. To contextualize our intuition in the settings of bug detection, it is clear that a large part of the buggy code is usually irrelevant to the semantics bugs denote, therefore, we can downplay a considerable number of nodes from the graph, which not only helps to reduce the cost of the underlying message-passing process, but more importantly, increases GNN’s overall accuracy in recognizing the bug patterns from the few remaining nodes.

Through this global attention mechanism, an attention weight will be allocated to every node in the graph based on its relation with a meta node, a new node that we add into the graph for approximating the global properties of the graph. The attention weight then determines the level of influence a node can have on GNN’s message-passing procedure. Specifically, nodes at the core of the graph will receive higher attention weights, thus, start dominating how GNN learns from the graph. On the contrary, nodes on the fringe of the graph will receive lower attention weights. As a result, they become marginalized and eventually fade out of the graph. Section 3.3 presents the details of our global attention mechanism.

**5 MBD’s Loss Function** (Section 3.4). In order to be bug-generic, MBD cannot adopt the approach of existing bug detectors which directly learns the patterns from specific types of bugs. However, since bugs are always inconsistent with the correct programs regardless of their types, we can train MBD to recognize the inconsistency in a bug-type generic manner. As described in ① *Training Data Collector*, every inconsistency code contains far more correct programs than buggy programs. Therefore, we can train MBD to recognize the inconsistency in an inconsistent code by placing every bug further away from the average of all programs than every correct program in the embedding
space. Equation 3 gives the mathematical formulation.

\[ \forall p \in P \forall q \in Q |g_\theta(p) - \bar{r}| > |g_\theta(q) - \bar{r}| \]  

(3)

where \( g_\theta \) is the GNN with parameter \( \theta; P/Q \) denotes the set of buggy/correct programs for one type of inconsistency, and \( \bar{r} \), the average program, is defined as follows.

\[ \bar{r} = \frac{\sum_{p \in P} g_\theta(p) + \sum_{q \in Q} g_\theta(q)}{|P \cup Q|} \]  

(4)

In Section 3.4, we discuss how to convert Equation 3 into MBD’s loss function based on the triplet loss paradigm [Schultz and Joachims 2003].

6 Relational Embedding. A potential issue with MBD’s learning approach (formulated by Equation 3) is that the embeddings are myopic in the sense that each program \( x \in P \cup Q \) gets embedded by \( g_\theta \) independently of other programs. Instead, we argue \((P \cup Q) \setminus \{x\}\) should have an influence on how we embed \( x \) through \( g_\theta \). The reason is that some program \( y \) in \((P \cup Q) \setminus \{x\}\) may be particularly indicative of \( x \)’s correctness such as the program in Figure 1b is of the program in Figure 1a, in which case it is beneficial to account \( y \) while embedding \( x \). Some evidence of this is presented in Section 4.5. For this reason, we propose to embed each program \( x \in P \cup Q \) through a function which takes as input \((P \cup Q) \setminus \{x\}\) in addition to \( x \) (i.e., \( g_\theta(x) \) becomes \( g_\theta(x, (P \cup Q) \setminus \{x\}) \)). Thus, \((P \cup Q) \setminus \{x\}\) now can affect how we embed \( x \). Technically, we apply the attention mechanism to compute a weighted sum of the embeddings of programs in \((P \cup Q) \setminus \{x\}\) w.r.t \( x \) (Equation 5). This way programs that should have played a bigger role in embedding \( x \) will receive more attention.

\[ R = (P \cup Q) \setminus \{x\} \]

\[ z = \sum_{y \in R} \alpha(g_\theta(x), g_\theta(y))g_\theta(y) \]  

(5)

\[ \alpha(g_\theta(x), g_\theta(y)) = \frac{\exp(g_\theta(x) \odot g_\theta(y))}{\sum_{w \in R} \exp(g_\theta(x) \odot g_\theta(w))} \]  

(6)

Instead of directly taking \( z \) as the relational embedding of \( x \), we then perform several steps of reads from a Long-Short Term Memory (LSTM) [Hochreiter and Schmidhuber 1997] in an attempt to add depth to the attention. In principle, this allows the model to increasingly attend to the crucial programs in \( R \) and de-emphasize the rest. Technically, we follow Vinyals et al. [2016] from their “Process” block — a module designed for similar purposes to ours.

\[ \hat{\delta}_\theta^{k+1}(x), c^{k+1} = LSTM(g_\theta(x), [\delta_\theta^k(x), z^k], c^k) \]

\[ \delta_\theta^{k+1}(x) = \hat{\delta}_\theta^{k+1}(x) + g_\theta(x) \]  

(7)

where \( \delta_\theta(x) \) is the relational embedding of \( x \) after \( k \)-steps of read, \( \delta_\theta^0(x) \) is initialized to \( g_\theta(x) \), \( z^0 \) is defined in Equation 5. Later, \( g_\theta(x)/g_\theta(y) \) will be replaced with \( \delta_\theta^k(x)/\delta_\theta^k(y) \) for computing \( z^k; c^0 \) is assigned randomly.

7 Test Data Collector. Directly applying a trained MBD to find bugs in a test codebase is not the ideal way forward. This is because the notion of inconsistency is valid only if test programs share common properties (e.g., inconsistency has no validity concerning programs that trim a string and sort an array). Deploying MBD to scan an entire codebase will unavoidably make MBD work with totally unrelated code, which clearly violates the premise of MBD’s approach. Therefore, the first task we need to resolve is to collect the set of test programs to which MBD is applicable for identifying the inconsistencies. A simple solution is to group together test programs that could potentially induce the same type of bugs. Technically, w.r.t the nature of each common type of bugs, we develop a data-flow analysis to collect test programs. Considering the machine learning model
Chi Zhang, Yu Wang, and Linzhang Wang

(further down MBD’s workflow) will not be able to bring back any bugs if they are missed by this analysis, we design our analysis to be sound. Below, we illustrate this analysis using null pointer dereference as an example. We first find out the variables \( V \) in the entire program that are assigned the value \( \textit{null} \) at a program point \( l_1 \) and dereferenced later at \( l_2 \) while ignoring the feasibility of the path from \( l_1 \) to \( l_2 \). For each \( v \in V \), we then use the dereference on \( v \) at \( l_2 \) as the criterion to compute a program slice [Weiser 1981] in which the null pointer dereference is guaranteed to be triggered under the same condition as it is in the original program. Finally, we regard all such slices to be the test code for null pointer dereference. As we explained above, the goal of this analysis is not to miss any potential bugs in the entire program. Therefore, the precision of the analysis is not a concern because the machine learning model, which is at the core of MBD, is designed precisely for separating real bugs from false alarms. Interested readers may refer to the supplemental material (Section A) for the details of the data-flow analysis for some common types of bugs.

8 Prediction Interpreter (Section 3.5). After we identify the test programs for each type of bugs, MBD first computes the raw embedding (i.e., \( g_\theta(\cdot) \)) and then the relational embedding (i.e., \( \delta^K_\theta(\cdot) \)) of each test program. Next, MBD ranks test programs based on how distant their embeddings is to the mean point of this test set. Since in test-time the vast amount of the code MBD sees will be bug-free, the mean point reflects the characteristic of the main cluster formed by the correct code, therefore, following the methodology of the training of MBD, programs that are distant to the mean point are deemed to be the bugs inconsistent with the correct code.

Given test programs that MBD predicts to be buggy, we proceed to explain its predictions. In general, we utilize the global attention mechanism as the basis of our explanations. In particular, we set out to find, in the buggy program, the path of statements that receive the highest global attention scores. Our intuition is that because of the goal of MBD’s global attention mechanism — emphasizing the most essential part of a program graph that separates bugs from correct code, and a fact about the test code — the inconsistent program constructs are the constant differentiator between buggy and correct programs, the statements that are highly attended will be parts of the program where the inconsistency lies. Technically, we compute a buggy trace out of those statements to explain the prediction that MBD makes for the buggy programs. This is a standard practice we follow from the classic static analysis tools, which ensures that MBD is as useful as static analysis tools regarding bug reporting. In Section 3.5, we explain in detail how to find the path covering the highest attended statements in the buggy program efficiently.

3 MBD’S GRAPH LEARNING APPROACH

In this section, we present MBD’s learning approach, in particular, we give the technical details for the key components in the training and test of MBD.

3.1 Definitions

First, we define a standard bug detector. Conceptually, a bug detector is a model that assigns a label to an input program.

Definition 3.1. (Bug Detector) A bug detector is a K-way classifier \( M : x \rightarrow \rho \in \mathbb{R}^K \) that for each label class predicts the probability \( \rho_i \in [0, 1] \) (s.t. \( \sum_{i=1}^{K} \rho_i = 1 \)) that a program \( x \) is a member of the class \( i \).

Definition 3.1 does not exclude \( P \) for being a correct program since bug free can also be a prediction class. Given such a bug detector, the label of \( x \) is \( L \) s.t. \( L = \text{argmax}_i \rho_i \) where \( L \in [1, K] \), meaning, it can not predict any label outside of the \( K \) classes provided in the training set. Now we define MBD as follows:
Definition 3.2. (Meta Bug Detector) A meta bug detector is a ranking system $T$ for a set of programs $X$ s.t. $T : \forall x \in X(x, X/\{x\}) \rightarrow v \in \mathbb{R}$ that produces a scalar $v$ for $x$, which is the distance between $x$ and the mean point of $X$ in the embedding space.

In essence, MBD only examines how $x$ contradicts with or conforms to the other programs in $X$, thus the label of $x$ is irrelevant. As a result, MBD is capable of catching the type of bugs that are unobserved during training, a property that standard bug detectors do not possess. Interpreting MBD’s results is also straightforward: programs that have a larger distance $v$ are more likely to be buggy, and the label of the bug is already determined from the data-flow analysis introduced in Test Data Collector in Section 2.

Definition 3.3. (Correctness of MBD) Suppose all bug-free programs are assigned a label $\epsilon$. MBD’s prediction for a bug-free program $P$ is correct if MBD predicts the label of $P$ to be $\epsilon$. Suppose a program $Q$ is buggy and for each bug $\mu$ in $Q$, there exists a sequence statements $T$ s.t. the execution of $T$ with an input $i$ triggers $\mu$ (denoted by $exe(T, i) = \mu$). Let $\overline{T}$ be the global minimum trace for $T$ i.e., $\exists \overline{T} (exe(\overline{T}, i) = \mu) \land (|\overline{T}| < |T|)$ where $|\cdot|$ denotes the length of a sequence. MBD’s prediction for $Q$ is correct if MBD outputs a sequence of statements $T'$ for each $\mu$ s.t. $\overline{T} \leq T' \leq T$ where $\leq$ denotes subsequence and read as ‘is a subsequence of’.

Our intuition is that $\overline{T}$ represents in theory the gold standard trace for explaining the bug $\mu$ because executing $\overline{T}$ with some input triggers $\mu$, and every statement in $\overline{T}$ is necessary. Therefore, if MBD outputs a sequence $T'$ that subsumes $\overline{T}$, then $T'$ is a correct explanation for $\mu$. Since we design our analysis for collecting test programs to be sound, those test programs may have false positives (e.g., statements that $\mu$ depends on are in fact unnecessary to the occurrence of $\mu$), therefore, $T'$, a path of one such test program, may also include false positives. This is the reason we do not require $T'$ to be identical to $\overline{T}$. Regarding $T'$ being a subsequence of $T$, this is because the analysis ruled out true negatives in $T$ (e.g., statements completely unrelated to $\mu$) that no longer appear in $T'$. For interested readers, we left an example in the supplemental material (Section B) to further illustrate the validity of Definition 3.3.

3.2 An Interprocedural Program Dependence Graph

We design MBD to be an interprocedural bug finder, in particular, we extend our PDG-based Program Representation in Section 2 to include procedures and procedure calls. We use “procedure” as a generic term referring to both the main function and the auxiliary procedures when the distinction between the two is irrelevant.

In simplest terms, an interprocedural PDG includes a program dependence graph, which represents the main function, procedure dependence graphs, which represent the auxiliary procedures, and additional edges, which represent direct dependences between a call site and the called procedure. Regarding the design of the representation of parameter passing between procedures, we adopt the approach proposed in Horwitz et al. [1988]. Specifically, we incorporate five new kinds of dependencies.
of vertices to our PDG-based program graphs. A call site is represented using a call-site vertex; information transfer is represented using four kinds of parameter vertices. On the calling side, information transfer is represented by a set of vertices called actual-in and actual-out vertices. These vertices, which are control dependent on the call-site vertex, represent assignment statements that copy the values of the actual parameters to the call temporaries and from the return temporaries, respectively. Similarly, information transfer in the called procedure is represented by a set of vertices called formal-in and formal-out vertices. These vertices, which are control dependent on the procedure’s entry vertex, represent assignment statements that copy the values of the formal parameters from the call temporaries and to the return temporaries, respectively. Using the program in Figure 5, Figure 6 depicts an example of the five new kinds of vertices. The rectangle represents the call site vertex. Ovals with dotted/dashed lines represent the actual in/out vertices whereas diamonds with dotted/dashed lines represent the formal in/out vertices. Note that we do not consider the actual-out and formal-out vertices if the corresponding parameter is passed by value like the parameter $j$ passed to the swap function in Figure 5. As before, all the new vertices are also represented by ASTs in the interprocedural PDG.

As for data dependences between procedures, they flow from actual-in vertices to formal-in vertices, and from formal-out vertices to actual-out vertices. Connecting procedure dependence graphs to form an interprocedural program dependence graph is straightforward, involving the addition of three kinds of edges: (1) a control dependence edge is added from each call-site vertex to the corresponding procedure entry vertex; (2) a data dependence edge is added from each actual-in vertex at a call site to the corresponding formal-in vertex in the called procedure; (3) another data dependence edge is added from each formal-out vertex in the called procedure to the corresponding actual-out vertex at the call site. For interesting readers, we left the entire interprocedural program dependence graph of the program in Figure 5 to the supplemental material (Section C).
Another issue in constructing an interprocedural program dependence graph is identifying, at a
given method invocation, which method is being invoked amid dynamic dispatch or virtual function
mechanism. This will not be a problem for programs used in the training phase because we know
the exact paths along which bugs are triggered. For test code, we run interprocedural points-to
analysis to track the origin of an object (i.e., its allocation site) on which a virtual method is invoked.
Technically, we perform flow and context-sensitive Andersen style points-to analysis with call-site
sensitivity on the entire codebase. If the analysis returns multiple origins of an object, for each one,
we create a procedure dependence graph to represent its version of the method implementation,
and then connect all procedure graphs to the program dependence graph of the calling method.

3.3 Global Graph Attention
As a result of our adoption of the interprocedural program dependence graph, the size of program
graphs increase, which makes GNN’s message-passing mechanism, an already expensive procedure,
even more costly. To overcome this challenge, we propose a global attention mechanism which helps
to improve GNN’s learning precision in meta bug detection. Our insight is that a few statements
already sufficiently display the inconsistencies between correct and buggy code. Therefore, GNN
should learn to emphasize such nodes and downplay the rest in the graph. We realize this idea based
on the classic attention mechanism [Bahdanau et al. 2015; Luong et al. 2015; Vaswani et al. 2017].
Below, we first present the technical details of the global graph attention, and then we explain how
it makes GNN more effective in detecting the inconsistencies between correct and buggy code.

We first add to the graph another node, called meta node, which connects every native node
in the graph via a new type of edge. Next, we compute an attention score (Equation 8) for each
native node by taking the dot product between its embedding and the embedding of the meta
node $H$. Equation 9 then converts the attention scores into attention weights which can be deemed
as a quota that limits the role each node plays in the message-passing procedure, in particular,
Equation 10 presents how the message $m_n$ for node $n$ is produced after the attention weight of each
of its neighboring nodes, $\alpha_u$, is taken into account; Equation 11 shows how node $n$’s representation
is updated with its own attention weight $\alpha_n$.

\[
e_n = h_n \odot H \tag{8}
\]
\[
\alpha_n = \frac{exp(e_n)}{\sum_{i=1}^{N} exp(e_i)} \tag{9}
\]
\[
m_n = \sum_{u \in \Omega_n} \alpha_u f(h_u) \tag{10}
\]
\[
h' = GRU(m_n, \alpha_n h_n) \tag{11}
\]

Initially, the embedding of the meta node is assigned randomly. After a round of message-passing,
we update it as follows:

\[
H' = \sum_{n \in V} \alpha_n h_n
\]

where $V$ denotes the set of native nodes in the graph.

To achieve MBD’ learning objective, or in essence minimize GNN’s training loss, GNN will
inevitably attend to the nodes that display the inconsistency between buggy and correct program
graphs since they are the biggest differentiators of the two sets of graphs. In other words, even if
some buggy programs substantially differ from some correct programs in surface-level textual/syn-
tactic features, given nodes that display the inconsistencies between the two sets of graphs are the
constant difference maker for every buggy and correct program, GNN will nudge toward them in
order to minimize its training loss. While such nodes are receiving higher attention weights, the
meta node starts to reflect their characteristics, as a result, more attention will be given to those
nodes in later rounds of message-passing, subsequently, GNN separates the two sets of graphs even further, and the virtuous circle carries on.

We note that our global graph attention differs from the widely used graph attention mechanism (e.g., GAT [Veličković et al. 2018]). The goal of global attention mechanisms is to emphasize nodes that reflect the essential characteristics of the entire graph while downplaying the rest. We realize this goal by assigning an attention weight for every node based on how it relates to every other node in the graph (e.g., through meta node). In contrast, GAT prioritizes nodes only within the neighborhood of a node (denoted by node \( n \)). The attention weight of each neighboring node (denoted by node \( u \)) is purely determined by node \( n \) and \( u \). It is rather unclear how such a localized attention mechanism can be applied to prioritize nodes within the entire graph. Although graph pooling [Cangea et al. 2018; Gao and Ji 2021; Lee et al. 2019] share some similar characteristics with the global attention mechanism, they are designed for a different purpose: preventing models from overfitting by reducing the size of data representations. For this reason, they cut nodes from graphs in an explicit manner; and thus require users to specify the number of nodes to be retained in the graph via a hyperparameter. In contrast, our global graph attention aims to improve the precision of GNN, and it does so by highlighting the key nodes without altering the graph structure.

3.4 MBD’s Loss Function

First, we give an overview of the triple loss function from which we derive MBD’s loss function. Consider a setting where the training data is prepared into a set of triples \((\bar{x}, x_+, x_-)\) where \( \bar{x} \) is an anchor input, \( x_+ \) is a positive input of the same class as \( \bar{x} \), \( x_- \) is a negative input of a different class from \( \bar{x} \). The objective of triplet loss models is to embed each triple in the way that samples with the same label, \( \bar{x} \) and \( x_+ \), are smaller in distance than those with different labels, \( \bar{x} \) and \( x_- \). The equation below formulates this concept.

\[
L(\bar{x}, x_+, x_-) = \max(\|\phi(\bar{x}) - \phi(x_+)\|^2 - \|\phi(\bar{x}) - \phi(x_-)\|^2 + \epsilon, 0)
\]

where \( \|\cdot\|^2 \) denotes the Euclidean distance function and \( \phi(\cdot) \) is the embedding function.

Clearly, \( L(\bar{x}, x_+, x_-) \) equals to 0 when the distance between \( \bar{x} \) and \( x_- \) is greater than that between any \( \bar{x} \) and \( x_+ \) by at least \( \epsilon \). The purpose of \( \epsilon \) is that, when the representations produced for \( \bar{x} \) and \( x_- \) are distant enough, further training will save the effort from continuing to enlarge that distance, instead, it can focus on other more difficult triplets. Following the convention in the literature, we set the value of \( \epsilon \) to be 1.

Summing over all triplets in the training set, we obtain the triplet loss function:

\[
\mathcal{L} = \sum_{i=1}^{N} L(\bar{x}^i, x_+^i, x_-^i)
\]

Now, we can instantiate the above equation to create MBD’s loss function according to its learning objective (Equation 3). To sort our training programs into a set of triplets, we first create a set of pairs by pairing each buggy program with each correct program for each type of inconsistency. Then we use the average embedding of those programs as the anchor for each pair. By substituting \( p \), a correct program, for \( x_+ \), \( q \), a buggy program, for \( x_- \) and \( \bar{r} \) for \( \bar{x} \), we create MBD’s loss function for one type of inconsistency below:

\[
\mathcal{L}(P, Q) = \sum_{q \in Q} \sum_{p \in P} \max(\|\bar{r} - \delta^k_q(p)\|^2 - \|\bar{r} - \delta^k(p)\|^2 + \epsilon, 0)
\]
where $\bar{r}$ is defined in Equation 4 and $\delta_\alpha^k(\cdot)$ is defined in Equation 7. Summing losses over all types of inconsistencies, we obtain MBD’s loss function:

$$L(U) = \sum_{U_i \in \mathcal{U}} L(U_i)$$

(12)

where $\mathcal{U}$ denotes the entire set of training programs, $U_i$ is the training programs for one type of inconsistency, $U_i$ represents the buggy/correct programs in $U$.

### 3.5 Explaining MBD’s Predictions

In this section, we give a detailed presentation on MBD’s self-explainability, specifically, how MBD produces a buggy traces out of the buggy program. We omit MBD’s procedure for detecting buggy programs during test, which is already presented in Prediction Interpreter in Section 2.

As we explained earlier, the global attention mechanism in principle would attend to program statements that constitute the inconsistency between buggy and correct code. However, directly presenting the highest attended statements in the buggy program as explanations lacks clarity, and thus, hinders the utility of MBD. Instead, we set out to find the program path in the buggy program that contains the highest attended statements, the same way static analysis tools educate developers when they report bugs.

First, we formulate the problem of trace generation more precisely: find a feasible program path along which statements add up to the highest attention scores. Note that a path satisfying this requirement is not guaranteed to trigger the bug. However, this should not refute our formulation of the problem. Because ultimately it’s the model’s fault for not learning the right features to describe the bug. Considering the interpreter’s job is to merely present what the model has learned, we believe our formulation is reasonable.

Given this problem formulation, an optimal solution requires an exponential time algorithm, instead, we propose a 1-basic block off algorithm that can solve this problem much more efficiently. The high-level key steps are as follows: (1) we automatically take into account statements that are integral to the type of the bug (e.g., null assignment and dereferencing statement for null pointer dereference), in addition, we mark the criterion we use for slicing (described in Test Data Collector in Section 2) to be the point at which the bug occurs. (2) we then consider 10 other statements that receive the highest attention scores. We select 10 because they together make up over 99 percent of the attention scores for any program in our extensive experiments; (3) out of the statements identified in step (1) and (2), we identify the path of the highest attention scores without considering the path feasibility, meaning, we ignore statements in an if/else clause only if the opposite else/if clause contains statements combined into a higher attention score. An exception, which in fact applies to the whole procedure, is that we never remove integral statements even if they have the lowest attention scores. (4) Next, we check the feasibility of the path found in the previous step, if it is infeasible, we identify a new (i.e., unexplored) path by deselecting statements with the lowest possible attention scores. In other words, the new path either circumvents the if branch of deselected statements (if it is a stand-alone if block) or takes the opposite branch to deselected statements; (5) if the new path is again unfeasible, we first recover the path found at step (3), and then deselect statements with the next lowest possible attention scores. This is the reason we name our algorithm 1-basic block off as the new path identified always covers exact 1 different basic block from the original path identified in step (3); (6) we repeat step (5) until we find a feasible program path or deselect all statements identified in step (2). In theory, this algorithm may not find the path that satisfies the formulation above, however, this is not a concern because the formulation should not be deemed as a requirement — because solutions guaranteed to satisfy the formulation may still violate Definition 3.3 — but rather a guidance that leads MBD
Algorithm 1 Find a feasible path with the highest attention score.

1: **procedure** FINDFEASIBLEPATH(program, attention_score)
   \> Get the integral statements from the program.
2: \hspace{1em} integral_statements ← GetIntegralStatements(program)
   \> Get top-10 highest-attended statements.
3: \hspace{1em} statements ← ReturnStmtsWithTopNScore(program, attention_score, 10)
4: \hspace{1em} statements.Incorporate(integral_statements)
   \> Calculate the scores of the branches covering these statements.
5: \hspace{1em} branches_score_map ← CalcBranchScores(statements, attention_score)
   \> Select the path with the highest score and get its branches.
6: \hspace{1em} path, branches ← SelectPathAndBranches(branches_score_map)
   \> Each candidate path is constructed from variable path instead of the last candidate path.
7: \hspace{1em} cand_path ← path
8: \hspace{1em} **while** !IsPathFeasible(cand_path) and !branches.Empty() **do**
   \> Pop the lowest score branch in the branch set.
9: \hspace{2em} lowest_score_branch ← PopBranchWithLowestScore(branches)
10: \hspace{2em} opposite_branches ← FindOppositeBranches(lowest_score_branch)
11: \hspace{2em} if opposite_branches.Empty() **then**
12: \hspace{3em} cand_path ← RemoveBranch(path, lowest_score_branch)
13: \hspace{3em} **end if**
14: \hspace{2em} else
15: \hspace{3em} cand_path ← FindSubpath(path, lowest_score_branch, opposite_branches)
16: \hspace{2em} **end if**
17: \hspace{1em} **end while**
18: \hspace{1em} return cand_path
19: **end procedure**

19: **function** FINDSUBPATH(path, branch, branches)
   \> We enumerate all subpaths within the branches including their sub-branches.
20: \hspace{1em} subpaths ← EnumerateAllSubpathsInBranches(branches)
21: \hspace{1em} **while** !subpaths.Empty() **do**
22: \hspace{2em} subpath ← PopSubpathWithHighestScore(subpaths)
23: \hspace{2em} **end if**
24: \hspace{2em} cand_path ← ReplaceSubpath(path, branch, subpath)
25: \hspace{1em} **end while**
26: **end function**

to discover the intended path. In practice, the algorithm works well and the reason is simple: the
global attention weights are already accurate reflection of the buggy trace, if anything, it only
needs minor corrections which is precisely what this 1-basic block off approach does.

Algorithm 1 gives the details. We first obtain the integral statements according to the type of the
bug. Next, we collect ten statements with the highest attention score via function ReturnStmtsWith-
TopNScore. If some of those statements happen to reside in branches, we compute the score for
a branch as the sum of its statements (CalcBranchAndScore). Note that we compute the branch
score in a recursive manner to properly handle the nested branches. SelectPathAndBranches then
selects a path that covers the branches with the highest scores. If neither of the opposite branches contain any top-10 highest-attended statement, we randomly select a branch to cover. Then we determine whether or not the path is feasible, if not, we retrieve the branch in the path with the lowest attention score, and proceed with the following two scenarios. (1) if the retrieved branch is a single `if` block, we just remove it; (2) if there are other branches associated with the retrieved branch (e.g., one or multiple `else` blocks), we replace the retrieved branch with a subpath within an associated branch which makes the entire path feasible. In particular, we enumerate all subpaths within each associated branch (taking into account the nested branches) through \textit{EnumerateAllSubpathsInBranches}, and look for the subpath with the highest score (\textit{PopSubpathWithHighestScore}) to replace the retrieved branch (\textit{ReplaceSubPath}). The first one we find that produces a feasible path is declared as the buggy trace. The algorithm will always terminate after, in the worst case, it has dealt with all branches in the path discovered at line 6.

As a common practice, we perform under-constrained symbolic execution [Ramos and Engler 2015] to determine the feasibility of a path. Specifically, we start executing a test program from the entry point of the procedure that contains the first statement (e.g., an integral statement or a top-10 attended statement) and unroll every loop once to reduce the number of paths to be explored.

Concerning the usability of MBD, we decorate the buggy trace as follows. First, we mark the statement at which the bug occurs within a bounding box, and cut off all subsequent statements in the trace. Second, we number statements in the trace according to the order in which they are executed. Finally, we underline a few highest attended statements as key steps of the trace.

4 EVALUATION

We implement MBD into Infrared, and evaluate it on three benchmarks including the well-established datasets like Defects4J [Just et al. 2014] and JBench [Gao et al. 2018]. We also examine in-depth the results of MBD’s to reveal the strengths and weaknesses of MBD. Finally, we set out to confirm the necessity of each key component of Infrared.

4.1 Implementation

Due to the ample availability of Java datasets, we target a variety of bugs in Java programs. For a different language, Infrared only requires a front-end analyzer that extracts program dependence graphs for programs in the new language, the rest of the workflow is language-agnostic. We use DECKARD [Jiang et al. 2007] to detect syntactic similar programs. We construct program dependence graphs using Spoon [Pawlak et al. 2015], an open-source library for analyzing Java source code. We use Doop [Bravenboer and Smaragdakis 2009] for pointer analysis. We realize the prediction interpreter on top of angr [Shoshtaihshili et al. 2016], a fairly mature framework that supports the symbolic execution in Java. We realize Infrared in Tensorflow based on GGNN’s implementation. All experiments are performed on a 3.7GHz i7-8700K machine with 32GB RAM and NVIDIA GTX 1080 GPU.

4.2 Proof-of-Concept

First and foremost, we conduct a proof-of-concept for meta bug detection. That is can Infrared at the very least separate bugs, the kind of which it has never observed during training, from the correct code that we automatically mined for our dataset.

\textbf{Dataset.} We build an automated pipeline (presented in \textbf{Training Data Collector} in Section 2) to curate the dataset. We collect bugs from Bugs.jar [Saha et al. 2018] and BugSwarm [Tomassi et al. 2019b], two mature datasets of real bugs in Java programs. A caveat is that both datasets trace bugs from exceptions that programs throw at the run time, which can induce noise to their data. As an
evidence, we notice some exceptions are thrown independently of the program that the datasets include as buggy. For example, some bugs in Bugs.jar are linked to the FileNot FoundException, which is thrown when the file to be accessed is not found in the system. After a careful investigation on the root cause of those bugs, we conclude that they are unrelated to programming mistakes. In some cases, it’s due to the lack of sufficient privilege to write the file at the first place. A few others are caused by the network issues that prevent the file to be transmitted from a remote server. Also, there are exceptions triggered due to the violation of the project-specific assumptions. For example, in reality APIs are never called in the order specified in the test code; or input arrays are always of a certain size. Furthermore, we found some exceptions are triggered by the runtime environment which do not have static features (e.g., OutOfMemoryException). Finally, we only consider bugs that are reproducible through the automated scripts provided by the dataset. We sort bugs in our dataset into five categories: null pointer dereference, number format exception, array index out-of-bound, stack overflow exception, and file handle leak — under which bugs are guaranteed to be valid.

Cross Validation. To evaluate the effectiveness of Infrared, we conduct a cross-validation on our dataset. We withdraw programs for each type of bugs from the entire dataset in turn as the validation set, and leave the rest as the training set. Our goal is to test if Infrared can indeed catch the type of bugs it has never seen during training. Concerning the validity of the experiment, we manually confirm that the type of inconsistency overlaps between the validation set and the training set for every type of bugs.

Metric. We emphasize again that we examine MBD’s performance strictly according to Definition 3.3 throughout our evaluation. Because Infrared is, in essence, a ranking system of test programs, we define a cut-off point $N$ as the separation between buggy and correct programs. In other words, we consider the top $N$ programs to be buggy and inspect how many bugs are covered (denoted by TP), how many bugs are missed (denoted by FN), and how many false alarms are reported (denoted by FP). The value of $N$ can be set according to developers’ intention whether they favor MBD to produce less false warnings or identify more bugs. For instance, $N = 1$ can represent an opportunistic approach developers tend to adopt due to their unwillingness for an extensive bug inspection. $N = 5$ (or even a larger number), on the other hand, represents a conservative approach in which all bugs should have been caught because developers will not inspect reports further down the list. Considering this evaluation only serves as a proof-as-concept, we simply set $N$ to be the number of buggy programs w.r.t. the type of bugs withdrawn to the validation set.

Baselines. We use GGNN, the base model of Infrared, and GINN [Wang et al. 2020], one of the most recent advances in learning-based bug detection, as the baselines for this experiment. Because both Infrared and GINN are built upon GGNN, the three models have the same set of hyperparameters.

| Bug Types                        | # Programs |
|----------------------------------|------------|
| Null pointer dereference (NPD)   | 35         |
| Number format exception (NFE)    | 16         |
| Array index out-of-bound (AIO)   | 23         |
| Stack overflow exception (SOE)   | 14         |
| File handle leak (FHL)           | 11         |
| Total                            | 99         |

Table 1. Details about our dataset.
For a fair comparison, we use the default values provided by GGNN’s implementation for all models. We train GGNN and GINN to predict only two labels regardless of the type of the bugs: buggy or correct. To counter the lopsided ratio between the correct and bug code in the training set, we give six times higher weights to the loss that models incur on buggy programs than correct programs so that models will prioritize the minimization of its loss on buggy code over correct code. Since GGNN and GINN directly learn bug patterns, we expect them to perform poorly on this dataset.

**Results.** Table 2 shows Infrared’s results in cross validation. Clearly, Infrared is quite precise across all types of bugs, yielding around 0.35 false positive/negative ratio, which is significantly better than GGNN and GINN. This study confirms that Infrared, which learns the manner in which buggy code is inconsistent with correct code, can detect the type of bugs that are unobserved during training. On the contrary, existing models, which learn specific bug patterns, can not deal with the type of bugs which they are not trained on.

### 4.3 Evaluating Infrared on Defects4J

In this experiment, we evaluate the effectiveness of Infrared in catching bugs from real-world codebases. This is the typical way static bug finders are used except in our case bugs are already identified. For groundtruth, we extract 79 bugs from Defects4J [Just et al. 2014], arguably the most well-established dataset of bugs in Java programs. Our bugs come in three different types: null pointer dereference (39), array index out-of-bound (34), and number format exception (6). In addition to reasons for which we can not use all bugs in Bug.jar, we also exclude bugs in Defects4J that are related to the test suite themselves (e.g., test code written in JUnit). We ignore them because they are far less significant than bugs occurred in the real codebases. For null pointer dereference, the 39 bugs used in our evaluation is a superset of the 26 bugs that Tomassi and Rubio-González [2021] extract from Defect4J for their evaluation.

**Baselines.** Although Defects4J has been the standard benchmark for evaluating fault localization works [Masri 2015; Zou et al. 2021], we do not choose them as baselines because they are predominately powered by dynamic approaches that assume the pre-existence of test cases that can trigger the bug. It is clear such test cases are not available in the setting of static bug finding. Instead, we pick baseline methods based on an extensive study conducted by Tomassi and Rubio-González [2021] on the real-world effectiveness of several noteworthy static bug finders, especially that they also use Defects4J as an evaluation benchmark. According to their results, we select the top performers: Eradicate, which find the most bugs, and Infer, which has the highest precision. Eradicate is a type checker that checks for @Nullable annotations in Java programs by performing a flow-sensitive analysis to propagate null-related information through assignments and calls. Infer, a prominent static-analysis tool developed by Facebook, uses bi-abduction analysis to find bugs. Despite being built upon sound programming language theory [Calcagno et al. 2011], Infer is an unsound tool in practice. Tomassi and Rubio-González [2021] discuss in detail sources of Infer’s unsoundness (at Table II). Nonetheless, our experiments set aside the soundness notion while focusing on the effectiveness of each tool in finding bugs in real-world programs. For a learning-based method, we pick FICS [Ahmadi et al. 2021], arguably the state-of-the-art anomaly detection tool, as another

### Table 2. Results of cross validation. The acronyms of bug types are introduced in Table 1.

|       | NPD | NFE | AIO | SOE | FHL |
|-------|-----|-----|-----|-----|-----|
|       | TP  | FP  | FN  | TP  | FP  | FN  | TP  | FP  | FN  | TP  | FP  | FN  |
| GINN  | 8   | 36  | 27  | 2   | 12  | 14  | 3   | 18  | 11  | 2   | 15  | 9   |
| GGNN  | 4   | 42  | 31  | 2   | 14  | 14  | 1   | 18  | 22  | 3   | 16  | 11  |
| Infrared | 24  | 11  | 11  | 11  | 5   | 5   | 14  | 9   | 9   | 9   | 5   | 5   | 7   | 4   | 4   |

Proc. ACM Program. Lang., Vol. 1, No. CONF, Article 1. Publication date: January 2018.
Table 3. Comparing Infrared against Eradicate, Infer and FICS on Defects4J. The acronyms of bug types are introduced in Table 1. + is the number of bugs a tool finds but not others. When counting + for a baseline, we consider Infrared in top-5 prediction mode which is Infrared’s the best performance. Because there does not exist a static analysis tool in the literature that can check AIOE and NFE in a fully automated manner, we only compare Infrared against FICS. For interested readers, we have left a comprehensive, 1-on-1 bug comparison among all tools to the supplemental material (Section D). We emphasize again all tools are unsound, however, this comparison sets aside the soundness issue while focusing on the effectiveness of each tool in catching bugs in Defects4J, the same way Tomassi and Rubio-González [2021] evaluate static bug finders.

|                | NPD          | AIO          | NFE          |
|----------------|--------------|--------------|--------------|
|                | TP | FP | FN | +  | TP | FP | FN | +  | TP | FP | FN | +  |
| Infer          | 3  | 459| 36 | 0  | -  | -  | -  | -  | -  | -  | -  | -  |
| Eradicate      | 11 | 16476| 28 | 6  | -  | -  | -  | -  | -  | -  | -  | -  |
| FICS           | 3  | 4182| 36 | 2  | 6  | 3555| 28 | 6  | 1  | 732| 5  | 0  |
| Infrared       | top-1 | 6  | 4  | 33 | 3  | 6  | 3  | 28 | 6  | 2  | 1  | 4  | 1  |
|                | top-5 | 11 | 39 | 28 | 8  | 10 | 35 | 24 | 10 | 4  | 11 | 2  | 3  |

baseline of this experiment. FICS is aimed at C/C++ which works on LLVM IR [LLVM 2021]. To make it compatible with Java programs, we use JLang [Daniel Donenfeld 2019] as a front-end to generate LLVM IR of Java programs, which then get passed to FICS. To ensure the correctness of our implementations, we perform differential testing on the behavior of two executable files: one is x86 machine code obtained via Java source code $\rightarrow$ LLVM IR $\rightarrow$ x86 machine code and the other is Java class file directly compiled from Java source code. Our implementation passes all test cases in the test suite provided by JLang. A key hyperparameter of FICS’s is the code granularity at which the analysis is performed. We use 1-Con (i.e., one basic block) and full-Con (i.e., the whole data dependence graph) with a similarity threshold of 95%, which gives the best performance of FICS on this dataset. We have also tuned the other hyperparameters to make sure FICS performs to the best of its ability. We compare Infrared against Eradicate and Infer in catching null pointer dereference bugs, the only kind of bugs used in this experiment that Eradicate and Infer can handle. Since we did not find any static analysis tool in the literature that can catch array index out-of-bound or number format exception in a fully automated fashion, we only compare Infrared with FICS on those two kinds of bugs.

Test Procedure. We follow MBD’s testing workflow depicted in Figure 3b. At the end of the workflow, Infrared produces a ranking of test programs w.r.t. each type of bugs. Given that Infrared is evaluated in a real setting of static bug finding, we set $N$ to be 1 and 5 to mimic two contrasted approaches developers can adopt to inspecting bug reports. As we explained in the previous section, $N = 1$ enables developers to focus on bugs that are highly likely to be buggy, alternatively, $N = 5$ would let developers inspect all code that are potentially buggy. The model used in this experiment is retrained with the entire dataset (Table 1) with the same set of hyperparameters as before.

Results. Considering Eradicate and Infer are regarded as the state-of-the-art in static bug finding, we make a tremendous effort to confirm the correctness of their results in our evaluation. First, we compare the results of Eradicate and Infer to those reported in Tomassi and Rubio-González [2021] on the 26 bugs that both evaluations use. After a thorough inspection, we find the two sets of results indeed match. We describe the details of our inspection in the supplemental material (Section E). Second, since Eradicate and Infer analyze the remaining 13 null pointer dereference in the exact same manner as they analyze the 26 shared bugs (e.g., the same version of Eradicate and Infer, the same commands for invoking Eradicate and Infer), we believe their results are also correct. Finally, in case Infer have found new bugs not included in Defects4J, we manually examine
Infrared: A Meta Bug Detector

Infer’s 459 false reports. We rule out 428 on our own as clear false alarms, and left the developers the rest which are all declared to be false in the end. We did not manually inspect Eradicate’s (or FICS’s) false reports for two reasons: first, given the number of false alarms they produce, such a manual inspection will be extremely time-consuming; and second, even if Defect4J missed bugs for some projects, all tools will be affected the same way, thus, the results will not bias toward Infrared. Below, we discuss the results of our evaluation.

For null pointer dereference, in top-1 prediction mode, Infrared already finds more bugs than all baselines except Eradicate. In particular, it produces orders of magnitude less false alarms than all baselines. In top-5 prediction mode, Infrared finds the same number of bugs as Eradicate but with 400+ times less false warnings. Results for array index out-of-bound and number format exception tell the same story: Infrared not only finds more bugs than FICS, but has a substantially higher precision. Despite Infrared’s far superior effectiveness, we find that the baselines are in fact complementary to Infrared. Specifically, Eradicate and Infrared find only two common null pointer dereference while FICS does not overlap with Infrared at all in array index out-of-bound bugs (more details can be found in Section D in supplemental material). This finding suggest that each tool has its own strengths and weaknesses, and no tool is completely replaceable.

We use a buggy (Figure 7a) and its reference program (Figure 7b) to showcase how Infrared detects null pointer dereference. The reference program is the correct program that receives the highest attention weight (Equation 6) w.r.t. the embedding of the buggy program. As we explained in § Relational Embedding in Section 2, the program of the highest attention weight is the most indicative of the incorrectness of the bug, thus, we use it as a reference to illustrate the bug. The statements receiving the highest attention scores in both programs are highlighted in shadowbox. The inconsistency between the two programs is that the correct program always checks the object to be dereferenced (e.g., line 9, 15, and 20) whereas the buggy program misses the check for the last dereference at line 26.

(a) A null pointer dereference. (b) The reference program.

Fig. 7. A bug in Defects4J that Infrared caught.

Infer’s 459 false reports. We rule out 428 on our own as clear false alarms, and left the developers the rest which are all declared to be false in the end. We did not manually inspect Eradicate’s (or FICS’s) false reports for two reasons: first, given the number of false alarms they produce, such a manual inspection will be extremely time-consuming; and second, even if Defect4J missed bugs for some projects, all tools will be affected the same way, thus, the results will not bias toward Infrared. Below, we discuss the results of our evaluation.

For null pointer dereference, in top-1 prediction mode, Infrared already finds more bugs than all baselines except Eradicate. In particular, it produces orders of magnitude less false alarms than all baselines. In top-5 prediction mode, Infrared finds the same number of bugs as Eradicate but with 400+ times less false warnings. Results for array index out-of-bound and number format exception tell the same story: Infrared not only finds more bugs than FICS, but has a substantially higher precision. Despite Infrared’s far superior effectiveness, we find that the baselines are in fact complementary to Infrared. Specifically, Eradicate and Infrared find only two common null pointer dereference while FICS does not overlap with Infrared at all in array index out-of-bound bugs (more details can be found in Section D in supplemental material). This finding suggest that each tool has its own strengths and weaknesses, and no tool is completely replaceable.

We use a buggy (Figure 7a) and its reference program (Figure 7b) to showcase how Infrared detects null pointer dereference. The reference program is the correct program that receives the highest attention weight (Equation 6) w.r.t. the embedding of the buggy program. As we explained in § Relational Embedding in Section 2, the program of the highest attention weight is the most indicative of the incorrectness of the bug, thus, we use it as a reference to illustrate the bug. The statements receiving the highest attention scores in both programs are highlighted in shadowbox. The inconsistency between the two programs is that the correct program always checks the object to be dereferenced (e.g., line 9, 15, and 20) whereas the buggy program misses the check for the last dereference at line 26.
### 4.4 Evaluating Infrared on JBench

Next, we evaluate the effectiveness of Infrared in catching data races, an important class of programming errors in concurrency that is notoriously difficult to detect and fix. This is a specially challenging task considering Infrared has never seen data races or even concurrent programs from training programs. Nevertheless, we think Infrared can still perform well against data races considering many of the common errors that lead to data races are similar to those that appeared in our benchmark. For example, missing locks can be deemed as missing checks/statements; using non-atomic operations (e.g., `i++` instead of `i.getAndIncrement()`) are often related to API misuse.

Similar to the previous experiment, we evaluate Infrared in real scenarios of static bug finding. We use bugs provided in JBench [Gao et al. 2018], one of the most recent benchmark suites of data races in Java programs, as the groundtruth for this experiment. JBench contains 985 data races in total. We discard 204 extracted from research prototypes (e.g., Joshi et al. [2009], Huang et al. [2014], etc.) as they are quite different from concurrent programs in real-world applications. We also exclude races triggered in library methods for which we can not generate the program dependence graph. We left with 657 for this evaluation.

As the first step, we design the light-weight analysis to collect test programs that potentially have data races. A crucial task is to find shared variables. We follow the algorithm proposed by Kahlon et al. [2007]. Specifically, we perform a data-flow analysis to detect complete update sequences from $p$ to $q$ (e.g., $p_1 = p; p_2 = p_1; \ldots; q = p_k$) that are followed by the modification of a variable accessed via $q$ (e.g., $q.a = v$), where $p$ either points to a global variable or is passed as a parameter to an API function. A variable is declared as shared if (1) it is propagated from a complete update sequence; and (2) gets assigned by a local variable or an expression. After a shared variable is discovered, we regard all procedures that use the variable as the test program. Furthermore, we mark the write statement (e.g., $q.a = v$) and every statement that reads the variable in the update sequence (e.g., any read to a variable among $p, p_1, p_2, \ldots, p_k, q$) to be the point at which a data race can occur. Regarding the graph format of test programs, we first represent each set with its interprocedural PDG. Then, we link all the interprocedural PDGs via a data dependence edge (from $p$ to $q$) through the node of the shared variable on each interprocedural PDG when the corresponding operations include at least one write. We left an example in the supplemental material (Section F) for reviewers’ perusal. For prediction interpreter, we introduce a simple change to our procedure for determining the feasibility of program paths that trigger data races. Because data races are triggered by concurrent sub-paths, both of which access the same shared variable. Thus, we solve the union of the path constraints collected from both sub-paths to ensure their concurrent feasibility.

O2 [Liu et al. 2021] is the state-of-the-art for static race detection, at the same time, it’s a commercial tool that is not publicly accessible. For this reason, we compare Infrared against RacerD [Blackshear et al. 2018], the then state-if-the-art static race detector. By default, RacerD does not perform a whole program analysis, therefore, it may miss code that Infrared covers. To ensure RacerD having at least the same coverage as Infrared, we manually annotate all classes (with @ThreadSafe annotations) in each project (included in JBench) that are covered by the test code Infrared identified with shared variables. We also include FICS as another baseline since FICS claims to be capable of detecting any type of bugs. FICS has the same configuration as the previous experiment which produces the best performance again on JBench. For Infrared, we reuse the model trained for Defects4J evaluation. Same as before, we set the cut-off value $N$ to be 1 and 5 for this experiment.

**Results.** As shown in Table 4, Infrared in top-1 and top-5 display almost a perfect precision, reporting very few false warnings. However, this is insignificant considering the number of data
Table 4. Comparing Infrared against RacerD and FICS on JBench. top-max sets the cut-off value \( N \) to be the total number of bugs in a project. + is the number of bugs a tool finds but not others. When counting + for RacerD and FICS, we consider Infrared in top-max which achieves the best performance. For clarity, we split the six projects in JBench into two groups. The first row shows the results of each tool on dbcps2, ftpserver and guava while the second row shows the results of each tool on the rest.

|       | dbcps2 | ftpserver | guava |
|-------|--------|-----------|-------|
|       | TP     | FP        | FN    | +     |
|       |        |           |       |       |
| RacerD| 11     | 96        | 17    | 4     |
| FICS  | 0      | 52        | 28    | 0     |
| Infrared | top-1 | 1        | 0     | 27    |
|        | top-5  | 4        | 1     | 24    |
|        | top-max| 9       | 19    | 19    |

|       | log4j | tomcat | zookeeper |
|-------|-------|--------|-----------|
|       | TP     | FP        | FN   | +     |
|       |        |           |       |       |
| RacerD| 16     | 102       | 120  | 0     |
| FICS  | 4      | 291       | 132  | 0     |
| Infrared | top-1 | 1       | 0     | 135   |
|        | top-5  | 5       | 0     | 131   |
|        | top-max| 44      | 92    | 92    |

race that each project has far exceeds 5. For a more fair evaluation, we set \( N \) to be the number of data races a project has in total, denoted by top-max in Table 4. Infrared again comes on top. Compared to RacerD, which is clearly the better baseline, Infrared on average has both significantly lower false positive rate (68\% vs. 89\%) and false negative rate (67\% vs. 78\%). Overall, we conclude that Infrared is capable of detecting data races from concurrent programs, the type of code that is vastly different from what Infrared has encountered during training.

4.5 A Comprehensive Investigation on the Results of Infrared

In this section, we perform an in-depth analysis of Infrared’s results on Defects4J and JBench. Our goal is to gain an insight on why Infrared is accurate in detecting bugs in some cases while lacks precision in others.

On the Strengths of Infrared. We study 53 bugs Infrared caught in top-5 prediction mode which is a good sample size for our analysis. To understand the reason why Infrared predicts them to be buggy, for each bug, we gather the reference program under the relational embedding approach.\(^2\) First, we confirm the inconsistency indeed exists for each bug and its reference, subsequently, we manually annotate the part of the program in the bug and reference program that constitutes the inconsistency. Because every inconsistency can be expressed with no greater than 6 statements in the bug or reference program, we retrieve 6 statements assigned the highest global attention weight (Equation 9) from both programs. After a thorough examination, we find that in 42 out of 53 bugs the annotated code are fully covered by the 6 highest attended statements in the bug and reference program, even for the few remaining bugs, the 6 highest attended statements miss at most 2 statements combined in the annotated code. This is a convincing piece of evidence that Infrared has learned the right features that capture the semantic inconsistency between buggy and reference programs. Another interesting finding is that 17 reference programs used in this

\(^2\)we explain what is a reference program for a given bug in the last paragraph of Section 4.3.
study exhibit program features (e.g., unknown APIs, new language features like lambda expression, reflections, etc.) that never appear in the training set of Infrared and rarely occur in other test programs collected in the same batch. Under this circumstance, Infrared could have been easily tricked to predict such reference programs as bugs, if it relied on syntactic variations. Instead, Infrared is not confused. Even without the knowledge of those rare features, Infrared can still determine whether or not they constitute a semantic inconsistency by analyzing how they are used in test programs. This findings testify the precision of Infrared in recognizing deep, semantic inconsistency beyond syntactic variations, which is the primary reason behind Infrared’s superior effectiveness to traditional static analysis tools like Eradicate, Infer or RacerD. The reason that Infrared significantly outperforms FICS is also straightforward: FICS adopts an unsupervised approach; without a training phase like Infrared’s FICS do not have a chance to learn to pinpoint the kind of semantic inconsistency that leads to bugs. As a result, it seems to be easily confused by syntactic variations: flagging anything it sees as an inconsistency.

**On the Limitations of Infrared.** We also attempt to understand why Infrared fails to find bugs. We find that the main source of Infrared’s inaccuracy stems from its handling of exceedingly complicated programs. For example, Infrared is shown to be ineffective against quite a few data races triggered by the interleaving of many threads, each of which yields a lengthy call chain spanning across dozens of procedures. The cause of the phenomenon is the well-known deficiency of graph neural networks in processing large, complex graphs. Despite for different reasons, this issue is in fact common to all tools used in our evaluation including both state-of-the-art static analysis tools and learning-based bug detectors. Nevertheless, it points to a worthwhile direction for future work on top of MBD’s contribution in this paper. Another factor that contribute to Infrared’s imprecision is that bugs may not always be inconsistent with correct programs. For example, several bugs in array index out-of-bound involve specialized, heavy arithmetic operations for computing the bound of an array. Although the computations turn out to be erroneous, they are not necessarily inconsistent with any correct program. Thus, Infrared did not flag them as buggy. This may be a limitation for Infrared, however, considering bugs not showing inconsistency with correct programs are minorities, Infrared’s approach — meta bug detection — is still valid.

### 4.6 On the Necessity of Key Components of Infrared

To confirm the necessity of each key component of Infrared, we perform an ablation study to reveal their contributions. We compare Infrared’s performance on both Defects4J and JBench before and after the removal of a component. We use true positives as the only metric because false positives/negatives convey the same information given Infrared generates the same number of reports before and after the ablation. For convenience, we sum the number of true positives Infrared finds in Defects4J and JBench. Since the prediction mode is not a factor for this study, we present our results only in top-5. First, we drop the global attention mechanism from the GNN, and re-run experiments in the same procedure as before. As depicted in the second row of Table 5, Infrared catches 9 fewer bugs than before, a significant downgrade from its original configuration. This indicates the idea that the global graph attention realizes — focusing on nodes that express the essential characteristics of graphs — improves GNN’s precision in meta bug detection.

| Configuration                               | # of Bugs Found |
|---------------------------------------------|-----------------|
| original                                    | 53              |
| w/o global graph attention                   | 44              |
| w/o relational embedding                     | 46              |

---

3 for each bug, we collect approximately 20 test programs on average. 17 reference programs refer to those that contain features that rarely occur in the other 19 test programs collected in the same batch.
Next, we forgo the relational embedding approach. As a result, Infrared misses 7 bugs. In fact, only adding back the attention mechanism in the relational embedding (i.e., without read operations) still has Infrared catch 4 fewer bugs, not to mention, the complete relational embedding approach embeds every buggy program further away from the average program than both the partial (i.e., without read operations in relational embedding) and the plain embedding approach (i.e., without relational embedding at all). This strongly suggests that relational embedding helps to improve the precision of the learning approach, furthermore, the read operations succeeded in amplifying the effect of attention in this relational embedding approach. Overall, our ablation study shows the components are all necessary for Infrared to be effective.

5 RELATED WORK

We survey related work from anomaly detection, formal methods- and learning-based bug finders.

5.1 Anomaly Detection

Engler et al. [2001] proposed to analyze bugs as deviant behavior, a seminal work in anomaly detection. DIDUCE [Hangal and Lam 2002], also based on anomaly detection, monitors the violation of invariants during the execution of programs. Bixie [McCarthy et al. 2015; Schäf et al. 2013] regards a code fragment to be buggy if it is not part of any normal terminating executions. These works are by nature rule-based and focused on a few specific types of programming errors. Some bug detectors take a more general approach to identifying coding inconsistencies. APisan [Yun et al. 2016] infers correct API usages in source code through symbolic execution and semantic cross-checking. Similar to AntMiner [Liang et al. 2016] detects API usage inconsistencies via programming rules minded from the program dependence graph. Li et al. [2021] also detect API Misuses base on active learning. The work that is the closest to ours is FICS [Ahmadi et al. 2021], which is also a bug-generic detector. However, FICS is built upon an unsupervised approach without a training phase. As a result, it is shown to be significantly less precise than Infrared in our experiments.

5.2 Formal Methods-based Bug Finding

Model checking has been an important technique for static bug finding. SPIN [Holzmann 1997], Java PathFinder [Visser et al. 2000], and CBMC [Clarke et al. 2004, 2003] are notable examples. SLAM [Ball and Rajamani 2002] and BLAST [Henzinger et al. 2002] improve the scalability of the above tools with the idea of abstraction refinement. Data-flow analysis has also been applied extensively in this field (e.g., IFDS [Reps et al. 1995], Saturn [Xie and Aiken 2005], and CALYSTO [Babic and Hu 2008]). But these tools often suffer from scalability issues. Sparse value-flow analysis [Cherem et al. 2007; Livshits and Lam 2003; Shi et al. 2018; Sui and Xue 2016] mitigates this problem by tracking the flow of values sparsely through def-use chains or static single assignment form. Infer[Berdine et al. 2006; Calcagno et al. 2015] finds bugs by automatically inferring separation logic assertions over statements. Compared to the static analysis tools, Infrared benefits from a simpler design and higher utility, specifically, it can detect the types of bugs that are unobserved during training.

5.3 Learning-based Bug Detection

Allamanis et al. [2018] develop a new program graph by incorporating the data flow and type hierarchies information into ASTs for predicting the name of a variable and detecting the misuse of variable. Vasic et al. [2019] present multi-headed pointer networks for detecting the same variable misuse bug. Hellendoorn et al. [2020] further improve the joint model [Vasic et al. 2019] by combining sequence model (e.g., RNN, transformer) with a structure model like GNN. DeepBugs [Pradel and Sen 2018] presents a learning approach to name-based bug detection. Wang et al. [2020]
develop an interval-based graph abstraction method to improve the scalability of GNNs. MBD is fundamentally different from all the works above which are dedicated to learning bug patterns.

6 CONCLUSION
In this paper, we propose meta bug detection, a fundamentally different concept from the methodology of existing learning-based bug detection. Our insight is to learn how bugs are inconsistent with correct programs independently of the type of bugs. Built on top of this insight, MBD (1) requires substantially less training data than the existing bug detectors, (2) is capable of predicting the type of bugs that are totally unobserved during training, and (3) can explain its prediction without any external interpretability methods. We realize MBD into a tool, Infrared, and extensively evaluated it. On well-established datasets like Defects4J and JBench, results show Infrared is effective, catching a large number of bugs, in particular it significantly outperforms noteworthy static analysis tools like Facebook Infer and RacerD, and learning-based anomaly or bug detectors like FICS and GINN.
REFERENCES

2022. Eradicate. https://fbinfer.com/docs/checker-eradicate/. Accessed: 2022-07-07.
Mansour Ahmadi, Reza Mirzazadeh farkhani, Ryan Williams, and Long Lu. 2021. Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code. In 30th USENIX Security Symposium (USENIX Security 21). USENIX Association, 2025–2040. https://www.usenix.org/conference/usenixsecurity21/presentation/ahmadi
Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In International Conference on Learning Representations.
Domagoj Babic and Alan J. Hu. 2008. Calysto: Scalable and Precise Extended Static Checking. In Proceedings of the 30th International Conference on Software Engineering (Leipzig, Germany) (ICSE ’08). Association for Computing Machinery, New York, NY, USA, 211–220.
Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In International Conference on Learning Representations.
Thomas Ball and Srimat K. Rajamani. 2002. The SLAM Project: Debugging System Software via Static Analysis. SIGPLAN Not. 37, 1 (Jan. 2002), 1–3. https://doi.org/10.1145/565816.503274
Josh Berdine, Cristiano Calcagno, and Peter W. O’Hearn. 2006. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In Proceedings of the 4th International Conference on Formal Methods for Components and Objects (FMCO’05).
Sam Blackshear, Nikos Gorgiannos, Peter W. O’Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. Proc. ACM Program. Lang. 2, OOPSLA, Article 144 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276514
Martin Bravenboer and Yannis Smaragdakis. 2009. Exception Analysis and Points-to Analysis: Better Together. In Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (Chicago, IL, USA) (ISSTA ’09). Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1572272.1572274
Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O’Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In NASA Formal Methods, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi (Eds.).
Cristiano Calcagno, Dino Distefano, Peter W. O’Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. J. ACM 58, 6, Article 26 (Dec. 2011), 66 pages. https://doi.org/10.1145/2049697.2049700
Cătălina Cangea, Petar Velicković, Nikola Jovanović, Thomas Kipf, and Pietro Liò. 2018. Towards sparse hierarchical graph classifiers. arXiv preprint arXiv:1811.01287 (2018).
Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. IEEE Transactions on Software Engineering (2019).
Sigmund Cherev, Lonnie Princehouse, and Radu Rugina. 2007. Practical Memory Leak Detection Using Guarded Value-Flow Analysis. SIGPLAN Not. 42, 6 (June 2007), 480–491.
Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. 2004. Predicate Abstraction of ANSI-C Programs Using SAT. Form. Methods Syst. Des. 25, 2–3 (Sept. 2004), 105–127. https://doi.org/10.1023/B:FORM.0000040025.89719.f3
Edmund Clarke, Daniel Kroening, and Karen Yorav. 2003. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. In Proceedings of the 40th Annual Design Automation Conference (Anaheim, CA, USA) (DAC ’03), Association for Computing Machinery, New York, NY, USA, 368–371. https://doi.org/10.1145/775832.775928
Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 2000. Model Checking. MIT Press, Cambridge, MA, USA.
Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (Los Angeles, California) (POPL ’77). Association for Computing Machinery, New York, NY, USA, 238–252.
Matthew Gharrity Daniel Donenfeld. 2019. JLang, An LLVM backend for the Polyglot compiler. https://polyglot-compiler.github.io/JLang/
Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In International Conference on Learning Representations.
Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. SIGOPS Oper. Syst. Rev. 35, 5 (Oct. 2001), 57–72. https://doi.org/10.1145/502059.502041
Rudolf Ferenc, Peter Gyimesi, Gábor Gyimesi, Zoltán Tóth, and Tibor Gyimóthy. 2020. An automatically created novel bug dataset and its validation in bug prediction. Journal of Systems and Software 169 (2020), 110691.
Rudolf Ferenc, Zoltán Tóth, Gergely Ladányi, István Siket, and Tibor Gyimóthy. 2018. A public unified bug dataset for Java. In Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering. 12–21.
Hongyang Gao and Shuiwang Ji. 2021. Graph U-Nets. IEEE Transactions on Pattern Analysis and Machine Intelligence (2021), 1–1. https://doi.org/10.1109/TPAMI.2021.3081010
Proc. ACM Program. Lang., Vol. 1, No. CONF, Article 1. Publication date: January 2018.
Jian Gao, Xin Yang, Yu Jiang, Han Liu, Weiliang Ying, and Xian Zhang. 2018. Jbench: A Dataset of Data Races for Concurrency Testing. In Proceedings of the 15th International Conference on Mining Software Repositories (Gothenburg, Sweden) (MSR ’18). Association for Computing Machinery, New York, NY, USA, 6–9. https://doi.org/10.1145/3196398.3196451

Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In Proceedings of the 34th International Conference on Machine Learning - Volume 70 (Sydney, NSW, Australia) (ICML ’17). JMLR.org, 1263–1272.

M. Gori, G. Monfardini, and F. Scarselli. 2005. A new model for learning in graph domains. In IEEE International Joint Conference on Neural Networks, Vol. 2. 729–734 vol. 2. https://doi.org/10.1109/IJCNN.2005.1559542

Sudheendra Hangal and Monica S. Lam. 2002. Tracking down Software Bugs Using Automatic Anomaly Detection. In Proceedings of the 24th International Conference on Software Engineering (Orlando, Florida) (ICSE ’02). Association for Computing Machinery, New York, NY, USA, 291–301. https://doi.org/10.1145/581339.581377

Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global Relational Models of Source Code. In International Conference on Learning Representations. https://openreview.net/forum?id=B1hnRNIwtr

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. 2002. Lazy Abstraction. SIGPLAN Not. 37, 1 (Jan. 2002), 58–70. https://doi.org/10.1145/565816.503279

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. Neural Computation 9, 8 (11 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Gerard J. Holzmann. 1997. The Model Checker SPIN. IEEE Trans. Softw. Eng. 23, 5 (May 1997), 279–295.

S. Horwitz, T. Reps, and D. Binkley. 1988. Interprocedural Slicing Using Dependence Graphs. In Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (Atlanta, Georgia, USA) (PLDI ’88). Association for Computing Machinery, New York, NY, USA, 35–46. https://doi.org/10.1145/53990.53994

Jeff Huang, Patrick O’Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. Association for Computing Machinery, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

Lingxiao Jiang, Ghassan Misserghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In 29th International Conference on Software Engineering (ICSE’07). IEEE, 96–105.

D’Angelo John Philip and Douglas Brent West. 1997. Mathematical thinking: problem-solving and proofs. Prentice-Hall.

Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. 2009. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In Computer Aided Verification, Ahmed Bouajjani and Oded Maler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 675–681.

René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (San Jose, CA, USA) (ISSTA 2014). Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. 2007. Fast and accurate static data-race detection for concurrent programs. In International Conference on Computer Aided Verification. Springer, 226–239.

James C. King. 1975. A New Approach to Program Testing. SIGPLAN Not. 10, 6 (April 1975), 228–233. https://doi.org/10.1145/390016.808444

Junhyun Lee, Inyeeop Lee, and Jaewoo Kang. 2019. Self-attention graph pooling. In International Conference on Machine Learning. PMLR, 3734–3743.

Yuja Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2016. Gated graph sequence neural networks. In International Conference on Learning Representations.

Ziyang Li, Aravind Machiry, Binghong Chen, Mayur Naik, Ke Wang, and Le Song. 2021. ARBITRAR: User-Guided API Misuse Detection. In 2021 IEEE Symposium on Security and Privacy (SP). 1400–1415. https://doi.org/10.1109/SP40001.2021.00090

Bin Liang, Pan Bian, Yan Zhang, Wenchang Shi, Wei You, and Yan Cai. 2016. AntMiner: Mining More Bugs by Reducing Noise Interference. In 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). 333–344. https://doi.org/10.1109/ICSE.2016.7466645

Bozhen Liu, Peiming Liu, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang. 2021. When Threads Meet Events: Efficient and Precise Static Race Detection with Origins (PLDI 2021). Association for Computing Machinery, New York, NY, USA, 725–739. https://doi.org/10.1145/3453483.3454073

V. Benjamin Livshits and Monica S. Lam. 2003. Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs. SIGSOFT Softw. Eng. Notes 28, 5 (Sept. 2003), 317–326.

LLVM. 2021. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing.
Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER ’19). https://arxiv.org/abs/1901.06024

Wes Masri. 2015. Chapter Three - Automated Fault Localization: Advances and Challenges. Advances in Computers, Vol. 99. Elsevier, 103–156. https://doi.org/10.1016/B978-0-12-382007-9.00003-4

Tim McCarthy, Philipp Rümmer, and Martin Schäf. 2015. Bixie: Finding and understanding inconsistent code. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. IEEE, 645–648.

Martin Monperrus, Marcel Bruch, and Mira Mezini. 2010. Detecting Missing Method Calls in Object-Oriented Software. In Proceedings of the 24th European Conference on Object-Oriented Programming (Maribor, Slovenia) (ECOOP’10). Springer-Verlag, Berlin, Heidelberg, 2–25.

Robert O’Callahan and Jong-Deok Choi. 2003. Hybrid Dynamic Data Race Detection. SIGPLAN Not. 38, 10 (June 2003), 167–178. https://doi.org/10.1145/966049.781528

Renata Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Nogueira, and Lionel Seunier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. Software Practice and Experience 46 (2015), 1155–1179. https://doi.org/10.1002/spe.2346

Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. Proceedings of the ACM on Programming Languages 2, OOPSLA (2018), 1–25.

David A. Ramos and Dawson Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In Proceedings of the 24th USENIX Conference on Security Symposium (Washington, D.C.) (SEC’15).USENIX Association, USA, 49–64.

Thomas Reps, Susan Horwitz, and Moody Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 49–61.

Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. 2018. BugS. jar: a large-scale, diverse dataset of real-world java bugs. In Proceedings of the 15th international conference on mining software repositories. 10–13.

Martin Schäf, Daniel Schwartz-Narbonne, and Thomas Wies. 2013. Explaining Inconsistent Code. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (Saint Petersburg, Russia) (ESEC/FSE 2013). Association for Computing Machinery, New York, NY, USA, 521–531. https://doi.org/10.1145/2491411.2491448

David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019a. BugSwarm: mining and continuously growing a dataset of reproducible failures and fixes. In ICSE. IEEE / ACM, 339–349.

David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019b. BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes. In Proceedings of the 41st International Conference on Software Engineering (Montreal, Quebec, Canada) (ICSE ’19). IEEE Press, 339–349. https://doi.org/10.1109/ICSE.2019.00048

David A. Tomassi and Cindy Rubio-González. 2021. On the Real-World Effectiveness of Static Bug Detectors at Finding Null Pointer Exceptions. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). 292–303. https://doi.org/10.1109/ASE51524.2021.9678535

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh singh. 2019. Neural Program Repair by Jointly Learning to Localize and Repair. In International Conference on Learning Representations. https://openreview.net/forum?id=ByloJ20qtm

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, I. ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In Advances in Neural Information Processing Systems. I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In International Conference on Learning Representations. https://openreview.net/forum?id=rJXMpikCZ
Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. 2016. Order Matters: Sequence to Sequence for Sets. In *International Conference on Learning Representations*.

Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. 2000. Model Checking Programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE ’00)*. IEEE Computer Society, USA, 3.

Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning Semantic Program Embeddings with Graph Interval Neural Network. *Proc. ACM Program. Lang.*, 4, OOPSLA, Article 137 (Nov. 2020), 27 pages.

Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, California, USA) (*ICSE ’81*). IEEE Press, 439–449.

Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 2020. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1556–1560.

Yichen Xie and Alex Aiken. 2005. Scalable Error Detection Using Boolean Satisfiability. *SIGPLAN Not.*, 40, 1 (Jan. 2005), 351–363. https://doi.org/10.1145/1047659.1040334

Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APIsan: Sanitizing API Usages through Semantic Cross-Checking. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 363–378.

Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. 2021. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Transactions on Software Engineering* 47, 2 (2021), 332–347. https://doi.org/10.1109/TSE.2019.2892102
A TEST DATA COLLECTOR FOR COMMON TYPES OF BUGS

To collect test data for array index out-of-bound, we first find out the array operations $O$ including array read or write at a program point $l$. For each $o \in O$, we then use the array operation of array $arr$ and index $idx$ at $l$ as the criterion to compute a program slice which include the entire data dependence of $arr$ and $idx$. Finally, we regard all such slices to be the test code for array index out-of-bound.

For number format exception, we first find out the operations $O$ that parse strings $s$ to numbers. We then use each $o \in O$ as the criterion to compute a program slice which include the entire data dependence of $s$. Finally, we regard all such slices to be the test code for number format exception.
**B AN EXAMPLE FOR MBD’S CORRECTNESS DEFINITION**

```java
void foo (int x, int y) {
    x = 5;
    y = 6;
    log("value of x,y:", x, y);
    y = Math.abs(x) /* y is assign the absolute value of x */
    if (y >= 0)
    {
        str = null;
        str.length();
    }
}
```

Fig. 8. An example for Definition 3.3.

For the example in Figure 8, $T$ is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$, and $T'$ is $1 \rightarrow 4 \rightarrow 5 \rightarrow 6$, however, the correct trace $\bar{T}$ should be $4 \rightarrow 5 \rightarrow 6$ because the value of $y$ will always be non-negative. In fact, computing the minimal unsatisfiable cores of the symbolic trace will be useful in finding $\bar{T}$, specifically, $y = Math.abs(x) \land y > 0 \land str = null \land assert(str != null)^4$ is the minimal unsatisfiable cores of the entire trace. But it is out of the scope of our paper which focus on meta bug detection instead of error trace explanation.

**B.1 Definition of Subsequence**

A subsequence of $<a>$ is a sequence $<b>$ defined by $b_k = a_{n_k}$, where $n_1 < n_2 < ...$ is an increasing sequence of indices [John Philip and West 1997]. For example, if $a_n = 2n - 1$ and $n_k = k^2$, then $b_k = 2k^2 - 1$ [John Philip and West 1997].

| $n$ | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |
|-----|----|----|----|----|----|----|----|----|----|
| $a_n$ | 1  | 3  | 5  | 7  | 9  | 11 | 13 | 15 | 17 |
| $k$ | 1  | 2  |    |    |    |    |    |    |    |
| $b_k$ | 1  | 7  |    |    |    |    |    |    |    |

\[we represent the semantic constraint expressed by \texttt{str.length() with assert(str != null)}.\]

Proc. ACM Program. Lang., Vol. 1, No. CONF, Article 1. Publication date: January 2018.
C  AN INTERPROCEDURAL PROGRAM DEPENDENCE GRAPH

We show the entire interprocedural program dependence graph for the program in Figure 5 of the main paper.

Fig. 9. The entire interprocedural program dependence graph for the program in Figure 5 of the main paper. We omit AST nodes and edges, as well as execution order edges for brevity.
### D  DEFECTS4J BUGS DETECTED BY DIFFERENT TOOLS

Table 6. Null pointer dereference in Defects4J detected by Infer, Eradicate, FICS and Infrared.

| Proj.             | Infer | Eradicate | FICS | Infrared (Top-1) | Infrared (Top-5) |
|-------------------|-------|-----------|------|------------------|------------------|
| Chart-2           |       |           |      |                  |                  |
| Chart-4           |       |           |      |                  |                  |
| Chart-14          | ✔     |           |      |                  |                  |
| Chart-16          | ✔     |           |      |                  |                  |
| Cli-5             | ✔     | ✔         |      |                  |                  |
| Cli-30            |       |           |      |                  |                  |
| Codec-5           |       |           |      |                  |                  |
| Codec-13          | ✔     |           |      |                  |                  |
| Codec-17          | ✔     | ✔         | ✔    | ✔                |                  |
| Collections-25    | ✔     |           |      |                  |                  |
| Collections-28    |       |           |      |                  |                  |
| Csv-4             |       |           | ✔    |                  |                  |
| Csv-9             | ✔     |           |      |                  |                  |
| Csv-11            |       |           | ✔    | ✔                |                  |
| JacksonCore-8     | ✔     | ✔         | ✔    |                  |                  |
| JacksonDatabind-3 |       |           |      |                  |                  |
| JacksonDatabind-13|       |           |      |                  |                  |
| JacksonDatabind-36|       |           |      |                  |                  |
| JacksonDatabind-80| ✔     |           |      |                  |                  |
| JacksonDatabind-93|       |           |      |                  |                  |
| JacksonDatabind-95| ✔     |           |      |                  |                  |
| JacksonDatabind-107| ✔  ✔  ✔ | ✔         | ✔    |                  |                  |
| Jsoup-8           | ✔     | ✔         |      |                  |                  |
| Jsoup-22          |       |           |      |                  |                  |
| Jsoup-26          |       |           |      |                  |                  |
| Jsoup-66          |       |           |      |                  |                  |
| Jsoup-89          | ✔     |           |      |                  |                  |
| Lang-20           |       |           |      |                  |                  |
| Lang-33           | ✔     | ✔         | ✔    | ✔                |                  |
| Lang-39           | ✔     |           |      |                  |                  |
| Lang-47           |       |           |      |                  |                  |
| Lang-57           |       |           |      |                  |                  |
| Math-4            | ✔     | ✔         |      |                  |                  |
| Math-70           |       |           |      |                  |                  |
| Math-79           |       |           |      | ✔                |                  |
| Mockito-18         |       |           |      |                  |                  |
| Mockito-35         |       |           | ✔    | ✔                |                  |
| Mockito-36         |       |           |      |                  |                  |
| Mockito-38         | ✔     |           |      |                  |                  |

✓ indicates the bug was correctly detected by the tool.
Table 7. Array index out-of-bound in Defects4J detected by FICS and Infrared.

| Proj.          | FICS | Infrared (Top-1) | Infrared (Top-5) |
|---------------|------|-----------------|-----------------|
| Chart-5       |      | ✔               |                 |
| Chart-18      | ✔    | ✔               |                 |
| Chart-22      |      |                 |                 |
| Cli-32        | ✔    |                 |                 |
| Codec-8       |      |                 |                 |
| Codec-18      | ✔    | ✔               |                 |
| Compress-21   | ✔    |                 |                 |
| Compress-37   |      |                 |                 |
| JacksonCore-10| ✔    |                 |                 |
| JacksonCore-11| ✔    |                 |                 |
| JacksonCore-19| ✔    |                 |                 |
| JacksonCore-25|      |                 |                 |
| Jsoup-5       |      |                 |                 |
| Jsoup-34      |      |                 |                 |
| Jsoup-72      |      |                 |                 |
| Jsoup-80      |      |                 |                 |
| Jsoup-86      |      |                 |                 |
| Jsoup-90      |      |                 |                 |
| Lang-6        |      | ✔               |                 |
| Lang-12       | ✔    |                 |                 |
| Lang-19       | ✔    | ✔               |                 |
| Lang-27       |      |                 |                 |
| Lang-44       | ✔    |                 |                 |
| Lang-45       |      |                 |                 |
| Lang-51       |      |                 |                 |
| Lang-59       | ✔    | ✔               |                 |
| Lang-61       |      |                 |                 |
| Math-3        |      |                 |                 |
| Math-81       | ✔    |                 |                 |
| Math-98       |      |                 |                 |
| Math-100      | ✔    | ✔               |                 |
| Math-101      |      |                 |                 |
| Mockito-3     |      |                 |                 |
| Mockito-34    |      |                 | ✔               |

✔ indicates the bug was correctly detected by the tool.
Table 8. Number format exception in Defects4J detected by FICS and Infrared.

| Proj.          | FICS | Infrared (Top-1) | Infrared (Top-5) |
|---------------|------|------------------|------------------|
| Compress-32   | ✔    |                  | ✔                |
| JacksonCore-5 | ✔ ✔ ✔|                  | ✔                |
| Lang-1        |      |                  |                 |
| Lang-16       |      |                  | ✔                |
| Lang-36       | ✔ ✔  |                  | ✔                |
| Lang-58       |      |                  |                 |

✔ indicates the bug was correctly detected by the tool.
E COMPARING ERADICATE AND INFER’S RESULTS IN OUR EVALUATION AGAINST?

We describe how we confirm the results of baseline methods (e.g., Eradicate and Infer) match those of Tomassi and Rubio-González [2021]. After a thorough review of their detailed experiment results\(^5\), we summarize our findings below. First, for each bug in their dataset, they run Eradicate and Infer twice, one on the buggy version and the other on the fixed version\(^6\), and they count the same warning twice when reporting the total number of alarms. Third, bugs are always analyzed separately in Tomassi and Rubio-González [2021] even if they are from the same project. As a result, a warning occurs as many times as the project is analyzed. Since Tomassi and Rubio-González [2021] did not remove duplicate alarms, the same report produced by Eradicate and Infer can be countered multiple times. Finally, Tomassi and Rubio-González [2021] also count the warnings Eradicate and Infer report on code written for testing purposes. As we explained in the main paper, we do not consider bugs in test code for our evaluation, in the same way, we also ignore all of Eradicate and Infer’s reports on test code. Taking into account all factors above, we confirm their results indeed match ours.

\(^5\)released at https://github.com/ucd-plse/Static-Bug-Detectors-ASE-Artifact/tree/main/data

\(^6\)such that the presence/absence of a bug report on the buggy/fixed version of the project points to the bug.
A DATA RACE EXAMPLE

```java
public class Main {
    private ChildThread childThread;

    public void execute() {
        childThread = new ChildThread(this);
        childThread.start();
        ...
        synchronized (this) {
            if (childThread != null) {
                childThread.interrupt();
            }
        }
    }

    public class ChildThread extends Thread {
        private Main main;

        public ChildThread(Main main) {
            this.main = main;
        }

        public void run() {
            ...
            main.childThread = null;
        }
    }
}
```

Fig. 10. A program with data race.

![Diagram showing the program's PDG with control and data dependence edges.]

Fig. 11. The graph representation of the program in Figure 10. We add two data dependence edges between the PDG of Main::execute and ChildThread::run to denote uses of the shared variable.

We use the example from O'Callahan and Choi [2003] to illustrate our approach. In Figure 10, the main thread starts a new thread called ChildThread (Line 4) and then tries to terminate it (Line 9). Because the only lock at line 7 cannot prevent the simultaneous access to a shared variable childThread. In particular, when main.childThread = null (Line 22) is executed between if(childThread != null) (Line 8) and childThread.interrupt() (Line 9), a null pointer dereference will be thrown.

To analyze this program, we first found a shared variable this in method Main::execute according to (1) a complete update sequence: main = this; (the parameter passed to the constructor of ChildThread), and this.main = main; (inside of the constructor of ChildThread); and (2)
a subsequent modification of the field variable childThread accessed via this.main (inside of ChildThread::run). Then, we identify Main::execute and ChildThread::run as the methods that use this variable. Therefore, they become a test program. For the graph representation, we first construct the interprocedure PDG (each containing only one procedure in this specific case) for each method, and then we connect the use points of the shared variable in each interprocedure PDG as Figure 11 shows.

\[\text{The variable } \textbf{this} \text{ in } \text{Main::execute} \text{ and } \textbf{this.main} \text{ in } \text{ChildThread::run} \text{ can be regarded as } p \text{ and } q \text{ in the algorithm we explained in the main paper.}\]