Hunting for Truth: Analyzing Explanation Methods in Learning-based Vulnerability Discovery

Tom Ganz
SAP SE
Karlsruhe, Germany

Philipp Rall
Technische Universität Darmstadt
Darmstadt, Germany

Martin Härterich
SAP SE
Karlsruhe, Germany

Konrad Rieck
Technische Universität Berlin
Berlin, Germany

***Abstract*—Recent research has developed a series of methods for finding vulnerabilities in software using machine learning. While the proposed methods provide a remarkable performance in controlled experiments, their practical application is hampered by their black-box nature: A security practitioner cannot tell how these methods arrive at a decision and what code structures contribute to a reported security flaw. Explanation methods for machine learning may overcome this problem and guide the practitioner to relevant code. However, there exist a variety of competing explanation methods, each highlighting different code regions when given the same finding. So far, this inconsistency has made it impossible to select a suitable explanation method for practical use.**

In this paper, we address this problem and develop a method for analyzing and comparing explanations for learning-based vulnerability discovery. Given a predicted vulnerability, our approach uses directed fuzzing to create *local ground-truth* around code regions marked as relevant by an explanation method. This local ground-truth enables us to assess the veracity of the explanation. As a result, we can qualitatively compare different explanation methods and determine the most accurate one for a particular learning setup. In an empirical evaluation with different discovery and explanation methods, we demonstrate the utility of this approach and its capabilities in making learning-based vulnerability discovery more transparent.

1. Introduction

The automatic discovery of vulnerabilities in software is a long-standing challenge in security research. Several methods have been proposed for this task that combine static program analysis with machine learning techniques. Recent approaches build on deep neural networks that are trained on examples of vulnerable and non-vulnerable code and potentially identify security defects automatically. In controlled experiments, several of these learning-based methods reach a remarkable performance and outperform conventional techniques of static program analysis for vulnerability discovery [e.g., 13, 16, 33, 49, 56].

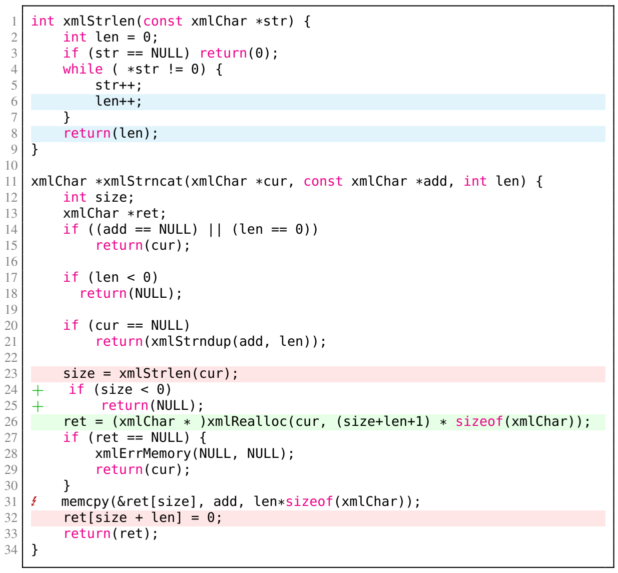
Methods for learning-based vulnerability discovery, however, suffer from a severe shortcoming: The employed deep neural networks are opaque to the practitioner. That is, it remains unclear how they arrive at a decision and which particular code structures are responsible for a predicted vulnerability. To make use of learning-based methods, a practitioner is forced to manually investigate each finding and validate its integrity, undermining the promise of automatic vulnerability discovery. Despite their excellent performance, learning-based methods are thus rarely employed in practice.

As a remedy, recent work has explored *explanation methods* for machine learning in vulnerability discovery [e.g., 21, 50, 58]. These methods enable to trace back the decisions of a neural network to particular code regions, thus creating the necessary context to assess a predicted vulnerability. However, there is no established standard for these explanations. A variety of competing concepts exists, each highlighting different code regions when given the same finding [21, 50]. As an example, Figure 1 shows three explanations for a security flaw identified by a deep neural network [13]. Each explanation marks different parts of the code, making it impossible to interpret the finding without further insights. This inconsistency poses a major hurdle in creating transparent and explainable methods for vulnerability discovery.

In this work, we address this problem and propose a method for analyzing and comparing explanation methods for learning-based vulnerability discovery. The core idea of our approach is to generate ground-truth around the code regions marked by an explanation method to determine their veracity. To this end, we guide a directed fuzzer toward their locations and inspect the relation between reported crashes and explanations. This strategy allows us to make a *qualitative* comparison of explanation methods and link marked code regions to actual vulnerabilities. Our method provides a novel view of explainable machine learning in security that addresses the lack of ground-truth in current frameworks for analyzing explanations.

We empirically evaluate our approach using different explanation methods suitable for vulnerability discovery. Our experiments show that commonly used intrinsic criteria, such as the descriptive accuracy of an explanation, do not adequately measure performance and lead to inconsistent results. In contrast, our approach allows for a reliable comparison, as it selectively constructs ground-truth on local code regions, defined as *local ground-truth* and thus evaluates the explanations against real vulnerabilities. Our analysis contradicts prior work on selecting explanation methods for vulnerability discovery[21, 50]: We find that graph-based explanation methods actually outperform other techniques when we base this comparison on local ground-truth rather than (arbitrary) intrinsic criteria.

Naturally, our method cannot uncover the ground-truth for any possible vulnerability, as it inherits the limitations of directed fuzzing. For example, explanations pointing to unreachable code cannot be analyzed and verified. Still,



```
int xmlStrlen(const xmlChar *str) {
    int len = 0;
    if (str == NULL) return(0);
    while ( *str != 0) {
        str++;
        len++;
    }
    return(len);
}

xmlChar *xmlStrncat(xmlChar *cur, const xmlChar *add, int len) {
    int size;
    xmlChar *ret;
    if ((add == NULL) || (len == 0))
        return(cur);

    if (len < 0)
      return(NULL);

    if (cur == NULL)
        return(xmlStrndup(add, len));

    size = xmlStrlen(cur);
+    if (size < 0)
+        return(NULL);
    ret = (xmlChar * )xmlRealloc(cur, (size+len+1) * sizeof(xmlChar));
    if (ret == NULL) {
        xmlErrMemory(NULL, NULL);
        return(cur);
    }
ϟ   memcpy(&ret[size], add, len*sizeof(xmlChar));
    ret[size + len] = 0;
    return(ret);
}
```

Figure 1: Vulnerability CVE-2016-1834 with highlighted explanations for ReVeal+GNNExplainer (green), ReVeal+Smoothgrad (blue), ReVeal+GradCam (red). ϟ and + denote the crash site and patch, respectively.

our method is the first approach to automatically assess the veracity of explanations and help practitioners select accurate explanation methods in practice.

The rest of this paper is organized as follows: In Section 2, we introduce learning-based vulnerability discovery and corresponding explanation methods. We then present our approach for validating explanations in Section 3 and evaluate its efficacy in Section 4. Limitations and related work follow in Sections 5 and 6, respectively, before we conclude in Section 7.

2. Vulnerability Discovery and Explanation

Let us first formalize the task of vulnerability discovery.

Definition 1. *A method for **static vulnerability discovery** is a decision function $f\colon x \mapsto P(vuln \mid x)$ that maps a piece of code x to its probability of being vulnerable.*

Several methods can be directly cast into this simple representation. For example, the classic tool Flawfinder[1] searches for known patterns of insecure code, including the usage of functions associated with buffer overflows (e.g., `strcpy`, `strcat`, `gets`), format string problems (e.g. `printf`, `snprintf`), and race conditions. Flawfinder takes the source code text representation, matches it against the above-mentioned function names and sorts them by risk which is a discrete approximation to $P(\text{vuln}|x)$. Other static code analysis tools, such as *Cppcheck*[2] or *SonarQube*[3], can be similarly described as a function f predicting vulnerabilities.

Learning-based methods for vulnerability discovery also fit into this generic representation. The methods build on a function $f = f_\theta$ (model) parameterized by weights θ that are obtained by training on a dataset of vulnerable and non-vulnerable code [22]. Compared to classic static analysis tools, learning-based approaches do not have a fixed rule set and thus can adapt to characteristics of different vulnerabilities in the training data. Conceptually, these learning-based approaches mainly differ in (a) the program representation used as input and (b) the learning model, that is, the way f depends on the weights θ.

1. https://dwheeler.com/flawfinder/
2. https://cppcheck.sourceforge.io/
3. https://www.sonarqube.org/features/multi-languages/cpp/

2.1. Program Representation

Learning algorithms typically require vector representations as input. Some methods for vulnerability discovery, therefore, apply techniques from natural language processing (NLP) to derive a suitable feature vector for a given source code. In this case, the statements in the code are regarded as *sentences* while keywords and literals form the *words*. Doing so yields a sequential data corpus that can be numerically encoded, for instance, by applying common word embeddings [33, 39, 42].

Source code can also be modeled as a directed graph $G = G(V, E)$ with vertices V, edges $E \subseteq V \times V$, and attributes from a suitable feature space, that are attached to nodes and edges [1, 7, 53]. We refer to the resulting program representations as *code graphs*. These graphs can capture syntactic and semantic relations between statements and expressions inside code. Popular graphs are abstract syntax trees (AST) and flow graphs encompassing data and control flow. Similarly, a structure called a *program dependence graph* (PDG) describes control and data dependencies in a joint form [19]. Based on these classic representations, combined graphs have been developed for vulnerability discovery, in particular, the *code property graph* (CPG) by Yamaguchi et al. that resembles a combination of the AST, CFG and PDG [53].

2.2. Learning Model

Several learning models have been considered for the discovery of vulnerabilities, ranging from simple ones to deep neural networks [32, 33, 38, 42]. In particular, graph neural networks (GNN) are a promising approach to process structured program representations. They are deep learning models that take advantage of the graph structure in the input and realize an embedding $i\colon G(V, E) \mapsto y \in \mathbb{R}^d$ that can be used for classification tasks [43]. The most popular GNN types belong to so-called message-passing networks (MPN) where the prediction function is computed by iteratively aggregating and updating information from neighboring nodes. Several message passing types exist that use different aggregation and update schemes [51].

Due to the rich semantics captured by code graphs, GNNs have been applied in a series of works for vulnerability discovery [16, 49]. The resulting approaches outperform the former introduced sequential models, like VulDeepecker [33] and Draper [42]. In this work, we thus focus on approaches using GNNs on code graphs. In particular, we consider the graph-based methods Devign [56], ReVeal [13] and the token-based methods VulDeeLocator [34] and LineVul [20] that are state-of-the-art in learning-based vulnerability discovery. Nonetheless, our approach for creating local ground-truth is applicable to all learning models that allow tracing back explanations to code regions.

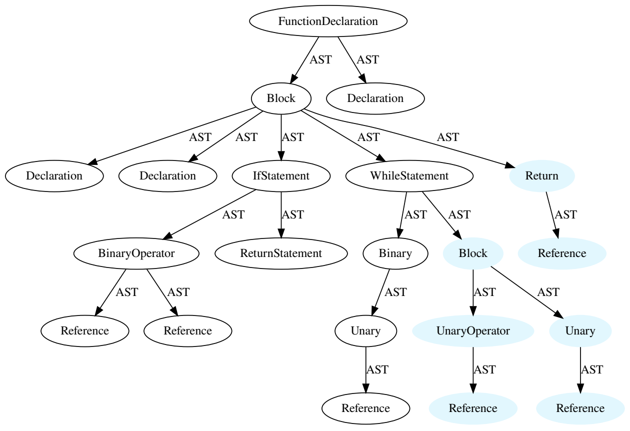


Figure 2: Explanation in the excerpt of a code graph. Relevant nodes are highlighted in blue.

2.3. Explainable Learning

While learning-based vulnerability discovery has made significant progress over the last years, a security practitioner faces the problem that their decisions must be verifiable. Learning-based approaches, however, only yield a binary decision as output for a given source code, which is hardly helpful for this task and requires a manual investigation. Hence, explainable learning has been studied as a remedy.

Given a vulnerability discovery method f we formalize explanation methods as producing heatmaps $\mathcal{M}$ from pairs of source code x and the predicted output $y = f(x)$. *Heatmaps* (or interchangeably *explanations*) attribute numerical relevance scores to locations in the code, i.e. to nodes and edges, if f is a GNN.

Definition 2. *An **explanation method** (EM) is a function $e_f\colon (x, f(x)) \mapsto \mathcal{M}$ where f is a vulnerability discovery method, x a piece of code, and $\mathcal{M}$ a heatmap defined over x.*

EMs are commonly classified into two categories: *black-box EMs* require no knowledge of the learning model, as for instance GNNExplainer, while *white-box EMs* have access to the weights of the learning model [50]. Furthermore, we discriminate *graph-specific EMs* that account for the topology of the provided graphs and *graph-agnostic EMs* that do not [21]. Throughout the paper, we apply different EMs to graph representations of code. We assume that a list of relevant lines of code can be extracted from the inferred heatmap $\mathcal{M}$ over the features. The precise process depends on the learning model that is used. For NLP-based approaches, as for instance VulDeeLocator and LineVul, the embedded code slices have to be converted back to their string representation, while for code graphs the respective code lines have to be attached to each node.

Explanation algorithms for GNNs attribute relevance to nodes, edges or subgraphs. We apply the heatmaps to the nodes as depicted in Figure 2. Here we see the `xmlStrlen` function from Example 1 with the corresponding graph representation. Data and control flow edges have been removed for visualization purposes. The relevant nodes are highlighted in accordance with the explanation method Smoothgrad on the ReVeal model.

2.4. Comparing Explanations

In view of the variety of available EMs, it becomes important to select an appropriate method for a given task, in our case vulnerability discovery. Unfortunately, ground-truth about relevant code regions is not available, and if there is, then only under laboratory conditions. To compare explanation methods in practice, we require metrics that capture the quality of the explanations delivered by an explanation method w.r.t. to a dataset and a model. Such a metric can be defined as a criterion function taking an EM as input and mapping it to a numerical score.

Definition 3. *A **criterion** is a function $c : e_f \to \mathbb{R}$ that measures the quality of e_f. An explanation method e_f outperforms $\hat{e}_f$ on a particular dataset D if $c(e_f|D) > c(\hat{e}_f|D)$.*

A frequently used criterion is the *descriptive accuracy* (DA) that measures the relative importance of samples comparing the prediction outcome of the model [9, 23, 35]. By removing the top features in $\mathcal{M}$ from x and re-evaluating $f(\hat{x})$ [50] we measure the relative drop in performance, for instance, the accuracy. We expect the model to arrive at a poorer decision without its relevant features. In this case, the vulnerability discovery model is not only used as the decision method but also as an *oracle* to assess the quality of an explanation.

Definition 4. *An **explanation oracle** is a function $o\colon \mathcal{M} \to [0, 1]$ which assesses the attributed relevance in a heatmap.*

Another popular criterion measures the *sparsity* of the explanation since we expect fewer relevant lines of code to be more human-interpretable [50]. The sparsity is calculated by simply counting the relevant code lines from $\mathcal{M}$. Furthermore, some works from the security domain also measure the *robustness* of explanations, giving intuitions about the influence of noise. Based on these intrinsic criteria, Warnecke et al. [50] and Ganz et al. [21] assess the *suitability* of explainable learning in security.

Definition 5. ***Suitability** is the property of an explanation that describes the potential interpretability in practical scenarios.*

We refer to criteria characterizing suitability as *intrinsic* criteria since they only draw conclusions between the learning model and the explanations—and not the task at hand. Consequently, intrinsic criteria do not compare EMs by their ability to explain decisions but rather by their *potential* to generate interpretable explanations. Since this is fundamentally different to *interpretability*, we introduce the term *suitability* for intrinsic comparisons.

For example, an explanation method might be suitable for vulnerability discovery, yet it may still be incorrect in the sense that the highlighted code is unrelated to the identified vulnerabilities. Validating the veracity of explanations is only possible with ground-truth, that is, *extrinsic* criteria that incorporate external knowledge about vulnerable code.

The ground-truth represents another oracle, which however is only available for trivial cases [5].

Definition 6. ***Veracity*** *is the property of an explanation that describes how the relevant lines of code of a model actually correspond to the examined task.*

When it comes to vulnerability detection, the veracity of an explanation is arguably more important than its suitability. The veracity is to an explanation what soundness is to a static analyzer. A static analyzer is considered sound if it claims that a property of a program is true, while this property is in fact true [30]. Similarly, we expect a code region highlighted by an explanation method to be linked to the underlying vulnerability.

Let us consider the example shown in Figure 1 which we use throughout the paper. It shows a vulnerability in Libxml2 (CVE-2016-1834). The code uses `xmlStrncat` to concatenate two strings together. In particular, it reallocates the first string to a larger contiguous memory area using the calculated lengths from `xmlStrlen`. If the length of the string is too large, the variable `len` overflows in line 6 and eventually results in a negative size used for reallocation. The memory block now becomes too small for `memcpy` in line 31 and yields a buffer overflow. The patch checks whether `size` is negative and is denoted by + in line 24 and 25. The crash-site is indicated by ↯ in line 31.

We highlight explanations from three EMs on this vulnerability to illustrate their inconsistency. GNNExplainer has the worst results according to criteria proposed by Ganz et al. [21], yet it comes close to the crash-site. GradCam highlights the size assignment in line 23, which is also a good indicator; however, line 32 is definitely a false positive. Smoothgrad highlights lines 6 and 8 equally. While line 6 may be relevant to the integer overflow, the other line is a false positive as well. The problem is that the EMs arrive at completely different explanations with varying suitability. Thus, we wonder how can they can be compared with respect to their *veracity*?

3. Methodology

We argue that the veracity of an explanation is key for practical vulnerability discovery. However, it requires ground-truth about the location of vulnerabilities in source code, which is tedious to obtain or not available at all. As a remedy, we propose to apply directed fuzzing as an incomplete but sound strategy for generating local ground-truth around a region highlighted by an explanation method. An overview of the resulting method is shown in Figure 3 and formalized in Algorithm 1. Technically, the method is composed of four components: a *learning model*, an *explanation* generated for a predicted vulnerability, a *directed fuzzer* for comparing the veracity and a *crash analysis* to provide fine-grained insights.

In the first step, the learning model receives a code sample x as input, runs an inference process and outputs the decision $f_\theta(x)$. If a white-box explanation is employed, the model's weights, gradients and log-probabilities, are additionally returned. Next, the explanation method under test generates a heatmap $\mathcal{M}$ given the results of the model for the input x. We interpret $\mathcal{M}$ as pointers to interesting code regions by assigning a relative score to each feature. A *directed fuzzer* [8, 14, 37] is then used as an

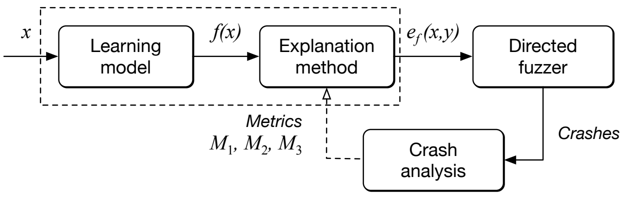


Figure 3: Overview of our approach for generating local ground-truth for explanation methods.

explanation oracle, which executes the given sample with inputs directed toward the highlighted code regions. As the last step, we employ a *crash analysis* by reproducing any discovered crash. This enables us to obtain a detailed view of the execution flow and, most importantly, the executed lines of code.

The directed fuzzing and the crash analysis are repeated for a fixed period, where the targets are processed according to their relevance. This loop ensures that we obtain comparative quantities for the top-k code lines highlighted by the explanation method. We proceed to describe these steps in more detail.

Algorithm 1: Method for generating ground-truth for explanation methods.

```
Input: Discovery method $f$, explanation method $e$, dataset of
       source code $\mathcal{X}$
1 repeat $n$ times
2     for $x \in \mathcal{X}$ do
3         $y \leftarrow f(x)$  // Inference on sample
4         $\mathcal{M} \leftarrow e_f(x, y)$  // Generation of explanation
5         Highlight code lines $L$ from $\mathcal{M}$
6         Run directed fuzzer towards $L$
7         Reproduce and analyze crashes
8 return $M_1, M_2, M_3$ (Extrinsic criteria)
```

Let us assume we use the highlighted line in Figure 1 from GNNExplainer to direct the fuzzer. As soon as the fuzzer generates a seed large enough for an integer overflow, the application crashes at line 31, since the reserved memory block for `memcpy` is too small. Line 32 is obviously not on the execution trace of the crash and the assignment in line 26 is closer to line 32 than lines 4 and 8. Hence, we argue that the fuzzer would reveal GNNExplainer as the best EM in this example.

3.1. Prediction and Explanation

Our approach is applicable to any combination of learning models and EMs that allow a mapping of the predictions back to locations in the source code. For example, it can be applied to all combinations of models and EMs considered in the studies by Ganz et al. [21], Zou et al. [58], and Warnecke et al. [50].

Some vulnerability discovery models come with their own integrated EM that is optimized jointly with the learning process, as for instance, VulDeeLocator [34] or LineVul [20]. These explanation can be similarly used with our method, as they are equivalent to an API wrapped around a learning model and EM as indicated by the dashed box in Figure 3. We refer to these methods as *model-integrated* EMs.

Furthermore, some EMs generate negative and positive scores, as they discriminate between the influences of the two prediction classes. To unify these methods with other explanations, we consider a code line that negatively contributes to a vulnerability as one that indicates it is not vulnerable. That is, we first apply the explanation method to a decision of the learning model and then normalize the returned relevance scores to the range $\mathcal{M} \in [0,1]^F$, where F is the number of code lines in a sample. After this normalization, we select the top k most relevant lines of code as indicated by the relevance values. The number k to be selected is subject to parametric choice but should be kept small and constant since more sampled lines lead to a more involved manual assessment.

3.2. Code Highlighting

If the considered learning model operates on a sequential representation of the source code, mapping the highlighted features to code regions is straightforward. For graph-based approaches, however, we need to design a way to extract the relevant code locations from the graph to be able to feed them as targets into a directed fuzzer.

To this end, we attach to each node its starting and end lines in the source code. Some of these nodes appear higher in the AST hierarchy, like `IfStatements`, whereas others denote the leaf nodes, such as literals [21]. The hierarchy defines the number of lines a node spans. Since the same code line can be marked as relevant by multiple nodes from different layers, we traverse the AST in a depth-first search and add relevance to each line as we walk from the root to the leaf nodes.

Ganz et al. observe that the relevance scores from higher nodes aggregate the score of their child nodes [21], causing higher nodes to be more relevant than lower ones. This phenomenon can be observed in Figure 2 and is likely due to the aggregation property from the message passing algorithm in GNNs. We solve this issue by weighting the relevance per node by the inverse of the number of lines a node spans, such that, for example, the EM attributes more relevance to an `IfStatement` than to its surrounding `FunctionDeclaration`.

3.3. Directed Fuzzing

We employ a directed grey-box fuzzer to retrieve a set of target locations and to aim at reaching these by seeking inputs minimizing the distance to the locations [8]. For instance, the directed fuzzer *AFLGo* calculates control-flow and call-graph distances prior to the fuzzing process to guide this search. Similarly, approaches to directed fuzzing, such as Hawkeye [14], can be applied in this step to explore the highlighted code regions and test for the presence of vulnerabilities.

To emphasize this, we revisit the definition of a security vulnerability.

Definition 7. *A security vulnerability is a software defect that enables an adversary to violate a security goal, such as the confidentiality, the integrity, or the availability, through a specifically crafted input.*

A fuzzer is a program analysis tool used to generate inputs and provoke program crashes. These crashes indicate software defects, and since they are triggered by manipulated inputs, they often represent security vulnerabilities in the sense of Definition 7. As we direct a fuzzer towards potential vulnerabilities in software, it is likely that a crash is associated with a vulnerability rather than a software defect. While this correlation could be coincidental, that is, an independent defect is close to a vulnerability, this situation should be rare given the high performance of current methods for vulnerability discovery. Moreover, even if only a defect is found, its proximity to a potential vulnerability makes it necessary to investigate it anyway, indicating a security relevance.

3.4. Crash Analysis

With the help of a directed fuzzer, we can thus explore a program and seek to hit code regions indicated by an explanation. To pinpoint the exact location of a program crash, we utilize a debugger. The crash is a *fact* and thus represents a form of ground-truth derived from a genuine incident during the program's execution. This incident is *local*, as it pertains to individual statements rather than the program as a whole. Consequently, we define *local ground-truth* in Definition 8.

Definition 8. *Local ground-truth refers to the precise location within a program where a crash has been reported. This location serves as a reference point for identifying and addressing related issues in the code.*

Technically, we employ a software debugger, such as GDB or LLDB, that enables us to execute the programs under test (PUT) with the crashing seeds and pause the execution to collect information about the crash-site and the explanation. In particular, we select the code lines highlighted by the EMs as *breakpoints* during the crash reproduction. When a relevant line is hit, the debugger halts the program and starts to calculate metrics that serve as *extrinsic criteria* in our approach. Using these criteria, we are able to draw conclusions about the relation between the crash and the highlighted code regions.

3.5. Extrinsic Criteria

We introduce three metrics that provide extrinsic criteria to compare and assess explanation methods. These criteria *complement* each other and yield a comprehensive view of the veracity of an explanation.

M_1: **Crashes per path over time.** As the first criterion, we identify the number of unique crashes and hangs that are reported during the fuzzing processes. This enables us to argue about which EM identified targets that lead to more paths resulting in crashes or hangs.

Fuzzers like AFL and AFLGo report *unique crashes* and *unique paths* by first counting the overall crashes and paths, and then rejecting those that reach the same branches. This process is called *de-duplication*. However, Klees et al. argue that de-duplication based on the executed edges leads to false positives [29]. Furthermore, it is insufficient to measure only the number of crashes. An explanation that randomly assigns relevance will result in the fuzzer having greater test coverage, leading to more crashes, only a few of which are actually related to the code defect. We counteract both issues by consolidating the *crashes-over-time* criterion with the found *paths-over-time*. Calculating the proportion of unique crashes per path, effectively gives us a single notion of how many crashes per path on average have been found. More paths will decrease the score while fewer paths increase it.

M_2: **Mean breakpoint hits.** As the second criterion, we consider the average number of breakpoint hits during the reproduction of crashes. If a target line triggers a breakpoint during the reproduction of a crash, that line can be considered to be associated with the vulnerability. The more breakpoints are hit during the reproduction of the crash, the more lines from the explanation are relevant. The explanation method can highlight sections of code near the actual vulnerability, but which may not be part of the execution trace. Such lines may still be relevant, since they may help a security practitioner to pin down the vulnerability. On the other hand, this criterion alone measures only whether a code line from an explanation is executed at least once.

M_3: **Mean crash distance.** To overcome the gap left by M_2, we also measure the average executed statements between the breakpoint hits and the crash-site. If the lines from an EM are closer to the crash-site, they should be more helpful for a security practitioner to identify and locate the cause of the crash and hence more relevant. Thus, the highlighted line does not have to lie exactly on the crash-site to be helpful. A target line from which it takes longer to reach the crash-site is accordingly less relevant since a security practitioner would have to afford more time to traverse the code and find the connection between the explanation and the actual cause.

```
1 #include <iostream>
2 using namespace std;
3 int array[3] = {1,2,3};
4 unsigned int index = userinput();
5 if (index < 3)
6     cout << array[index];
7 else
8     cout << array[index]; // crash
```

Figure 4: Example vulnerability for extrinsic criteria.

Interplay of the criteria. We provide a hypothetical example to illustrate their interplay in Figure 4 and to establish an intuition for the three extrinsic criteria with their edge-cases.

In this example, there are two possible branches, with one leading to a crash if the user input `index` is larger than 2. There are eight lines of code, hence each heatmap $\mathcal{M}$ is specified by an 8–tuple of numbers from the interval $[0, 1]$ and consequently there are 8! possible orderings by the relevance of the code lines. We want to investigate what would be sets $\mathcal{M}^+$ and $\mathcal{M}^-$ leading to the best and worst scores, respectively.

In a fuzzing experiment, we can assume that the fuzzer finds the crash quickest if line 7 or line 8 are highlighted as relevant, resulting in an M_1 score close to 1 for these lines. When the lines before 5 are highlighted, the benign and the vulnerable path should appear evenly often, resulting in an M_1 score of $1/2$. When only line 5 or 6 are selected, the result is an M_1 score of 0. Hence, we have $[0, 0, 0, 0, 0, 0, 1, 1] \subseteq \mathcal{M}^+$ as the optimal explanation.

M_2 measures the explained lines that have been executed prior to the crash. Line 1 through line 5 are always considered relevant by M_2, since they are linear (non-branching) starting from the program entry. In contrast, line 5 and line 6 will never be hit in a crash and can therefore only negatively impact the criterion. Line 7 and line 8 are executed within the crash reproduction. Hence, the worst score is achieved by labeling line 5 and 6, and the best score by labeling the complement: line 1 through 4 and line 7 and 8. This leads to $[1, 1, 1, 1, 0, 0, 1, 1] \subseteq \mathcal{M}^+$.

M_3 measures the distance between the explained lines to the crash-site. Hence, marking line 1 would result in the worst and line 8 in the best score. Clearly, the most relevant line is line 8 since it has a distance of 0 to the crash-site. For fairness, we can also take $k = 2$ lines prior to the crash-site as sufficiently close, therefore we have $[0, 0, 0, 1, 0, 0, 1, 1] \subseteq \mathcal{M}^+$ as the optimal explanation.

We conclude that an EM placing the most relevance on line 7 and 8 would yield the best-combined results, since there is the largest overlap of the different $\mathcal{M}^+$. While M_1 already captures this property, it depends on the number of successful fuzzing runs. When the number of repetitions is insufficient, the heatmap becomes ambiguous and thus also M_2 and M_3 are necessary to distill the local ground-truth around the crash-site.

Relationship to intrinsic criteria. To better understand the introduced extrinsic criteria as an incomplete yet sound replacement for the intrinsic criteria, let us discuss their commonalities and differences below.

The DA is the response of an intrinsic explanation oracle, namely f, while our extrinsic criterion M_1 evaluates the response of an extrinsic explanation oracle, namely a directed fuzzer, to measure the relevance of an explanation. Thus, in both cases, the heatmaps are interpreted as the localization of a weakness.

Let us consider Figure 1 as an example again: After removing line 6 and 8, ReVeal is unable to classify the code snippet as vulnerable. This corresponds to a beneficial DA, however, it wrongly suggests that Smoothgrad has advantageous detection capabilities. If we focus on GNNExplainer and remove line 26, ReVeal still classifies the snippet as vulnerable, leading to a disadvantageous DA. This effect is due to the fact that features in x have varying degrees of influence on the prediction output.

The intrinsic criterion *sparsity* counts the relevant lines [50]. A sparser $\mathcal{M}$ is considered more human interpretable, while M_2 and M_3 measure the distance of the executed relevant lines to a recorded crash. For sparsity, the goal is to minimize the highlighted code lines while for M_2 and M_3, as many relevant lines as possible should be executed close to the crash. Thus, both criteria provide intuitions about the conciseness of the EM. Consider again Example 1, GNNExplainer has the sparsest score but compared to LineVul is further away from the actual crash location. LineVul, however, has a worse sparsity score.

Some studies measure the *stability* [21] or *robustness* [50] as the resiliency of an EM to noise in the feature space. Ganz et al. measure the variance in the descriptive accuracy [21] to this end. One EM is more robust than another if the variance in descriptive accuracy is lower for the former than for the latter. Intuitively, it makes sense that the same procedure can be trivially applied to the extrinsic criteria as well.

Local ground-truth vs. vulnerabilities. After examining the behavior of the extrinsic criteria M_1, M_2, and M_3 and their relationship to intrinsic criteria, we can now consider the benefits of using extrinsic criteria.

A program crash is a clear indicator of a software defect and likely a vulnerability, as demonstrated by our running example (Figure 1). For instance, in line 31, we can observe that the program allocates insufficient memory, which directly leads to the crash. While highlighting line 31 can precisely pinpoint the vulnerability, other lines executed close to the crash might provide additional insights into the root cause of it. For instance, line 23 or 26 may indicate an integer overflow that contributed to the crash and could be helpful in fixing the vulnerability. On the other hand, lines such as 32 cannot be part of the crash execution trace, even if they are in close proximity to the crash. Consequently, GradCam's accuracy is lower when considering our extrinsic criteria.

The lines of code highlighted by an EM indicate the most relevant features for predicting a security vulnerability. However, EMs may identify code locations with artifacts resulting from overfitting, such as noise or outliers [50]. Conversely, a crash triggered by a fuzzer indicates the direct consequence of a security vulnerability [37], and the crash-site pinpoints the exact location of the problem. Therefore, a security analyst needs to reason about the vulnerability using the crash-site and the associated execution trace. We expect a veracious EM to highlight regions related to the crash-site and execution trace.

Our evaluation criteria assess the accuracy and proximity of the highlighted code locations to the local ground-truth, which is based on the actual incident where the program behavior diverged from its intended functionality, potentially exposing the system to security risks. This is different from intrinsic criteria that rely solely on the discovery model's feedback or properties directly derived from $\mathcal{M}$, which may be unreliable.

4. Evaluation

We proceed to experimentally demonstrate the applicability of our approach and evaluate the veracity of different explanation methods. In the course of this, we provide answers to the following research questions:

RQ1 Can we establish ground-truth around vulnerabilities predicted by learning models?

RQ2 Which explanation method provides the best explanations according to extrinsic criteria?

RQ3 How do extrinsic and intrinsic criteria differ when comparing explanation methods?

RQ4 How do rule-based auditing tools perform against explanation methods?

These questions naturally arise during software auditing with learning-based methods for vulnerability discovery and thus reflect typical decisions that must be made by security practitioners.

4.1. Experimental Setup

Before addressing these questions, we first introduce our experimental setup and the different methods for learning-based vulnerability discovery and generating explanations for their predictions.

Vulnerability discovery methods. For identifying security flaws, we employ the methods Devign [56] and ReVeal [13], which are state-of-the-art in learning-based vulnerability discovery.

1) Devign: This method uses code property graphs as a basis for detecting vulnerabilities. The graphs are extended with edges connecting leaf nodes with succeeding statements. These edges represent the natural order of the statements. The employed learning model is a gated graph neural network (GGNN) with six-time steps [56].

2) ReVeal: This discovery method takes an unmodified code property graph as input. The learning model consists of an eight-time-step GGNN followed by a sum aggregation and a fully connected network as the prediction head. The training involves a triplet loss that includes binary cross-entropy, L2 regularization, and a projection loss that minimizes the distances between similar classes and maximizes the difference between different classes.

Choosing two different models enables us to identify effects that are specific to the model. Both models are trained on 70% of the dataset while the remaining 30% are used for testing. Devign achieves $70.33 \pm 0.23\%$ and ReVeal $77.89 \pm 0.11\%$ accuracy on the test dataset using five-fold cross-validation.

Furthermore, we apply two recent learning-based detection models that come with their own explanation methods (model-integrated EMs).

3) LineVul: This model is a transformer-based discovery model that jointly trains a self-attention [27] layer used for line-level explanations. LineVul uses a pre-trained large language model based on BERT and fine-tunes on patch diffs represented as tokens [20].

4) VulDeeLocator: This model optimizes an attention layer after a bidirectional LSTM layer and achieves granularity refinement with a top-k pooling layer by filtering out unimportant statements [34]. VulDeeLocator uses a token-based representation extracted from graph slices on the LLVM intermediate representation.

We use LineVul's official implementation[4] and their pre-trained model and retrain VulDeeLocator[5] with their official implementation on their original data. Both models are evaluated on the same dataset as Devign and ReVeal achieving $68.15 \pm 0.43\%$ for VulDeeLocator and $81.11 \pm 0.20\%$ accuracy for LineVul using five-fold cross-validation. LineVul and VulDeeLocator both apply jointly optimized attention layers.

Training dataset. For our experiments, we consider a combined training dataset assembled from the works by Chakraborty et al. [13], Zhou et al. [56], and Russell et al. [42]. To ensure strict separation of training and test data, the programs under test (PUTs) are not part of this dataset. In particular, our training dataset is derived from the following sources:

4. https://github.com/awsm-research/LineVul/
5. https://github.com/VulDeeLocator/VulDeeLocator

1) FFmpeg+Qemu: The Devign dataset comprises vulnerabilities extracted from commits associated with bug fixes. These commits are taken from the FFmpeg and Qemu open-source projects. The bugs are manually annotated and balanced with non-vulnerable code [56].

2) Debian+Chromium: The ReVeal dataset is scraped from patches of Chromium's Bugzilla bug tracker and issues from the Debian Linux security tracker. The dataset is imbalanced and manually annotated [13].

3) Draper dataset: The Draper dataset is a partly synthetic dataset and builds on the software assurance reference dataset. It is partly labeled by static analyzers and has over one million functions with around 6% of them being vulnerable [42].

In total, our combined training dataset contains about one million vulnerable C and C++ functions. It is likely representative of a large set of CWEs in source code and thus provides a good basis for training learning models for vulnerability discovery.

Explanation methods. As subjects for our study, we consider four common explanation methods for machine learning. In particular, we focus on the graph-agnostic methods Smoothgrad [46] and GradCAM [45] that are widely applied in computer vision and the graph-specific methods GNNExplainer [54] and PGExplainer [36] tailored towards explaining GNNs. We chose these methods since they yield the best performances according to other studies focusing on software security [21, 50].

1) GradCAM: This explanation method applies a linear approximation to the intermediate activations of GNN layers [45]. In this work, we take the GradCAM variant where activations of the last convolutional layer before the readout layer are used [21].

2) SmoothGrad: This method averages the node feature gradients on multiple noisy inputs [46]. We use noise sampled from a normal distribution ($\sigma = 0.15$) with 50 samples following Ganz et al. [21].

3) GNNExplainer: This method employs a black-box forward technique for GNNs. For a given graph, it tries to maximize the *mutual information* (MI) of the prediction [54]. Since the method returns edge relevance, we attribute the relevance of each node according to the harmonic mean of adjacent edges. We train the GNNExplainer for 100 epochs with a learning rate of 0.01.

4) PGExplainer: This method uses a so-called explanation network on an embedding of the graph edges [36]. We train the network for 20 epochs with a learning rate of 0.01 using an SGD optimizer.

Since graph-agnostic methods explain only vector-spaced features, we aggregate the feature vectors associated with graph nodes, so that all considered methods yield node-level explanations. Moreover, since each method returns a heatmap with relevant scores, different thresholds will result in different numbers of false positives and false negatives. Compared to Ganz et al. [21] we select the ten most relevant code lines per vulnerable function instead of a number relative to the graph size. Generally speaking, the number of highlighted lines should be small to avoid extensive manual assessment.

Baselines. In addition, we compare our approach against three simple baselines: A *random baseline* assigns relevance to random lines in the functions known to contain bugs. This allows us to draw conclusions about the relevance of EMs. Note that this baseline has prior knowledge about the vulnerable functions and is therefore a strong baseline. Moreover, vanilla *AFL* is included to show the general effectiveness of EM-driven target generation and its use as an oracle. Compared to the behavior of the EM-directed fuzzing, we expect AFL to take longer until crash and to find more unrelated paths. We also compare the explanation methods against two popular open-source rule-based static analyzers *CPPCheck* and *Flawfinder* that already have been used in several studies [56]. We enable CPPCheck's *bug hunting* option to reduce false positives.

Programs under test. For comparing the selected explanation methods under realistic conditions, we consider a set of programs under test (PUTs) with known vulnerabilities in several previous versions. We choose these programs since they are commonly used in fuzzing literature [8, 14, 40] and different fuzzing harnesses are readily available. Note that the source code of these programs is not included in the training set of the learning models and hence unknown to them.

1) Libxml2: The first program is an XML parser written in C with around 70 known CVEs and around 5000 public commits. The input seeds for the fuzzer are based on DTD documents from the respective Git repository.

2) Libming: This program is a flash utility written in C that has around 70 known CVEs associated with overflow or DoS vulnerabilities. The initial seed is an exemplary SWF file. We use the available fuzzing instrumentation script from the AFLGo repository.

3) Giflib: The third program is a library to manipulate GIF image files. Its repository has around 700 commits with only eight publicly known CVEs. The input seed is an empty string. We also use the available fuzzing instrumentation code from the AFLGo repository.

To find potential vulnerabilities, we extract commits that are associated with CVEs and bug fixes from their respective versioning control systems, since this is currently the state-of-the-art approach to build vulnerability datasets [13, 56]. This extraction technique offers insights into whether we can use our method to successfully assess the explanation methods on popular models. We have 65, 69 and 8 vulnerable versions respectively for Libxml2, Libming and Giflib.

Directed fuzzer. For establishing local ground-truth, we rely on the directed fuzzer AFLGo which is a modified version of the coverage-guided fuzzer AFL. Experiments show that AFLGo provides a significant speed-up compared to AFL when trying to reproduce crashes given known targets [8]. We set a time budget of one hour to measure the average crashes per path, the mean breakpoint hits and the average crash distances.

All PUTs are compiled with an *address sanitizer* (ASAN) to increase the yield of address-based crashes. Sanitizers alter the instrumented code by inserting inline reference monitors. This leads to crashes when policy violations, e.g., reading from uninitialized memory, happen. Consequently, we are capable of detecting several more defect types related to memory violations. Lastly, we use the GNU Debugger (GDB)[6] for our crash analysis and determine our extrinsic measures M_1, M_2 and M_3.

Intrinsic criteria. We compare our approach for establishing local ground-truth against intrinsic criteria presented in the introduction, which are commonly used to assess explanation methods [9, 21, 50, 58]. To measure the *descriptive accuracy* (DA), we follow the strategy by Warnecke et al. [50] and calculate the decrease in performance of a learning model when removing the top ten relevant lines of code. If an explanation method correctly identifies important code, the performance will drop significantly and we get a high DA. Conversely, if the explanation method is not able to mark relevant code, the DA will be close to zero.

Furthermore, we calculate the *sparsity* of an explanation by calculating the *mass around zero* (MAZ) [21] of the explanations for all samples. In reality, EMs produce heatmaps $\mathcal{M}$ with vastly different numerical scores, hence we project these relevance scores to the range $[0, 1]$ and count the number of lines lower than a 0.5 threshold averaged by the number of lines. This indicates how much of the *relevance mass* lies lower than a 0.5 threshold. The larger, the fewer lines have been marked as relevant.

All discovery and explanation methods are implemented on top of Pytorch Geometric and all experiments are run on separate AWS EC2 g4dn instances so that the runs do not interfere with each other. We repeat them $n = 5$ times as suggested by Klees et al. [29] since the fuzzing process depends on randomness.

4.2. Experimental Results

We organize the discussion of the experimental results along with the four research questions posed at the beginning of this section. Our goal is to develop an understanding of how local ground-truth can help in analyzing explanation methods in vulnerability discovery and how it improves current practices in software auditing.

RQ1. — *Can we establish ground-truth around vulnerabilities predicted by learning-models?* Given Figure 5, we see that random relevance attribution is by far the worst explanation method on Giflib with only around 3.5% crashes per path on average according to M_1. However, it for example, beats PGExplainer and SmoothGrad on Libming using Devign, proving it to be a strong baseline. Hence, on some PUTs, graph-agnostic explanation methods exhibit the same or worse M_1 score as randomly annotating code lines with relevance. Since the random baseline attributes relevance to random code lines within a vulnerable function, this method will generally find more unique crashes per path over time compared to AFL alone, as seen in the experiments with Libming and Libxml2.

Figures 6 present the mean breakpoint hits and mean crash distances per explanation method for Devign and ReVeal. The more left an EM is located on the map, the closer are the targets to the crash-site (M_3) and the higher an EM, the more target lines lie on the crash's execution trace (M_2). Hence, the green shading denotes a better placement, while the red suggests worse performances

6. https://www.sourceware.org/gdb/

TABLE 1: The time needed to reproduce crashes from CVEs.

CVE	Project	Devign+SmoothGrad	AFL
CVE-2018-11226	Libming	**3h30m4s**±75s	>24h
CVE-2018-7866	Libming	**15m21s**±23s	39m00s±65s
CVE-2014-0191	Libxml2	**2m41s**±54s	13m12s±73s
CVE-2016-5131	Libxml2	**4m43s**±14s	11m15s±29s
CVE-2016-4658	Libxml2	**11m22s**±33s	30m15s±22s
CVE-2014-3660	Libxml2	**26m09s**±19s	56m14s±22s
CVE-2015-7500	Libxml2	**38m04s**±04s	1h13m07s±43s

regarding the two criteria. Including M_2 and M_3 from Figure 6 however, random is the worst method. Lastly, there is always at least one graph-specific method per PUT that outperforms random relevance assignment.

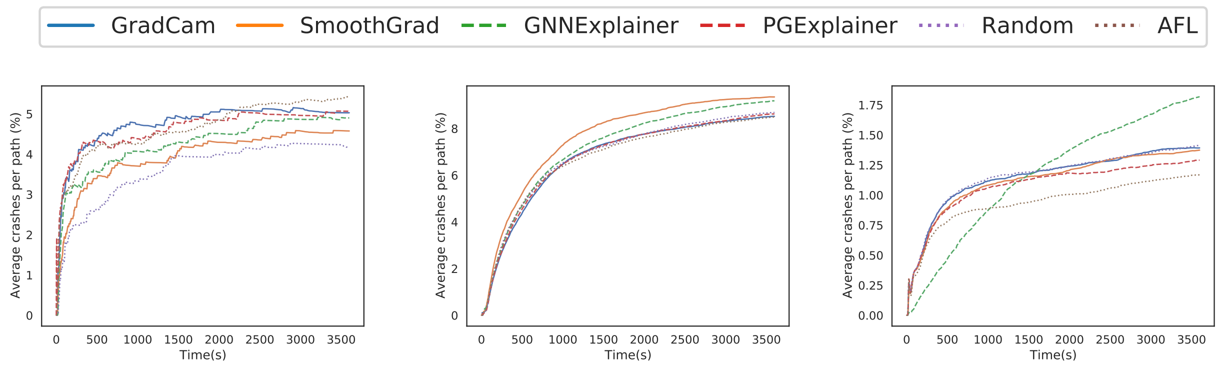
Except for Giflib, vanilla AFL finds fewer crashes per path on average than the other explanation methods. Surprisingly, AFL is among the best methods on Devign for Giflib. This could be due to the overall bad performance of Devign on Giflib compared to ReVeal. Table 1 shows known CVEs contained in the PUTs with the average time needed until the fuzzer reproduces the crash with and without targets from the EM within a fixed period of 24 hours. In our experiments, the fuzzer in combination with an EM reproduces the crash of every CVE substantially faster than AFL alone.

Discussion. According to Table 1, the crash reproduction is faster using the extracted lines from explanation methods compared to vanilla AFL without any targets. After re-executing the generated seeds from the directed fuzzer during the crash analysis, using the debugger, we observe that the lines were indeed hit and close to a crash-site, given the results from Figure 6, since all EMs do have a beneficial M_2 and M_3 score over random. This verifies that the seeds are indeed targeted to the explained lines and that the lines are in fact associated with the crash.

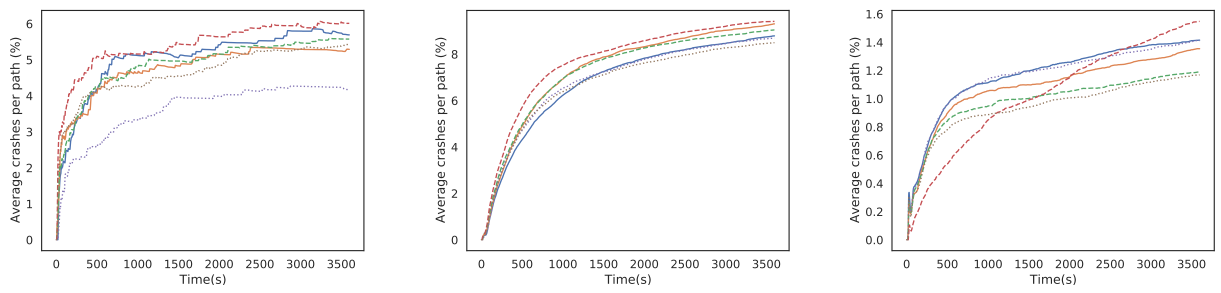
The heatmaps of the EMs advantageously direct the fuzzer to the crash-sites. Thus, we can interpret the heatmaps as local ground-truth around vulnerabilities.

RQ2. — *Which explanation method provides the best explanations according to extrinsic criteria?* For visualization purposes, consider Figure 5 showing the average crashes per path (M_1) over the fuzzing period for five runs for Devign and ReVeal on all three PUTs. AFLGo uses *simulated annealing* [28] to schedule the energy assignment for the generated seeds [8]. We can see that the fuzzing process gets more and more targeted until we observe an asymptotic progression suggesting an optimum.

We can fit this behavior to a logarithmic function parameterized by time: $f(x) = a \cdot log(x) + b$, where a is the *logarithmic stretch* and b denotes an intuition for the initial found crashes per path. A higher a corresponds to a steeper approximated slope and consequently denotes the speed by the targets that let the fuzzer find crashes. Since the first crashes are hardly targeted, we are more interested in the steepness of the progression a and not the initial performance b. The logarithmic stretch allows us to simplify the comparison for the M_1 and is depicted in Table 2.



(a) Average unique crashes per path over time (M_1) for Giflib, Libming and Libxml2 with Devign.



(b) Average unique crashes per path over time (M_1) for Giflib, Libming and Libxml2 with ReVeal.

Figure 5: Fuzzing results for models *Devign* and *ReVeal*.

TABLE 2: Logarithmic stretch a per EM, PUT and model for average unique crashes per path over time (M_1).

PUT	Gradcam		GNNExplainer		SmoothGrad		PGExplainer		LineVul	VulDeeLocator	Flawfinder	CPPCheck	AFL	Random
	Devign	ReVeal	Devign	ReVeal	Devign	ReVeal	Devign	ReVeal						
Libming	2.01	2.09	2.15	2.06	2.14	**2,16**	2.00	2.04	2.12	2.10	1.50	1.54	1.91	1.98
Libxml2	0.26	0.26	**0.48**	0.21	0.26	0.25	0.23	0.37	0.23	0.25	0.09	0.19	0.21	0.25
Giflib	0.60	**0.94**	0.73	0.81	0.71	0.76	0.54	0.70	0.78	0.72	0.53	0.70	0.71	0.30

With the logarithmic stretch, we first assess the influence of the model on the EM's output, for instance, GradCam, SmoothGrad, GNNExplainer and PGExplainer in Figure 5. Overall we can see that all EMs show a beneficial progression over time for at least one dataset and model combination since a higher a value denotes a steeper increase. Judging by the best scores, graph-specific EMs score four out of six and graph-agnostic EMs score two out of six. PGExplainer finds more crashes on ReVeal than all other methods on Devign for Libxml2 given Table 2. GNNExplainer outperforms all other methods on Libxml2 for Devign, since the others show an asymptotic progression to a lower plateau. On Libming, however, they are equally performing.

According to the M_2 and M_3 scores from Figure 6, GNNExplainer gives the most concise explanations for Devign regarding the mean breakpoint hits while Smoothgrad yields the best score concerning the mean crash distance. This means that the explanations by GNNExplainer result in targets that are more often part of the execution trace but the distance to the crash-site is further away from the crash-site. On the other hand, SmoothGrad does not have as many breakpoints but the relevant lines are closer to the crash-site. Conversely, GradCam gives the worst results since only a few lines are relevant and those lines marked are further away from the crash-site than for the other EMs. Even the baseline highlighted more lines that have been part of crash traces than GradCam. Furthermore, Figure 6b shows that GNNExplainer yields the most relevant lines measured by the breakpoints that were hit during crash reproduction and the average distance to the crash-site. Although the average crashes per path are best for PGExplainer, GNNExplainer has the most precise explanations. GNNExplainer is on the Pareto front for both maps.

At first sight, the model-integrated EMs perform similarly compared to the separate EMs as indicated by Figure 8b and Table 2. LineVul is slightly better on Libming and Giflib, while VulDeeLocator is better on Libxml2 but still only on par with Random. Given Table 3 VulDeeLocator, however, has a far better M_2 score, which indicates that the explanation is more accurate, while less close to the local ground-truth given the M_3 metric compared to LineVul. Both methods outperform GradCam and PGExplainer but are inferior to GNNExplainer and SmoothGrad.

Discussion. Our evaluation reveals, that the choice of EM is also dependent on the choice of discovery model. A different model might work better with different EMs. In general, however, are graph-specific EMs, for instance, GNNExplainer, achieving the best results in our experiments for graph-based vulnerability detection models as seen in Table 3. The relevant explained lines are all close to the examined crashes and are, compared to the others, more often part of the crashing execution trace. We also see that graph-specific methods also perform better regarding

TABLE 3: M_2 and M_3 comparison between model-integrated and model-independent EMs with standard deviation.

Model	Mean Breakpoint Hits M_2	Mean Crash Distance M_3
Random	8.21 ± 0.40%	0.124 ± 0.014s
GNNExplainer	**15.48** ± 0.22%	0.084 ± 0.002s
SmoothGrad	11.06 ± 0.13%	**0.077** ± 0.003s
PGExplainer	10.04 ± 0.37%	0.088 ± 0.010s
GradCam	9.26 ± 0.05%	0.097 ± 0.006s
VulDeeLocator	10.54 ± 0.14%	0.094 ± 0.002s
LineVul	9.97 ± 0.16%	0.084 ± 0.003s

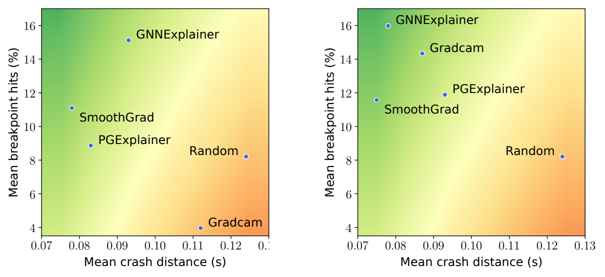
M_1 where graph-specific EMs achieve the best scores in four out of six while graph-agnostics only in two out of six experiments according to Table 2. Contrary to the empirical study of Ganz et al. [21] we find that graph-specific methods are more veracious than graph-agnostic EMs. We thus conclude that the extrinsic criteria enable us to qualitatively compare EMs, which has not been possible before for vulnerability discovery.

Model-integrated EMs are a viable replacement for separate vulnerability discovery models and EM combinations. In our experimental study, however, we observe two model-separate beat their performance, namely GNNExplainer and SmoothGrad.

Our extrinsic criteria indicate that graph-specific explanation methods highlight vulnerabilities best. When possible, these methods should be used to explain code in graph-based vulnerability discovery.

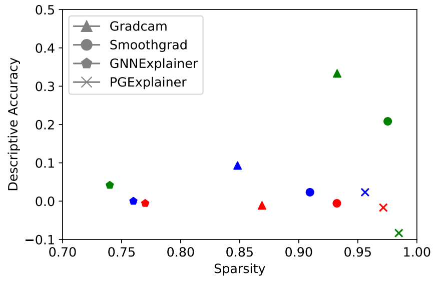
RQ3. — *How do extrinsic and intrinsic criteria differ when comparing explanation methods?* We evaluate two commonly used intrinsic criteria for ReVeal, Devign, VulDeeLocator and LineVul on the PUTs in Figure 7. We measure the sparsity (MAZ) and descriptive accuracy for each EM and PUT. The higher and the further to the right the result is in the graph, the better the EM is according to the intrinsic criteria. Overall, Gradcam and Smoothgrad yield the best DA for ReVeal. Most explanations however lie closely around zero according to the DA. Considering the sparsity, only about 75% of all code lines' relevance scores lie within the range of $[0, 0.5]$ for GNNExplainer. PGExplainer has the best MAZ for all PUTs and for both models since around $0.95 - 0.98\%$ of the code lines score an accumulated relevance lower than 50%. Our results are in line with Ganz et al. [21] since according to them, Smoothgrad is among the best candidates considering the DA and PGExplainer produce the most concise explanations.

Reviewing the snippet in Figure 1, we note that VulDeeLocator does not detect the sample, while LineVul highlights lines 14, 23 and 31. Interestingly, it is able to precisely pinpoint the crash-site, however, it seems to also detect more false positives. Given their intrinsic evaluation in Figure 7c, we can support the observation that model-integrated methods appear to have an abundant heatmap, given their sparsity score, but achieve an up to 400% better DA than the separate EMs, i.e. those that are not coupled with the discovery model. VulDeeLocator, on the other hand, has an inferior DA than LineVul. Only the gradient-based EM GradCam (on ReVeal) has a similar Descriptive Accuracy as LineVul, which mirrors the results from their intrinsic evaluation [20].

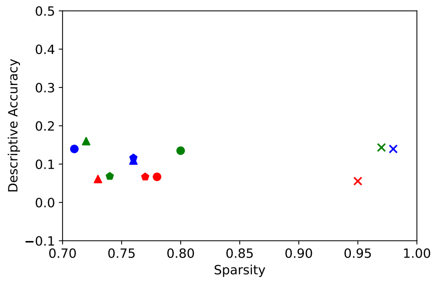


(a) Pareto map for Devign. (b) Pareto map for ReVeal.

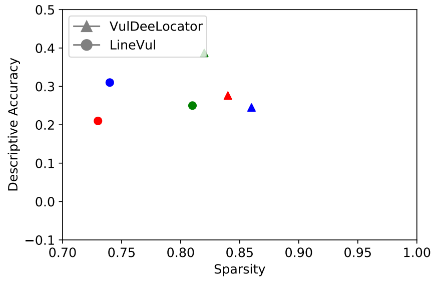
Figure 6: Mean breakpoint hits (M_2) and mean crash distance (M_3).



(a) ReVeal



(b) Devign



(c) Models with integrated EMs

Figure 7: Intrinsic evaluation. Red, Green and Blue denote Libxml2, Giflib and Libming respectively.

Discussion. We can see that the DA for Devign yields no particular information whatsoever. Gradcam and Smoothgrad yield a better DA for ReVeal. Although their DA is superior compared to GNNExplainer and PGExplainer, their located code lines are, however, unrelated to the actual underlying vulnerability concluding from our extrinsic results. Furthermore, Smoothgrad has a low DA on Libming although its speedup of the crash reproduction was the fastest, resulting in the highest M_1 score given Table 2. The performance of the EMs measured by their intrinsic criteria in Figure 7 shows a large discrepancy from the performances measured by our extrinsic criteria. The descriptive accuracy is similar across all PUTs, EMs and even models. In contrast, our extrinsic criteria show vastly different performance plateaus among the EMs taken from Table 2 and the Pareto maps from Figure 6a and 6b.

It has been shown that the DA is sensitive to learned artifacts in the model, such as feature overfitting [21, 58]. This can be explained analogously with an image recognition task: Imagine evaluating a model that classifies *boats* and *cars*. Intuitively, the model could learn to focus on whether the image contains water or not. Consequently, removing features such as *coastline* and *water* causes the model's performance to drop significantly, resulting in a higher DA. Hence, the most relevant features are not important for solving the actual underlying task and the DA fails to capture this. Moreover, if a model generalizes well over its features, it may be robust to the removal of its top features, resulting in a low DA while still performing well in its task.

It turns out that this feature overfitting, measured by the discrepancy between intrinsic and extrinsic criteria, is even more extreme when the model and the EM are jointly trained. The tighter coupling between VulDeeLocator and LineVul's model and EM causes the EM to be more sensitive to the overfit artifacts, i.e. , noise, under or overrepresented features, or outliers. LineVul and VulDeeLocator, for instance, are trained on vulnerable functions and their associated patches. The modified lines in the patch are used to train their EM. This introduces bias, for instance in Example 1, another possible fix might as well change the return type of `xmlStrlen` from `int` to `size_t`. However, the maintainers decided against this[7]. Thus, the patch and the bug location can be very different. This might be the reason why LineVul highlights line 23, since buffer length calculations might be often part of a patch in their datasets.

More formally speaking, the disadvantage of the descriptive accuracy is the double use of model f: firstly as a model from which the relevant code lines are calculated and secondly as an oracle to evaluate them. Since M_1 uses a directed fuzzer, we can remove any bias by the decoupling of the oracle from the discovery model. Another advantage of our extrinsic criteria is that they rely on an oracle based on dynamic analysis instead of a static analyzer f. Christakis et al. and Dietrich et al. state that dynamic analysis should be preferred to validate the soundness of a static analyzer [17, 18]. Hence, we can conclude that M_1 uses a more faithful oracle for the real-world task of vulnerability discovery.

Considering the sparsity, GNNExplainer and LineVul achieve the worst MAZ results, and PGExplainer and Smoothgrad yield the best. However, judging by how often their lines lie on the execution trace and how close they were to the crash, they perform exactly counter-factually using our extrinsic criteria.

Suppose a line from an explanation was neither close to the crash-site nor was it even executed during a fuzzing iteration. In that case, we can clearly say that removing this feature does not affect M_1 but increases the conciseness of an explanation. On the other hand, we can take into account what it means for an explanation to be maximally concise. Consider an example where every highlighted line is part of the execution trace and maximally close to the crash-site. Removing a single line might give us a more concise explanation but at the cost of valuable information. We conclude that M_2 and M_3 measure sparsity, too, but if we rely on the intrinsic sparsity to compare EMs, we may end up with an EM that labels a few code lines as relevant but none of these actually deliver information to locate or fix the bug. As an example of this phenomenon, consider Smoothgrad in Figure 1. Therefore, the ability of our extrinsic criteria to measure EMs is closer to the underlying task of vulnerability discovery and less susceptible to learned biases.

Our criterion M_1 relates to the descriptive accuracy, while M_2 and M_3 describe sparsity. However, our extrinsic criteria provide a better basis for comparison, as their results turn out to be more diverse and consistent compared to intrinsic criteria.

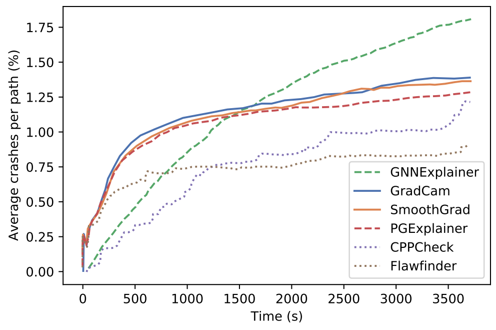
RQ4. — *How do rule-based auditing tools perform against explanation methods?* We compare Flawfinder and CPPCheck against the explanation methods in Table 2. All static analyzers are inferior in our experiments and only beat the random baseline on Giflib. Figure 8a exemplary visualizes the average crashes per path (M_1) for the explanation methods and the static analyzers over the fuzzing duration for Libxml2 and Devign. Over time, except GNNExplainer and Flawfinder, all methods converge to a similar plateau. Towards the end, CPPCheck identifies more crashes than Flawfinder. It has already been observed in the work of Arusoaie et al. [3], that CPPCheck is more effective than Flawfinder. In general, the improved effectiveness of the explanation methods compared to the static analyzers is likely due to the fact that Devign and ReVeal are better at detecting vulnerable code artifacts.

For visualization purposes, we want again to point to Figure 1 which shows a critical vulnerability in Libxml2[8]. Flawfinder reports a possible flaw in `memcpy` due to `len` not being checked. Hence, Flawfinder is the only method to correctly detect the crashing location without any false positive, although the accompanying description is wrong whatsoever since `len` is not the problem. CPPCheck on the other hand does not detect the flaw, even with the bug-hunting option disabled, resulting in potentially more false positives.

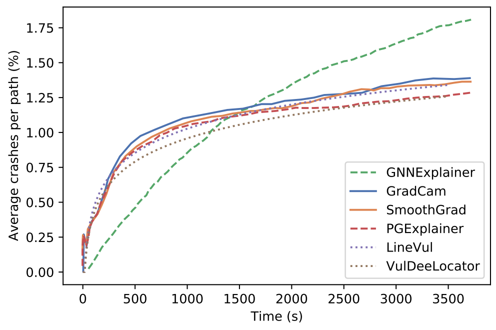
Discussion. As previously pointed out, another advantage is that our extrinsic criteria allow us to benchmark vulnerability discovery models with their EMs against classical rule-based static analyzers, which is not possible with intrinsic criteria. In our experiments given Figure 8a, we see that all EMs beat the open-source static analyzers when comparing the average found crashes over time in our experimental study.

7. https://bugzilla.gnome.org/show_bug.cgi?id=763071

8. https://gitlab.gnome.org/GNOME/libxml2/-/commit/8fbbf55



(a) M_1 for EMs with Devign against Static Analyzers.



(b) M_1 for EMs with Devign against Model-integrated EMs

Figure 8: Exemplary plot for the average unique crashes per path over time for Libxml2.

Our extrinsic criteria indicate that the combination of EMs and learning-based vulnerability discovery models reveal code relevant to vulnerabilities better than traditional static analyzers.

5. Limitations and Implications

We proceed to discuss the limitations and implications of our approach and provide recommendations for its application in practice.

Limits of fuzzing. If we assume that a vulnerability is present in the program under test, it is not known in advance whether a fuzzer can reach it due to time constraints or roadblocks. Worse, it is not even certain whether the defect actually causes a crash. Clearly, our approach is limited to vulnerabilities that can be generally identified through a fuzzer. Interestingly, however, such vulnerabilities largely overlap with those that can be uncovered using learning-based vulnerability discovery

To illustrate this relation, we compile a non-exhaustive list of Common Weakness Enumeration (CWE) numbers that vulnerability discovery models, such as [16, 20, 49], are capable of detecting statically. We then cross-reference these CWE numbers with those that fuzzers, such as [24], are able to find, and present the results in Table 4 in the Appendix. We conclude that not all vulnerabilities in a dataset can be successfully uncovered by fuzzing, but we can still gather enough evidence using some descriptive samples to evaluate one explanation method in preference to another, which after all, is our main proposition.

Moreover, AFL, the fuzzer we employ, suffers from hash collisions in the way it stores visited branches[9]. As a consequence, a random portion of the seeds that have been found is dropped before the crash analysis. However, this (statistically) does not influence the outcome for M_2 and M_3 because both average quantities (breakpoint hits, or crash distance) over the set of seeds.

Implications. We have seen that intrinsic criteria can lead to misleading results. Although our approach is not perfect, it is an important step forward that helps to better compare and evaluate explanation methods. In particular, when learning-based vulnerability discovery methods are jointly applied with fuzzing, for example as part of a security audit, our approach is a natural fit and allows determining the best explanations methods for the program under test. Moreover, our method is applicable in all scenarios where fuzzing is effective and thus can serve as an oracle to improve learning-based vulnerability discovery. This, for instance, holds for all open-source software currently investigated in the OSS-Fuzz project.

However, there are also scenarios where our approach is not suitable for evaluating explanation methods. If the program under test is small and only a few samples are available, the proposed criteria may not provide meaningful results because not many bugs can be validated by the fuzzer. Similarly, if the time budget is limited, the fuzzer may not have enough processing time to go through the important branches. In these cases, our extrinsic criteria may not be meaningful. Still, we recommend sticking to a manual assessment of samples in these cases, rather than relying on intrinsic criteria.

6. Related Work

Our work provides a novel link between two active areas of security research: vulnerability discovery using machine learning and directed fuzzing. As a result, there exist different prior work related to our approach that we discuss in the following.

Learning-based vulnerability discovery. The combination of GNNs and code graphs, considered in our work, has proven successful in the discovery of bugs and security vulnerabilities in a series of research [11, 15, 48, 56]. For example, Zhou et al. introduce the first gated graph neural network on code property graphs to identify bugs and vulnerabilities collected from real-world commits. Their approach outperforms popular open-source and commercial static analyzers as well as token-based learning models [56]. Cao et al. choose a different graph representation of the underlying source code. They combine data-flow and control-flow graphs with the abstract syntax tree to the code composite graph [11].

Recently, Chakraborty et al. reveal that several state-of-the-art datasets to evaluate these models are not realistic [12]. In a similar vein, Arp et al. [2] discuss common pitfalls when working with methods learning-based vulnerability discovery. We argue that these problems can only be tackled if appropriate explanation methods are employed and hence the process of vulnerability discovery becomes transparent to the practitioner.

9. https://github.com/mirrorer/afl/blob/master/docs/technical_details.txt

Explanation methods. Several recent surveys have developed taxonomies, algorithms and evaluation criteria for explanation methods in machine learning [6, 47]. With the rise of graph neural networks, several works ported the underlying classic explanation concepts to the graph domain [4, 41], as well as completely new graph-specific algorithms have been invented [36, 44, 54]. For instance, Yuan et al. describes a comprehensive taxonomy for explainable graph-specific machine learning with a categorization of current algorithms and evaluation methods [55]. Guo et al. introduce a black-box explanation method for security-critical machine learning models [23]. Some recent approaches incorporate EMs in the vulnerability discovery task to integrate interpretability directly in the learning process [20, 25, 34]. However, all work in this direction focuses on either intrinsic criteria evaluating the explanations by the *descriptive accuracy* or *sparsity* or suggesting human expert studies to validate the actual usefulness. Closest to our approach, Sanchez-Lengeling et al. compare explanation methods using several types of ground-truth for molecule graphs [43].

Comparing explanation in security. Explanation methods have already been applied to learning models in security in different studies. Warnecke et al. show that it is non-trivial to validate security-critical models from explanations given by several algorithms with a predefined set of evaluation criteria [50]. However, explaining the decisions of such models is crucial [2]. Zou et al. present a method to extract important tokens from token-based vulnerability discovery models. The extraction works by perturbing input source code pieces such that the classification label switches from 1 to 0. Black-box explanation methods yield better portability between different models but the overall performance deteriorates. Furthermore, they use *descriptive accuracy* to measure the performance. Since this is an intrinsic metric, it is impossible to make any assumptions about the veracity [58]. Finally, Linardatos et al. state that it is unfeasible to rank EMs by their ability to make a model's decision interpretable [35].

Directed fuzzers. Since we rely on directed fuzzers, we briefly discuss popular approaches, including AFLGo [8] and Hawkeye [14]. Both model the targeted input generation as a power-schedule problem. Beacon [26] tries to incorporate path pruning into the seed selection process and hereby accelerates crash reproduction compared to AFLGo and Hawkeye. Targetfuzz [10] prioritizes the initial seed selection to speed up directed fuzzing. Other works focus on improving the instrumentation of grey-box fuzzers by heuristically extracting potentially interesting code regions, for example in the work by Österlund et al. [40]. Zhu et al. use explanation methods in conjunction with an NLP-based bug-detecting model to speed up the directed fuzzer AFLGo. V-Fuzz also speeds up fuzzing with learning techniques: it uses a neural network to detect likely vulnerable spots in binary programs [31].

Static analysis report verification. From a broader perspective, our work compares static code analysis methods using dynamic analysis. This approach has been also persuaded in other contexts. For example, Christakis et al. [17] validate unverified and potentially unsound static code analysis reports using dynamic code execution to reduce false positives. Similarly, Wüstholz and Christakis [52] build upon this work and use online static analysis to guide a fuzzer by analyzing each path during the fuzzing process right before a new input is selected. Closely related to our explanation oracle, Barr et al. [5] define testing oracles as mechanisms that decide whether a set of system tests are relevant or not. Dietrich et al. [18] state that it makes more sense to validate static analysis results using oracles based upon dynamic analysis. All these approaches are related to our work, yet they focus on different types of static tools and do not consider learning-based discovery methods and their explanation.

7. Conclusion

In this work, we present a novel method to compare explanation methods for learning-based vulnerability discovery models by their veracity. Current advances in the field consider vulnerability discovery as a classical machine learning task. They fail to connect it to the underlying problem, which is static program analysis. Since there is a large pool of explanation methods available to choose from, with each yielding vastly different explanations, we present an appropriate and novel method to systematically and automatically evaluate extracted explanations for deep learning-based vulnerability-detecting models.

We propose directed fuzzing to selectively generate ground-truth and verify and compare the relevance of explanations. We show that several general assumptions drawn from past experimental studies are biased. For instance, recent work uses inadequate oracles or none at all to compare EMs. This leads to results that advise against black-box or graph-specific explanation methods in past works, such as GNNExplainer. However, by using dynamic execution as a more appropriate oracle, our results suggest to still consider black-box and graph-specific EMs for vulnerability discovery. In addition, we present evidence that integrating explanation methods directly into the learning task to discover weaknesses can further compromise performance comparison. We conclude that our method is suitable for testing explanation methods and verifying practical considerations of whether or not learning-based vulnerability discovery models should be incorporated into everyday secure software development. Based on our results, we hope to foster research in the fields of explainable machine learning for vulnerability discovery.

Acknowledgments

This work has been funded by the Federal Ministry of Education and Research (BMBF, Germany) in the project IVAN (FKZ: 16KIS1165K).

References

[1] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL https://openreview.net/forum?id=BJOFETxR-.

[2] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck. *Dos and Don'ts of Machine Learning in Computer Security*. Usenix Security Symposium (USENIX). USENIX, 2022 edition, July 2021.

[3] A. Arusoaie, S. Ciobâca, V. Craciun, D. Gavrilut, and D. Lucanu. A comparison of open-source static analysis tools for vulnerability detection in c/c++ code. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 161–168, 2017. doi: 10.1109/SYNASC.2017.00035.

[4] F. Baldassarre and H. Azizpour. Explainability techniques for graph convolutional networks. *ArXiv*, abs/1905.13686, 2019.

[5] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41 (5):507–525, 2015. doi: 10.1109/TSE.2014.2372785.

[6] V. Belle and I. Papantonis. Principles and practice of explainable machine learning. *Frontiers in Big Data*, 4, 2021. ISSN 2624-909X. doi: 10.3389/fdata.2021.688969. URL https://www.frontiersin.org/article/10.3389/fdata.2021.688969.

[7] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler. Neural code comprehension: A learnable representation of code semantics. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 3589–3601, Red Hook, NY, USA, 2018. Curran Associates Inc.

[8] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349468. doi: 10.1145/3133956.3134020. URL https://doi.org/10.1145/3133956.3134020.

[9] N. Burkart and M. F. Huber. A survey on the explainability of supervised machine learning. *J. Artif. Intell. Res.*, 70:245–317, 2021.

[10] S. Canakci, N. Matyunin, K. Graffi, A. Joshi, and M. Egele. Targetfuzz: Using darts to guide directed greybox fuzzers. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, page 561–573, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391405. doi: 10.1145/3488932.3501276. URL https://doi.org/10.1145/3488932.3501276.

[11] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li. Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection. *Information and Software Technology*, 136:106576, 2021. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2021.106576. URL https://www.sciencedirect.com/science/article/pii/S0950584921000586.

[12] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, TBD:1, 2020. URL https://git.io/Jf6IA.

[13] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering*, 2021.

[14] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu. Hawkeye: Towards a desired directed greybox fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2095–2108, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243849. URL https://doi.org/10.1145/3243734.3243849.

[15] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Trans. Softw. Eng. Methodol. 1, 1, Article*, 1:32, 2020. doi: 10.1145/3436877. URL https://doi.org/10.1145/3436877.

[16] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Trans. Softw. Eng. Methodol. 1, 1, Article*, 1:32, 2020. doi: 10.1145/3436877. URL https://doi.org/10.1145/3436877.

[17] M. Christakis, P. Müller, and V. Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 144–155, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884843. URL https://doi.org/10.1145/2884781.2884843.

[18] J. Dietrich, L. Sui, S. Rasheed, and A. Tahir. On the construction of soundness oracles. pages 37–42, 06 2017. doi: 10.1145/3088515.3088520.

[19] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987. ISSN 0164-0925. doi: 10.1145/24039.24041. URL https://doi.org/10.1145/24039.24041.

[20] M. Fu and C. Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE, 2022.

[21] T. Ganz, M. Härterich, A. Warnecke, and K. Rieck. Explaining graph neural networks for vulnerability discovery. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*, AISec '21, page 145–156, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386579. doi: 10.1145/3474369.3486866. URL https://doi.org/10.1145/3474369.3486866.

[22] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, page

85–96, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339353. doi: 10.1145/2857705.2857720. URL https://doi.org/10.1145/2857705.2857720.

[23] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing. Lemna: Explaining deep learning based security applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 364–379, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243792. URL https://doi.org/10.1145/3243734.3243792.

[24] A. Hazimeh, A. Herrera, and M. Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), jun 2021. doi: 10.1145/3428334. URL https://doi.org/10.1145/3428334.

[25] D. Hin, A. Kan, H. Chen, and M. A. Babar. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 596–607, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393034. doi: 10.1145/3524842.3527949. URL https://doi.org/10.1145/3524842.3527949.

[26] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang. Beacon : Directed grey-box fuzzing with provable path pruning. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 104–118, Los Alamitos, CA, USA, may 2022. IEEE Computer Society. doi: 10.1109/SP46214.2022.00007. URL https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00007.

[27] S. Jain and B. C. Wallace. Attention is not Explanation. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3543–3556, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1357. URL https://aclanthology.org/N19-1357.

[28] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220 (4598):671–680, 1983. doi: 10.1126/science.220.4598.671. URL https://www.science.org/doi/abs/10.1126/science.220.4598.671.

[29] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243804. URL https://doi.org/10.1145/3243734.3243804.

[30] D. C. Kozen. *Rice's Theorem*, pages 245–248. Springer Berlin Heidelberg, Berlin, Heidelberg, 1977. ISBN 978-3-642-85706-5. doi: 10.1007/978-3-642-85706-5_42. URL https://doi.org/10.1007/978-3-642-85706-5_42.

[31] Y. Li, S. Ji, C. Lyu, Y. Chen, J. Chen, Q. Gu, C. Wu, and R. Beyah. V-fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs. *IEEE Transactions on Cybernetics*, 52(5):3745–3756, 2022. doi: 10.1109/TCYB.2020.3013675.

[32] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, S. Wang, and J. Wang. Sysevr: A framework for using deep learning to detect software vulnerabilities. 2018. doi: 10.21227/fhg0-1b35. URL https://dx.doi.org/10.21227/fhg0-1b35.

[33] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. 01 2018. doi: 10.14722/ndss.2018.23165.

[34] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin. Vuldeelocator: A deep learning-based fine-grained vulnerability detector. *CoRR*, abs/2001.02350, 2020. URL http://arxiv.org/abs/2001.02350.

[35] P. Linardatos, V. Papastefanopoulos, and S. Kotsiantis. Explainable ai: A review of machine learning interpretability methods. *Entropy*, 23(1), 2021. ISSN 1099-4300. doi: 10.3390/e23010018. URL https://www.mdpi.com/1099-4300/23/1/18.

[36] D. Luo, W. Cheng, D. Xu, W. Yu, B. Zong, H. Chen, and X. Zhang. Parameterized explainer for graph neural network. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

[37] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47:2312–2331, 2021.

[38] A. Mazuera-Rozo, A. Mojica-Hanke, M. Linares-V'asquez, and G. Bavota. Shallow or deep? an empirical study on detecting vulnerabilities using deep learning. *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 276–287, 2021.

[39] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, page 3111–3119, Red Hook, NY, USA, 2013. Curran Associates Inc.

[40] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida. ParmeSan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2289–2306. USENIX Association, Aug. 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund.

[41] P. E. Pope, S. Kolouri, M. Rostami, C. E. Martin, and H. Hoffmann. Explainability methods for graph convolutional neural networks. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10764–10773, 2019. doi: 10.1109/CVPR.2019.01103.

[42] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762, 2018. doi: 10.1109/ICMLA.2018.00120.

[43] B. Sanchez-Lengeling, J. Wei, B. Lee, E. Reif,

P. Wang, W. Qian, K. McCloskey, L. Colwell, and A. Wiltschko. Evaluating attribution for graph neural networks. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 5898–5910. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/417fbbf2e9d5a28a855a11894b2e795a-Paper.pdf.

[44] T. Schnake, O. Eberle, J. Lederer, S. Nakajima, K. Schütt, K.-R. Mueller, and G. Montavon. Higher-order explanations of graph neural networks via relevant walks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PP:1–1, 09 2021. doi: 10.1109/TPAMI.2021.3115452.

[45] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017.

[46] D. Smilkov, N. Thorat, B. Kim, F. B. Viégas, and M. Wattenberg. Smoothgrad: removing noise by adding noise. *ArXiv*, abs/1706.03825, 2017.

[47] E. Tjoa and C. Guan. A survey on explainable artificial intelligence (xai): Toward medical xai. *IEEE Transactions on Neural Networks and Learning Systems*, 32:4793–4813, 2021.

[48] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 16:1943–1958, 2021. ISSN 15566021. doi: 10.1109/TIFS.2020.3044773.

[49] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 16:1943–1958, 2021. ISSN 15566021. doi: 10.1109/TIFS.2020.3044773.

[50] A. Warnecke, D. Arp, C. Wressnegger, and K. Rieck. Evaluating Explanation Methods for Deep Learning in Security. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 158–174, Genoa, Italy, Sept. 2020. IEEE. ISBN 9781728150871. doi: 10.1109/EuroSP48549.2020.00018. URL https://ieeexplore.ieee.org/document/9230374/.

[51] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32:4–24, 2019.

[52] V. Wüstholz and M. Christakis. Targeted greybox fuzzing with static lookahead analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 789–800, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380388. URL https://doi.org/10.1145/3377811.3380388.

[53] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, page 590–604, USA, 2014. IEEE Computer Society. ISBN 9781479946860. doi: 10.1109/SP.2014.44. URL https://doi.org/10.1109/SP.2014.44.

[54] R. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec. *GNNExplainer: Generating Explanations for Graph Neural Networks*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[55] H. Yuan, H. Yu, S. Gui, and S. Ji. Explainability in graph neural networks: A taxonomic survey. *ArXiv*, abs/2012.15445, 2020.

[56] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[57] X. Zhu, S. Liu, X. Li, S. Wen, J. Zhang, S. A. Çamtepe, and Y. Xiang. Defuzz: Deep learning guided directed fuzzing. *CoRR*, abs/2010.12149, 2020. URL https://arxiv.org/abs/2010.12149.

[58] D. Zou, Y. Zhu, S. Xu, Z. Li, H. Jin, and H. Ye. Interpreting deep learning-based vulnerability detector predictions based on heuristic searching. *ACM Trans. Softw. Eng. Methodol.*, 30(2), mar 2021. ISSN 1049-331X. doi: 10.1145/3429444. URL https://doi.org/10.1145/3429444.

A. Appendix

TABLE 4: CWEs detected by fuzzers (F) and ML models (M) as reported by [16, 20, 24, 49].

CWE	Description	Detected By	Sanitizer needed
20	Improper Input Validation	FM	
22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	FM	
74	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')	M	
77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	M	
78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	M	
119	Improper Restriction of Operations within the Bounds of a Memory Buffer	FM	
125	Out-of-bounds Read	FM	ASAN
130	Improper Handling of Length Parameter Inconsistency	F	
131	Incorrect Calculation of Buffer Size	F	
133	String Errors	F	
138	Improper Neutralization of Special Elements	FM	
172	CWE-172: Encoding Error	F	
189	Numeric Errors	F	
190	Integer Overflow or Wraparound	FM	UBSAN
191	Integer Underflow (Wrap or Wraparound)	FM	UBSAN
200	Exposure of Sensitive Information to an Unauthorized Actor	FM	
269	Improper Privilege Management	M	
284	Improper Access Control	M	
285	Improper Authorization	M	
287	Improper Authentication	FM	
310	Cryptographic Issues	F	
362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	FM	TSAN
369	Divide By Zero	FM	
393	Return of Wrong Status Code	F	
399	Resource Management Errors	FM	
400	Uncontrolled Resource Consumption	FM	
404	Improper Resource Shutdown or Release	FM	
415	Double Free	FM	
416	Use After Free	F	
434	Unrestricted Upload of File with Dangerous Type	F	
457	Use of Uninitialized Variable	F	
465	C: Pointer Issues	F	
467	Use of sizeof() on a Pointer Type	M	
469	Use of Pointer Subtraction to Determine Size	FM	UBSAN
476	NULL Pointer Dereference	FM	MSAN
514	Covert Channel	F	
573	Improper Following of Specification by Caller	M	
610	Externally Controlled Reference to a Resource in Another Sphere	M	
611	Improper Restriction of XML External Entity Reference	F	
617	Reachable Assertion	FM	
662	Improper Synchronization	F	
665	Improper Initialization	FM	
666	Operation on Resource in Wrong Phase of Lifetime	M	
668	Exposure of Resource to Wrong Sphere	M	
670	Always-Incorrect Control Flow Implementation	M	
674	Uncontrolled Recursion	F	
681	Incorrect Conversion between Numeric Types	F	
682	Incorrect Calculation	F	
703	Improper Check or Handling of Exceptional Conditions	F	
704	Incorrect Type Conversion or Cast	FM	
706	Use of Incorrectly-Resolved Name or Reference	F	
754	Improper Check for Unusual or Exceptional Conditions	FM	
758	Reliance on Undefined, Unspecified, or Implementation-Defined Behavior	FM	UBSAN
770	Allocation of Resources Without Limits or Throttling	FM	
772	Missing Release of Resource after Effective Lifetime	FM	LeakSAN
787	Out-of-bounds Write	FM	
834	Excessive Iteration	FM	
835	Loop with Unreachable Exit Condition ('Infinite Loop')	F	