Software Defect Localization Using Explainable Deep Learning

vorgelegt von
Master of Science
Tom Ganz

an der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
Dr. rer. nat.

genehmigte Dissertation

Promotionsausschuss:
Vorsitzender: Prof. Wojciech Samek
Gutachter: Prof. Konrad Rieck
Gutachter: Prof. Lorenzo Cavallaro
Gutachter: Prof. Martin Johns
Tag der wissenschaftlichen Aussprache: 26.04.2024

Berlin 2024

Zusammenfassung

Mit den wachsenden Anwendungsbereichen für Computerprogramme steigt auch die Sicherheitsbedrohung durch Cyber-Angriffe, welche schwerwiegende Auswirkungen und erhebliche finanzielle Schäden für Nutzer und Betreiber zur Folge haben können. Um die Risiken einzudämmen, stehen Softwareentwickler heute vor der Herausforderung, Schwachstellen in der Programmierung schnellstmöglich identifizieren und entschärfen zu können. Die automatisierte Fehlererkennung auf Basis von maschinellen Lernen hat sich hierbei als vielversprechendes Tool zur Unterstützung abgezeichnet. Dabei werden Modelle trainiert, die in der Lage sind, Schwachstellen im Programmcode, welche für einen Angriff potentiell anfällig sind, zu erkennen. Gegenüber herkömmlichen, regelbasierten statischen Anwendungstests, bieten maschinell trainierte Modelle den Vorteil, dass sie auf projektspezifische Codes angepasst werden und individuelle Entscheidungsgrenzen festgelegt werden können. Sie zeigen eine vielversprechende Leistung und übertreffen teilweise sogar traditionelle Analysetechniken.

Trotzdem weisen auf maschinellem Lernen basierende Fehlerdetektoren noch Defizite auf. Sie sind nur begrenzt auf andere Projekte übertragbar und somit wenig verallgemeinerbar, d. h. für jedes Projekt muss eigens ein Modell trainiert werden. Der Mangel an hochwertigen Trainingsdaten sowie Probleme im Zusammenhang mit der Interpretierbarkeit von Modellen stellen zusätzliche Hürden dar. Viele Deep-Learning-Modelle werden als Blackboxen benutzt, wodurch die Entscheidungsfindung nur schwer nachzuvollziehen ist.

Im Rahmen dieser Arbeit untersuchen wir die Herausforderungen, die mit auf maschinellem Lernen basierenden Methoden zur Entdeckung von Schwachstellen verbunden sind, und gehen diese an. Unser Fokus zielt auf vier entscheidende Dimensionen ab: Datenqualität, Modell-Interpretierbarkeit, Robustheit und Kontextsensitivität. Um dem Mangel an Daten zu begegnen, stellen wir neue, auf Code zugeschnittene Augmentationstechniken vor und steigern dadurch die Modellgenauigkeit. Außerdem kombinieren wir Erklärungsmethoden mit dynamischer Programmanalyse, um effektivere Vergleiche zu ermöglichen. Um die Robustheit der Erkennung zu erhöhen, setzen wir kausale Lerntechniken ein und senken somit die Modellbeeinträchtigung durch Störfaktoren bis zu 50%. Letztlich unterstützen wir die Fehlererkennung durch den Einsatz von Taint-Analysen und erweitern so den Kontext, ohne auf das Problem exponentiell wachsender Merkmalsräume zu stoßen und verbessern damit Erkennungsraten gegenüber traditionellen Schwachstellendetektoren.

Diese Thesis präsentiert Lösungen, um die Anwendbarkeit der lernbasierten Schwachstellenerkennung im produktiven Einsatz zu gewährleisten. Desweiteren liefert diese Arbeit einen umfassenden Überblick über die Herausforderungen auf diesem Gebiet und bietet Einblicke in die Zukunft der auf maschinellem Lernen basierenden Modelle zur Schwachstellenerkennung.

Abstract

The rapid proliferation of software has led to an increase in security threats, causing data breaches that have severe privacy implications and substantial financial consequences. As a result, software developers are under pressure to efficiently identify and mitigate vulnerabilities. One category of tools that has gained prominence in supporting developers in this regard is the field of machine learning-based software vulnerability detection, where models are trained to classify code as either vulnerable or clean. These models offer advantages over traditional static application testing tools, including adaptability to project-specific code and tunable decision boundaries. They have shown promising performance in vulnerability discovery, outperforming traditional static analysis techniques.

Despite their potential, machine learning-based vulnerability detectors face challenges. They exhibit low transferability and generalizability, meaning that models trained on one project may not perform well on another. The scarcity of high-quality training data, along with issues related to model interpretability, poses additional hurdles. Many deep learning models are used as black boxes, making it difficult for security practitioners to understand their reasoning.

Explainable AI (XAI) methods have been proposed to address the interpretability issue, allowing practitioners to gain insights into the model's decision process. However, these explanations can be noisy, and small changes in the input can lead to different results. Additionally, the choice of the right explanation method remains a challenge. The context-sensitivity of discovery models, or their ability to detect defects that span multiple modules or analyze code interprocedurally, also influences their detection capabilities.

In the scope of this thesis, we explore and tackle the challenges associated with machine learning-based vulnerability discovery methods. Our focus encompasses four crucial dimensions: data quality, model interpretability, robustness, and context sensitivity. To address the scarcity of data, we employ novel augmentation techniques specifically tailored to code, which helps to increase model accuracy. Furthermore, we integrate explanation methods with dynamic program analysis to enable more effective comparisons. In terms of enhancing detection robustness, we employ causal learning techniques to effectively reduce confounding effects by up to 50%. Finally, we bolster defect detection by leveraging taint analysis, thereby expanding the context without encountering the issue of exponentially increasing feature spaces and, most prominently, increasing the detection rate.

In this thesis, we propose solutions for learning-based vulnerability discovery to be more effectively applied in real-world scenarios. Finally, this work also provides a comprehensive overview of the challenges and advancements in the field, offering insights into the future of machine learning-based vulnerability discovery models.

Acknowledgements

I want to express my heartfelt gratitude to the following individuals whose unwavering support and encouragement have played an instrumental role in the completion of my doctoral thesis.

First, I want to thank Martin Härterich, my dedicated supervisor, for your patience and guidance not only through my professional but also through my personal challenges. Also, my respected professor, Konrad Rieck, deserves my utmost gratitude, his guidance and mentorship have been invaluable. Your expertise and willingness to share knowledge have not only shaped my academic and research path but also my personal growth. I don't want to miss thanking my colleagues, including Daniel Bernau, Anh Phan Duc, and my manager, Mathias Kohler, for their collaborative spirit and advice. I thank my students, including Erik Imgrund, Felix John, Inaam Ashraf, Konrad Hartwig, and Philipp Rall, for their trust, support, and collaboration. Of course, I would also like to thank my reviewers, Martin Johns and Lorenzo Cavallaro, for taking the time to read this work.

Moreover, I want to express my deepest gratitude to all my friends, including but not limited to Alexander Volker, Carlo (slayer of feedback-loop) Götz, Christian Quindt, Felix Kaufmann, Michael Galetzka, Philipp Münch, Robin Baumann, and Tim Heisterklaus, for encouraging me, pleasant distractions, and for proofreading this thesis.

Furthermore, I want to thank my family, especially my mother, Elvira Ganz, and my brother, Frieder Ganz, whose boundless support and belief in my abilities and continual encouragement have guided me through challenging times. Last but not least, my partner, Tamara Schütte, stood by my side through the long hours of research, offering constant love and understanding.

I extend my deepest appreciation to each of you for contributing to the successful completion of my doctoral thesis. Your collective presence has made this academic endeavor a truly rewarding and memorable experience.

Contents

1 Introduction

In recent times, there has been a remarkable surge in the abundance of software products, a phenomenon that has presented substantial security challenges [92]. Over the past decade, this increase in software usage has been accompanied by a series of data breaches [181], each of which has wide ramifications for users. Many security vulnerabilities in application software are exploited even years after disclosure [159], while at the same time, the number of security vulnerability exposures increases every year [183]. Reportedly, the number of disclosed vulnerabilities grew by more than 25% from 2021 to 2022. According to recent statistics from the USA, an average data breach costs around 9.48 million US dollars [182] and is therefore of significant financial interest for software companies [56]. There is increasing interest in tools and frameworks that assist software developers in identifying and mitigating risks. Tools that warn developers about potential vulnerabilities and bugs are valuable. However, since time is limited, these tools are required to be efficient and only add a minimum of manual labor [92].

One such category of tools is static program analysis [119, 149]. Static program analysis, as opposed to dynamic program analysis, describes the process of inferring properties of a program without actually executing it. Such techniques are commonly used for compiling [34], type inference [77], or program optimization [7]. A more important and critical application of these tools is the detection and localization of defects in software. Methods often referred to as static application security testing (SAST) tools digest the source code of a program and return a human-interpretable list of potential defects. SAST tools can be integrated into the software development life-cycle of a project to detect defects before they manifest themselves as security vulnerabilities after shipping. However, as promising as this sounds, these tools are bound by theoretical limitations by Rice's theorem, posing vulnerability discovery as an undecidable problem [99]. The consequences can be summarized in such that SAST tools may never be sound and complete at the same time. More specifically, not every reported finding may be an actual incident [68, 124, 13], while at the same time, not every defect can be detected [33]. Any static analysis tool is forced to either under- or over-approximate the runtime semantics of a program. While an over-approximation adds false positives, an under-approximation yields more false negatives. In the end, a manual triage is required to filter for true and false positives. This process requires manual labor and throws a spanner in the works of automatic

vulnerability detection [10]. Despite this theoretical bound, there are practical requirements, such as calculation speed, which must also be taken into account. Developers need SAST tools to deliver results within a few minutes at most [188].

As a remedy, research interest arose in the domain of machine learning-based defect detection, where the process of statically detecting flaws has been translated into a learning task [222]. Such models are trained in a supervised fashion on secure and vulnerable samples to classify unseen code during inference. Instead of relying on hand-crafted and hard-coded inference rules such as conventional SAST tools, they learn these rules from the data sets at hand [113]. Another advantage is the adaptation of project-specific code compared to rule-based SAST tools. Finally, learning-based models have a fuzzy and tuneable decision boundary. A decision threshold can be tuned to balance precision and recall, which can be directly interpreted as the balance between soundness and completeness. In controlled experiments, several of these learning-based methods reach a remarkable performance and outperform conventional techniques of static program analysis for vulnerability discovery [228, 113] measured by their detection rate while pertaining to a lower false positive rate. Machine learning-based tools differ in their architecture, dataset, and data preprocessing technique. For instance, recent models borrow techniques from natural language processing using recurrent neural networks (RNNs) or long short-term memorys (LSTMs), where the source code is processed as a flat sequence of code tokens [113, 112]. More recent approaches apply graph neural networks (GNNs), thereby leveraging code graphs as a compact structure to represent the syntactic and semantic properties of programs. Finally, the latest advances in large language transformer models have been successfully implemented to solve the vulnerability discovery task [55].

Unfortunately, there is a catch to it: Learning-based vulnerability detectors suffer from low transferability and generalizability [30, 27]. In this work, we will investigate several limitations responsible for the lack of practical adaptations of machine learning-based vulnerability discovery methods. We identify four significant shortcomings in the recent literature on learning-based SAST tools, which we categorize in four categories, namely, *the data*, *the interpretability*, *the robustness*, and *context sensitivity*.

First, models trained in one project cannot simply be applied to another project with the same initial reported performance. This may be due, for example, to unrealistic datasets [27], spurious correlation [6, 87], or label inaccuracies [197]. The vulnerability discovery task requires a large amount of high-quality data that is only insufficiently available. Previous works rely on pattern-generated data [113], for example, the software assurance reference dataset (SARD) [16] or the Juliet dataset [90, 27]. Others use static analyzers to generate ground truth [165]. As a remedy, we demonstrate a novel neural augmentation strategy to artificially generate new vulnerable samples from high-quality datasets to improve model generalizability. Using our augmentation approach, we can significantly increase the performance of vulnerability discovery models on unseen projects.

Moreover, models lack human interpretability since deep learning models are generally treated as black boxes [12, 191, 23]. Needless to say, vulnerability discovery is security critical; hence, it must be evident to practitioners why a model arrives at a particular conclusion [203]. Fortunately, Explainable AI (XAI) can be used to open up these black boxes [203, 57, 230]

and enable security practitioners to reason about a finding. For example, some models only classify bugs at the function level [228, 27, 31] without providing explanations. However, why a particular function is vulnerable is beyond their scope. There are several XAI methods that can be easily used to enhance their interpretability [69], though, since all differ, it remains challenging to choose the appropriate one. Some more recent vulnerability detectors even come with their own integrated explanation mechanism [55, 111]. It is hard to select and assess the proper explanation method for the vulnerability discovery task. Thus, this work presents a novel approach to compare XAI methods using dynamic program analysis as a validation oracle. We show that many explanation techniques reveal irrelevant information and impede practical applicability.

Third, learning-based models may be easily affected by noise or minor deviations from the input domain [161]. A piece of code may be correctly identified as vulnerable, but after some simple syntactic manipulations, the model can be deceived into classifying the semantic-equivalent function as unproblematic. This effectively hinders the transfer to other projects or unseen samples from other distributions [30] while it may also describe the decrease in performance over time due to concept drift. This work demonstrates a new evaluation scheme using causal learning to find potential sources of bias in the models, called *confounders*, and proposes countermeasures to alleviate these issues. We reveal that current state-of-the-art models have at least 30% decreased performance potential due to confounding effects.

Finally, the features must be carefully designed. A statement- or function-level vulnerability detector may never detect vulnerabilities that span multiple modules [227]. Just like regular static analyzers, learning-based analyzers may contain different levels of context sensitivity. Some methods only consider code metrics [134] or hashes of functions [95], some operate only intraprocedurally [113, 111] on token-level, some rely on flow-sensitive data preprocessing methods [27, 228] and others even incorporate call-sensitivity to some extend enabling the model to analyze interprocedurally [227, 112]. The context-sensitivity of the model directly influences the detection capabilities. As a first approach, we demonstrate a model that utilizes whole-program graphs and taint analysis to widen its context width without suffering from exploding feature spaces. Using this model, we achieve a 50% increased detection rate compared to other detection models.

Thesis Contributions This thesis proposes solutions to improve the applicability of learning-based vulnerability to real-world settings. The thesis condenses five peer-reviewed security conference publications. Note that the publications are not sorted by publication date. Following our empirical and experimental studies we propose enhancements to current literature to achieve a practical approach to *software defect localization using XAI* and provide the following concrete contributions:

1. *Neural Vulnerable Code Augmentation.* We analyze the shortcomings of current state-of-the-art datasets and enhance them with a novel neural code augmentation technique. The work is based on the publication "CodeGraphSMOTE - Data Augmentation for Vulnerability Discovery" [58]. The full paper can be found in Appendix C.

2. *Explainability of Vulnerability Discovery Models.* This thesis further assesses the interpretability of learning-based vulnerability discovery models and studies new

directions to the accurate comparison of explanation techniques based on the publications "Explaining Graph Neural Networks for Vulnerability Discovery" [57], which can be found in Appendix D and "Hunting for Truth: Analyzing Explanation Methods in Learning-based Vulnerability Discovery" [60] attached in Appendix E.

3. *Controlling Confounding Effects.* We present a novel evaluation scheme using causal learning to benchmark models with respect to their learned biases. This work is based explicitly on the publication "Broken Promises: Measuring Confounding Effects in Learning-based Vulnerability Discovery" [87]. The full paper can be found in Appendix F.

4. *Learning-based Taint Analysis.* Learning-based vulnerability discovery models lack context and do not fully grasp the semantics of code. As a novel approach, we apply classical taint-style analysis to learners based on the publication "PAVUDI: Patch-based Vulnerability Discovery using Machine Learning" [59]. We refer the reader to the full paper in Appendix G.

Structure of Thesis The rest of this thesis is structured as follows: Chapter 2 is split into two parts, starting with a theoretical introduction to software vulnerabilities and vulnerability discovery. The second part constitutes a recap of the related works. Chapter 3 then presents the main results of this thesis structured in the main issues of learning-based vulnerability discovery, each based upon the respective publications. We conclude in Chapter 4 with a summary and a future outlook.

2

Background

We start with an outline of the necessary preliminaries in Section 2.1 followed by a summary condensing the respective related work from the publications discussed in this thesis in Section 2.2.

2.1 Preliminaries

We proceed to discuss the theoretical foundation necessary for the rest of this work. We start with the notion of software vulnerabilities, rule-based vulnerability detection methods, code graph representations, and taint analysis. For a comprehensive overview of learning-based vulnerability discovery, we proceed with a brief description of machine and deep learning, classification performance metrics, and a description of Explainable AI.

Software Vulnerabilities

The National Institute of Standards and Technology (NIST) glossary describes a software vulnerability as *a security flaw, glitch, or weakness found in software code that an attacker could exploit [...] [180].*

According to this, a software vulnerability is a flaw in the program logic that has the potential to be leveraged by an attacker. Such a flaw, also denoted as *bug*, may emerge from poor design or mistakes in the implementation. A bug can have several manifestations on a program, depending on the software, use case, and the surrounding organization itself, thus, we focus on a set of the most basic required policies to ensure the security of software [49, p.199]. We can derive security policies leaning on the three information security principles, often referred to as *CIA* triad [93]:

1. *Confidentiality.* This principle ensures that sensitive information not meant to be disclosed by the system's design remains protected from unauthorized access. For example, a client should not have the capability to access the raw memory of a web server.

2. *Integrity.* Integrity pertains to safeguarding the integrity of a program's execution flow from manipulation by potential attackers. For instance, attacks involving remote or local code execution generally aim to compromise this aspect.

3. *Availability.* Availability underscores the importance of preventing malicious actions that could lead to the software's termination. Even a simple act like supplying malformed input that causes a program to panic and terminate should be guarded against.

Since not every code defect has to be a security vulnerability necessarily, we can derive three important criteria for a program that have to hold in order for it to be vulnerable given the NIST formulation:

1. *Attacker-Controlled.* This property denotes that an attacker must have the capability to exploit vulnerabilities through, for example, malformed or deliberately crafted input data. Consider scenarios where user inputs can be manipulated to compromise system security and violate security policies.

2. *Inadequate Validation.* In cases where user input is not validated correctly or sanitized, it creates the potential for security vulnerabilities to surface. Robust validation mechanisms should be in place to mitigate such issues. The absence of effective validation in the presence of a software defect is a critical concern.

3. *Critical Operation.* A security vulnerability is more likely to emerge when a sensitive operation exists within the system. To enable a user to violate a security policy, there must be a critical process or function that they can target. Identifying and safeguarding these critical operations is essential for strengthening the system's security [212].

From the opposite perspective, we can say a program is *secure* if it does not violate any of these three security policies [118] and a program is *safe* when it's free from flaws [105] or free from unacceptable risk as per *IEC 61508* [19]. It can be challenging to differentiate between a flaw that only affects the safety of a program and one that presents a threat if exploited by an adversary [61][49, p.7]. We briefly outline both, even if our focus is on the program's security.

IEC 61508: Functional safety of control systems defines Safety Integrity Levels (SILs) as a measure for the risk, taking into account the likelihood of a failure and the potential damage it could cause [19]. For low-risk services with low demand, SIL 1 is sufficient, which only requires basic functionality testing, for instance, per *ISO 9001* [22, 101]. The specification dictates that higher-risk services conforming to SIL 2 require extensive testing, and SIL 3 and 4 even require formal verification methods. It is noteworthy that static analyzers are recommended for use from SIL 2 upwards in terms of defensive programming to prevent adversary effects, for instance, input validation [118]. Similarly, *ISO 15408* [17] titled *Common Criteria for Information Technology Security Evaluation* defines Evaluation Assurance Level (EAL) as a security-focused and risk-based system assessment. More confidence in a software's security posture requires a higher EAL level and more rigorous testing. In the rest of this thesis, we refer to a program that is free from exploitable flaws as *clean*.

Lastly, there are several classifications of defects. However, this thesis considers the most popular common weakness enumeration (CWE) [138]. Common weaknesses are mapped to identifiers. For instance, a stack-based buffer overflow corresponds to *CWE-121*, while a

format string vulnerability to *CWE-134*. Disclosed vulnerabilities from the past are instances of CWEs and labeled by a unique common vulnerability enumeration (CVE) number. The famous heartbleed vulnerability, for example, has the identifier *CVE-2014-0160* but belongs to *CWE-130: Improper Handling of Length Parameter Inconsistency*. The risk of a vulnerability can be calculated using the Common Weakness Scoring System (CWSS) score, which considers the environment, for instance, the business impact and the likelihood of exploitation, the attack surface, for example, the required attacker's privilege, and the attack vector, and the base finding, which includes the severity by its technical impact. Finally, we formally define a security vulnerability for the rest of this thesis in Definition 1.

Definition 1 *A security vulnerability is a software defect that enables an adversary to violate a security goal, such as confidentiality, integrity, or availability, through a specific input.*

Rule-based Vulnerability Discovery

Static program analysis is the process of analyzing the run-time behavior of a program without executing it. More formally, it makes it possible to infer whether a property b of a program $p \in \mathcal{P}$ holds, denoted as $p \vDash b$ [149]. The task of static vulnerability discovery can be consequently formulated as in Definition 2.

Definition 2 *A method for* ***static vulnerability discovery*** *is a decision function* $f : \mathcal{P} \mapsto \{0, 1\}$ *returning* 1 *if* $p \in \mathcal{P}$ *is* VULNERABLE*, implying* $p \vDash b$ *or* 0 *if it is* CLEAN *otherwise. In this scenario, property* b *refers to a defect being present in program* p*.*

There is already a vast number of SAST tools available to support developers in their daily work to find and disclose defects, all representing an instance of this decision function [60]. The conventional SAST tools arrive at such a function f by applying handcrafted rules to the source or binary representations. Most rules are project-agnostic; however, in software development companies, it is expected to elaborately design domain-specific detection and linting rules [73, 221]. In conclusion, all these tools may differ in their analysis technique, context, and implementation [34]. There are several properties of a static analysis framework introducing different computational complexities:

1. *Flow sensitivity.* If an analysis is flow-insensitive, the program is considered an unordered set of statements. If an analysis is flow-sensitive, it accounts for the flow of control of the program.

2. *Context sensitivity.* If an analysis is context-insensitive, the analysis yields the same results for all possible invocations. Otherwise, the call context, arguments, and return values are also considered.

3. *Interprocedural/intraprocedural.* If an analysis is intraprocedural, only a single function is analyzed in isolation. If an analysis is interprocedural, the whole program is analyzed.

In addition, we can categorize SAST tools into three coarse-grained categories, classifying their detection capabilities. We stick to a brief overview of currently popular SAST tools that are freely available.

1. *Lexical analyzers.* Lexical analyzers are deterministic finite automata. For instance, Flawfinder [206] is a popular open-source SAST tool using a set of C function names[1] expressed by regular expressions. These rules are then matched against symbols such as `strcpy` or `strcat`. Flawfinder consequently suggests replacing them with secure alternatives, for example, `strlcpy` or `strncat`, respectively.

2. *Syntactical analyzers.* Syntactical analyzers implement abstract syntax tree traversal strategies. Adding the capability of, for example, finding insecure argument bugs [211]. PMD [156], or Cppcheck [132] are SAST tools that rely on abstract syntax tree (AST) traversals and have no or minimal flow-sensitive scanning capabilities [34].

3. *Semantical analyzers.* Semantical analyzers incorporate context-sensitive detection rules. These tools often define operations over broader context [34]. Tools may be even field-sensitive or type-sensitive like Infer [152], flow-sensitive like CodeQL [171], or path-sensitive like the Clang analyzer [123].

Both lexical and syntactical analyzers have a higher false positive rate compared to semantical analyzers [100, 119]. One reason is that Flawfinder and Cppcheck over-approximate the program semantics: For example, an unreachable but insecure C function already triggers a false positive. We would like to refer the reader to Appendix A for a brief SAST tool comparison. Although SAST tools may support flow-sensitive and interprocedural analysis, detection capabilities depend on the defined rules at hand. For example, Joern [211] is a semantical analyzer with interprocedural analysis capabilities, however, its C/C++ detection rules are all intraprocedural.

From a theoretical point of view, there is a limitation that makes both rule-based and learning-based static analysis in general an undecidable problem. Consider again $f(p)$ as a function computing whether $p \vDash b$ with b being the property of *software defect is present*. f is sound if $f(p)$ implies $p \vDash b$ and f is complete if $\neg f(p)$ implies $p \vDash \neg b$. Since this is a decider over the semantics of $p \in \mathcal{P}$, its computation can be equivalently formulated as determining whether the program will terminate. This is generally known as the halting problem and is proven to be undecidable [99][208, p.337]. For further details, please refer to the proof provided in Appendix A.

Thus, we have to over- or under-approximate the program's runtime semantics, which has consequences for the soundness and completeness of the static analyzer. Over-approximation can report more findings, striving for a complete analysis, while under-approximation approximates a sound analyzer with fewer false positives.

[1] I.e. https://github.com/david-a-wheeler/flawfinder/blob/master/flawfinder.py#L1082

```
void flawed_strdup(const char *input) {
char *copy;
/* Fail to allocate space for terminating '\0' */
copy = (char *)malloc( strlen (input));
strcpy(copy, input);
return copy;
}
```

Figure 2.1: *CWE-131*: Null termination string weakness.

Figure 2.1 shows a software defect caused by a string allocation that does not account for null termination, resulting in a buffer overflow. Flawfinder's regular expression for finding incorrect calculations of the buffer size for strings `*strcpy(*`, namely *CWE-131*, may undoubtedly detect the flawed string copy operation in Figure 2.1 simply due to the occurrence of the function call. Unfortunately, it will also report legitimate sanitized occurrences.

```
exists ( StrlenCall strlen | DataFlow::localExprFlow( strlen , malloc.getSizeExpr())) and
exists (ArrayFunction af, FunctionCall fc, int arg |
DataFlow::localExprFlow(malloc, fc.getArgument(arg)) and
fc.getTarget() = af and
(
  af.hasArrayWithNullTerminator(arg)
  or
  af.hasArrayWithUnknownSize(arg)
  or
  formatArgumentMustBeNullTerminated(fc, fc.getArgument(arg))
)
```

Figure 2.2: CodeQL query to find flawed string buffer calculations.

Suppose we consider the rule by CodeQL, a context-sensitive SAST tool. In that case, we notice a more involved detection rule in Figure 2.2, which not only checks for a call to `strlen` but also whether it flows into a function that implies null-terminated strings. However, the rule only applies intraprocedurally, within a single function, and may not catch, for example, wrapper functions around `malloc`.

```
val allocations = cpg.method(".*malloc$").callIn.argument(1)
cpg
  .method("(?i)strncpy")
  .callIn
  .map { c =>
    (c.method, c.argument(1), c.argument(3))
  }
  .filter { case (method, dst, size) =>
    dst.reachableBy( allocations ).codeExact(size.code).nonEmpty &&
    method.assignment
      .where(_.target.arrayAccess.code(s"${dst.code}.*\\[.*"))
```

Figure 2.3: Joern query to find flawed string buffer calculations.

We can find a similar rule by Joern in its C bug detection engine depicted in Figure 2.3, which considers `malloc` as the source but only `strncpy` as a sink.

```
1 for (const Token* tok = scope->bodyStart->next(); tok != scope->bodyEnd; tok = tok->next()) {
2  [...]
3     const Token* varTok = tok->astOperand1();
4     const Token* litTok = tok->astOperand2();
5      [...]
6     } else if (litTok->tokType() == Token::eChar && varType && varType->pointer) {
7         suspiciousStringCompareError_char(tok, varTok->expressionString());
8     }
9 }
```

Figure 2.4: Cppcheck query to find flawed string comparisons.

On the other hand, tools such as CPPcheck are not capable of performing such a flow analysis, they rather operate on the AST. Thus, there is no specific rule for *CWE-131*. CPPcheck can be used to find incorrect string comparisons that frequently occur, for instance, `str == '\0'` vs `*str == '\0'`. An excerpt of the rule can be seen in Figure 2.4 where two operands in the AST are being compared.

Code Graphs

We have seen that the syntactical SAST tools describe traversals over the AST and semantical analyzers over flow graphs. Hence, it is advantageous to model programs as directed graphs [3, 211, 14]. Recent learning-based approaches also make use of such graph representations [228, 31, 197] for source code rather than utilizing flat token sequences [165, 113]. We refer to the resulting representations as *code graphs* and denote the underlying directed graphs as $G = G(V, E)$ with vertices V and edges $E \subseteq V \times V$. Moreover, the nodes and edges are attributed, that is, elements of V or E are assigned properties. However, different code graphs capture various syntactic and semantic features [59]. Recent work, for instance, relies only on syntactic features for neural code comprehension using the AST [3]. This is a tree representing the syntactic structure of the source code of a program $p \in \mathcal{P}$ as defined in Definition 3.

Definition 3 *The abstract syntax tree (AST) is the result of parsing the source code of a function such that the leaf nodes in the resulting tree $G_A = G(V_A, E_A)$ are literals and the edges E_A describe the composition of syntactic elements [2, p.8, p.58].*

The semantic attributes of a function can be captured in flow graphs, for instance, with the flow of control as in Definition 4 or the flow in information defined as in Definition 5.

Definition 4 *The control flow graph (CFG) within a function is $G_C = G(V_C, E_C)$ with the nodes $V_C \subset V_A$ being statements, and where the directed edges E_C describe the execution order of the statements $V_C \subset V_A$ [211, 34].*

Definition 5 *The data flow graph (DFG) within a function is $G_D = G(V_D, E_D)$ with the nodes $V_D \subset V_A$ being variable assignments and references, and where the directed edges E_D describe read or write access from or, respectively, to a variable [26].*

These graph representations allow us to reason about the order of the executed statements and the flow of information between variables. An analysis using these properties is considered flow-sensitive [149, p.5]. Finally, a call graph connects the call sites of the functions with the definitions of the functions as defined in Definition 6.

Definition 6 *The call graph (CG) within a program is defined as* $G_{CG} = G(V_{CG}, E_{CG})$ *where the nodes* $V_{CG} \subset V_A$ *are the call sites and definitions of the functions, while the edges* E_{CG} *connect the caller with the respective definition of the function.*

An analysis using the call context of a program is considered context-sensitive [149]. Code graphs capture the semantic and syntactic relationships between statements and expressions in programs. Based on these classical representations, combined graphs have been developed for vulnerability discovery. A popular one is the code property graph (CPG) by Yamaguchi et al. [211], which is a combination of the AST, CFG, and program dependence graph [52]. Other approaches use different combinations, for instance, combining the AST with the CFG and the DFG [26] is called code composite graph (CCG) as defined in Definition 7.

Definition 7 *The code composite graph (CCG) is a disconnected graph* G_{CCG} *for a program consisting of multiple functions* $p = \{p_1, p_2, .., p_n\}$ *with* $V = \bigcup_{i=1}^{n} V_A^i$ *and* $E = E_A \cup E_D \cup E_C$ *combining the AST with the semantic information from the CFG and DFG.*

This thesis uses the CCG, since the components of a CCG are easily obtained during compilation and capture syntactic features and information flow.

Static Taint Analysis

Classic taint analysis is originally a dynamic program analysis approach where particular statements or expressions are marked and monitored at run-time [169]. This analysis allows security practitioners to identify, for instance, potential attacker-controlled sources flowing into critical program regions. Yamaguchi et al. [211] define an over-approximate static approach by tainting program parts and statically propagating tainted values along the control and data flow as depicted in Definition 8.

Definition 8 *A taint-style analysis for vulnerability detection is a 3-tuple* $(V_{SOURCE}, V_{SAN}, V_{SINK})$ *consisting of the nodes in the code graph of a program* $p \in \mathcal{P}$ *denoting the taint source, sink and sanitizer nodes as depicted from* V_A *[211].*

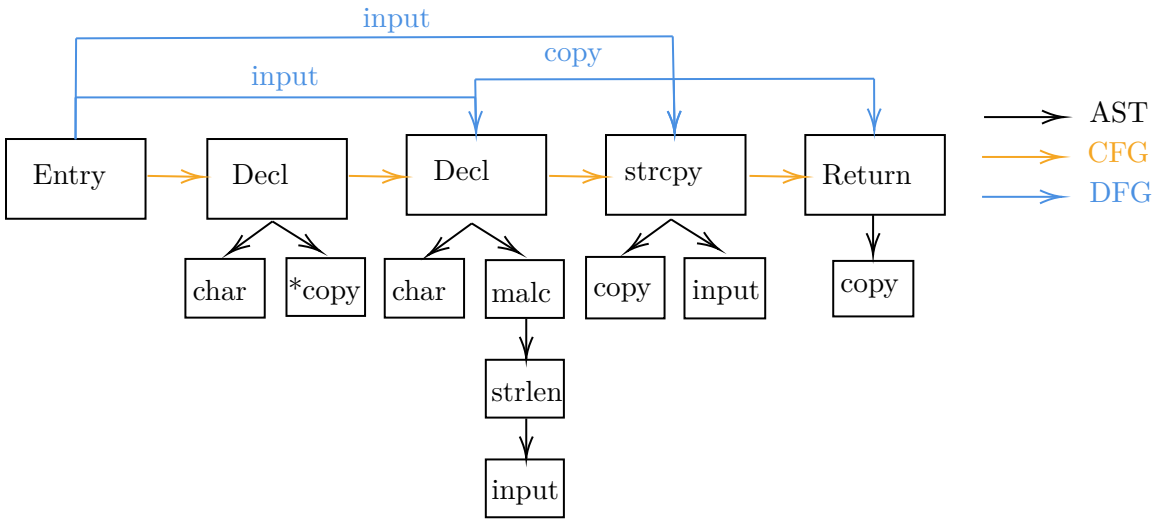


Figure 2.5: Simplified code composite graph [26] from Figure 2.1.

Figure 2.5 shows a code graph for the code sample from Figure 2.1. Such a property graph can be stored in graph databases and queried using well-formed traversals [162]. For instance, we can define traversals as mappings from one powerset of nodes to another:

$$\mathcal{T} : V^* \to V^*$$

Given two traversals, we can combine them by composition: $\mathcal{T} \circ \mathcal{T}$. New traversals can be defined by composing primitive traversal operations. For example the primitive operation *Filter* selects all nodes X that fulfill a predicate b:

$$\text{FILTER}_b(X) = \{v \in X : b(v)\}$$

Furthermore, neighboring outgoing nodes can be retrieved by the following operation:

$$\text{OUT}(X) = \bigcup_{v \in X} \{u : (v, u) \in E\}$$

Searching, for example, for user-controlled format string vulnerabilities can then be expressed using a single traversal finding user input that flows into variable declarations ending in formatted string outputs.

$$\text{FILTER}_{printf} \circ \text{OUT} \circ \text{FILTER}_{decl} \circ \text{OUT} \circ \text{FILTER}_{cin}$$

In practice, these traversals are implemented using graph database queries or implementation-specific APIs, as we have seen in Figure 2.2 and Figure 2.3. One recent idea of learning-based static analyzers relying on graph neural networks is that these models learn such inference rules automatically [228]. We will revisit taint analysis in Section 3.4.

Learning-based Vulnerability Discovery

This chapter introduces the concept of learning-based vulnerability discovery. First, we outline supervised machine learning, which is the process of learning a task given labeled data. Then, since most models rely on deep learning, we move on to introducing the concept of deep neural networks, optimization strategies, different deep learning architectures, classification tasks, performance metrics used throughout this thesis, and finally, explainable AI.

Supervised Machine Learning

Machine learning (ML) is a discipline arising from computer science and statistics. ML intends to gain new knowledge by finding recurring patterns in datasets without or with only minimal use of human interaction [137, p.2][166]. Machine Learning is mainly used for complex data and, therefore, replaces traditional statistical methods. Most often, problems requiring the use of machine learning (ML) are separated into classification tasks where a model needs to classify a given data point to a given class or perform regression tasks where a model predicts numerical values [104]. Supervised machine learning is the process of approximating an unknown or complex function using a model estimate [63, p.139]. It typically requires a dataset $D = \{x_0, \ldots, x_{|D|}\}$ with $x \in \mathbb{R}^F$.

F and $|D|$ denote the number of features and the number of samples in the dataset, respectively. Furthermore, there is a set of labels $\vec{y} = \{y_0 \dots y_{|D|}\}$ typically denoted as $y \subseteq [0,1]^{|D|}$ in a binary classification task. The goal of supervised machine learning is to find a mapping $f_\theta : x \to P(y \mid x)$ where θ is some form of optimizable parameter set. In the rest of this thesis, we assume, without loss of generality, the label $y = 0$ corresponds to a *clean* sample, while $y = 1$ denotes a *vulnerable* sample.

Deep Neural Networks

There are several deep neural network architectures, with the simplest one being a multilayer perceptron, sometimes referred to as a feed-forward network, which can be seen as a universal function approximator [36]. Three components can summarize the architecture of such a neural network: the neurons (units), the layers (set of nodes), and the weights as the connection between the neurons. Every layer consists of a configured amount of neurons [63, p.196]. All *Deep Neural Networks* (DNN) have multiple hidden layers and the ability to learn complex data representations in common. Each hidden layer retains a specific abstraction of representations from a given dataset. The input layer denotes the dimension of a data point, and the output layer denotes the dimension of the desired prediction vector. The output of one layer is forwarded to the input of the next layer. Estimating the weights really just corresponds to estimating a function [83].

Every layer consists of a fixed amount of neurons as part of their configuration.

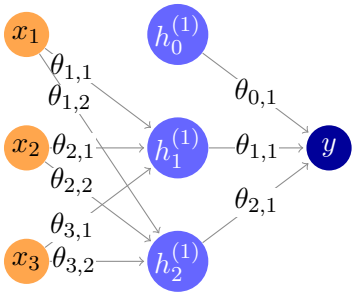


Figure 2.6: Multi-layer perceptron layer configuration.

Consider a multilayer perceptron (MLP) as in Figure 2.6 and let $h_j^{(i)}$ denote the j-th neuron of the i-th layer. Every node $h_j^{(i)}$ is connected to every node $h_k^{(i+1)}$ by an edge with weight $\theta_{j,k}^{(i)} \in \mathbb{R}$. The activation value for a specific node is given by:

$$h_k^{(i+1)} = \sigma\left(\sum_{j=0}^{n} \theta_{j,k}^{(i)} \cdot h_j^{(i)}\right)$$

A layer $h(x)$ incorporates a non-linear activation function, for instance, *rectifying linear unit* (ReLu) or *tangens hyperbolicus* (tanh). An activation function $\sigma(x)$ is used to add non-linear properties to the learning ability of a model. Given a data point x, typically denoted as $D = \{x_0 \dots x_m\}$, x is passed through the first layer, in other words: $h^{(0)} = x$. Using a vector notation, we can come up with a recursive definition $h^{i+1} = \sigma(h^i\theta)$ with $i > 0$ [128]. Suppose we abstract the calculation rule between the weights and the previous activation values. In that case, we obtain $h_i = \sigma(\text{CALCULATE}(h^{i-1}, \theta))$ where in $\text{CALCULATE}(\cdot,\cdot)$ of an

MLP would correspond to a matrix multiplication. Different neural network architectures have different calculation rules.

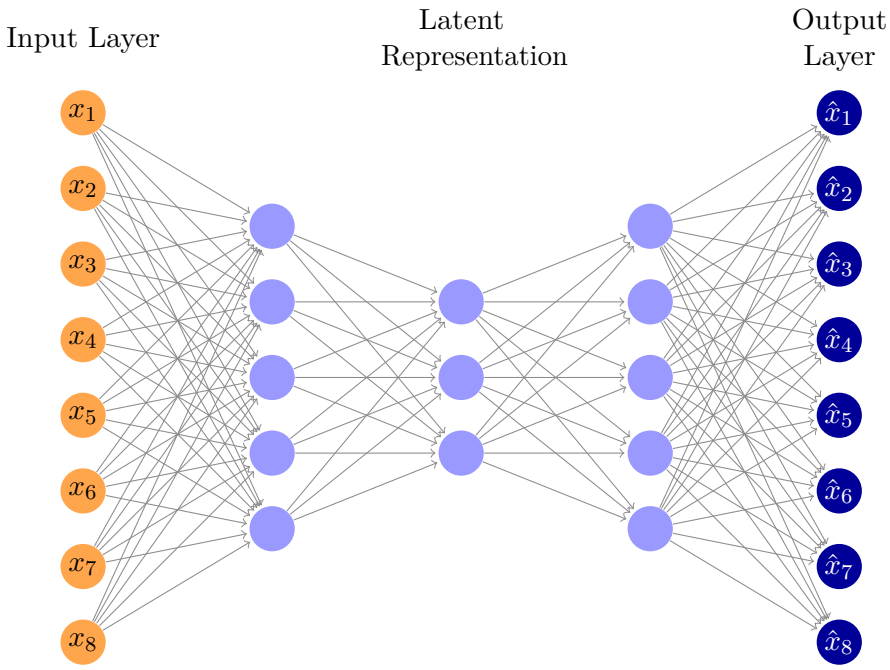


Figure 2.7: Neural autoencoder architecture.

Several layer and neuron configurations have been proposed [104] for different purposes. For instance, the architecture from Figure 2.7 depicts an under-complete autoencoder that can be trained to reconstruct the input [63, p.504]. The left part of the dense hidden layer is the encoder $E(\cdot)$ and the right part is the decoder $D(\cdot)$ the combined application to the input $\hat{x} = D(E(x))$ is the approximated reconstruction. The compressed, dense latent space should represent interesting properties [63, p.505]. We will revisit a similar architecture in Section 3.1.

Neural Network Architectures

Other architectures build on the idea of MLPs. In this thesis, different architectures are drawn with different calculation rules. This thesis has a particular emphasis on graph neural networks since they neatly connect with the concept of code graphs.

Graph Neural Networks These networks are a class of deep learning models realizing a function $f\colon G(V,E) \mapsto y \in \mathbb{R}$ that can be used for the classification of graph-structured data [167]. A graph convolutional network needs two mandatory input parameters, that is, an initial feature vector $x \in \mathbb{R}^{N\times F}$, with N being the number of nodes in the graph and F the number of features per node, and the topology commonly described by the adjacency matrix $A \in [0,1]^{N\times N}$. Furthermore, we denote by $\mathcal{N}(v)$ the neighboring nodes of any $v \in V$ [71, 194].

One of the simplest graph convolutional networks (GCNs) is the message passing network defined by Kipf and Welling [97]:

$$h^{(l)} = \sigma(\hat{A}\, h^{(l-1)}\theta^{(l-1)}) \tag{2.1}$$

with $h^0 = x$ and $\theta \in \mathbb{R}^{F^{(l-1)} \times F^{(l)}}$ with $F^{(l-1)}$ and $F^{(l)}$ being the previous and next layer's feature dimension. Here, the intermediate representations are linearly projected and sum-wise aggregated according to the normalized adjacency matrix $\hat{A}$ with self-loops followed by a non-linear activation function. This GCN can be stacked to learn filters with respect to larger neighborhoods. Other GCN layers use different aggregation and update mechanisms, for instance, instead of an MLP, gated graph neural networks (GGNNs) use gated recurrent units (GRUs) to update the hidden state of nodes [109], while graph attention network (GAT) layers use attention mechanisms [194]. Due to the fitting premise of GCNs, they have been widely adopted for representation learning on code graphs [228].

Transformer Models A transformer model typically consists of either an encoder, a decoder, or both [115]. Each part is further composed of multiple blocks consisting of bidirectional multi-head self-attention mechanisms and MLPs [193]. Compared to RNNs, transformer models do not rely on computation rules with recurrences. Instead, attention matrices produce an attention vector for each token, denoting the influence of each other token in the sequence [55].

$$\textsc{Attention}(Q, K, V) = \textsc{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}V\right) \tag{2.2}$$

The matrix $Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$ denotes a query containing the set of representations for the current tokens, which is then multiplied with the key matrix $K \in \mathbb{R}^{d_{\text{model}} \times d_k}$. The result is scaled by the inverse square root of the size of the embedding d_k and finally, after a softmax, used as an index to the value matrix $V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ that produces the attention vector. d_{model} denotes the size of the vocabulary, and the Q, K, and V matrices can be split into multiple attention heads to capture richer semantics, while d_k and d_v constitute hyperparameters.

Optimization

Since it is often intractable to find a θ directly, such that a function $f_\theta : x \to y$, we seek to find a model θ from a potentially infinite hypothesis space Ω to minimize a cost function:

$$J : \Omega \to \mathbb{R}$$

Gradient Descent (GD) is a first-order optimization algorithm that can be used to find local minima (the steepest descent) in a differentiable function [185]. In this context, gradient descent is used to fit the parameters of a neural network to a suiting solution by minimizing a loss function. The algorithm calculates the partial derivatives of a loss function concerning its weights $\frac{\delta J}{\delta \theta}$. Gradient descent differentiates, with respect to each parameter, the loss function J and proceeds to calculate the error for each sample $x \in D$ and, in case of a neural network, propagates the changes through every layer. The calculated gradients are then used to take an update in each direction. The calculated gradients are used to update the weights by:

$$\theta_{new} = \theta_{old} - \eta \nabla J(f(x; \theta_{old}, y)$$

More advanced optimization algorithms like ADAM and AdaGrad are commonly used in practice, but will not be discussed for brevity [63, p.308]. The step size and, therefore, the

convergence speed are controlled by a hyperparameter η. Eventually, the model and its configuration θ should encode the task to find vulnerabilities.

Classification

The output of the neural network prediction is given by consecutively passing the neurons' values to the next layer by calculating the activation values of the neurons with the particular calculation rule. To obtain a probability vector, typically, the output vector is normalized by applying the *softmax* or *sigmoid* function [63, p.184].

In conclusion, we can start with the Definition 2 and use a probabilistic output and a parametrizable weight vector θ given some code representation of $p \in \mathcal{P}$ to obtain static vulnerability discovery based on machine learning techniques as defined in Definition 9. In the end, the probability can be translated to a final binary output CLEAN or VULNERABLE using a decision threshold typically in the range of $t \in [0, 1]$. A t closer to 1 accounts for f_θ being more complete and a t closer to 0 for a f_θ being more sound.

Definition 9 *A method for* ***static learning-based vulnerability discovery*** *is a parametrized hypothesis function* $f_\theta \colon \mathcal{P} \mapsto [0, 1]$ *that extracts a representation* x *of a program* $p \in \mathcal{P}$ *and maps* p *to a probability* $P(\textsc{vulnerable} \,|\, x)$*.* p *is clean if the probability is smaller than a threshold* t *or vulnerable if it is greater or equal to a threshold* t*.*

Performance Metrics

Within this thesis, we stick to performance metrics for machine learning classification tasks since vulnerability discovery is essentially a binary classification task. We consider true positives (TPs) represented by samples that were labeled correctly as VULNERABLE and true negatives (TNs) as records that were correctly labeled as CLEAN. false positives (FPs) and false negatives (FNs) are represented by samples that were incorrectly classified as VULNERABLE or CLEAN respectively. We can further calculate their respective rates by dividing them by the total number of samples of either class in the test set, yielding true positive rate (TPR), true negative rate (TNR), false positive rate (FPR), and false negative rate (FNR). To achieve an unbiased view of the performance of the models, we proceed to define four metrics to measure the models on their test sets [6].

Definition 10 *We use the balanced accuracy as a replacement for the classical accuracy score to account for imbalanced datasets. This metric yields a prevalence-agnostic score to measure the detection quality of a classifier.*

$$\textsc{Balanced Accuracy} = \frac{TPR + TNR}{2}$$

Definition 11 *The precision measures the number of correctly identified defects with respect to the wrongly classified ones. It, therefore, measures the soundness of the SAST tool.*

$$\textsc{Precision} = \frac{TP}{TP + FP}$$

Definition 12 *The recall measures the amount of correctly classified defects in relation to all defects in the dataset. It, therefore, measures the completeness of the SAST tool.*

$$\textsc{Recall} = \frac{TP}{TP + FN}$$

Definition 13 *The F1-score is the harmonic mean between the precision and recall, giving a single numeric score for the balance between soundness and completeness of the SAST tool.*

$$F1 = 2 \cdot \frac{\textsc{Precision} * \textsc{Recall}}{\textsc{Precision} + \textsc{Recall}}$$

Furthermore, we calculate the area under the receiver operating characteristic curve, which is a chart that visualizes the trade-off between TPR and FPR for different decision thresholds. Balanced accuracy, F1-Score, and area under receiver operating characteristic (ROC) curve are especially recommended when handling heavily imbalanced datasets as opposed to, for example, the accuracy score [6].

Explainable AI

Molnar [140, p.6] refers to explainability as the models and methods that make machine learning predictions understandable to humans. While Rudin et al. [164] define interpretability as machine learning models that obey domain-specific rules to make their decisions more easily understood by humans. It is noteworthy that there is a lack of agreement between the two notions *explainability* and *interpretability* in recent research [54].

Definition 14 *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.*

In this work, we specifically leverage explanation methods (EMs) for machine learning to alleviate the problem of opaque decisions and to verify learning-based security systems. Given a vulnerability discovery method f, we formalize the explanation methods as producing heat maps $\mathcal{M}$ from pairs of source code x and the predicted output $\hat{y} = f_\theta(x)$. *Heatmaps* (or interchangeably *explanations*) attribute numerical relevance scores to locations in the code, that is, to nodes and edges, if f is a GNN.

2.2 Related Work

This work provides novel directions for the discovery of static vulnerabilities with explainable AI and dynamic program analysis. As a result, there exist different prior works tangent to this thesis. We proceed with an assembly and discussion of the related works of the publications from the appendix.

Program Analysis

We have already discussed static program analysis from a security testing perspective and now proceed to address other categories of program analysis, starting with dynamic program analysis with a particular emphasis on works with a security focus.

Dynamic Analysis Dynamic program analysis, as opposed to static analysis, is not limited to the source code but also includes the analysis of the runtime behavior of a program. Compared to static program analysis, dynamic analysis tends to be more precise, but a single analysis report only holds for one particular analyzed execution [64].

Generally, the dynamic analysis consists of two parts, the instrumentation and the profiling of an application, to monitor the runtime behavior [139]. Tools like Valgrind [145] and Purify [74] both instrument an executable and monitor its behavior. Instrumentation techniques may track undefined bits in registers [173], taint values in memory [146], or, for example, store type information to variables based on their operations [24]. Other approaches may include the automatic analysis of runtime logs [76].

One technique that lies in this category is fuzzing. Since this work uses directed fuzzers in Section 3.2 to compare explanation methods, we briefly discuss popular approaches. Common coverage-guided fuzzers like AFL or Libfuzz generate test inputs to observe crashes or hangs in a program under test (PUT) [53, 172, 131]. Directed fuzzers also try to reach certain execution points during runtime. For example, AFLGo [18] and Hawkeye [29] model targeted input generation as a power-schedule problem. Beacon [84] tries to incorporate path pruning into the seed selection process and, as a result of this, accelerates crash reproduction compared to AFLGo and Hawkeye. Targetfuzz [25] prioritizes the initial seed selection to speed up directed fuzzing. Other works focus on improving the instrumentation of gray-box fuzzers by heuristically extracting potentially interesting code regions [153].

Zhu et al. use explanation methods in conjunction with an NLP-based vulnerability-detecting model to speed up AFLGo. V-fuzz also speeds up fuzzing with learning techniques: It uses a neural network to detect likely vulnerable spots in binary programs [110]. Similarly, we use fuzzing to compare EMs in Section 3.2.

Symbolic Analysis Symbolic program analysis can be used to detect violations of safety and security properties [9]. Symbolic execution can explore a program's execution paths considering several potential inputs. For instance, it can be used to reason about buffer or null checks or to determine domain invariants [114].

To such symbolic program analysis techniques belong the abstract interpretation [4] and symbolic execution [157]. Here, the program is partially interpreted, while symbolic representatives may replace specific library calls or user inputs. This can be useful for value set analysis or test generation. SymCC [157] is a symbolic execution framework that relies on compiled code and uses a Satisfiability Modulo Theorem (SMT) solver. Otter [129], on the other hand, is based solely on the source code. Ma et al. [129] propose directed symbolic execution to reach specific points during runtime analysis to, for example, guide fuzzers.

Static Analysis Static program analysis reasons about properties of a program without executing it [143, 100]. There are static analysis applications for particular focuses, for instance, finding liveness bugs and safety violations in distributed systems [94], dependency of ordering bugs in smart contracts [62, 51]. Shen et al. [174] compares SAST tools for embedded systems, while Kinder and Veith [96] proposes static analysis on disassembled binaries. The fundamental problems of static analysis include deciding the reachability of certain instructions, the deduction of data types [77], the liveliness of variables, dead code, the avoidance of superfluous calculations [204], the reduction of the likelihood of unintended behavior, or the reduction of redundant calculations [149, p.1].

Recent works propose new algorithms that better exploit parallelism for faster analysis [133], support incremental analysis, or larger context-width [189]. Interprocedural pointer analysis yields significant improvements for static taint analysis [127], while incremental analysis speeds up developer feedback [187]. We apply similar ideas to learning-based vulnerability detectors, as for instance, methods to utilize broader context and incremental analysis as in Section 3.4.

SAST tools are still underused and have an average false positive rate of 35% to 93%, while the manual triage is time-consuming and costly [143]. There is an entire research field dedicated to improving static analyzer precision within theoretical boundaries [68, 124, 13]. This work focuses on improving learning-based SAST tools.

Code Metric Correlation The first successes that have been achieved by using heuristics to detect vulnerabilities in code bases are based on the correlation of software quality metrics, for instance, *code size*, *code churn*, *developer activity*, or *code complexity* [175]. Meneely and Williams [135] use Bayesian networks to predict software defects based on a set of quality metrics. Walden and Doyle [196] propose to rank software vulnerability indicator scores to predict new vulnerabilities. Perl et al. [155] take a different path and identify commits that are likely to introduce vulnerabilities based on a support vector machine. The works suggest a low correlation between the metrics and actual vulnerabilities [196], and some works report disappointing detection rates paired with high false positive rates [135].

Anomaly Detection Another set of interesting works treats vulnerability discovery as an anomaly detection task. Chang, Podgurski, and Yang [28] use program mining techniques to identify neglected conditional rules. Similarly, Yamaguchi et al. [214] find missing sanitization checks by combining taint analysis and bag-of-words techniques to calculate an anomaly score. These approaches are either only applicable to large software projects without the capability to transfer knowledge from others or suffer from high false positive rates [28].

Vulnerable Clone Detection Some works redefine vulnerability discovery in terms of finding vulnerable code clones, which often includes finding recurring patterns or functions that were known to be vulnerable. For instance, Yamaguchi, Lindner, and Rieck [213] embed symbols to vectors using principal component analysis and search for similar functions. Kim et al. [95] similarly define a vulnerable code clone detection mechanism based on hashed function abstractions. These methods trade off soundness for completeness and thus have a low detection rate in performance benchmarks [111].

Vulnerable Pattern Recognition Several past works already target the problem of automatically discovering vulnerabilities and defects using loosely defined rules (heuristics) or machine learning [222]. One of the first works to use supervised machine learning for vulnerability discovery is Wang, Liu, and Tan [198] from 2016. They extract feature vectors from the AST. More works followed, for instance, *Draper* by Russell et al. [165] using flat token sequences and a convolutional network and *VulDeePecker* by Li et al. [113] who first define code gadgets as multiple code lines depicting vulnerable control or data dependencies. Similarly, Grieco et al. [67] use Word2Vec representations from the AST and shallow learners.

This work has a slight focus on graph neural networks since they currently provide the most promising results [136]. The combination of GNNs and code graphs, considered in our work, has been proven to be successful in the discovery of defects and security vulnerabilities in a series of research [197, 228, 26, 31, 80, 46, 227]. For example, Zhou et al. [228] introduce the first gated graph neural network on code property graphs to identify defects and vulnerabilities collected from real-world commits. Their approach outperforms popular open-source and commercial rule-based static analyzers as well as token-based learning models. Cao et al. [26] 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 composite code graph. Other works also rely on different graph representations [37].

With recent advances in large language models, transformer models with several billion parameters have been fine-tuned for vulnerability discovery. They already show extreme success for natural language [115], and it can be assumed that we are able to achieve similar success in representation learning on code [81]. Recent large language model (LLM)-based vulnerability detectors achieve astonishing performance results according to their respective publications [55, 30, 190]. Several different architectures and pre-trained large code models have been used for this purpose, like Bert [43], RoBerta [122] or CodeT5 [201, 202].

XAI in Security

Explainable AI (XAI) is the field of reasoning about the decision process of machine learning models. There is active research with recent surveys providing taxonomies, algorithms, and evaluation criteria for explanation methods in machine learning [12, 191, 220].

A variety of general techniques for explaining learning models exist that can be broadly categorized into white-box and black-box explanation methods, where the latter correlate input features to the model output, for instance, SHAP [125] or LEMNA [69] or approximate them with smaller, better interpretable models like LIME [160]. This thesis focuses more on the former, white-box methods, since these methods tend to be more precise [203] and we assume full access to the vulnerability discovery models. These methods may use the activations from neurons like class activation maps (CAM) [215] or rely on the gradients of the model, for example, the linear approximation [167], grad-CAM [170], Smoothgrad [177] or integrated gradients [186]. There are also decomposition-based methods such as layer-wise relevance propagation [103], excitation backpropagation [225], or guided backpropagation [178].

With the rise of graph neural networks, several works have ported the underlying classic explanation concepts to the graph domain [8, 158, 167], and completely new graph-specific algorithms have been invented [126, 219, 168] building heat maps over nodes or edges.

Large language models use attention mechanisms where their respective attentions can be directly interpreted as their relevance scores for a specific input/output pair. However, it is currently disputed whether they actually offer interpretability [88, 207]. For instance, attention scores can be decoupled for their respective prediction and be altered while pertaining to the same prediction [88].

Bricken et al. [21] find monosemantic features, i.e., features that are uniquely descriptive for a decision outcome, using autoencoders. They argue that models trained on cross-entropy loss prefer many polysemantic features over fewer *true* monosemantic features.

Some recent approaches incorporate EMs in the vulnerability discovery task to integrate interpretability directly into the learning process [80, 55, 111].

Comparing Explanations in Security One large corpus of works discusses the comparison of EMs since they frequently arrive at vastly different explanations [203, 57, 230, 12, 158]. To quantitatively compare these EMs, there exist several metrics.

Warnecke et al. [203] provide security-specific comparisons of explanation methods. However, other works also provide comparison strategies [176, 23]. 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 [203]. Hence, explaining the decisions of such models is crucial [6]. Zou et al. present a method to extract essential tokens from token-based vulnerability discovery models. The extraction works by perturbing input source code pieces such that the binary classification label switches from one class to the other. 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 [230]. All works in this direction focus on intrinsic criteria that evaluate explanations by *descriptive accuracy* or *sparsity* or even suggest costly human expert studies to validate the actual usefulness. Sanchez-Lengeling et al. compare explanation methods using several types of ground-truth for molecule graphs [167]. Nadeem et al. [144] criticize that current security-related explainable AI research rarely conducts user studies, and it is not obvious how to integrate XAI into analyst workflows. Finally, Linardatos, Papastefanopoulos, and Kotsiantis state that it is not possible to rank EMs by their ability to make a model decision interpretable [117].

Limitations of SAST

Next, we will consider work that critically examines static and learning-based analysis methods. First, we we discuss related work that attempts to validate potentially incorrect SAST reports, and second, we will look at previous work that discusses the limitations of learning-based models for vulnerability discovery.

Static Analysis Verification From a broader perspective, some parts of this work compare and benchmark static code analysis methods. This has already been done in other non-learning-based contexts. Christakis, Müller, and Wüstholz [33] validate unverified and potentially unsound static code analysis reports using dynamic code execution to reduce false positives. Similarly, Wüstholz and Christakis [210] 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 from Definition 16 in Section 3.2, Barr et al. [10] define testing oracles as mechanisms that decide whether a set of system tests are relevant or not. Dietrich et al. [44] state that it makes more sense to validate static analysis results using oracles based upon dynamic analysis. Hence, Ma et al. [129] propose to apply directed symbolic execution to verify SAST reports. 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.

Limits of Vulnerability Discovery Models Recently, Chakraborty et al. revealed that several state-of-the-art datasets for evaluating vulnerability discovery models are not realistic [27]. In a similar vein, Arp et al. [6] discuss common pitfalls when working with learning-based vulnerability discovery methods. They 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. Wang et al. [197] criticize datasets obtained through biased approaches like filtering commit messages by certain keywords. They propose to filter samples using a classifier identifying security-relevant patches. Croft, Babar, and Kholoosi [35] state that most vulnerability discovery models suffer from label inaccuracies with up to 42.5% irrelevant samples. We leverage these insights in our experimental design, as well as in our metrics and dataset choices.

Risse and Böhme [161] discover that models seem to overfit certain artifacts in the dataset and hence lack generalizability. Chen et al. [30] analyze LLM-based vulnerability detector models and conclude that they lack transferability to other projects and, therefore, propose a novel dataset as a remedy. Krishnan et al. [100] define four major problems in learning-based vulnerability detection that hinder real-world adaptation. For instance, learning code semantics is hard, data is insufficient, the assessment is only considering laboratory settings, and there is a missing explanation for their decisions.

Harzevili et al. [72] argue that existing models may struggle to accurately capture the context of a vulnerability, either under-approximating program semantics through overly narrow contexts or treating code as a natural sequence. Over-approximating programs may involve considering infeasible paths [32], resulting in a higher FPR, while under-approximation leads to a high FNR. Zheng, Jiang, and Su [227] criticize that current vulnerability discovery models lack interprocedural analysis. Addressing these challenges, some recent approaches, such as those adopted by DiverseVul [30], abstain from identifier masking, as opposed to Draper [165], aiming to glean contextual information from variable and function names.

3

Software Defect Localization Using XAI

The following sections present solutions to recent limitations of machine learning-based vulnerability discovery models developed during this research. Our ultimate goal is to achieve a practical approach to localizing software defects using XAI. We specifically look at deep-learning-based vulnerability discovery models and how state-of-the-art methods struggle with transferability and generalizability. In general, models that help developers detect bugs and vulnerabilities are beneficial. However, they do not always transfer to new projects or code within the project they were initially trained on. Therefore, designing a vulnerability discovery model requires careful consideration.

As depicted in Figure 3.1, we suggest four dimensions that are important in the conceptualization and development of a vulnerability discovery model. These dimensions correspond to distinct stages that a model traverses. First and foremost, there is the preprocessing phase, with a strong emphasis on meticulous data curation. Following that, the model is employed by practitioners, highlighting the importance of the interpretability of the model results. In the third stage, during evaluation and post-deployment, the model must demonstrate applicability to previously unseen data. Lastly, the model should be conceived

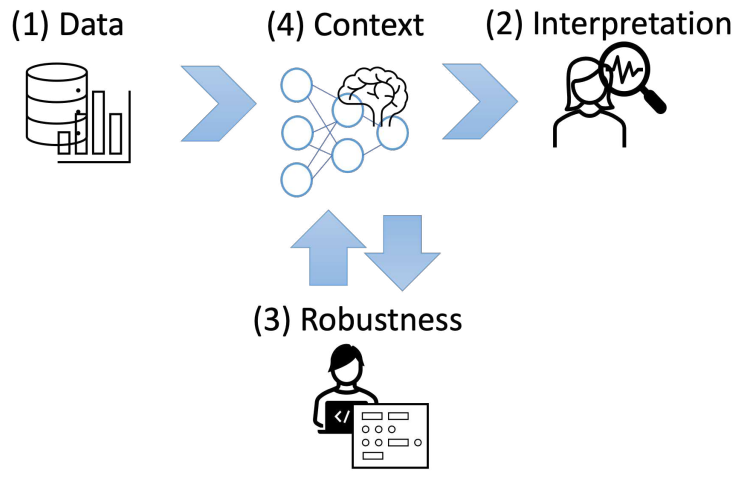


Figure 3.1: Brief overview of this thesis.

and constructed with the vulnerability detection task as its primary focus. In conclusion, we emphasize four perspectives that are indispensable for crafting production-ready vulnerability detectors:

(1) The Data The training data used to train a supervised machine learning model must be suitable for the task. In our case, we have to deal with datasets that reflect both *clean* as well as *vulnerable* code. Unfortunately, It is commonly known that *vulnerable* samples are scarce, and we have to assert that there are enough samples for the training process of the model. We will address this issue in Section 3.1 *Neural Vulnerable Code Augmentation* and present a novel neural vulnerable code augmentation technique based on our publication in Appendix C.

(2) The Interpretation Vulnerability detection models are designed to assist developers in finding software defects before they manifest as vulnerabilities in production code. We need to ensure that security experts can interpret the prediction results. However, we need to evaluate different explanation methods and even confirm their applicability to real software. We will benchmark different EMs and provide a novel approach using dynamic program analysis to effectively compare EMs specifically for vulnerability discovery in Section 3.2 *Explainability of Vulnerability Discovery Models* based on our works in Appendix D and Appendix E.

(3) The Robustness We show that excessive focus on irrelevant artifacts in the dataset can skew the reported effectiveness of learning-based vulnerability detectors during testing and after deployment. They lack robustness and are too sensitive to small perturbations in the input space. We present a novel evaluation scheme using causal learning to measure these confounding effects in Section 3.3 *Controlling Confounding Effects*, furthermore we demonstrate several easy-to-use preprocessing and model modifications to established vulnerability discovery models to combat confounding effects based on our work in Appendix F.

(4) The Context We have already discussed the importance of context sensitivity to rule-based detection models. We show that the same applies to learning-based SAST tools. Models with a too narrow context will over-approximate a program and suffer from a high FPR. It is challenging to enrich the model's perception of the code under analysis with a broader context. We propose a solution combining static taint analysis with deep learning in Section 3.4 *Learning-based Taint Analysis for Patches* based on our paper in Appendix G.

Each section outlines the problem setting and motivation for the specific problem and proceeds to briefly discuss our proposed approach and the results. For more in-depth details, we refer to the respective publication.

3.1 Neural Vulnerable Code Augmentation

I only hope that when the data's analyzed, a weakness can be found.
-Princess Leia Organa [85]

In learning-based vulnerability discovery, one major obstacle is obtaining enough representative code samples for supervised learning. First, vulnerable samples are sparsely available, and second, depending on the code base, there might be a massive variance under different code samples and different projects. One code author, for example, may prefer one programming pattern or paradigm over another, and one project might have a completely different linting and formatting guideline than another. If the datasets portray a limited view of how the C / C++ code is written, they may not understand the full diversity of the language [65]. In conclusion, in order for a learning-based system to derive meaningful patterns, the dataset must be sufficiently large and diverse [121].

Table 3.1: Popular datasets for vulnerability discovery.

Dataset	Level	#Vulnerable	#clean	Imbalance Ratio	Used in Thesis	#Projects
SARD [16]	Files	30000	30000	50.00%	✗	N/A
Devign [228]	Functions	11888	14149	45.65%	✓	2
Juliet [90]	Files	5393	6503	45.00%	✗	N/A
Draper [165]	Functions	87804	1198458	6.82%	✗	N/A
BGNN4VD [26]	Functions	3867	92058	4.02%	✗	4
Reveal [27]	Functions	1664	16505	9.15%	✓	2
PatchDB [199]	Functions	3441	30149	10.24%	✓	313
FUNDED [197]	Functions	5200	5200	50.00%	✗	1000
SySeVR [112]	Slices	2091	13502	15.48%	✗	N/A
BigVul [50]	Functions	11823	253096	4.44%	✓	348
CrossVul [150]	Functions	6884	127242	5.13%	✗	498
CVEFixes [15]	Functions	8932	159157	5.31%	✓	564
DiverseVul [30]	Functions	18945	311547	5.73%	✓	797

In Table 3.1, the most popular and common vulnerability datasets are listed to ensure the reader a good overview of the vastly different datasets in the wild. The datasets vary between $10k$ and $100k$ samples. The samples are curated from different numbers of open-source projects ranging from 2 to 1000. Projects with *N/A* do not or to an unknown number contain real-world extracted software projects. Many datasets lack vulnerable samples, which is observable in their large imbalance ratio. This directly leads to a less diverse representation of real vulnerabilities.

Problem Setting

Chakraborty et al. [27] state that the imbalance ratio is important to reflect real-world circumstances, and hence, a lower imbalance ratio is a preferable characteristic. However, it poses threats to the training and validation of machine learning models [6]. Pattern-generated datasets like SARD or Juliet and data labeled as ground-truth by rule-based SAST tools, such as in the vulnerability detector Draper [165], lead to over-promised results and less transferable detection models [27, 151]. While clean code samples are vastly available and can be gathered easily from different sources, for example, GitHub, StackOverflow, or SourceForge, vulnerable samples are scarce [6].

Often Github, vulnerability disclosure databases like the national vulnerability database (NVD) or CVE or bug trackers are consulted to collect vulnerable samples [27]. Although these databases may also suffer from minor inaccuracies [195], they are the best sources at present.

Another issue we encounter in the work of Zhou et al. [228] is generating datasets obtained from GitHub using security-related keyword extraction. This approach relies on keywords like `security`, `fix`, or `buffer overflow` to find commits potentially fixing security vulnerabilities. The assumption is that the same code before that patch contains the vulnerability. Unfortunately, many clean samples sneak in as vulnerabilities [35]. This problem is known as label inaccuracy [6] and can be best visualized with an example as in Figure 3.2. The patch is from the Devign dataset extracted from the QEMU open-source project with its commit title displaying *qemu-iotests: step clock after each test iteration* and its commit message happens to contain the keyword *fix* and *hang*, however, this is neither a vulnerability nor a relevant defect, as it solves some minor hangs during testing[1]. The dataset even contains samples that are not C or C++ functions but, for instance, markup or configuration files[2]. These improperly labeled samples impose a bias on the final vulnerability detector. Inspired by Dunlap et al. [48], we can conduct a differential dataset analysis to obtain a lower bound on the true proportion of defects in the dataset. We want to refer the reader to Appendix A for this experiment.

```
1 ...
2 + # Allow remaining requests to finish. We submitted twice as many to
3 + # ensure the throttle limit is reached.
4 + self.vm.qtest("clock_step %d" % ns)
5   ...
```

Figure 3.2: Wrongly labeled sample from the Devign dataset [228].

Equipped with the state-of-the-art datasets and their shortcomings, we address the problem of scarce vulnerable data and imbalanced datasets and propose a solution utilizing only high-quality datasets extracted from more reliable sources such as NVD or the CVE database [27] and enhancing them.

Approach

In classical machine learning, there are several approaches to handle imbalanced datasets [116, 70, 79]. A naive approach is to downsample the dataset, throwing away *clean* samples. A more sophisticated approach is to augment the dataset with new synthetic *vulnerable* samples. A popular method is synthetic minority oversampling technique (SMOTE), which takes existing similar samples and pairwise interpolates between them. Similar samples can be selected by, for example, their k nearest neighbor set of the same class [20, 98]. New samples are then generated by interpolating between the features of the two selected samples, yielding a new feature vector $\tilde{x} = \lambda x_1 + (1 - \lambda)x_2$, where x_1, x_2 are the features of the original samples and $\lambda \in [0, 1]$ is a uniformly distributed random number.

This may work well with continuous feature vectors, but code is naturally discrete. Inspired by the work of Dablain, Krawczyk, and Chawla [38] and Zhao, Zhang, and Wang [226], where the former applies SMOTE to image classification and the latter to node classification, this

[1] https://github.com/qemu/qemu/commit/cbaddb
[2] E.g. https://github.com/ffmpeg/ffmpeg/commit/301ab1

thesis proposes a variational autoencoder as a generative model for vulnerability augmentation. More specifically, our approach called *CodeGraphSMOTE* enables the generation of new vulnerable code graphs by interpolating in the latent space of a *variational* graph autoencoder.

During training, a vulnerable code property graph (CPG) x is passed through an encoder graph neural network E that produces two output vectors, namely a mean $E_\mu(x)$ and a variance $E_\sigma(x)$ vector.
Then a latent representation $z \sim \mathcal{N}(E_\mu(x), E_\sigma(x))$ can be sampled, and the decoder network $D(z)$ should approximate x as its reconstruction [63]. The training objective has two parts. First, we would like $D(z)$ to be similar to x, so the first optimization goal is to minimize the reconstruction loss $||D(z) - x||$. Second, as regularization and to prevent the model from memorizing, we constrain the latent space distribution to be close to a standard Gaussian distribution using the Kullback-Leibler divergence $KL\left(\mathcal{N}(E_\mu(x), E_\sigma(x))||\mathcal{N}(0, 1)\right)$.

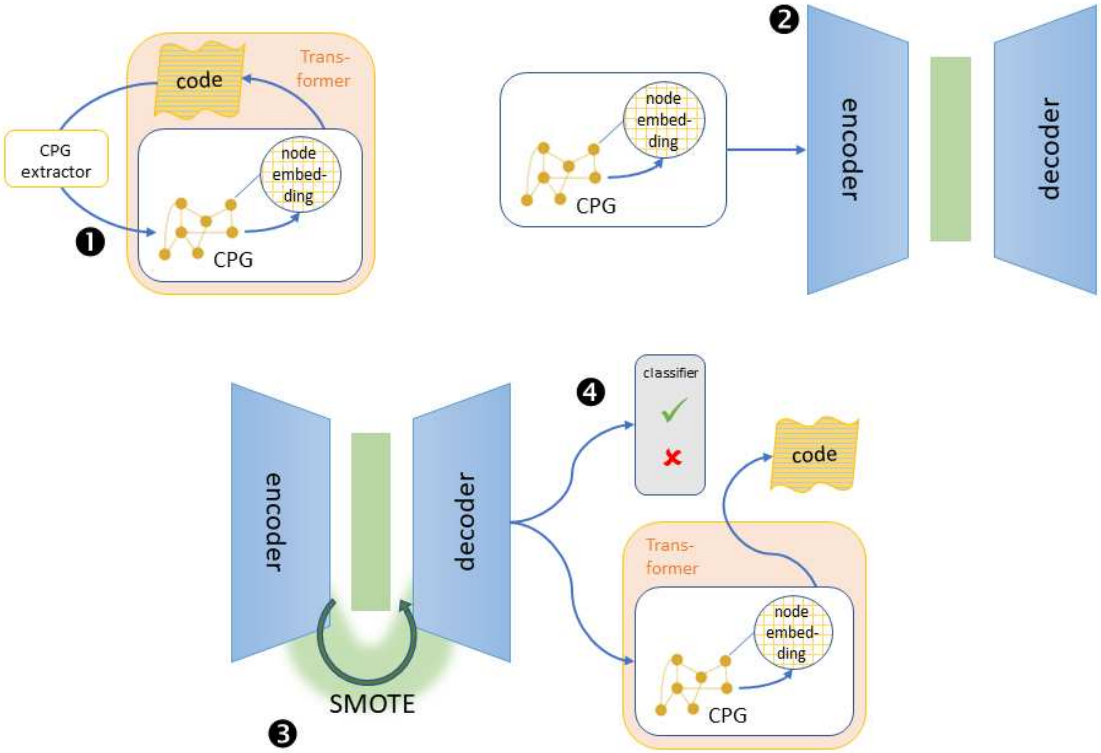


Figure 3.3: The CodeGraphSMOTE training and inference pipeline.

Figure 3.3 depicts the architecture of CodeGraphSMOTE: A sequence-to-sequence BART [106] transformer network is used to learn a descriptive embedding of source code attached at the nodes (1), and a variational graph autoencoder is used to compress an entire code graph into a continuous latent space preserving the most relevant information of the code graph (2). Eventually, SMOTE is used on this latent representation (3). The graph topology can be reconstructed using the decoder layer of the autoencoder, and ultimately, the source code can be reconstructed using the transformer network.

$$
\begin{aligned}
z_1 &\sim \mathcal{N}(E_\mu(x_1), E_\sigma(x_1)) \\
z_2 &\sim \mathcal{N}(E_\mu(x_2), E_\sigma(x_2)) \\
\tilde{x} &= D(\lambda z_1 + (1-\lambda) z_2)
\end{aligned} \tag{3.1}
$$

Formally, we can interpolate between a code graph sample x_1 and x_2 by plugging the latent space representation into the SMOTE formula as outlined in Equation (3.1).

Result

Using CodeGraphSMOTE, we circumvent the problem of interpolating discrete structures as code or graphs. If we visualize the latent vector per sample using t-distributed stochastic neighbor embedding (t-SNE) [130], we observe a natural clustering of similar CWEs as in Figure 3.4.

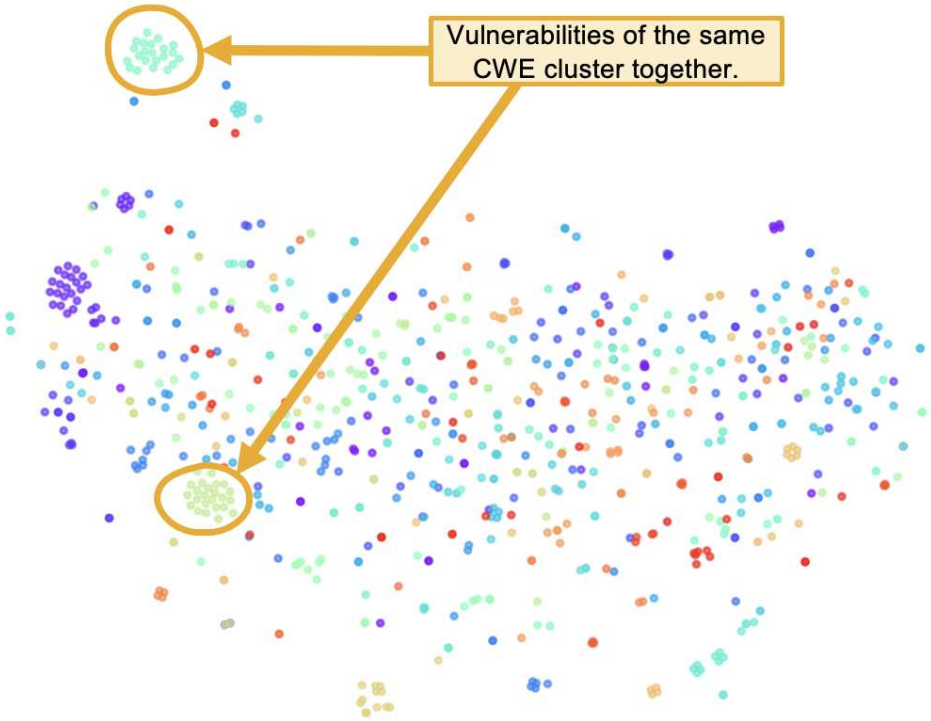


Figure 3.4: A t-SNE latent space visualization of CodeGraphSMOTE. Every point is an encoded vulnerable sample and every color is a CWE.

A visualized artificially interpolated example can be seen in Figure 3.5. The two upper real-world vulnerabilities truncate a string but incorrectly calculate its size. Consequently, the interpolated vulnerability is also containing a buffer overflow. It may not be guaranteed that the code is syntactically or semantically correct, for instance, the interpolated sample unnecessarily defines a `char *dest` in the signature, however, this may not even be necessary since it provides a more diverse view of the C and C++ programs to the learning model.

In addition to this, our experiments suggest that we can achieve a performance increase in downstream vulnerability models with up to 21% measured in balanced accuracy compared to other approaches, including simple downsampling.

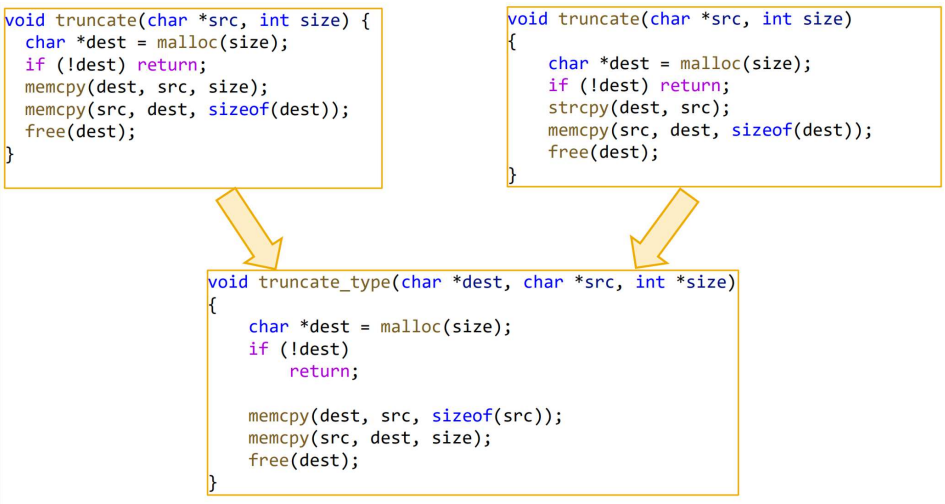


Figure 3.5: An interpolated artificial vulnerability.

CodeGraphSMOTE performs better than state-of-the-art neural code editing methods such as Graph2Edit [217]. And while there are also deterministic code editing techniques, such as identifier renaming [218, 78, 224, 216, 223], insertion of statements [218, 223, 78, 179] or replacement of code elements with equivalent elements [108, 5] which might be suitable for code augmentation, the model may memorize the set of transformations and we are faced with the same problem as with the SARD dataset.

Although there are a large number of datasets for vulnerability discovery available, they all lack in *vulnerable* samples and suffer from imbalance.
Moreover, some datasets are of lower quality than others but provide more variety. It is evident that the performance of the discovery model depends on the underlying dataset. Thus, we propose CodeGraphSMOTE as a probabilistic strategy to augment high-quality datasets.

3.2 Explainability of Vulnerability Discovery Models

> *The answer to the ultimate question of life, the universe, and everything is 42.*
> *-DeepMind [1]*

The efficacy of vulnerability discovery models comes at a price: neural networks are black-box models due to their deep structure and complex connectivity. While these models produce remarkable results in lab-only experiments, their decisions are opaque to security practitioners and remain unclear, which hinders their adoption in practice. Identifying security vulnerabilities is a subtle and nontrivial task. Therefore, interaction with human experts is indispensable when looking for vulnerabilities. For them, it is pivotal to understand the decision process behind a method to analyze its findings and decide whether a piece of code is vulnerable. Thus, any technique for their discovery must be *interpretable*.

Problem Setting

One promising direction to address this problem is the field of Explainable AI (XAI). A large body of recent work has focused on explaining the decisions of neural networks, including feed-forward, recurrent, and convolutional networks [215, 170, 177, 142, 103].

Similarly, some specific methods have been proposed to make GNNs interpretable [168, 126, 219]. However, it is unclear whether and which of the methods of this broad selection can support and explain decisions in vulnerability discovery models [203].

```
1  int xmlStrlen(const xmlChar *str) {
2      int len = 0;
3      if (str == NULL) return(0);
4      while (*str != 0) {
5          str++;
6          len++;
7      }
8      return(len);
9  }
10
11 xmlChar *xmlStrncat(xmlChar *cur, const xmlChar *add, int len) {
12     int size;
13     xmlChar *ret;
14     if ((add == NULL) || (len == 0))
15         return(cur);
16
17     if (len < 0)
18       return(NULL);
19
20     if (cur == NULL)
21         return(xmlStrndup(add, len));
22
23     size = xmlStrlen(cur);
24    if (size < 0)
25        return(NULL);
26     ret = xmlRealloc(cur, (size+len+1) *sizeof(xmlChar));
27     if (ret == NULL) {
28         xmlErrMemory(NULL, NULL);
29         return(cur);
30     }
31     memcpy(&ret[size], add, len*sizeof(xmlChar));
32     ret[size + len] = 0;
33     return(ret);
34 }
```

Figure 3.6: Vulnerability *CVE-2016-1834* with highlighted explanations for ReVeal+GNNExplainer (green), ReVeal+Smoothgrad (blue), ReVeal+GradCam (red).

Consider the vulnerability identified by *CVE-2016-1834* in *libxml2*, as shown in Figure 3.6, where several lines are highlighted. The vulnerability discovery model ReVeal [27] correctly asserts that there is a potential buffer overflow. However, three different explanation methods show three different lines as the root cause of the vulnerability. It is unknown which explanation yields the best highlighting for a human security practitioner.
The question therefore arises as to which method of explanation is to be favored and which is actually correct. In order to compare them, we need a quantitative measure. Therefore, we define a metric to evaluate explanations as an explanation criterion in Definition 15 [60].

Definition 15 *An **explanation 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 $c(e_f|D) > c(\hat{e}_f|D)$.*

To make things worse, there exist many different quantitative criteria [220, 176]. In the following, we present six common criteria that are usually used in recent literature for an automatic evaluation of EMs to illustrate the diversity to the reader [57]:

1. *Descriptive accuracy.* To determine whether an EM captures relevant structures in a vulnerability discovery model, we remove the most relevant features associated with the classification.

2. *Robustness.* To measure the robustness of an input graph for a given EM, we compute the relevant features before and after perturbing the input code sample.

3. *Contrastivity.* This criterion calculates the discrepancy of the relevant statements between the *vulnerable* and *clean* classes.

4. *Sparsity.* An explanation method must stay concise during operation. To this end, we count the normalized amount of relevant code.

5. *Stability.* Some EMs are nondeterministic and do not provide identical results during different runs. To account for this problem, we can measure the stability in terms of the standard deviation.

6. *Efficiency.* Finally, the runtime of an explanation method should not drastically increase the time a security specialist needs for her traditional workflow.

In our work from Appendix D, we conduct an extensive experiment comparing these criteria in Table 3.2 using two models, namely Devign and ReVeal and 12 different EMs [57].

Table 3.2: The ↑descriptive accuracy, ↑robustness, ↑contrastivity, ↑sparsity and ↓efficiency for the EMs. The standard deviation is omitted for deterministic methods as well as SmoothGrad as it is neglectable. ↑ means higher numbers are better. ↓ means lower numbers are better.

Criteria	**Descriptive Accuracy**		**Robustness**		**Contrastivity**		**Sparsity**		**Efficiency**	
Model	**Devign**	**ReVeal**	**Devign**	**ReVeal**	**Devign**	**ReVeal**	**Devign**	**ReVeal**	**Devign**	**ReVeal**
GNNExplainer	0.08 ±0.003	0.15 ±0.008	0.55 ±0.000	0.58 ±0.010	0.09 ±0.01	0.19 ±0.01	0.73 ±0.000	**0.73** ±0.001	3.99 ±0.000	5.07 ±0.001
PGExplainer	0.09 ±0.003	0.16 ±0.002	0.37 ±0.000	0.57 ±0.010	0.10 ±0.010	0.21 ±0.030	**0.81** ±0.010	**0.73** ±0.001	1.22 ±0.010	47.04 ±0.001
Graph-LRP	0.09 ±0.002	0.10 ±0.000	0.13 ±0.000	0.71 ±0.000	**0.11** ±0.000	**0.35** ±0.010	0.79 ±0.000	0.14 ±0.000	22.24 ±0.000	33.01 ±0.000
Random	0.08 ±0.003	0.18 ±0.014	0.07 ±0.000	0.07 ±0.010	0.12 ±0.000	0.40 ±0.000	0.51 ±0.000	0.52 ±0.000	0.01 ±0.000	0.02 ±0.000
EB	0.09	0.10	0.48	0.71	0.02	0.00	0.80	0.14	0.11	**0.07**
GB	0.10	0.10	0.40	0.71	0.05	0.00	0.80	0.14	**0.10**	0.16
Gradient	0.10	0.10	0.40	0.71	0.05	0.00	0.80	0.14	**0.10**	0.16
LRP	0.09	0.10	0.16	0.71	0.08	0.00	0.77	0.14	0.14	0.21
CAM	0.26	**0.29**	0.45	0.49	0.01	0.07	0.48	0.14	0.12	0.17
SmoothGrad	0.08	0.10	0.30	0.71	0.03	0.00	0.77	0.15	1.66	1.72
GradCAM	0.11	0.10	0.42	0.71	0.01	0.00	0.56	0.14	0.11	0.16
Linear-Approx	0.09	0.10	0.42	0.71	0.02	0.00	0.80	0.14	0.11	0.16
IG	**0.31**	0.14	**0.71**	**0.72**	0.00	0.06	0.15	0.19	1.63	2.52

Looking at Table 3.2, we observe a lack of agreement between the different EMs. There is no one-fits-all solution since not only do the EMs perform differently from model to model, but also very differently concerning different criteria. Although graph-specific EMs achieve good results in sparsity, they lack runtime efficiency. Furthermore, their descriptive accuracy is not much better than just randomly highlighting lines.

Ultimately, we are not only dealing with a large number of different EMs that perform differently but also with a large number of different criteria for comparison that do not agree with each other.

Approach

As a remedy, we propose to compare the EMs in their ability to locate defects correctly. Essentially, we can associate any EM criterion to an oracle [44, 10] given Definition 16. In the case of the criteria from Table 3.2, the vulnerability discovery model itself is used as an oracle to assess the explanation. For example, consider the descriptive accuracy (DA): The model is observed whether its performance decreases given different features. Since the explanation method is only assessed by the model that generated the decision in the first place, we refer to such criteria as *intrinsic*. Intrinsic criteria are prone to noise in the input space of the model and reflect its biases or completely neglect the task at hand, for instance, the *sparsity* criterion.

Definition 16 *An* ***explanation oracle*** *is a function* $o\colon \mathcal{M} \to [0, 1]$ *that evaluates the relevance attributed in a heatmap.*

We propose *extrinsic criteria* based on feedback from extrinsic oracles, where we achieve a separation of the assessor and the predictor. In theory, a human security expert could pose as an oracle and manually assess each explanation. However, in reality, this is too costly. Therefore, to eliminate any potential model bias and still compare the explanations with respect to the underlying vulnerability discovery task, we can automatically assess EMs by using dynamic program analysis. To this end, we can use any dynamic program analysis technique, for instance, model checkers [129], abstract interpretation [68], or log analyzers [76]. In this thesis, however, we will stick to fuzzing [131, 75], more concretely, to directed fuzzers and runtime debuggers.

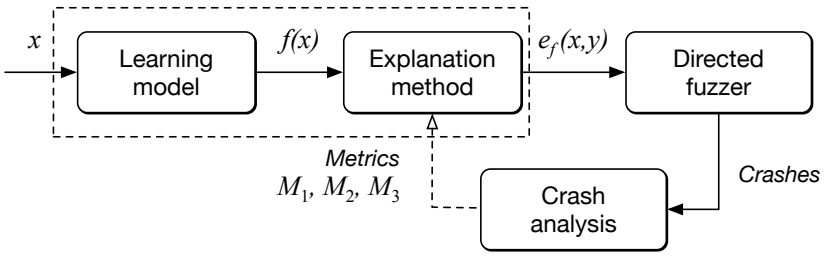


Figure 3.7: Overview of our approach for generating local ground-truth for explanation methods.

A fuzzer is a program analysis tool used to generate inputs and provoke program crashes [131, 53]. These crashes indicate software defects, and since manipulated inputs trigger them, they often represent security vulnerabilities in the sense of Definition 1. We employ a *directed* gray-box fuzzer to retrieve a set of target locations given by the heatmap from the EMs and aim at reaching these by seeking inputs that minimize the distance to the locations [18, 153]. For instance, the directed fuzzer *AFLGo* calculates control-flow and call-graph distances before the fuzzing process to guide this search. Similarly, approaches to directed fuzzing, such as Hawkeye [29], can be applied in this step to explore the highlighted code regions and test for the presence of vulnerabilities. Algorithm 1 summarizes the basic procedure of directed fuzzers, as can be found in AFLGo and Hawkeye [18, 29].

AFLGo [18] uses simulated annealing to ensure convergence: An initial high temperature is selected, which exponentially decreases throughout the fuzzing duration.
At first, inputs are assigned uniformly distributed energy and, for example, are prioritized even though they do not necessarily reduce the distance to the targets as indicated by `AssignEnergy()`

Algorithm 1: Directed Fuzzing [18].

```
   input  : Seed S, Targets T
   output : Crashes C
 1 repeat
 2     s = ChooseNext(S)
 3     E = AssignEnergy(s)
 4     repeat E times
 5         s' = MutateInput(s)
 6         if LeadsToCrash(s') then
 7             add s' to C
 8         else if IsInteresting(s', T) then
 9             add s' to S
10 until Time budget is exhausted
```

in Algorithm 1. During cool-down and towards the decrease of the temperature, the input seeds are more constrained to decrease the target's distances until eventually hitting an optimum.

To pinpoint the exact location of a crash, we utilize a debugger, for instance the GNU Debugger (GDB) [42], to reproduce the runtime state with the fuzzed input.
This crash is an evident *fact* and thus represents a form of ground-truth derived from a genuine incident during the program execution. In Figure 3.7, we can see the overall pipeline of the EM validation scheme. Using a directed fuzzer, we can come up with new extrinsic criteria that can be summarized as follows:

1. *Crashes per path over time (M_1)* 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 lead to more paths with crashes or hangs.

2. *Mean breakpoint hits (M_2)* 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 of the explanation are relevant.

3. *Mean crash distance (M_3)* To overcome the gap left by M_2, we also measure the average statements executed 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.

We compare different EMs and model combinations using these extrinsic criteria using three open-source programs under analysis, namely Libming, Libxml2, giflib, and the ReVeal [27] and Devign [228] models. We average the scores over the models to obtain a single score per EM. The results of this work suggest a beneficial guidance of the fuzzer using vulnerability discovery models and EMs.

Result

Given Table 3.3, we can see that all methods significantly outperform random line annotations. Furthermore, we can see that the intrinsic results from Table 3.2 are misleading since graph-

based EMs actually perform better than graph-agnostic EMs. VulDeeLocator [111] and LineVul [55] are both line-level vulnerability discovery tools. The former uses node classification and the latter uses attention mechanisms to locate the flawed code line. Both tools underperform heavily against the extrinsic and model agnostic EMs.
We could not measure M_2 and M_3 for the rule-based static analyzers since both tools could not find a significant amount of software defects in the samples. However, we can see that the Flawfinder lexical rule-based analyzer and the Cppcheck syntactic analyzer have at least a worse M_1 metric than the Learning-based analyzers, which are only slightly above Random.

Table 3.3: M_2 and M_3 comparison between EM.

EM	Crashes per path over time M_1	Mean Breakpoint Hits M_2	Mean Crash Distance M_3
Random	0.51 ± 0.42%	8.21 ± 0.40%	0.124 ± 0.014s
GNNExplainer	2.09 ± 0.30%	**15.48** ± 0.22%	0.084 ± 0.002s
SmoothGrad	1.96 ± 0.37%	11.06 ± 0.13%	**0.077** ± 0.003s
PGExplainer	**2.15** ± 0.41%	10.04 ± 0.37%	0.088 ± 0.010s
GradCam	2.05 ± 0.12%	9.26 ± 0.05%	0.097 ± 0.006s
VulDeeLocator	1.02 ± 0.28%	10.54 ± 0.14%	0.094 ± 0.002s
LineVul	1.04 ± 0.34%	9.97 ± 0.16%	0.084 ± 0.003s
Flawfinder	0.71 ± 0.15%	-	-
Cppcheck	0.81 ± 0.16%	-	-

The use of EMs has been shown to produce varying results depending on the method, which can greatly hinder their application in identifying vulnerabilities. Every incorrectly labeled line of code must be manually assessed. The effectiveness of selecting a particular EM depends on the dataset and the model being used and cannot be determined using intrinsic metrics alone. To make an informed selection, input from human security experts or dynamic program analysis can be used cost-effectively and automatically. We propose to benchmark EMs using our extrinsic metrics as a more accurate comparison.

3.3 Controlling Confounding Effects

> *It has long been an axiom of mine that the little things are infinitely the most important.*
> *-Sherlock Holmes [47]*

Kassar et al. [91] show that different code patterns can impede the efficiency of rule-based SAST tools. We show that similar effects are present in learning-based SAST tools. Although learning-based vulnerability detectors have achieved remarkable accuracy in recent work [55], these benchmarks are conducted only in laboratory settings. They cannot be projected to real-world applications [136]. In reality, the models fail to generalize and lack transferability [27, 30]. On the one hand, this means that the trained models are not able to detect out-of-distribution vulnerabilities from, for example, unseen projects or repositories. Still, on the other hand, this also means that they are unable to detect defects in slightly modified samples. This effectively hinders their adaptation to non-academic or real-world settings.

Problem Setting

In Figure 3.8, we see a type confusion bug from the PHP Zend Engine[3], LineVul [55] correctly classifies it as vulnerable.
However, suppose we remove the green line and replace it with the semantically equivalent but slightly obfuscated red line. In that case, LineVul fails to detect the bug, even if this line is entirely unrelated to the underlying flaw. In essence, the bug occurs due to user-provided data `&var2`, without checks, being cast to a double. Models overfit to label-unrelated features, making them unable to detect bugs that don't share common characteristics from the train set [161].

```
1 ...
2 float matrix[3][3] = {0,0,0, 0,0,0, 0,0,0};
3 float matrix[3 & 0xF][3 & 0xF] = {{0o0, 0o0, 0o0}, {0o0, 0o0, 0o0}, {0o0, 0o0, 0o0}};
4 if (zend_hash_index_find(Z_ARRVAL_PP(var), (j), (void **) &var2) == SUCCESS) {
5     SEPARATE_ZVAL(var2);
6     convert_to_double( *var2);
7     matrix[i][j] = (float)Z_DVAL_PP(var2);
8  ...
```

Figure 3.8: Confounding type confusion bug in the PHP Zend engine.

We can visualize this effect of the so-called spurious correlation in Figure 3.9, which derives a simple causal model for a vulnerability discovery function f_θ [184].
Here, X is the input data, and Y is the label, being either *vulnerable* or *clean*. The learning goal of f_θ is to find a relationship between the learned representation R and the label Y. If we inspect Figure 3.9 further, it is trivial to see that the representation learned by a model f directly influences the predicted label Y. But instead of using the sample under analysis to directly control the representation, we model it so that X has a causal and a trivial part [184]. We call the latter the confounding variable, shortcut feature, or spurious correlation [154]. As a result, we have three relationships: $A \leftarrow X \rightarrow C$, $C \rightarrow R \leftarrow A$, and $R \rightarrow Y$. The relationship $A \leftarrow X \rightarrow C$ denotes the causal features C while A denotes the trivial or biased feature patterns. Both influence the final latent representation R.

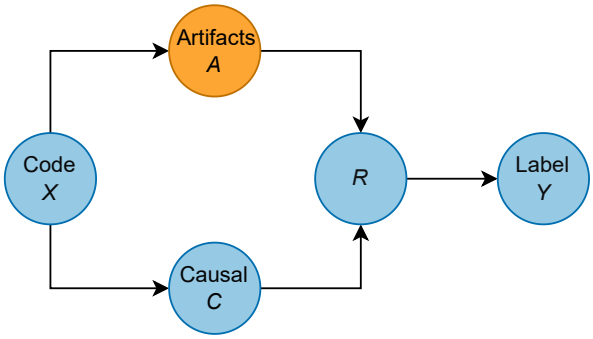


Figure 3.9: A causal model for learning-based vulnerability discovery.

Borrowed from causal learning, we call such artifacts *confounders*, and in our case, these originate from irrelevant features that cause the model to learn biased representations [184]. A

[3]https://cve.report/CVE-2014-2020

simple example for an artifact would be a common code style in all *vulnerable* samples and a different style for all *clean* samples [161]. To this end, we identify three sources of bias that can manifest themselves as artifacts in program code [59]. Beware that this list is non-exhaustive since there could be infinite possible confounders in a model.

1. *Coding style.* Every collected sample has an implicit coding style. Since many open-source projects use automatic linting, samples from one project to another are likely to differ in their styles. If there are more vulnerable samples from one project than another, the coding style correlates with Y.

2. *Control flow.* Projects often contain different calling hierarchies or indirections due to programming patterns, for instance, object-oriented design principles. Several projects and authors prefer one pattern over another, which may introduce confounding bias.

3. *Naming.* Different samples from different projects naturally vary in their naming conventions. Hence, vulnerable samples may potentially differ in variable naming compared to clean samples. Although it is common to mask such symbols, recent works desist from normalizing them.

A model that is too sensitive to noise in the input space is not considered robust [192]. We have already seen *robustness* in Section 3.2 to measure the difference in the explanation heatmaps with slightly perturbed input features. In this section, we are interested in the *robustness* form a model perspective.

Approach

To measure the true causal correlation and to remove confounding variables in causal learning, it is common to calculate the influence of one variable affecting the other by *intervention* [154]. This can be done using do-calculus, that is, we can stratify the confounder by calculating the influence of $C \rightarrow Y$ given all possible artifacts from $a \in A$. Since the possible artifacts are infinite, we have to approximate the distribution of A by calculating the estimated likelihood of the code samples $X = (x_0, \ldots, x_n)$ using a subset $A' \subset A$ and define an artifact $a \in A'$ to be a specific semantically equivalent variant of the code samples $k_a(X) = (k_a(x_0), \ldots, k_a(x)_n)$ obtained by one semantic-preserving perturbation $a \in A'$. To remove the confounding effects, we can use the calculation from Equation (3.2).

$$P(Y|do(C)) \approx \sum_{a \in A'} P(Y|C, A = a)P(a). \tag{3.2}$$

For $k_a(x)$ with $a \in A'$, we define a specific set of code transformations to obtain different variants of the dataset for training and evaluation. These transformations can be categorized into three classes:

1. *Styling.* We can apply style formatting using *clang-format* with different popular predefined styles [82].

2. *Uglification.* We test two different kinds of "uglification". The first variant consists of removing comments, randomly renaming all variables, and applying a code style. The second variant is the same except for additionally removing all whitespaces and newlines.

3. *Obfuscation.* The obfuscation consists of randomly renaming variables and functions, removing comments, adding unneeded statements, adding function definitions, and replacing numbers with an equivalent number representation.

Using the transformations and the stratification from Equation (3.2), we can derive a function to measure the confounding effect of the artifacts in Equation (3.3).

$$c = \frac{\sum_{a \in A'} P(Y|C, A = a)P(a) - P(Y|C, A)}{P(Y|C, A)} \tag{3.3}$$

As an intuition, consider $c = 0$, meaning that $P(Y|C, A) = P(Y|do(C))$. The more c deviates from 0, the greater the influence of the artifacts and $P(Y|C, A) \neq P(Y|do(C))$.

The application of different perturbations to the code should resemble a causal intervention. A non-confounded model should perform equally on semantically equivalent but perturbed code since the decision should solely depend on the causal feature part. Let us define the model predictions on the code samples under different perturbations as $\forall a \in A' : f(k_a(x))$. Consequently, this intervention provides insight into how the model behaves under different artifacts and yields a more robust basis for model evaluation and comparison. In a more practical sense, suppose that we have a metric $M \colon f \to \mathbb{R}$ that assesses the quality of a learning-based vulnerability discovery model, such as accuracy. We can then measure the influence of confounders using Equation (3.4).

$$c = \frac{\sum_{a \in A'} M(f(k_a(x))P(a) - M(f(x))}{M(f(x))} \tag{3.4}$$

Measuring the confounding effects is one side of the coin. Thus, this thesis also proposes four solutions to significantly reduce the confounding effects of different types for vulnerability detection models.

1. *LLMs with normalized code.* To remove the effect of style artifacts on LLMs, one naive solution is to normalize the code. Code normalization is the modification of code so that it conforms to a given style guide, which reduces, but does not remove, the impact of the personal style on the code [82].

2. *LLMs with pre-tokenized code.* Another solution follows from the work of Roziere et al. [163], who propose to tokenize the code before applying the byte-pair encoding using a programming language-specific lexer. They feed the resulting tokens as space-separated plain text into the model, a process we refer to as *pre-tokenization.*

3. *Causal Models.* A more principled approach arises from the work of Sui et al. [184] in the domain of GNNs: By applying an intervention directly to the learning model, they can mitigate the impact of confounding variables. This is done by conditioning the causal input features, in this case, a code graph, per sample with all possible trivial subgraphs obtained during training [87].

4. *LSTMs with differentiable memory.* Delétang et al. [41] show that transformer models cannot generalize well over different-sized input token lengths. The authors state that classical LSTMs and RNNs with differentiable memory provide stronger generalization

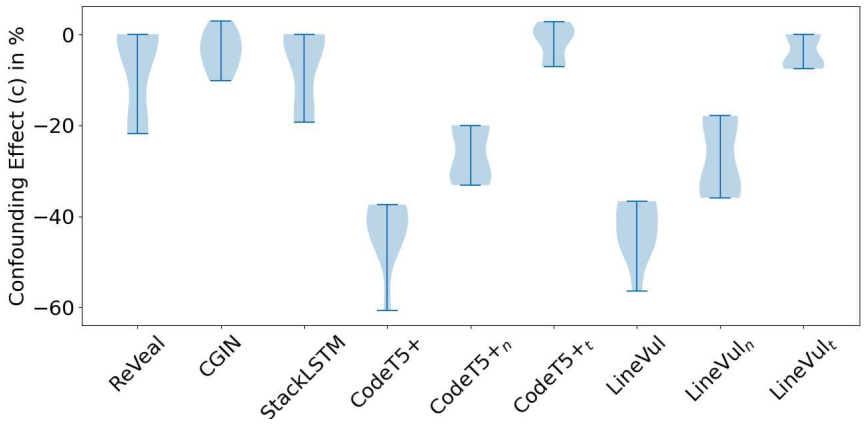


Figure 3.10: The confounding effect on the function-level detection performance measured by balanced accuracy.

performance than transformer models on increasingly complex tasks. Since the number of tokens within samples can impose another bias, we propose to extend LSTM-based vulnerability detection models with a differentiable stack [102].

Result

In Figure 3.10, we can see that the LLMs, LineVul, and CodeT5+ [201] have the most disadvantageous confounding effects with up to 60% performance decrease measured by their balanced accuracy. However, using the normalization trick, LineVul$_n$ and CodeT5+$_n$, arrive at a much better confounding effect of around 40% decrease. Using the tokenization trick, we can even further reduce the effect to only 5%.
ReVeal, the GNN, has a confounding effect of 20% but can be reduced using the causal graph replacement by Sui et al. [184] to around 10%. StackLSTM is an extension to VulDeePecker [113] and relies on a differentiable stack. Its confounding effect is slightly better than ReVeal, however not as good as the causal graph isomorphism network (CGIN).

The novel evaluation scheme relying on causal learning has unequivocally exposed the presence of numerous false correlations within the dataset, which led to an underperforming vulnerability detection model and wrong line-level explanations. However, implementing the proper preprocessing techniques or modifying the model architectures can significantly reduce the confounding effect on the task of discovering vulnerabilities.

3.4 Learning-based Taint Analysis for Patches

I'm trying to free your mind, Neo. But I can only show you the door.
-Morpheus [86]

Current vulnerability discovery models can't look behind function definitions and have limited context and worse so, recent transformer models process tokens in a very limited window [147, 39] and possibly fail to analyze functions that have more tokens than their limit [227]. Examining the historical evolution of vulnerability discovery models unveils a progression from models considering entire functions as observations [228, 27] to those focusing on local

code regions surrounding point of interests (PoIs) [112, 31]. The latter category views functions as classification targets as too coarse-grained, containing excessive, unnecessary information. Thus, they identify PoIs as specific statements involving array index calculations, function calls, pointer arithmetic, or memory management, treating them as syntactic vulnerability candidates. These models generally achieve better performances compared to function classifiers. In general, these models incorporate a small interprocedural context by inlining functions within the region of interest [227]. This inclusion of an interprocedural context is considered a key factor contributing to the enhanced performance, a hypothesis supported by various works [136, 32].

Problem Setting

Examining Figure 3.11, the Clang analyzer suspects a potential memory leak at the end of the code snippet. The function `free_ftrace_hash` acts as a wrapper for the standard library function `free`. The SAST tool is unaware of the actual semantics, underlining the crucial necessity for interprocedural context, especially in the realm of learning-based SAST tools [227]. Inlining functions would indeed solve this problem, but with deeper nesting and larger call hierarchies, we quickly reach limits.

```
1 [...]
2     new_hash->flags &= ~FTRACE_HASH_FL_MOD;
3     mutex_lock(&ftrace_lock);
4     ret = ftrace_hash_move_and_update_ops(ops, orig_hash, new_hash, enable);
5     mutex_unlock(&ftrace_lock);
6 out:
7     mutex_unlock (&ops->func_hash->regex_lock);
8     free_ftrace_hash(new_hash);
9 [...]
```

Figure 3.11: Linux kernel code sample with an alleged memory leak.

It has been demonstrated by Beba and Karlsen [11] that by adding flow-sensitive analysis methods to SAST tools, the false positive rate can be reduced. Similarly, combining learning-based SAST tools with taint analysis should improve precision and enable the discovery models to leverage context sensitivity.

Approach

Definition 1 tells us that a security vulnerability requires human input. Thus, as a novel contribution, we extend the definition of point of interests (PoIs) beyond the potential manifestation of software defects, as proposed by Mirsky et al. [136], to also encompass areas associated with human-provided input, such as `gets` or `cin`. Given a code graph, we can label all PoIs and arrive at a set of nodes V_{SOURCE} and V_{SINK} just as in the taint-style analysis Definition 8. If we were to trace all possible paths between the two sets by following every control and data flow, we would end up with an interprocedural graph that is too large to handle. Consider Figure 3.12, we can see two code graphs of a hypothetical program. At the same time, numbered nodes represent functions or statements, *SRC* and *SINK* denote source and sink nodes, respectively. The directed edges represent the flow of control or information. The left graph depicts a hypothetical interprocedural code graph for an entire program.

Such large graphs introduce severe problems for machine learning applications. For example, using GNNs, we would need to stack many layers to consider information that extends over larger neighborhoods, which would not only be computationally ineffective but also introduce problems such as Laplacian over-smoothing [107]. Moreover, information between two distant nodes is diluted, which would also affect models like RNNs or even LSTMs. LLMs suffer from a quadratic runtime complexity with respect to the input size, which, in addition to their limited processing context, would further impair their practical suitability. Minimizing the code graph by reformulating the problem from *vulnerability discovery* to *vulnerable patch detection* enables us to consider larger contexts without having to deal with exploding feature spaces. We will call this novel approach patch-based vulnerability discovery (PAVUDI).

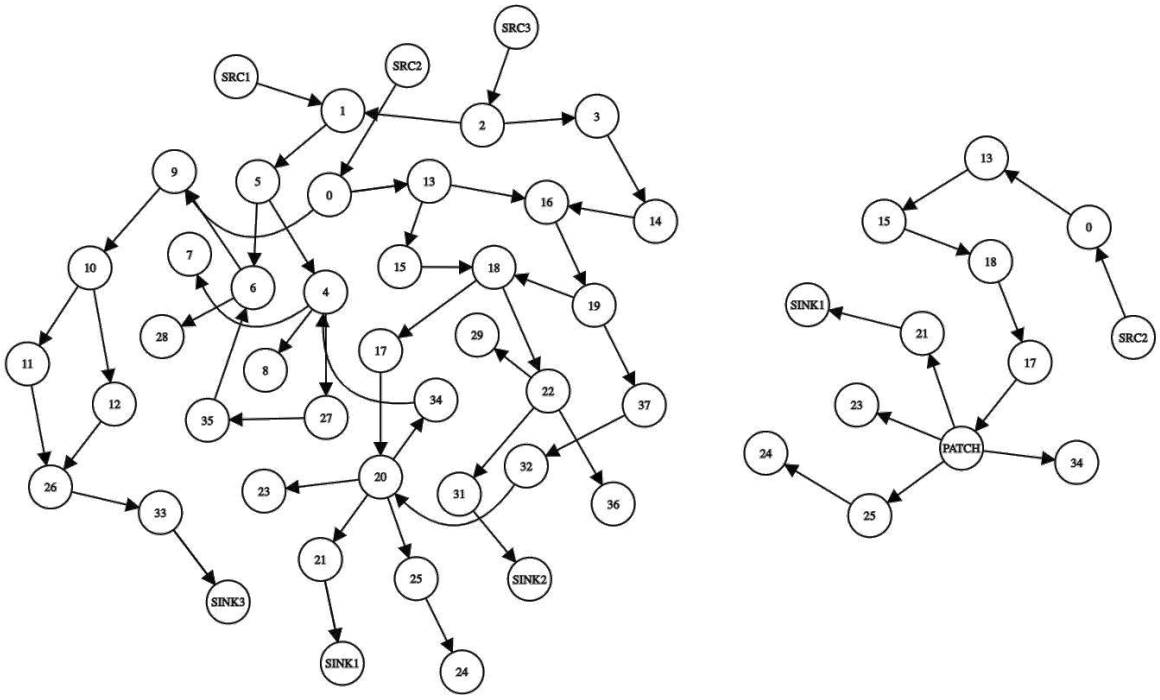


Figure 3.12: Hypothetical code graphs for a program. The left depicts an interprocedural whole-program graph, and the right its taint graph.

This not only emphasizes the necessity of context sensitivity, since the model has to analyze patches with respect to the entire application but also leaves us with a new set of nodes V_{EDIT} and we arrive at a new taint-style analysis problem as defined in Definition 17. Consider now the right graph in Figure 3.12 where we now arrive at a much smaller code graph compared to the left one. However, it still contains all necessary taint flows for a specific patch or commit.

Definition 17 *a) A taint-style analysis for vulnerable patch detection is a 4-tuple* $(V_{SOURCE}, V_{SINK}, V_{SAN}, V_{EDIT})$ *consisting of the nodes in a code graph of a program* $p \in \mathcal{P}$ *denoting the taint source, sink and sanitizer nodes from* V_A *[211] as well as the nodes corresponding to code that is changed or newly created in a patch* $[p' \Rrightarrow p]$.
b) We say a patch $[p' \Rrightarrow p]$ *contains a defect if there exists a vulnerable data or control flow*

between any $v_0 \in V_{SOURCE}$ and $v_1 \in V_{SINK}$ with the constraint of not reaching any defined sanitizer but intersecting with at least one node from V_{EDIT}.

Previous works have only conducted tangent research in statically finding patches that silently introduce vulnerability fixes, for example, the works by Wang et al. [199][200]. In essence, our vulnerable patch detection approach consists of three steps:

(1) Graph representation. We define a new interprocedural patch graph representation that can be built efficiently using incremental program analysis [188]. In contrast to current discovery models, interprocedural graph representations are arguably more beneficial since they enable us to propagate taint information within the entire program, which is impossible with a function, local slice, or file-level graph.

For this endeavor, we represent a program under analysis $p \in \mathcal{P}$ by an interprocedural code composite graph (CCG). We define taint paths as a first step towards the definition of taint graphs. For this, we select taint sources $V_{\text{SOURCE}} \subset V$ providing user input and taint sinks $V_{\text{SINK}} \subset V$ denoting critical code regions and the subset $V_{\text{EDIT}} \subset V$ of all nodes that have been edited in a specific patch.

Definition 18 *A single* taint path $\hat{p}$ *of a patch* $[p' \Rrightarrow p]$ *is an oriented path with vertices* $v_0, \ldots, v_b, \ldots v_e$ *starting at* $v_0 \in V_{SOURCE}$*, passing through* $v_b \in V_{EDIT}$ *and ending in* $v_e \in V_{SINK}$ *where all the edges are in* E_{IDFG}*,* E_{DFG} *or* E_{CFG}*.*

To obtain taint paths, we perform forward slicing from V_{EDIT} to V_{SINK} following any edge from the interprocedural data flow graph (IDFG), DFG or CFG while neglecting the edges from the AST and CG. This leaves us with a set of paths that describe the changed spots within a patch potentially flowing into critical sinks. Likewise, we perform backward slices from V_{EDIT} to V_{SOURCE}. Combining both sets of slices leaves us with a set of paths describing all flows, starting with user-defined inputs that intersect the patched locations and reach the critical sinks. To provide a holistic view of a patch, we arrive at the taint graph, as defined in Definition 19, by combining all taint paths and gluing them together at their patch intersections V_{EDIT}. It is trivial to see that this graph representation fits nicely into the definition of vulnerable patch detection in taint-style from Definition 17.

Definition 19 *A taint graph (TG) of a patch* $[p' \Rrightarrow p]$ *is defined as* G_{TG} *joining its taint paths* $\{\hat{p}^1, \hat{p}^2, \ldots, \hat{p}^k\}$ *at their common AST nodes, starting from* V_{SOURCE} *flowing through* V_{EDIT} *and reaching* V_{SINK}*.*

(2) Value-set analysis. As another improvement over recent discovery models, we calculate a set of values to track the variable domains on the graphs. This assists in reasoning about potential buffer bounds and sanitizations. More specifically, whether or not the value of a user-controlled variable or buffer length is bounded beneficially affects the model's decision.

Given G_D we can select any variable assignment $v_e \in V_D$ and find $(v_s, v_e) \in E_D$ where v_e reads from v_s. If v_s is a constant and v_e is a Boolean, Float, or Integer operation, we can evaluate v_e. If v_s is not a constant, we can find $(v, v_s) \in E_D$ and repeat. This eventually boils down to constant propagation and folding. If we can evaluate v_e, we attach the evaluated value

to the node. Otherwise, if the operation cannot be evaluated because, for instance, one data flow dependent on v_d of v_e relies on I/O or external API calls, we annotate v_e with v_d. Lastly, we find all expressions within surrounding conditional blocks that may act as invariants [204]. If within this conditional block, we find a variable that appears in a conditional of the form `<var> <comparison> <expression>`, we annotate its bounds with its value if it could be evaluated in the previous step. As an example, in Figure 3.6, we can assert that `len` is greater or equal to zero and hence has a lower bound of 0 in the remainder of the function from line 19.

(3) Causal GNN model. Finally, we use graph isomorphism network (GIN) layers, as in Equation (3.5), to train an inductive model to infer detection rules applied to taint graphs. Our model is especially suited for processing long input graphs due to skip-connections [209], self-attention [193], and the optimizable parameter ϵ determining the influence of current nodes relative to neighboring nodes. We also use the causal attention module from Section 3.3, enabling a fine-granular software defect localization.

$$h^{(l)} = \theta^{(l-1)} \left((A + (1+\epsilon) \times I) \times \text{RELU}(h^{(l-1)}) \right) \tag{3.5}$$

The performance of PAVUDI is assessed on the FFmpeg and Qemu vulnerability dataset from Zhou et al. [228] against different state-of-the-art function and slice-based SAST tools. Since, to the best of our knowledge, there is no prior tool for vulnerable patch detection, we adopt different strategies for the local detectors:

1. *Max* is a strategy in which the maximum value of all the prediction scores of the slices or functions is used.
2. *Mean* strategy averages every prediction score from slices or functions.
3. *Probability* describes the likelihood of the patch being vulnerable depending on its k components, similar to a system's failure probability.
4. *Isotonic Probability* uses an isotonic regressor [148] followed by the probability strategy.
5. *Commit* merges all code components changed within a patch together.

The different strategies have different effects on the FPs and TPs. If the score for the vulnerable function in the patch is smoothed out with the MEAN-Strategy, we have fewer TPs and FPs. The MAX-Strategy, on the other hand, may be too sensitive with functions that are slightly above the threshold, which results in a higher FP and TP, resulting in a lower precision but higher recall.

Result

Figure 3.13 shows that using a normal GIN with local code regions results in a performance equal to or close to the baseline models with only 40% of PAVUDI's original efficacy.
The CGIN layer enhances the performance significantly while using the taint whole-program paths contributes the most to the improved model performance. The bound information only yields negligible benefits of at most 2%.

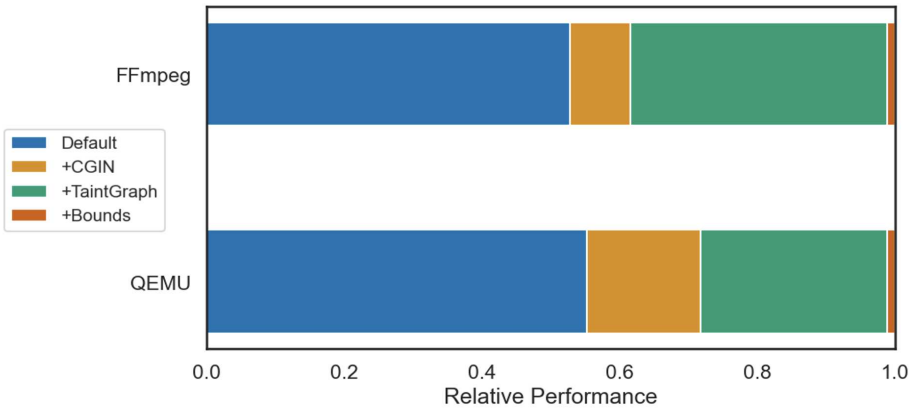


Figure 3.13: The performance contributions of all components of PAVUDI on the FFmpeg and the QEMU dataset.

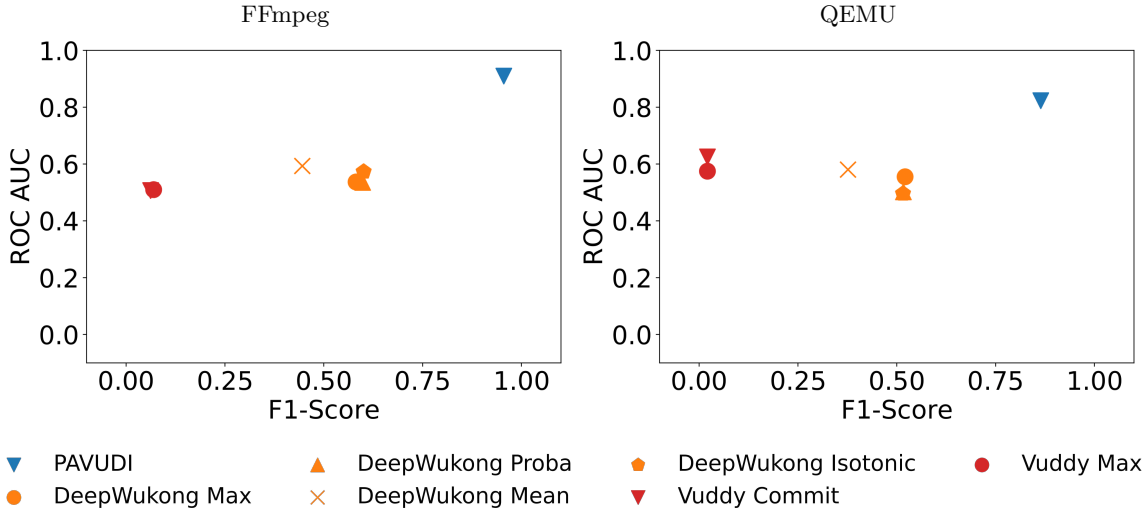


Figure 3.14: Performance comparison against DeepWukong and Vuddy.

We compare our approach against the seven baselines with the five aggregation strategies. In Figure 3.14, we can see that PAVUDI has a much higher area under receiver operating characteristic curve (AUROC) and F1-Score compared to Vuddy and DeepWukong. Since VUDDY is a deterministic method and not fine-tuned to our dataset, bad performance is expected. DeepWukong, however, is a sliced-based GNN approach and only achieves an F1-Score of 65% using the Isotonic-Strategy.

Figure 3.15 shows the result against other graph-based methods. Especially, BGNN4VD outperforms the previous token-based and slice-based methods with a AUROC of 80% and an F1 score of 75%. In all of our experiments, the use of the Mean-Strategy yields the worst scores. The Max and Commit strategies perform similarly poorly. Both increase the false positive rate too much, resulting in disadvantageous F1 and AUROC scores.

Given Figure 3.16, it is surprising that the simple token-based approach VulDeePecker with the Probability-Strategy achieves an AUROC of 79% on QEMU and FFmpeg beating SySEVR. Furthermore, we observe an overall beneficial score using the isotonic projection for all methods.

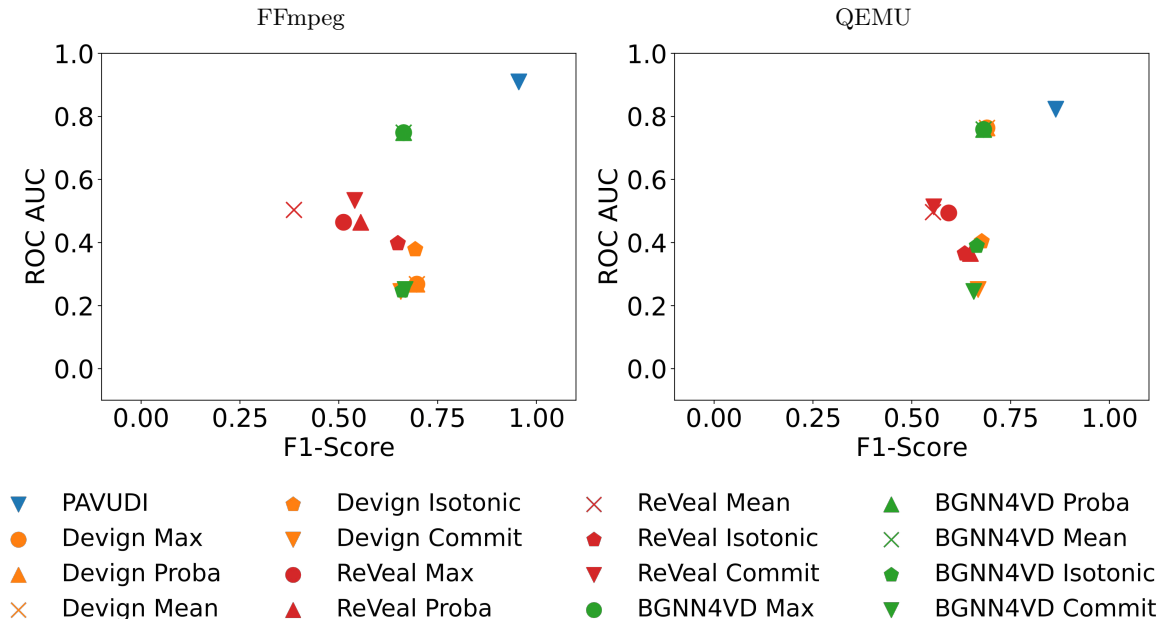


Figure 3.15: Performance comparison against Devign, ReVeal and BGNN4VD.

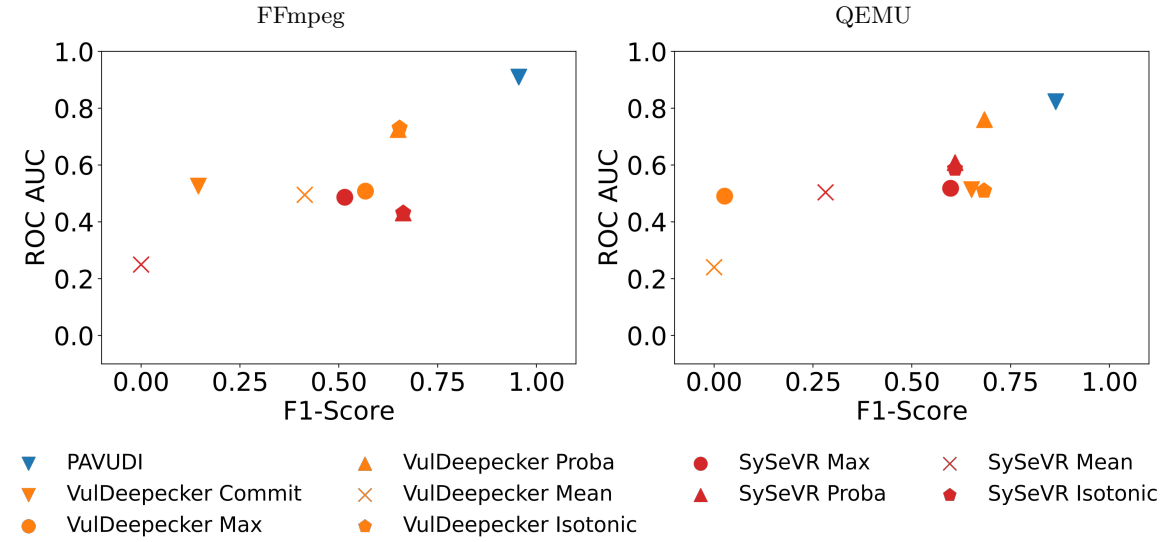


Figure 3.16: Performance comparison against VulDeePecker and SySEVR.

Our study has demonstrated that providing more context to learning-based models significantly enhances their detection capabilities. We recommend PAVUDI as the first whole-program learning-based taint analysis approach that can leverage large contexts without the problems of exploding feature spaces. While this approach may have some limitations, such as the requirement for pre-labeled sources and sinks, which may be prone to human error, we believe that this represents a significant step towards a new research direction for learning-based vulnerability discovery.

4

Conclusion

This thesis examines the limitations of current learning-based vulnerability discovery models and proposes enhancements to make them more suitable for realistic applications. We have seen four major limiting categories that hinder the adaption to real-world application, namely: insufficient, imbalanced, and low-quality data, a lack of model interpretation, unstable and nontransferable models, and finally, models that lack context. This chapter draws conclusions based on the issues and experimental results addressed.

Concluding Remarks Vulnerability discovery models achieve a high detection rate while pertaining to a low false positive rate in laboratory settings. However, the transfer to realistic scenarios has proven to be a more significant challenge [27, 30, 100]. This thesis proposes four potential culprits for impeding model generalization, that is, insufficient or low-quality training data, a lack of opaque model decision, confounding correlations, and missing context. We demonstrate CodeGraphSMOTE as a potential countermeasure for highly imbalanced datasets that lack *vulnerable* samples. Furthermore, we show that even though there exist many explanation methods to interpret ML models, they perform very differently, and popular benchmarking criteria suffer from the decoupling of the underlying vulnerability detection task. As a remedy, we propose to use dynamic program analysis for a more truthful comparison. Another major problem with current vulnerability detectors is that they tend to be rather input-sensitive and overfit on spurious correlations. Using causal learning methods, we can measure these confounding effects and effectively assess strategies that reduce the impediment. Lastly, all models have a limited context and hence may not learn more complex relationships in the program under analysis. We propose a new interprocedural whole-program graph that incorporates taint analysis to solve the vulnerable patch detection task. Considering all these problems and possible solutions drives us closer to models that are applicable to real-world secure software development life cycles.

Future Directions In the following, we want to discuss possible research directions. Current trends in machine learning, such as large language models or diffusion models, also foster research in (deep) learning-based vulnerability discovery. Large language models can process

larger token contexts compared to formerly used RNNs or LSTMs but are still limited. Novel approaches increase their processable context window [45] and their training and inference efficiency [40]. Diffusion models might provide better performances than variational auto-encoders in generating source code as they have already been adopted for natural language [231] or in generating code graphs [120].

Research in code representation for vulnerability detection tends to focus on graphs provided by intermediate representations from compilers, such as the LLVM-IR [136, 32]. This improves the data quality since it abstains from over-approximate graph representations such as the one from the fuzzy parser from Joern [211] or Fraunhofer-CPG [205]. Recent works also more and more incorporate dynamic analysis into the preprocessing stage as infeasible path removal [32]. It would be possible to add more and more sophisticated preprocessing steps, for instance, type deduction and annotation for small local samples if we lack context. In code gadgets, variables could be tainted and annotated, respectively.

We also see more work in local code region detection. Since it is obvious that vulnerable function classification provides too coarse-grained outputs that are barely helpful to human security experts, models like LineVul [55], LineVD [80], or Velvet [46] provide line-level or even statement-level results. This enables more interpretable outputs and less effort for manual validation and triage. However, the manifestation of vulnerability can differ from the predicted PoI. Patch locations might not correspond to the vulnerable location; hence, this can result in label inaccuracies. Very early research attempts to find vulnerable code clones by searching for similar code vector representations [213] or hashes of abstract code from historical vulnerabilities [95]. An exciting direction could be using high-performance vector databases [89] and latent representations by large code models [201] to efficiently find code that is very similar to historic CVEs. This yields a very scalable approach [66] and interpretable results since the predictions can be directly compared against historical vulnerabilities and mapped to concrete CWEs.

Learning-based vulnerability discovery currently assumes fixed data and a one-time trained model. In the future, with the increasing adoption in software development and delivery tools, it will be necessary to view it as a process rather than a static setting [141]. Thus, the data and model will change over time, which will pose a challenge for explainability and interpretability tools that have to adapt to these changes. Causal models [154] can be integrated into decision, data collection, and validation processes with better validation using domain knowledge by security practitioners.

References

[1] Douglas Adams. *The hitchhiker's guide to the galaxy*. Hitchhiker's Guide to the Galaxy. New York, NY: Random House, 2007.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. "Learning to Represent Programs with Graphs". In: *CoRR* (2017).

[4] Roberto Amadini, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. "Abstract Interpretation, Symbolic Execution and Constraints". In: *Recent Developments in the Design and Implementation of Programming Languages*. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[5] Leonhard Applis, Annibale Panichella, and Arie van Deursen. "Assessing Robustness of ML-Based Program Analysis Tools using Metamorphic Program Transformations". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2021.

[6] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. "Dos and Don'ts of Machine Learning in Computer Security". In: *CoRR* (2020).

[7] Andrei Arusoaie, Stefan Ciobaca, Vlad Craciun, Dragos Gavrilut, and Dorel Lucanu. "A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code". In: 2017.

[8] Federico Baldassarre and Hossein Azizpour. "Explainability Techniques for Graph Convolutional Networks". In: *CoRR* (2019).

[9] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. "A Survey of Symbolic Execution Techniques". In: *CoRR* (2016).

[10] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Transactions on Software Engineering* (2015).

[11] Sindre Beba and Magnus Melseth Karlsen. "Implementation Analysis of Open-Source Static Analysis Tools for Detecting Security Vulnerabilities". In: 2019.

[12] Vaishak Belle and Ioannis Papantonis. “Principles and Practice of Explainable Machine Learning”. In: *Frontiers in Big Data* (2021).

[13] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. “Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2016.

[14] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. “Neural Code Comprehension: A Learnable Representation of Code Semantics”. In: *CoRR* (2018).

[15] Guru Bhandari, Amara Naseer, and Leon Moonen. “CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software”. In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE ’21)*. ACM, 2021.

[16] Paul Black. *SARD: A Software Assurance Reference Dataset*. 1970.

[17] Gerardus Blokdyk. *ISO 15408 A Complete Guide - 2020 Edition*. 5starcooks, 2020.

[18] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. “Directed Greybox Fuzzing”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: Association for Computing Machinery, 2017.

[19] Josef Börcsök. *Funktionale Sicherheit: Grundzüge sicherheitstechnischer Systeme*. 2020.

[20] Kevin W. Bowyer, Nitesh V. Chawla, Lawrence O. Hall, and W. Philip Kegelmeyer. “SMOTE: Synthetic Minority Over-sampling Technique”. In: *CoRR* (2011).

[21] Trenton Bricken, Adly Templeton, Joshua Batson, Brian Chen, Adam Jermyn, Tom Conerly, Nick Turner, Cem Anil, Carson Denison, Amanda Askell, Robert Lasenby, Yifan Wu, Shauna Kravec, Nicholas Schiefer, Tim Maxwell, Nicholas Joseph, Zac Hatfield-Dodds, Alex Tamkin, Karina Nguyen, Brayden McLean, Josiah E Burke, Tristan Hume, Shan Carter, Tom Henighan, and Christopher Olah. “Towards Monosemanticity: Decomposing Language Models With Dictionary Learning”. In: *Transformer Circuits Thread* (2023).

[22] Simone Brugger-Gebhardt. *Die Din En ISO 9001:2015 Verstehen*. 2nd ed. Springer Gabler, 2016.

[23] Nadia Burkart and Marco F. Huber. “A Survey on the Explainability of Supervised Machine Learning”. In: *CoRR* (2020).

[24] Michael Burrows, Stephen N. Freund, and Janet L. Wiener. “Run-Time Type Checking for Binary Programs”. In: *Compiler Construction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[25] Sadullah Canakci, Nikolay Matyunin, Kalman Graffi, Manuel Egele, and Ajay Joshi. “TargetFuzz: Using DARTs to Guide Directed Greybox Fuzzers”. In: (2022).

[26] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. “BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection”. In: *Inf. Softw. Technol.* (2021).

[27] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. "Deep learning based vulnerability detection: Are we there yet". In: *IEEE Transactions on Software Engineering* (2021).

[28] Ray-Yaung Chang, Andy Podgurski, and Jiong Yang. "Discovering Neglected Conditions in Software by Mining Dependence Graphs". In: *IEEE Transactions on Software Engineering* (2008).

[29] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. "Hawkeye: Towards a Desired Directed Grey-Box Fuzzer". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018.

[30] Yizheng Chen, Zhoujie Ding, Xinyun Chen, and David A. Wagner. "DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection". In: *ArXiv* (2023).

[31] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. "DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network". In: *ACM Trans. Softw. Eng. Methodol.* (2021).

[32] Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. "Path-Sensitive Code Embedding via Contrastive Learning for Software Vulnerability Detection". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Virtual, South Korea: Association for Computing Machinery, 2022.

[33] Maria Christakis, Peter Müller, and Valentin Wüstholz. "Guiding Dynamic Symbolic Execution toward Unverified Program Executions". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 2016.

[34] Ciera Nicole Christopher. "Evaluating static analysis frameworks". In: *Analysis, pág* (2006).

[35] Roland Croft, M. Ali Babar, and M. Mehdi Kholoosi. "Data Quality for Software Vulnerability Datasets". In: ICSE '23. Melbourne, Victoria, Australia: IEEE Press, 2023.

[36] Balázs Csanád Csáji. *Approximation with Artificial Neural Networks*. Tech. rep. 2001.

[37] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, and Hugh Leather. *ProGraML: Graph-based Deep Learning for Program Optimization and Analysis*. 2020.

[38] Damien Dablain, Bartosz Krawczyk, and Nitesh Chawla. "DeepSMOTE: Fusing Deep Learning and SMOTE for Imbalanced Data". In: *IEEE Transactions on Neural Networks and Learning Systems* (2022).

[39] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. "Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context". In: *CoRR* (2019).

[40] Tri Dao. *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning*. 2023.

[41] Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A. Ortega. *Neural Networks and the Chomsky Hierarchy*. 2023.

[42] The GDB Developers. *GDB: The GNU Project Debugger*. 2023. URL: `https://www.sourceware.org/gdb/` (visited on 10/22/2023).

[43] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* (2018).

[44] Jens Dietrich, Li Sui, Shawn Rasheed, and Amjed Tahir. "On the Construction of Soundness Oracles". In: *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. SOAP 2017. Barcelona, Spain: Association for Computing Machinery, 2017.

[45] Lei Ding, Dong Lin, Shaofu Lin, Jing Zhang, Xiaojie Cui, Yuebin Wang, Hao Tang, and Lorenzo Bruzzone. "Looking Outside the Window: Wide-Context Transformer for the Semantic Segmentation of High-Resolution Remote Sensing Images". In: *IEEE Transactions on Geoscience and Remote Sensing* (2022).

[46] Yangruibo Ding, Sahil Suneja, Yunhui Zheng, Jim Laredo, Alessandro Morari, Gail Kaiser, and Baishakhi Ray. "VELVET: a noVel Ensemble Learning approach to automatically locate VulnErable sTatements". In: *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2022).

[47] Arthur Conan Doyle. *The Poison Belt*. 2017.

[48] Trevor Dunlap, Seaver Thorn, William Enck, and Bradley Reaves. "Finding Fixed Vulnerabilities with Off-the-Shelf Static Analysis". In: *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. 2023, pp. 489–505.

[49] Claudia Eckert. *It-Sicherheit*. 10th ed. de Gruyter Studium. de Gruyter, 2018.

[50] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. "A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries". In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR '20. Seoul, Republic of Korea: Association for Computing Machinery, 2020.

[51] Josselin Feist, Gustavo Grieco, and Alex Groce. "Slither: A Static Analysis Framework for Smart Contracts". In: *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 2019.

[52] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Trans. Program. Lang. Syst.* (1987).

[53] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. "AFL++: Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, 2020.

[54] Montgomery Flora, Corey Potvin, Amy McGovern, and Shawn Handler. *Comparing Explanation Methods for Traditional Machine Learning Models Part 1: An Overview of Current Methods and Quantifying Their Disagreement*. 2022.

[55] Michael Fu and Chakkrit Tantithamthavorn. “LineVul: A Transformer-based Line-Level Vulnerability Prediction”. In: *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE. 2022.

[56] Tom Ganz, Inaam Ashraf, Martin Härterich, and Konrad Rieck. “Detecting Backdoors in Collaboration Graphs of Software Repositories”. In: *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy*. CODASPY ’23. Charlotte, NC, USA: Association for Computing Machinery, 2023.

[57] Tom Ganz, Martin Härterich, Alexander Warnecke, and Konrad Rieck. “Explaining Graph Neural Networks for Vulnerability Discovery”. In: *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*. AISec ’21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021.

[58] Tom Ganz, Erik Imgrund, Martin Härterich, and Konrad Rieck. “CodeGraphSMOTE - Data Augmentation for Vulnerability Discovery”. In: *Data and Applications Security and Privacy XXXVII*. Cham: Springer Nature Switzerland, 2023.

[59] Tom Ganz, Erik Imgrund, Martin Härterich, and Konrad Rieck. “PAVUDI: Patch-based Vulnerability Discovery using Machine Learning”. In: *Proceedings of the 39th Annual Computer Security Applications Conference (ACSAC)*. New York, NY, USA: Association for Computing Machinery, 2023.

[60] Tom Ganz, Philipp Rall, Martin Härterich, and Konrad Rieck. “Hunting for Truth: Analyzing Explanation Methods in Learning-based Vulnerability Discovery”. In: *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. 2023.

[61] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. “Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey”. In: *ACM Comput. Surv.* (2017).

[62] Asem Ghaleb and Karthik Pattabiraman. “How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection”. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA ’20. ACM, 2020.

[63] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[64] Anjana Gosain and Ganga Sharma. “A Survey of Dynamic Program Analysis Techniques and Tools”. In: *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*. Cham: Springer International Publishing, 2015.

[65] D. Grahn and J. Zhang. “An Analysis of C/C++ Datasets for Machine Learning-Assisted Software Vulnerability Detection”. In: *Proceedings of the Conference on Applied Machine Learning for Information Security*. 2021.

[66] Luca Di Grazia and Michael Pradel. “Code Search: A Survey of Techniques for Finding Code”. In: *ACM Computing Surveys* (2023).

[67] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent 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. New Orleans, Louisiana, USA: Association for Computing Machinery, 2016.

[68] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. "Automatically Refining Abstract Interpretations". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[69] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. "LEMNA: Explaining Deep Learning Based Security Applications". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018.

[70] Guo Haixiang, Li Yijing, Jennifer Shang, Gu Mingyun, Huang Yuanyue, and Gong Bing. "Learning from class-imbalanced data: Review of methods and applications". In: *Expert Systems with Applications* (2017).

[71] William L Hamilton, Rex Ying, and Jure Leskovec. "Inductive representation learning on large graphs". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 2017.

[72] Nima Shiri Harzevili, Alvine Boaye Belle, Junjie Wang, Song Wang, Zhen Ming, Jiang, and Nachiappan Nagappan. *A Survey on Automated Software Vulnerability Detection Using Machine Learning and Deep Learning*. 2023.

[73] Nima Shiri Harzevili, Jiho Shin, Junjie Wang, Song Wang, and Nachiappan Nagappan. "Automatic Static Vulnerability Detection for Machine Learning Libraries: Are We There Yet?" In: *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023, Florence, Italy, October 9-12, 2023*. IEEE, 2023, pp. 795–806.

[74] Robert O. Hastings and Beverly A. Joyce. "Fast detection of memory leaks and access errors". In: *USENIX Summer 1992 Technical Conference*. USENIX Association, 1991.

[75] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. "Magma: A Ground-Truth Fuzzing Benchmark". In: *Proc. ACM Meas. Anal. Comput. Syst.* (2021).

[76] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. "Experience Report: System Log Analysis for Anomaly Detection". In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. 2016.

[77] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. "Generalizing Hindley-Milner Type Inference Algorithms". In: 2002.

[78] Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. "Semantic Robustness of Models of Source Code". In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022.

[79] Sophie Henning, William Beluch, Alexander Fraser, and Annemarie Friedrich. *A Survey of Methods for Addressing Class Imbalance in Deep-Learning Based Natural Language Processing*. 2023.

[80] David Hin, Andrey Kan, Huaming Chen, and Ali Babar. "LineVD: Statement-Level Vulnerability Detection Using Graph Neural Networks". In: *Proceedings of the 19th International Conference on Mining Software Repositories*. MSR '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022.

[81] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. "On the Naturalness of Software". In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012.

[82] Micha Horlboge, Erwin Quiring, Roland Meyer, and Konrad Rieck. *I still know it's you! On Challenges in Anonymizing Source Code*. 2022.

[83] Changcun Huang. "ReLU Networks Are Universal Approximators via Piecewise Linear or Constant Functions". In: *Neural Computation* (2020).

[84] 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)*. Los Alamitos, CA, USA: IEEE Computer Society, 2022.

[85] IMDB. *Carrie Fisher: Princess Leia Organa*. 2023. URL: `https://www.imdb.com/title/tt0076759/characters/nm0000402` (visited on 10/22/2023).

[86] IMDB. *Laurence Fishburne: Morpheus*. 2023. URL: `https://www.imdb.com/title/tt0133093/characters/nm0000401` (visited on 10/22/2023).

[87] Erik Imgrund, Tom Ganz, Martin Härterich, Niklas Risse, Lukas Pirch, and Konrad Rieck. "Broken Promises: Measuring Confounding Effects in Learning-based Vulnerability Discovery". In: *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISEC)*. AISec '23. Virtual Event, Republic of Korea: Association for Computing Machinery, 2023.

[88] Sarthak Jain and Byron 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, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics.

[89] Jeff Johnson, Matthijs Douze, and Hervé Jégou. "Billion-scale similarity search with GPUs". In: *IEEE Transactions on Big Data* (2019).

[90] Frederick Boland Jr. and Paul Black. *The Juliet 1.1 C/C++ and Java Test Suite*. 2012.

[91] Feras Al Kassar, Giulia Clerici, Luca Compagna, Davide Balzarotti, and Fabian Yamaguchi. "Testability Tarpits: the Impact of Code Patterns on the Security Testing of Web Applications". In: *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022.

[92] Evangelos Katsadouros and Charalampos Patrikakis. “A Survey on Vulnerability Prediction Using GNNs”. In: *Proceedings of the 26th Pan-Hellenic Conference on Informatics*. PCI ’22. Athens, Greece: Association for Computing Machinery, 2023.

[93] Tobias Keyser. “Security policy”. In: *The Information Governance Toolkit*. CRC Press, 2018.

[94] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. “Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code”. In: *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. Cambridge, MA: USENIX Association, 2007.

[95] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. “VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery”. In: *2017 IEEE Symposium on Security and Privacy (S&P)*. 2017.

[96] Johannes Kinder and Helmut Veith. “Jakstab: A Static Analysis Platform for Binaries”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[97] Thomas N. Kipf and Max Welling. “Semi-Supervised Classification with Graph Convolutional Networks”. In: *CoRR* (2016).

[98] György Kovács. “An empirical comparison and evaluation of minority oversampling techniques on a large number of imbalanced datasets”. In: *Applied Soft Computing* (2019).

[99] Dexter C. Kozen. “Rice’s Theorem”. In: *Automata and Computability*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1977.

[100] Padmanabhan Krishnan, Cristina Cifuentes, Li Li, Tegawendé F. Bissyandé, and Jacques Klein. “Why Is Static Application Security Testing Hard to Learn?” In: *IEEE Security & Privacy* (2023).

[101] Peter Ladkin. “Some Practical Issues in Statistically Evaluating Critical Software”. In: *Proceedings of the 10th IET System Safety and Cyber-Security Conference*. 2015.

[102] Guillaume Lample, Miguel Ballesteros, Kazuya Kawakami, Sandeep Subramanian, and Chris Dyer. “Neural Architectures for Named Entity Recognition”. In: *Proc. NAACL-HLT*. 2016.

[103] Sebastian Lapuschkin, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. “On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation”. In: *PLoS ONE* (2015).

[104] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. *Deep learning*. 2015.

[105] Nancy G Leveson. *Safeware*. Boston, MA: Addison Wesley, 1995.

[106] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. “BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension”. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, 2020.

[107] Qimai Li, Zhichao Han, and Xiao-Ming Wu. "Deeper Insights into Graph Convolutional Networks for Semi-Supervised Learning". In: *CoRR* (2018).

[108] Yaoxian Li, Shiyi Qi, Cuiyun Gao, Yun Peng, David Lo, Zenglin Xu, and Michael R. Lyu. *A Closer Look into Transformer-Based Code Intelligence Through Code Transformation: Challenges and Opportunities*. 2022.

[109] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. "Gated Graph Sequence Neural Networks". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. 2016.

[110] Yuwei Li, Shouling Ji, Chenyang Lv, Yuan Chen, Jianhai Chen, Qinchen Gu, and Chunming Wu. "V-Fuzz: Vulnerability-Oriented Evolutionary Fuzzing". In: *CoRR* (2019).

[111] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. "VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector". In: *CoRR* (2020).

[112] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, Zhaoxuan Chen, Sujuan Wang, and Jialai Wang. "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities". In: *CoRR* (2018).

[113] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection". In: *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[114] Jian Lin, Liehui Jiang, Yisen Wang, and Weiyu Dong. "A Value Set Analysis Refinement Approach Based on Conditional Merging and Lazy Constraint Solving". In: *IEEE Access* (2019).

[115] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. *A Survey of Transformers*. 2021.

[116] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. "Focal Loss for Dense Object Detection". In: *CoRR* (2017).

[117] Pantelis Linardatos, Vasilis Papastefanopoulos, and Sotiris Kotsiantis. "Explainable AI: A Review of Machine Learning Interpretability Methods". In: *Entropy* (2021).

[118] Maria B. Line, Odd Nordland, Lillian Røstad, and Inger Anne Tøndel. "Safety vs. Security? (PSAM-0148)". In: *Proceedings of the Eighth International Conference on Probabilistic Safety Assessment & Management (PSAM)*. ASME Press, 2006.

[119] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. "An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Virtual, South Korea: Association for Computing Machinery, 2022.

[120] Chengyi Liu, Wenqi Fan, Yunqing Liu, Jiatong Li, Hang Li, Hui Liu, Jiliang Tang, and Qing Li. *Generative Diffusion Models on Graphs: Methods and Applications*. 2023.

[121] Lili Liu, Zhen Li, Yu Wen, and Penglong Chen. "Investigating the impact of vulnerability datasets on deep learning-based vulnerability detectors". In: *PeerJ Computer Science* (2022).

[122] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019.

[123] LLVM. *Clang Analyzer*. 2003. URL: `https://clang-analyzer.llvm.org/` (visited on 10/29/2023).

[124] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. "A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models". In: *2009 International Symposium on Code Generation and Optimization*. 2009.

[125] Scott Lundberg and Su-In Lee. "A unified approach to interpreting model predictions". In: *CoRR* (2017).

[126] Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. *Parameterized Explainer for Graph Neural Network*. 2020.

[127] Linghui Luo. "Improving real-world applicability of static taint analysis." PhD thesis. University of Paderborn, Germany, 2021.

[128] Christopher M. *Pattern Recognition and Machine Learning*. 1st ed. Information Science and Statistics. New York, NY: Springer, 2006.

[129] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. "Directed Symbolic Execution". In: *Static Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[130] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* (2008).

[131] Valentin J. M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. *The Art, Science, and Engineering of Fuzzing: A Survey*. 2019.

[132] Daniel Marjamäki. *Cppcheck*. 2014. URL: `https://cppcheck.sourceforge.io/` (visited on 10/29/2023).

[133] Anders Alnor Mathiasen and Andreas Pavlogiannis. "The fine-grained and parallel complexity of andersen's pointer analysis". In: *Proc. ACM Program. Lang.* (2021).

[134] Nádia Medeiros, Naghmeh Ivaki, Pedro Costa, and Marco Vieira. "Vulnerable Code Detection Using Software Metrics and Machine Learning". In: *IEEE Access* (2020).

[135] Andrew Meneely and Laurie Williams. "Strengthening the Empirical Analysis of the Relationship between Linus' Law and Software Security". In: *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '10. Bolzano-Bozen, Italy: Association for Computing Machinery, 2010.

[136] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. "VulChecker: Graph-based Vulnerability Localization in Source Code". In: *USENIX Security*. 2023.

[137] Tom M Mitchell. *Machine learning*. McGraw-hill New York, 1997.

[138] MITRE. *Common Weakness Enumeration*. 2023. URL: `https://cwe.mitre.org/` (visited on 10/22/2023).

[139] Markus Mock. "Dynamic Analysis from the Bottom Up". In: *25th ICSE Workshop on Dynamic Analysis*. 2003.

[140] Christoph Molnar. *Interpretable Machine Learning. A Guide for Making Black Box Models Explainable*. 2nd ed. 2022.

[141] Christoph Molnar, Giuseppe Casalicchio, and Bernd Bischl. "Interpretable Machine Learning – A Brief History, State-of-the-Art and Challenges". In: *ECML PKDD 2020 Workshops*. Springer International Publishing, 2020.

[142] Grégoire Montavon, Alexander Binder, Sebastian Lapuschkin, Wojciech Samek, and Klaus-Robert Müller. "Layer-Wise Relevance Propagation: An Overview". In: *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*. Cham: Springer International Publishing, 2019.

[143] Tukaram Muske and Alexander Serebrenik. "Survey of Approaches for Postprocessing of Static Analysis Alarms". In: *ACM Comput. Surv.* (2022).

[144] Azqa Nadeem, Daniël Vos, Clinton Cao, Luca Pajola, Simon Dieck, Robert Baumgartner, and Sicco Verwer. "SoK: Explainable Machine Learning for Computer Security Applications". In: *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. 2023.

[145] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *SIGPLAN Not.* (2007).

[146] James Newsome and Dawn Song. "Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software." In: 2005.

[147] Minh-Tien Nguyen, Dung Tien Le, and Linh Le. "Transformers-based information extraction with limited data for domain-specific business documents". In: *Engineering Applications of Artificial Intelligence* (2021).

[148] Alexandru Niculescu-Mizil and Rich Caruana. "Predicting good probabilities with supervised learning". In: *ICML 2005 - Proceedings of the 22nd International Conference on Machine Learning*. 2005.

[149] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010.

[150] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. "CrossVul: A Cross-Language Vulnerability Dataset with Commit Data". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021.

[151] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. "Generating Realistic Vulnerabilities via Neural Code Editing: An Empirical Study". In: ESEC/FSE 2022. Singapore, Singapore, 2022.

[152] Peter O'Hearn, John Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures". In: *Computer Science Logic*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.

[153] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "ParmeSan: Sanitizer-guided Greybox Fuzzing". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[154] Judea Pearl. *Causality*. 2nd ed. Cambridge University Press, 2009.

[155] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. "VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015.

[156] PMD. *PMD - An extensible cross-language static code analyzer*. 2015. URL: `https://pmd.github.io/` (visited on 10/29/2023).

[157] Sebastian Poeplau and Aurélien Francillon. "Symbolic execution with SymCC: Don't interpret, compile!" In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[158] 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)*. 2019.

[159] Tenable Research. *TENABLE 2022 THREAT LANDSCAPE REPORT*. 2022. URL: `https://static.tenable.com/marketing/research-reports/Research-Report-2022_Threat_Landscape_Report.pdf` (visited on 10/22/2023).

[160] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier". In: *CoRR* (2016).

[161] Niklas Risse and Marcel Böhme. "Limits of Machine Learning for Automatic Vulnerability Detection". In: *arXiv e-prints*, arXiv:2306.17193 (2023).

[162] Marko A. Rodriguez and Peter Neubauer. "The Graph Traversal Pattern". In: *Graph Data Management: Techniques and Applications*. Ed. by Sherif Sakr and Eric Pardede. IGI Global.

[163] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. "Unsupervised Translation of Programming Languages". In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS'20. Vancouver, BC, Canada: Curran Associates Inc., 2020.

[164] Cynthia Rudin, Chaofan Chen, Zhi Chen, Haiyang Huang, Lesia Semenova, and Chudi Zhong. "Interpretable Machine Learning: Fundamental Principles and 10 Grand Challenges". In: *CoRR* (2021).

[165] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. "Automated Vulnerability Detection in Source Code Using Deep Representation Learning". In: *CoRR* (2018).

[166] A. L. Samuel. "Some Studies in Machine Learning Using the Game of Checkers". In: *IBM Journal of Research and Development* (1959).

[167] Benjamin Sanchez-Lengeling, Jennifer Wei, Brian Lee, Emily Reif, Peter Wang, Wesley Qian, Kevin McCloskey, Lucy Colwell, and Alexander Wiltschko. "Evaluating Attribution for Graph Neural Networks". In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2020.

[168] Thomas Schnake, Oliver Eberle, Jonas Lederer, Shinichi Nakajima, Kristof T. Schütt, Klaus-Robert Müller, and Grégoire Montavon. *Higher-Order Explanations of Graph Neural Networks via Relevant Walks*. 2020.

[169] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)". In: *2010 IEEE Symposium on Security and Privacy*. 2010.

[170] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. "Grad-cam: Visual explanations from deep networks via gradient-based localization". In: *Proceedings of the IEEE international conference on computer vision*. 2017.

[171] Semmle. *PMD - An extensible cross-language static code analyzer*. 2006. URL: `https://codeql.github.com/` (visited on 10/29/2023).

[172] Kosta Serebryany. "Continuous Fuzzing with libFuzzer and AddressSanitizer". In: *2016 IEEE Cybersecurity Development (SecDev)*. 2016.

[173] Julian Seward and Nicholas Nethercote. "Using Valgrind to Detect Undefined Value Errors with Bit-Precision". In: *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, 2005.

[174] Mingjie Shen, Akul Abhilash Pillai, Brian A. Yuan, James C. Davis, and Aravind Machiry. "An Empirical Study on the Use of Static Analysis Tools in Open Source Embedded Software". In: *CoRR* abs/2310.00205 (2023).

[175] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities". In: *IEEE Transactions on Software Engineering* (2011).

[176] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. "Learning Important Features Through Propagating Activation Differences". In: *CoRR* (2017).

[177] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda B. Viégas, and Martin Wattenberg. "SmoothGrad: removing noise by adding noise". In: *CoRR* (2017).

[178] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. "Striving for Simplicity: The All Convolutional Net". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings*. 2015.

[179] Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. "Generating Adversarial Computer Programs using Optimized Obfuscations". In: *International Conference on Learning Representations*. 2021.

[180] National Institute of Standards and Technology. *Software Vulnerability*. URL: `https://csrc.nist.gov/glossary/term/software_vulnerability` (visited on 11/14/2023).

[181] Statista. *Annual number of data compromises and individuals impacted in the United States from 2005 to 2022*. 2023. URL: `https://www.statista.com/statistics/273550/data-breaches-recorded-in-the-united-states-by-number-of-breaches-and-records-exposed/` (visited on 10/22/2023).

[182] Statista. *Average cost of a data breach worldwide from March 2022 to March 2023, by country or region*. 2023. URL: `https://www.statista.com/statistics/463714/cost-data-breach-by-country-or-region/` (visited on 10/22/2023).

[183] Statista. *Number of common IT security vulnerabilities and exposures (CVEs) worldwide from 2009 to 2023 YTD*. 2023. URL: `https://www.statista.com/statistics/500755/worldwide-common-vulnerabilities-and-exposures/` (visited on 10/22/2023).

[184] Yongduo Sui, Xiang Wang, Jiancan Wu, Min Lin, Xiangnan He, and Tat-Seng Chua. "Causal Attention for Interpretable and Generalizable Graph Classification". In: *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. KDD '22. Washington DC, USA: Association for Computing Machinery, 2022.

[185] Shiliang Sun, Zehui Cao, Han Zhu, and Jing Zhao. "A Survey of Optimization Methods From a Machine Learning Perspective". In: *IEEE Trans. Cybern.* 50.8 (2020), pp. 3668–3681.

[186] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. "Axiomatic Attribution for Deep Networks". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017.

[187] Tamás Szabó. "Incrementalizing Production CodeQL Analyses". In: *arXiv e-prints* (2023).

[188] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. "Incremental Whole-Program Analysis in Datalog with Lattices". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021.

[189] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. "Incremental Whole-Program Analysis in Datalog with Lattices". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021.

[190] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. "Transformer-Based Language Models for Software Vulnerability Detection". In: *Proceedings of the 38th Annual Computer Security Applications Conference*. ACSAC '22. Austin, TX, USA: Association for Computing Machinery, 2022.

[191] Erico Tjoa and Cuntai Guan. "A Survey on Explainable Artificial Intelligence (XAI): Towards Medical XAI". In: *CoRR* (2019).

[192] Andrea Tocchetti, Lorenzo Corti, Agathe Balayn, Mireia Yurrita, Philip Lippmann, Marco Brambilla, and Jie Yang. *A.I. Robustness: a Human-Centered Perspective on Technological Challenges and Opportunities*. 2022.

[193] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention is All you Need". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 2017, pp. 5998–6008.

[194] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. "Graph Attention Networks". In: *International Conference on Learning Representations* (2018).

[195] vin01. *Bogus CVEs (some of them at least ..)* 2023. URL: `https://github.com/vin01/bogus-cves` (visited on 10/22/2023).

[196] James Walden and Maureen Doyle. "SAVI: Static-Analysis Vulnerability Indicator". In: *IEEE Security & Privacy* (2012).

[197] 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". In: *IEEE Transactions on Information Forensics and Security* (2021).

[198] Song Wang, Taiyue Liu, and Lin Tan. "Automatically Learning Semantic Features for Defect Prediction". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: Association for Computing Machinery, 2016.

[199] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. "PatchDB: A Large-Scale Security Patch Dataset". In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2021.

[200] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. "PatchRNN: A Deep Learning-Based System for Security Patch Identification". In: *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*. 2021.

[201] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. "Codet5+: Open code large language models for code understanding and generation". In: *arXiv preprint arXiv:2305.07922* (2023).

[202] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation". In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, 2021.

[203] Alexander Warnecke, Daniel Arp, Christian Wressnegger, and Konrad Rieck. "Evaluating Explanation Methods for Deep Learning in Security". In: *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. Genoa, Italy: IEEE, 2020.

[204] Mark N. Wegman and F. Kenneth Zadeck. "Constant Propagation with Conditional Branches". In: *ACM Trans. Program. Lang. Syst.* (1991).

[205] Konrad Weiss and Christian Banse. *A Language-Independent Analysis Platform for Source Code*. 2022.

[206] David Wheeler. *Flawfinder*. 2013. URL: `https://dwheeler.com/flawfinder/` (visited on 10/29/2023).

[207] Sarah Wiegreffe and Yuval Pinter. "Attention is not not Explanation". In: *CoRR* (2019).

[208] Glynn Winskel. *The formal semantics of programming languages*. Foundations of Computing. London, England: MIT Press, 1993.

[209] Dongxian Wu, Yisen Wang, Shu-Tao Xia, James Bailey, and Xingjun Ma. *Skip Connections Matter: On the Transferability of Adversarial Examples Generated with ResNets*. 2020.

[210] Valentin Wüstholz and Maria Christakis. "Targeted Greybox Fuzzing with Static Lookahead Analysis". In: *CoRR* (2019).

[211] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. "Modeling and Discovering Vulnerabilities with Code Property Graphs". In: *2014 IEEE Symposium on Security and Privacy*. 2014.

[212] Fabian Yamaguchi. "Pattern-Based Vulnerability Discovery". PhD thesis. University of Göttingen, 2015.

[213] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. "Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning". In: *Proceedings of the 5th USENIX Conference on Offensive Technologies*. WOOT'11. San Francisco, CA: USENIX Association, 2011.

[214] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. "Chucky: Exposing Missing Checks in Source Code for Vulnerability Discovery". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. Berlin, Germany: Association for Computing Machinery, 2013.

[215] Wenjie Yang, Houjing Huang, Zhang Zhang, Xiaotang Chen, Kaiqi Huang, and Shu Zhang. "Towards Rich Feature Discovery With Class Activation Maps Augmentation for Person Re-Identification". In: (2019).

[216] Zhou Yang, Jieke Shi, Junda He, and David Lo. "Natural Attack for Pre-Trained Models of Code". In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022.

[217] Ziyu Yao, Frank F. Xu, Pengcheng Yin, Huan Sun, and Graham Neubig. "Learning Structural Edits via Incremental Tree Transformations". In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[218] Noam Yefet, Uri Alon, and Eran Yahav. "Adversarial Examples for Models of Code". In: *Proc. ACM Program. Lang.* (2020).

[219] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. "GNN Explainer: A Tool for Post-hoc Explanation of Graph Neural Networks". In: *CoRR* (2019).

[220] Hao Yuan, Haiyang Yu, Shurui Gui, and Shuiwang Ji. "Explainability in Graph Neural Networks: A Taxonomic Survey". In: *CoRR* (2020).

[221] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. "How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines". In: 2017.

[222] Peng Zeng, Gunjun Lin, Lei Pan, and Tai Yonghang. "Software Vulnerability Analysis and Discovery Using Deep Learning Techniques: A Survey". In: *IEEE Access* (2020).

[223] Huangzhao Zhang, Zhiyi Fu, Ge Li, Lei Ma, Zhehao Zhao, Hua'an Yang, Yizhe Sun, Yang Liu, and Zhi Jin. "Towards Robustness of Deep Program Processing Models—Detection, Estimation, and Enhancement". In: (2022).

[224] Huangzhao Zhang, Zhuo Li, Ge Li, L. Ma, Yang Liu, and Zhi Jin. "Generating Adversarial Examples for Holding Robustness of Source Code Processing Models". In: *AAAI Conference on Artificial Intelligence*. 2020.

[225] Jianming Zhang, Sarah Adel Bargal, Zhe Lin, Jonathan Brandt, Xiaohui Shen, and Stan Sclaroff. "Top-Down Neural Attention by Excitation Backprop". In: *Int. J. Comput. Vis.* 126 (2018), pp. 1084–1102.

[226] Tianxiang Zhao, Xiang Zhang, and Suhang Wang. "GraphSMOTE: Imbalanced Node Classification on Graphs with Graph Neural Networks". In: *Proceedings of the 14th ACM International Conference on Web Search and Data Mining* (2021).

[227] Weining Zheng, Yuan Jiang, and Xiaohong Su. "VulSPG: Vulnerability detection based on slice property graph representation learning". In: *32nd IEEE International Symposium on Software Reliability Engineering, ISSRE 2021, Wuhan, China, October 25-28, 2021*. IEEE, 2021, pp. 457–467.

[228] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 2019, pp. 10197–10207.

[229] Xiaogang Zhu, Shigang Liu, Xian Li, Sheng Wen, Jun Zhang, Seyit Ahmet Çamtepe, and Yang Xiang. "DeFuzz: Deep Learning Guided Directed Fuzzing". In: *CoRR* (2020).

[230] Deqing Zou, Yawei Zhu, Shouhuai Xu, Zhen Li, Hai Jin, and Hengkai Ye. "Interpreting Deep Learning-Based Vulnerability Detector Predictions Based on Heuristic Searching". In: *ACM Trans. Softw. Eng. Methodol.* (2021).

[231] Hao Zou, Zae Myung Kim, and Dongyeop Kang. *A Survey of Diffusion Models in Natural Language Processing*. 2023.

A

Supplementary Material

SAST is undecidable

We show that b from Definition 2 is undecidable using Rice's Theorem. First, b is a nontrivial property, since we have $\exists p_{\text{VULNERABLE}} \in \mathcal{P}$ and $\exists p_{\text{CLEAN}} \in \mathcal{P}$.

We can further reduce the problem to the undecidability of the halting Problem. Assume that there exists an algorithm f as in Definition 2 or Definition 9 that determines whether a program $p \in \mathcal{P}$ has property b.

Proof: Consider a program $O(p, w)$ that accepts a program $p \in \mathcal{P}$ and an input w. Also, without loss of generality, consider a program $p_{\text{VULNERABLE}} \in \mathcal{P}$ that contains a defect and terminates. Now define O as following:

Algorithm 2: Oracle O deciding the halting problem

Input: Program $p \in \mathcal{P}$, Input w
Output: True if p halts. False otherwise

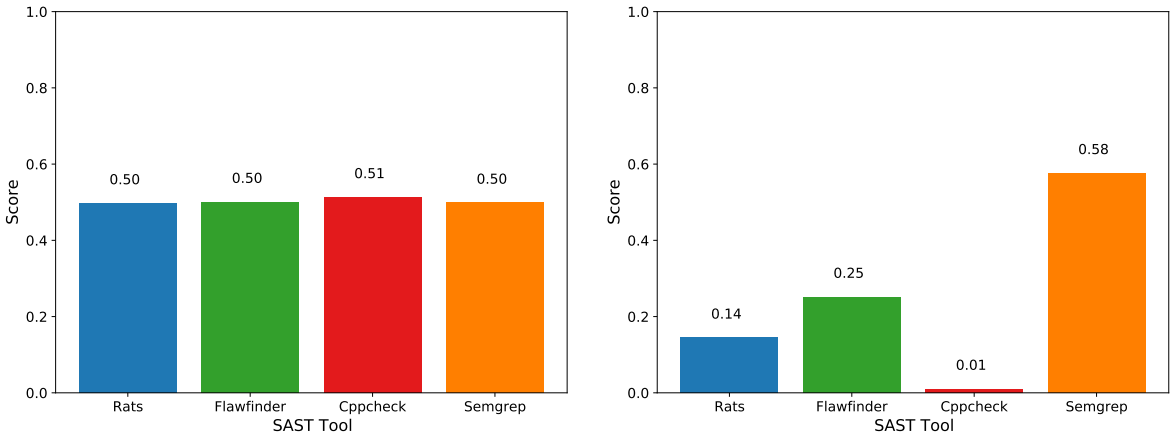
```
1 Procedure O(p, w):
2     Create a new program p' that simulates p(w) followed by p_VULNERABLE;
3     Return f(p');
```

If p halts, p' will contain a reachable defect, and O returns *true*. If p diverges, it will not contain a reachable defect, and O will return *false*. Since b is assumed to be decidable, this implies that we can use f to solve the halting problem, which is a contradiction. □

Rule-based SAST Tool Evaluation

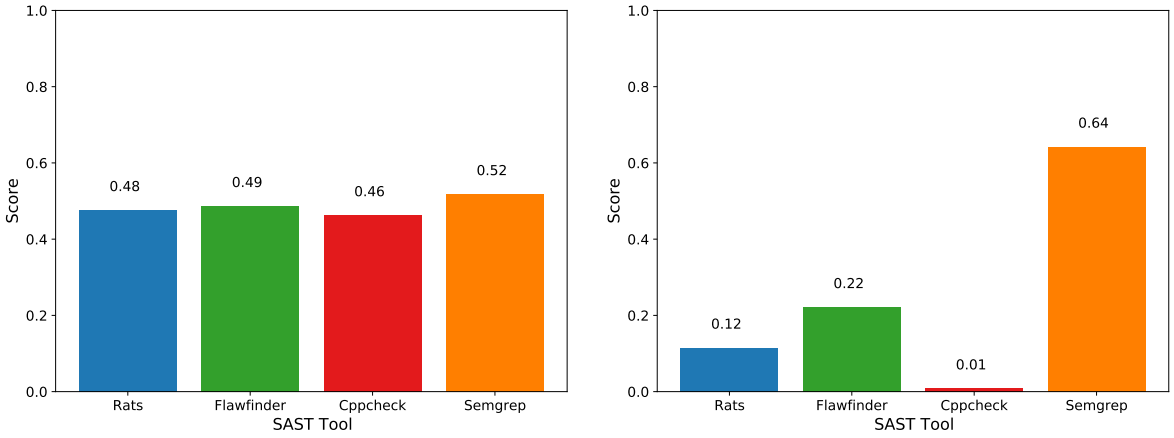
This section provides a brief evaluation of rule-based static application security testing (SAST) tools, focusing on precision and recall metrics for datasets that are specifically designed for training learning-based vulnerability discovery models. The precision and recall plots illustrate the effectiveness of *Flawfinder* (lexical), *Rats*, and *Cppcheck* (syntactical), as well as *Semgrep*

(semantical[1]) SAST tools in identifying vulnerabilities. Each set of subfigures corresponds to a specific dataset, including *BigVul*, *CVEFixes*, *Devign*, *DiverseVul*, and *ReVeal*. Precision plots emphasize soundness, while recall plots depict completeness. Notably, the semantical analyzer achieves the best trade-off between soundness and completeness. *Cppcheck* sacrifices completeness for soundness, with a consistently low recall score, while its precision remains comparable to the other tools. *Rats* scores the lowest in this evaluation. As a sidenote, all rule-based SAST tools perform worse than the learning-based vulnerability detectors from the respective publications.



(a) The precision. **(b)** The recall.

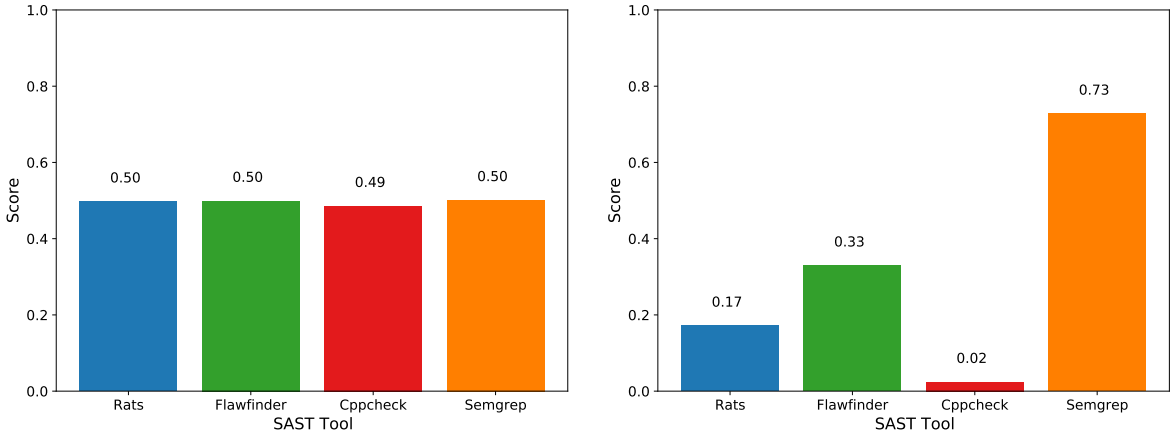
Figure A.1: The performance comparison on BigVul [50].



(a) The precision. **(b)** The recall.

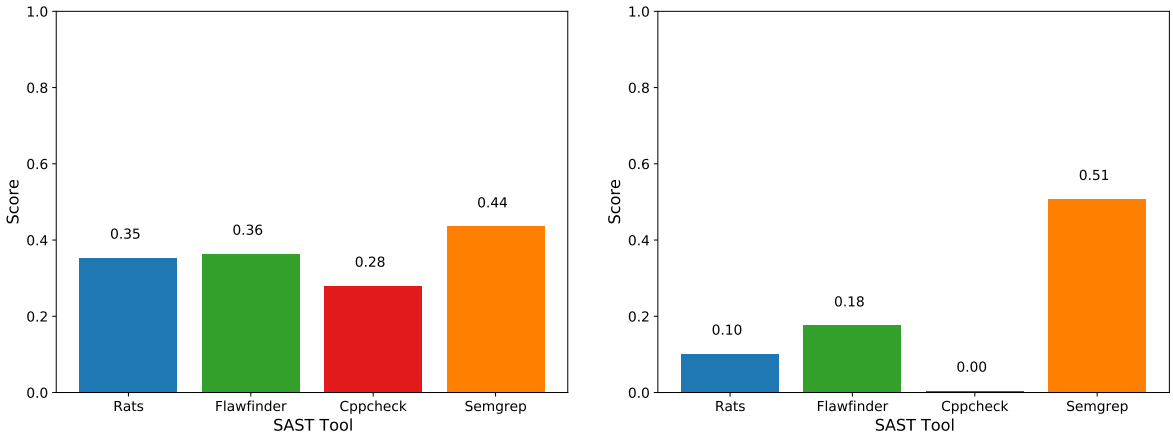
Figure A.2: The performance comparison on CVEFixes [15].

[1]Rules are taken from here https://github.com/anon767/semgrep-rules



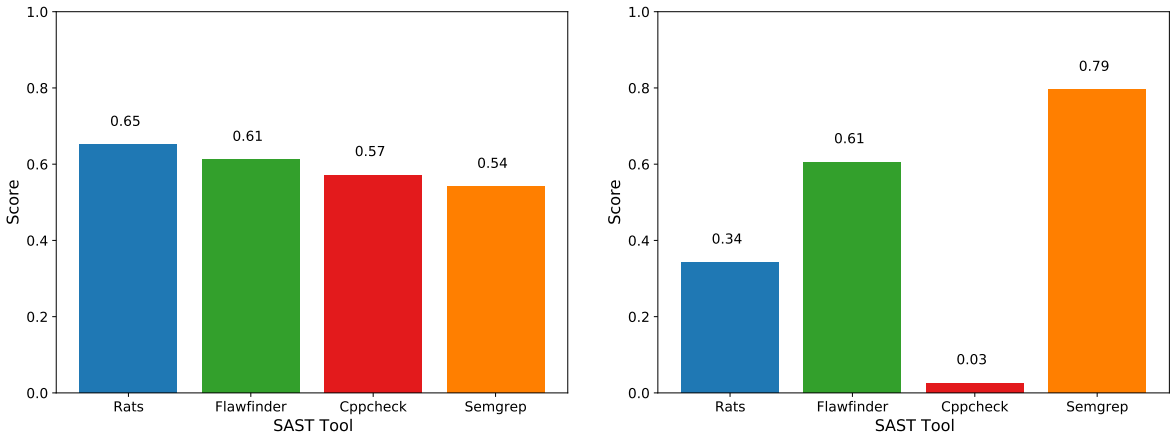
(a) The precision. **(b)** The recall.

Figure A.3: The performance comparison on Devign [228].



(a) The Precision. **(b)** The Recall.

Figure A.4: The performance comparison on DiverseVul [30].

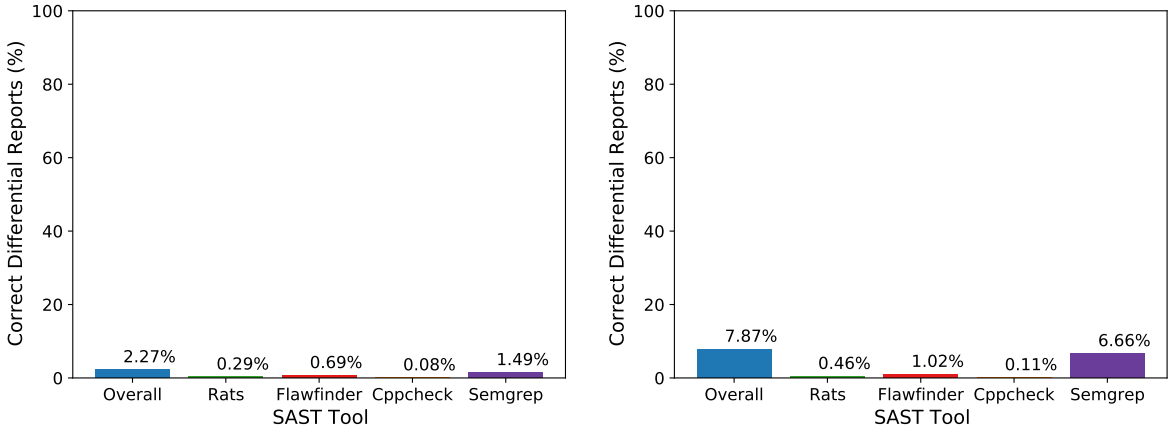


(a) The precision. **(b)** The recall.

Figure A.5: The performance comparison on ReVeal [27]

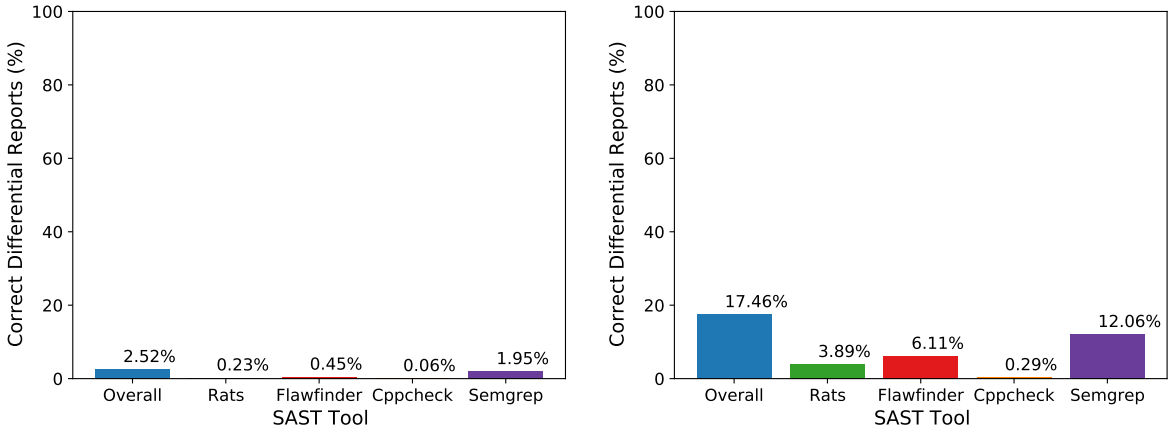
Differential SAST Dataset Analysis

In this section, a differential analysis of datasets is provided [48]. Differential data analysis is conducted using four SAST tools. Most vulnerability datasets consist of security patches. A *vulnerable* sample corresponds to the code before the patch, and a *clean* version represents the code after the patch. A higher ability of the SAST tools to identify patches, indicating their detection of bug removal or fixes, denotes a better-quality dataset. The code analysis plots offer visualizations for various datasets, namely *BigVul*, *CVEFixes*, *Devign*, *DiverseVul*, and *ReVeal*. Each subfigure provides insights into the differential analysis results for the corresponding dataset. The *Overall* score measures combined SAST performance, considering a hit whenever at least one SAST tool reported a fixed bug. Clearly, ReVeal achieves the highest score, directly followed by *DiverseVul*. The other datasets have lower scores, suggesting that these datasets may include more patches that are not directly security-relevant. Since no SAST tool is perfect, this, however, only gives us a lower bound.



(a) BigVul **(b)** CVEFixes

Figure A.6: Differential Analysis for BigVul [50] and CVEFixes [15].



(a) Devign **(b)** DiverseVul

Figure A.7: Differential Analysis for Devign [228] and DiverseVul [30].

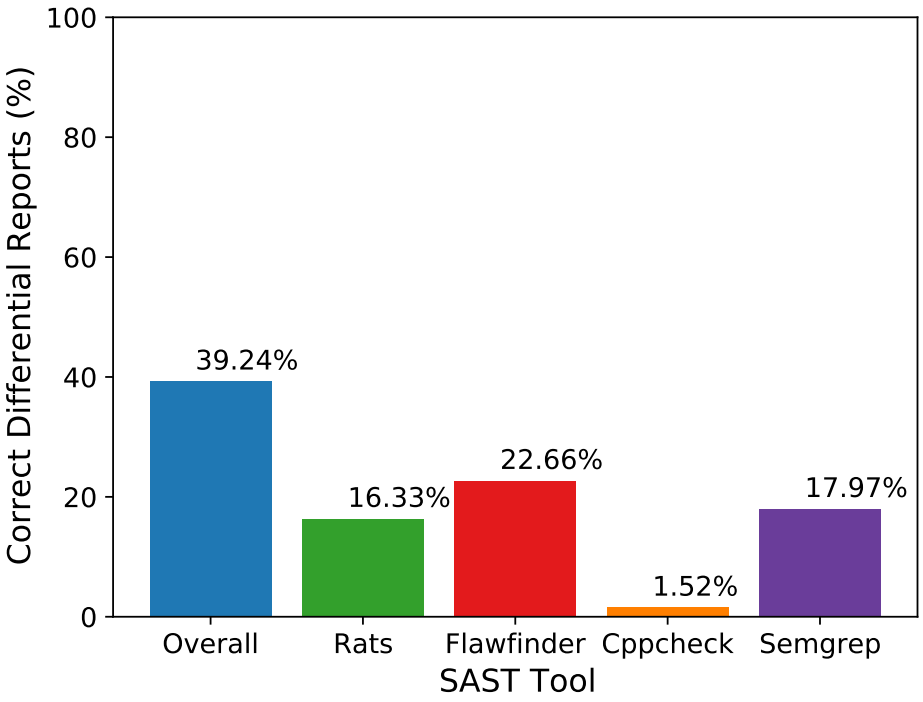


Figure A.8: Differential Analysis for ReVeal [27].

B

Publications

Publications in this Thesis

1. CodeGraphSMOTE: Data Augmentation for Vulnerability Discovery. Tom Ganz, Erik Imgrund, Martin Härterich and Konrad Rieck. Proc. of the IFIP Conference on Data and Applications Security and Privacy (DBSEC), 2023. Published version (https://doi.org/10.1007/978-3-031-37586-6_17). Material from: 'Ganz, T., Imgrund, E., Härterich, M., Rieck, K., CodeGraphSMOTE - Data Augmentation for Vulnerability Discovery, Data and Applications Security and Privacy XXXVII. DBSec 2023. Lecture Notes in Computer Science, vol 13942. published 2023, publisher - Springer'.

2. Explaining Graph Neural Networks for Vulnerability Discovery. Tom Ganz, Martin Härterich, Alexander Warnecke and Konrad Rieck. Proc. of the 14th ACM Workshop on Artificial Intelligence and Security (AISEC), 2021. Accepted manuscript (https://doi.org/10.1145/3474369.3486866). © ACM 2021. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in Proc. of the 14th ACM Workshop on Artificial Intelligence and Security (AISEC), https://doi.org/10.1145/3474369.3486866.

3. Hunting for Truth: Analyzing Explanation Methods in Learning-based Vulnerability Discovery. Tom Ganz, Philipp Rall, Martin Härterich and Konrad Rieck. Proc. of the 8th IEEE European Symposium on Security and Privacy (Euro S&P), 2023. Accepted manuscript (https://doi.org/10.1109/EuroSP57164.2023.00038). © 2023 IEEE. Reprinted, with permission, from Tom Ganz, Philipp Rall, Martin Härterich and Konrad Rieck, Hunting for Truth: Analyzing Explanation Methods in Learning-based Vulnerability Discovery, Proc. of the 8th IEEE European Symposium on Security and Privacy (Euro S&P), 7/2024.

4. Broken Promises: Measuring Confounding Effects in Learning-based Vulnerability Discovery. Erik Imgrund, Tom Ganz, Martin Härterich, Niklas Risse, Lukas Pirch and Konrad Rieck. Proc. of the 16th ACM Workshop on Artificial Intelligence and

Security (AISEC), 2023. Published version (https://doi.org/10.1145/3605764.3623915). This article is licensed under a Creative Commons license (CC-BY 4.0, https://creativecommons.org/licenses/by/4.0/).

5. PAVUDI: Patch-based Vulnerability Discovery using Machine Learning. Tom Ganz, Erik Imgrund, Martin Härterich and Konrad Rieck. Proc. of the 39th Annual Computer Security Applications Conference (ACSAC), 2023. Accepted manuscript (https://doi.org/10.1145/3627106.3627188). © ACM 2023. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in Proc. of the 39th Annual Computer Security Applications Conference (ACSAC), https://doi.org/10.1145/3627106.3627188.

Other Publications

6. Detecting Backdoors in Collaboration Graphs of Software Repositories. Tom Ganz, Inaam Ashraf, Martin Härterich and Konrad Rieck. Proc. of the 14th ACM Conference on Data and Applications Security and Privacy (CODASPY), 2023.

Supervised Theses

1. Erik Imgrund, DHBW Karlsruhe 2022, Bachelor of Science: Implementing and Evaluating Data Augmentation Techniques for Geometric Deep Learning.

2. Inaam Ashraf Muhammad, University of Hildesheim 2021, Master of Science: Anomaly detection in the software development process using Graph Neural Networks based on version history metadata and collaboration graphs.

3. Konrad Hartwig, DHBW Karlsruhe 2021, Bachelor of Science: Explainability of outputs from graph neural networks and their meaningfulness.

Open Source Software

Here, we provide a list of open-source software tools developed within the scope of our research endeavor. Each repository is accompanied by a brief description of its purpose and functionality.

1. Detecting Backdoors in Collaboration Graphs of Software Repositories[1]
   This repository contains a model and dataset for analyzing Git commits and finding anomalies. It can be used for finding backdoors or unusual development practices [56].

2. Taintgraph Extraction[2]
   This open-source tool is able to extract taint graphs and code composite graphs from C and C++ software projects. It also contains the patch-based vulnerability discovery (PAVUDI) model [59].

[1] https://github.com/SAP-samples/security-research-commit-anomaly-detection
[2] https://github.com/SAP-samples/security-research-taintgraphs

3. CodeGraphSMOTE - Data Augmentation for Vulnerability Discovery[3]
   This repository contains everything needed to reproduce our CodeGraphSMOTE results, including the model and baselines.

4. Cruzzer - A coverage-guided Webapplication Fuzzer[4]
   Cruzzer can be used to fuzz web applications by crawling HTML form fields and using coverage information from the backend for PHP and Java applications.

5. Java-Flow-Analyzer[5]
   Java Flow Analyzer is a tool written in Rust to find configurable data or control flow between sources and sinks within Java applications.

[3]https://github.com/SAP-samples/security-research-codegraphsmote
[4]https://github.com/SAP-samples/security-research-cruzzer
[5]https://github.com/anon767/Java-Flow-Analyzer

C

CodeGraphSMOTE - Data Augmentation for Vulnerability Discovery

CodeGraphSMOTE - Data Augmentation for Vulnerability Discovery

Tom Ganz[1(✉)], Erik Imgrund[1], Martin Härterich[1], and Konrad Rieck[2]

[1] SAP Security Research, Walldorf, Germany
{tom.ganz,erik.imgrund,martin.harterich}@sap.com
[2] Technische Universität Berlin, Berlin, Germany
rieck@tu-berlin.de

Abstract. The automated discovery of vulnerabilities at scale is a crucial area of research in software security. While numerous machine learning models for detecting vulnerabilities are known, recent studies show that their generalizability and transferability heavily depend on the quality of the training data. Due to the scarcity of real vulnerabilities, available datasets are highly imbalanced, making it difficult for deep learning models to learn and generalize effectively. Based on the fact that programs can inherently be represented by graphs and to leverage recent advances in graph neural networks, we propose a novel method to generate synthetic code graphs for data augmentation to enhance vulnerability discovery. Our method includes two significant contributions: a novel approach for generating synthetic code graphs and a graph-to-code transformer to convert code graphs into their code representation. Applying our augmentation strategy to vulnerability discovery models achieves the same originally reported F1-score with less than 20% of the original dataset and we outperform the F1-score of prior work on augmentation strategies by up to 25.6% in detection performance.

Keywords: Vulnerability Discovery · Data Augmentation · Graph Neural Networks

1 Introduction

The research in the field of automatic vulnerability discovery has made remarkable progress recently but is still far from complete. Traditional rule-based tools suffer from high false negative or false positive rates in their detection performance. Consequently, advances in deep learning spark interest in the development of learning-based vulnerability discovery models. For instance, recent models borrow techniques from natural language processing using recurrent neural networks (RNNs), in particular, long short-term memorys (LSTMs), where the source code is processed as a flat sequence of code tokens [6,30,31,38]. Even more recent approaches use graph neural networks (GNNs) thereby leveraging code graphs as a compact structure to represent the syntactic and semantic properties

Published by Springer Nature Switzerland AG 2023
V. Atluri and A. L. Ferrara (Eds.): DBSec 2023, LNCS 13942, pp. 282–301, 2023.
https://doi.org/10.1007/978-3-031-37586-6_17

of programs [6,10,43,55]. Graph learning is still a young field with a big room for improvement, but a promising technique to foster further research, especially in software security [15].

However, one major obstacle in learning-based vulnerability discovery, is obtaining enough representative code samples since most datasets available are either too small, unrealistic or imbalanced [6,34,44]. While clean code samples are vastly available and can be gathered easily, vulnerable samples, on the other hand, are scarce [2]. GNNs architectures suffer under that shortage the most, as they tend to overfit very easily and hence need balanced labels for training. Chakraborty et al. [6] report that models trained on inappropriate and imbalanced datasets are less transferable and have disadvantageous detection capabilities. The question arises then, on how to apply vulnerability discovery models to projects that lack a large history of vulnerabilities.

In traditional machine learning domains, data augmentation techniques are commonplace: For image data, random crops, offsets and rotations generate slightly different images with the same underlying meaning [40]. In tabular data, Synthetic Minority Oversampling Technique (SMOTE) is used to interpolate between minority samples and thus generate new samples [7]. Natural language processing uses techniques, such as synonym replacement, random word swaps, deletions or back translation [39]. While graph-based deep learning provides a unified method for neural networks on grids, groups, geodesics and gauges, no augmentation method for full graphs and even less so for code graphs exists.

Although augmentation techniques for node-level [54] and edge-level [52] tasks are available, techniques for graph classification are still unexplored [51]. The graph-specific augmentation methods that have been developed so far, either only perturb graphs [32,51], cannot generate new graphs with node attributes of the target domain [21] or only perturb the node attributes [26]. Even worse, these augmentation strategies are disconnected from the underlying vulnerability discovery task, causing the generated samples to be neither syntactically nor semantically correct rendering them effectively uninterpretable.

More promising approaches like Graph2Edit [50], SequenceR [9] or Hoppity [12] can generate new vulnerable samples by learning semantic edits applied to clean code samples [34]. Although they are better suited for balancing vulnerability datasets than random graph perturbations, they already require a large number of vulnerable samples for training, which is the problem we are trying to solve in the first place. Furthermore, Nong et al. [34] observe that neural code editing approaches for vulnerability injection only yield significant improvements if the generated samples are assessed and selected and thus require extensive manual labor. Hence, we need data augmentation strategies explicitly tailored for vulnerability discovery which do not require large amounts of vulnerable training data and produce human-readable code. We present CodeGraphSMOTE, a novel method to augment code graph samples for vulnerability discovery models. It generates new vulnerable samples for the minority class in a dataset by porting SMOTE to the graph domain, specifically for code graphs. It does so by interpolating in the latent space of a variational autoencoder.

Our approach focuses on interpretable and sound sample generation. In essence, the contributions we present are:

1. A novel method to generate sound and interpretable synthetic code graphs.
2. A graph-to-code transformer to translate code graphs back to source code.
3. An evaluation demonstrating the practicability of our method.

Moreover, we publish our implementation of this method to foster further research in this direction[1]. In the rest of this paper, we review Related Work in Sect. 2. Then we lay down the preliminaries for vulnerability discovery in Sect. 3 and for data augmentation in Sect. 4. We proceed to thoroughly describe our method in Sect. 5, then present our experimental evaluation in Sect. 6 and end with the Conclusion in Sect. 7.

2 Related Work

Some graph-specific data augmentation methods perturbing the given samples have been developed, while graph data augmentation methods that are extending the dataset by generating new graphs are heavily underdeveloped.

DropEdge [37] reduces overfitting and over-smoothing by removing random edges from the graph at training time, and several improvements over DropEdge have been made by choosing the dropped edges in a biased way [16,41]. Other methods are based on adding and removing edges [8,53], masking node attributes [56], sampling a random subset of the nodes [13,20] and cropping subgraphs [45]. DeepSMOTE interpolates images in the latent space of an autoencoder instead of the original pixel space.

This greatly improves downstream classification performance for imbalanced datasets by generating synthetic minority samples and works better than generating new samples based on generative adversarial networks [11]. The same idea is applied to graphs to generate new nodes for imbalanced node classification tasks in [54]. There, a GAE is trained to reconstruct the adjacency matrix, while simultaneously learning latent features of the edges. The nodes are then oversampled using SMOTE in the latent space obtained by the GAE, which is also used to generate edges connecting the new nodes with the rest of the graph. This method achieves better accuracy in the task of imbalanced node classification. However, no adaptation of this method has been published, that interpolates between graphs to be used in graph classification. Chakraborty et al. [6] already apply SMOTE on graph embeddings before using a vulnerability classifier, this was found to increase detection performance. However, their method is generally, hardly applicable since it uses intermediate representations from another vulnerability discovery model and does not reconstruct interpolated graphs let alone the underlying source code.

Other approaches have been proposed from different research branches. Neural code editing uses deep learning to generate syntactically and semantically

[1] https://github.com/SAP-samples/security-research-codegraphsmote.

valid code samples. Different approaches, for instance, Hoppity [12], SequenceR [9] and Graph2Edit [50] have been developed. A recent study found out, that these approaches do not work well for augmenting vulnerability datasets [34]. Furthermore, the generated additional samples by Graph2Edit were found to be unrealistic but still helpful as additional training data for a vulnerability discovery downstream task. Lastly, Evilcoder [36] allows for automatically inserting bugs using rule-based code modifications, for instance, modified/removed buffer checks. However, this method produces vulnerable code samples that are too trivial to distinguish from clean samples and hence not suitable for machine learning.

3 Vulnerability Discovery

The preliminary materials for our method concerning learning-based vulnerability discovery, program representations and representation learning on code graphs are discussed in this section.

3.1 Learning-Based Static Analyzer

We start by formalizing the vulnerability discovery task in the following section: Given a particular representation of a program, a static vulnerability discovery method is a decision function f that maps a code snippet x to a label $y \in \{\text{VULNERABLE}, \text{CLEAN}\}$.

Learning-based methods for vulnerability discovery build on such a decision function $f = f_\Theta$ parameterized by weights Θ that are obtained by training on a dataset of vulnerable and non-vulnerable code [18]. Compared to classical static analysis tools, learning-based approaches do not have a fixed rule set and can therefore adapt to the characteristics of different vulnerabilities in the training data. Current learning-based approaches differ in the program representation used as input and the inductive bias, that is, the way f depends on the weights Θ.

3.2 Program Representations

Different representations for programs have been used as a basis for vulnerability discovery models in the past. Popular natural language processing-based approaches represent a program as the natural token sequence that appears in the source code [38]. Since programs can be modeled inherently as directed graphs [1], more recent approaches make use of graph representations [10,43,55] for source code instead of flat token sequences achieving state-of-the-art performances [38]. We refer to the resulting program representation as a *code graph* and denote the underlying directed graphs as $G = G(V, E)$ with vertices V and edges $E \subseteq V \times V$. Moreover, for $v \in V$, we define $N(v)$ as the set of its neighboring nodes.

Code graphs differ in the syntactic and semantic features they capture. Recent works, for instance, rely only on syntactic features using the abstract syntax tree (AST) [1], while newer approaches also capture the semantic properties, as for instance using the control flow graph (CFG), which connects statements with edges in the order they will be executed or the data flow graph

(DFG) connecting the usages of variables. Based on these classical representations, combined graphs have been developed. A popular one is the code property graph (CPG) [49], which resembles a combination of the AST, CFG and program dependence graph (PDG). Other approaches use different combinations [5,46]. All these representations are denoted CPGs, however, to distinguish them from the original CPG proposed by Yamaguchi et al. [49], we formally define a code graph in Definition 1 as an attributed and combined graph structure representing programs.

Definition 1. *A code graph is an attributed graph* $G = (V, E, X_V, X_E)$ *derived from source code and providing a syntactic or semantic view of the program.*

Naturally, code graphs have attributes, for instance, a node could have code tokens or AST labels attached. Since deep learning algorithms expect input features to be numeric, recent works embed these attributes into vector spaces [6,10,43,55]. Hence a code graph extends the pair (V, E) by node attributes $X_V \in \mathbb{R}^{|V| \times d_V}$ of dimensionality d_V and edge attributes $X_E \in \mathbb{R}^{|\mathbf{E}| \times d_E}$ of dimensionality d_E [47].

3.3 Learning on Code Graphs

Vulnerability discovery using code graphs as input representation is a graph classification task. Building on a set of labels y and a set of attributed code graphs G it aims to learn a function $f_\Theta \colon G \mapsto y$. A set $\mathcal{G}_{\text{train}}$ of training graphs with known labels for each of those is given through which the parameters Θ of the function are optimized.

To build a graph neural network for code graphs, a convolutional and a global pooling block are needed [3]. Many graph convolutional blocks have been developed, the simplest of which is the graph convolutional network (GCN) [25]. The GCN can be formulated based on:

$$X' = \widehat{D}^{-1/2} \hat{A} \widehat{D}^{-1/2} X \Theta, \tag{1}$$

where $\hat{A} = A + I$ is the adjacency matrix with added self-loops, $\widehat{D}_{ii} = \sum_j \hat{A}_{ij}$ is the degree matrix and X the initial node feature matrix. Other types of convolutional blocks might be a gated graph neural network (GGNN) [29] or a graph isomorphism network (GIN) [48], where the former uses gated recurrent units instead of a feed-forward network and the latter has a separate optimizable parameter for the weights applied to the self-loops. In vulnerability discovery, however, we often lack a representative amount of vulnerable samples and, in consequence, have to deal with imbalanced graph classification [22].

4 Data Augmentation

Since there are few examples of vulnerabilities in the wild and the datasets for vulnerability discovery are unbalanced, as a remedy, we discuss the basics of data augmentation in this section.

4.1 SMOTE

The Synthetic Minority Oversampling Technique (SMOTE) [7] extends a dataset by generating new samples for all minority classes based on feature-space interpolation in the input domain. This way, the imbalance ratio of the dataset can be reduced and generalization to minority classes improved. New samples are generated by randomly selecting a sample and choosing a second sample from a random subset of the k nearest neighbors of the same class. New samples are generated by linearly interpolating between the features of the two selected nodes, yielding a new feature vector $\tilde{x} = \lambda x_1 + (1 - \lambda)x_2$, where x_1, x_2 are the features of the original samples and $\lambda \in [0, 1]$ is a uniformly random number.

4.2 Graph Autoencoder

Due to their discrete nature, SMOTE is not readily applicable to code graphs. It is not directly evident how one would interpolate between two graphs with both having, e.g., different numbers of nodes or edges. Some recent works apply SMOTE on the compressed latent space representation learned by an autoencoder in the computer vision domain [11], which learns to generate meaningful latent variables for samples from a data distribution [23] consisting of an encoder and a decoder. Moreover, the encoder of a variational autoencoder infers a probability distribution of the latent representation, by choosing a parametric probability distribution as the prior distribution for the latent variables. During training, the encoder infers the parameters of that distribution. For example, a variational autoencoder with a Gaussian distribution as the prior for the latent space would have two encoders $e_\mu(x) = z_\mu$ and $e_{\sigma^2}(x) = z_{\sigma^2}$. Then the latent representation needs to be decoded by sampling from $z \sim \mathcal{N}(z_\mu, z_{\sigma^2})$. This way, the decoder can still operate on a continuous latent representation, where it then tries to reconstruct the original input [24].

Graph autoencoders (GAEs) take this idea to the domain of graphs. They encode a graph into a latent space representation and decode it back into a graph. The latent space can be structured as a node- or graph-wise latent representation. The latter implies a single constant-sized vector for the complete graph, while the former latent space representation consists of a vector per node. Furthermore, the reconstruction target can be the adjacency matrix [24], the node or edge feature matrix [28]. Just like for the classical autoencoder, a variational variant exists, called variational graph autoencoder (VGAE).

5 CodeGraphSMOTE

CodeGraphSMOTE is applied on code graphs since not only do they provide state-of-the-art performance results on vulnerability discovery but also retain semantic and syntactic information in a compressed structure. Moreover, CodeGraphSMOTE is also equipped with a transformer to convert graphs back to source code representations. In particular, it consists of an autoencoder, interpolation method and graph-to-code transformation model. Figure 1 shows an overview of those blocks and their interplay.

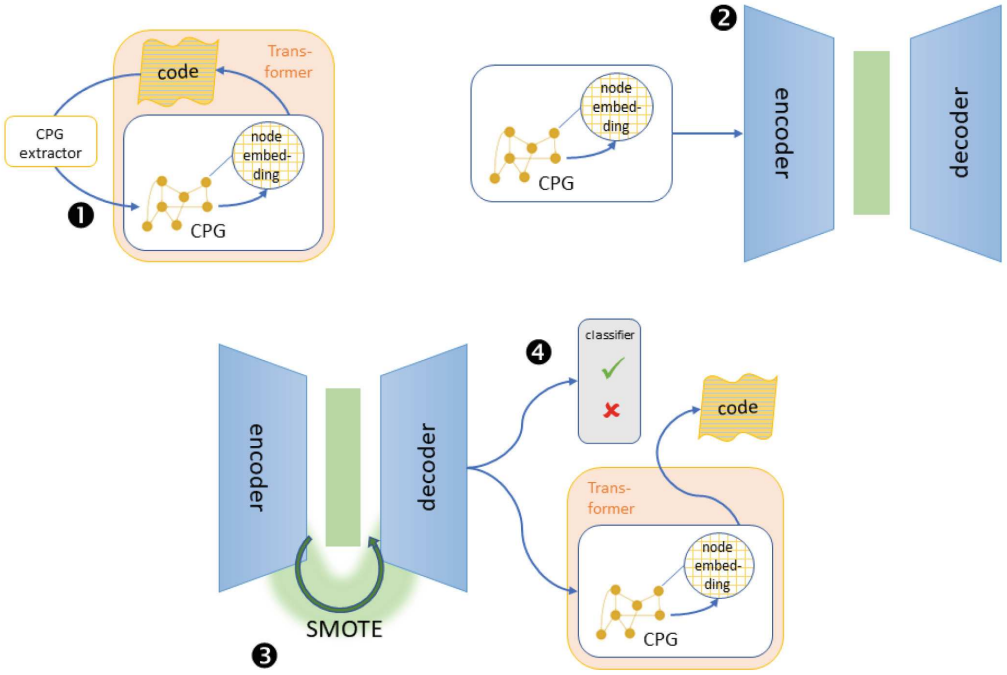


Fig. 1. Overview of the building blocks of CodeGraphSMOTE for training on imbalanced graph datasets.

5.1 Overview

The method has multiple training stages: The first stage is training a transformer model to convert code graphs to their original source code (1). The learned intermediate token embeddings of the transformer can then be used in the following stage to provide aggregated node embeddings for the VGAE. The second stage consists of training an autoencoder model to reconstruct code graphs in an unsupervised fashion (2). In the third stage, new samples can be generated by applying classic SMOTE to the VGAE latent representation (3). The last stage consists of training a vulnerability discovery model on an augmented version of the original dataset (4).

To augment the dataset, first, all code graphs of the original dataset are encoded by the autoencoder. Next, a balanced dataset is created by generating interpolated samples for the minority class. Lastly, the interpolated and original latent space representations are decoded by the autoencoder to generate new graphs for the vulnerability discovery downstream task. We proceed to explain our VGAE architecture and then provide insights into our transformer model.

5.2 Code Graph Generation

The input to the VGAE is a code graph, while the learning task is to pertain to as much information in the latent space as needed to reconstruct the original code

graph. This ensures a semantically structured latent representation of vulnerable and clean code samples.

Encoder. To produce latent space representations on the level of code graphs, a node-level autoencoder is implemented, where the intermediate node embeddings are calculated by applying a GNN to the input code graph. Conventional GNNs, such as GCNs, are low-pass filters and thus remove high-frequency features [35]. As it could be detrimental to decoding performance to smooth out the high-frequency features of the graph, an alternative architecture is used. The deconvolutional autoencoder [28] aims to preserve features of all components of the frequency spectrum by using more terms of the approximated eigendecomposition. Hence, a deconvolutional network with three layers and graph normalization [4] is used as the architecture for the encoder.

Decoder. The decoder needs to decode not only the edge and node features but also the graph's topology. For the node level decoder, we use a GNN from Li et al. [28] which employs an approximate inverse convolution operation, restoring high-frequency features and consequently alleviating the problem of GCNs being mostly low-pass filters [35]. Since the node feature decoder depends on the graph's topology, we first reconstruct the adjacency matrix using a topology decoder.

The most prominent topology decoder is the inner product decoder which connects two distinct nodes with latent representations X'_i and X'_j by an edge with probability $\sigma(X'_i {X'_j}^T)$. Therefore, we can sample edges given these probabilities or, in a deterministic setting, draw an edge iff $\sigma(X'_i {X'}^T_j) > p_0$ (usually $p_0 = 0.5$), or in other words iff

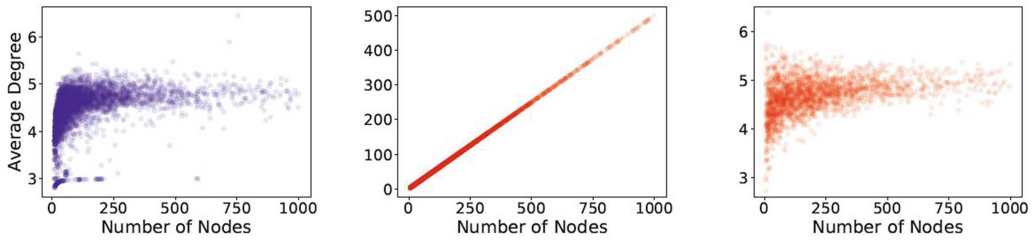
$$X'_i {X'}^T_j > t \tag{2}$$

for some threshold t (usually $t = 0$). Note that increasing t leads to fewer edges.

During the reconstruction of the adjacency matrix, nodes with similar features tend to have a very high probability of an edge between them. As a solution, we decode the node features and topology separately, by splitting the latent representation of each node in half and using only one part for each decoder. The topology decoder is then trained to reconstruct the adjacency matrix using a weighted binary cross-entropy loss due to the natural sparsity of adjacency matrices.

Another problem arises since decoders based on the inner product can only reconstruct undirected graphs due to their inherent symmetry. Hence, we implement an asymmetric inner-product-based decoder, by splitting the adjacency matrix into two halves by its diagonal. One half of the topology decoder's latent space is used for the upper and one for the lower part of the adjacency matrix.

Finally, a third problem with inner-product-based decoders stems from the random node embeddings causing the expected average degree to increase proportionally with the number of nodes. This is problematic since the average



(a) ReVeal (b) Unadjusted Decoding (c) Adjusted Decoding

Fig. 2. Average degree compared against the number of nodes for code graphs, decoded naively and decoded with our correction.

degree will be higher for larger graphs, while in reality, code graphs have the property that their average node degree is independent of the number of nodes, which is illustrated in Fig. 2a and Fig. 2b.

The number of reconstructed edges for a particular graph can be seen as a random variable

$$|E| \sim B(p, |V|^2 - |V|) \tag{3}$$

with p the binomial probability to decode a particular edge among the possible $|V|^2 - |V|$ edges. Note that we consider directed edges but no loops. Hence, we obtain $\mathbb{E}(|E|) = p \cdot (|V|^2 - |V|)$. Assuming a deterministic sampling, edges are decoded, when their similarity as defined by the inner product is above a certain threshold t. Adjusting the threshold by incorporating the expected average embedding distance of the closest embeddings reduces the effect of the proportionally increasing average degree as depicted in Fig. 2c. The derivation of this adjustment can be found in Appendix A.1.

Interpolation Method. To augment the code graph datasets, new samples need to be generated given a set of selected graphs. To do that, we propose a method to select and interpolate code graphs in their latent space representations.

A sample denotes an embedding matrix $X' \in \mathbb{R}^{|V| \times d_V}$ for a fixed latent space dimension d_V and number of nodes $|V|$. Since this matrix has different sizes for graphs with different numbers of nodes, no common distance metric can be applied to calculate the nearest neighbors. To mitigate this issue, the graphs are padded with zero vectors for non-existing nodes to the size of the largest graph in the dataset. Additionally, this same issue is found when interpolating the samples and solved in the same way. The interpolated embedding matrix

$$\hat{X}' = \lambda {X'}_a + (1 - \lambda) {X'}_b \tag{4}$$

for two chosen code graphs G_a, G_b and a uniformly random $\lambda \in [0, 1]$ is truncated to a number of nodes interpolated in the same way: $|\hat{V}| = \lambda |V_a| + (1 - \lambda)|V_b|$ with the same λ. This interpolation method is not permutation-equivariant, thus the node ordering affects the results. Since we use the same method in the nearest

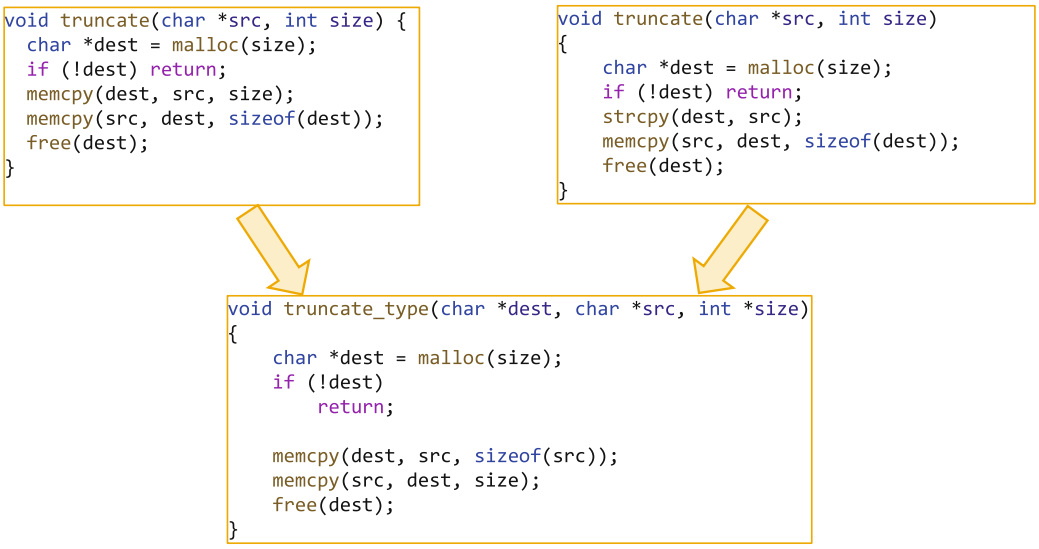


Fig. 3. An interpolated sample in its code representation.

neighbor search, however, this results in the corresponding nodes already being close to each other.

Finally, new latent space representation samples are generated, which can then be decoded using the decoder. To train a graph vulnerability discovery model on the augmented code graphs, we have to reconstruct both, edges and nodes. The node features are recovered using the node decoder's output which is trained using a cosine embedding loss. The adjacency matrices need to be discretized by sampling from a Bernoulli distribution with a probability conforming to the edge probability in the reconstructed adjacency matrix.

5.3 Graph to Code Transformation

The ability to transform code graphs back to source code adds three beneficial properties to our method: First, we can produce human-readable samples, second, we are no longer limited to GNNs and third, we can use the latent node embeddings as a fixed size vector for each node in the VGAE. Thus, we train a transformer [42] to decode non-interpolated graphs and eventually apply it to interpolated samples.

Similar to the variational autoencoder (VAE), a transformer model consists of an encoder and a decoder comprising multiple blocks each. A single block consists of two components, namely a bidirectional multi-head self-attention mechanism and a feed-forward neural network. The attention mechanism generates an attention vector for each code token providing a weight on how much one token affects the other.

We propose an auto-regressive transformer model with a BART-like architecture consisting of six encoder and decoder layers, a width of 128 and only two heads per layer [27]. The code graph is linearized by sequentially extracting

the tokens through a depth-first traversal of the AST and embedded using the pre-trained byte-pair tokenizer by Nijkamp et al. [33]. As each node consists of a variable number of tokens, the transformer has to learn a fixed-size token-level embedding of dimension $\mathbb{R}^{|N|\times d_V}$. To this end, an additional transformer encoder layer is trained jointly, that learns a normalized and pooled node vector.
This way, inter-dependencies of the tokens are encoded in a token-level representation for the node-level features for the code graphs in the GAE training.

Figure 3 shows a generated synthetic sample at the bottom. We take the graph representation from the upper two samples and interpolate between both latent representations. Then we translate the resulting graph to its code representation using the transformer model.

Both source methods truncate a string but run into buffer overflows due to potential size mismatches between src, dest and size. Note, that the upper samples are taken from two real vulnerabilities. The resulting generated function has a different name and signature, but its body is similar: It is obvious, that the resulting method is also an example of string truncation with a buffer overflow. The unused parameter char *dest in the signature may be erroneously caused by either the generation of the graph or the conversion of the graph to code but may be negligible due to the nature of data augmentation.

6 Evaluation

This section introduces the experiments designed to tackle three research questions, in particular, we describe our experimental setting and provide empirical results answering the following questions:

RQ1 *Does CodeGraphSMOTE provide a sound latent representation?*
RQ2 *Can CodeGraphSMOTE improve detection performance when we lack data?*
RQ3 *Do the augmented datasets yield better model transferability?*

6.1 Experimental Setting

We rely on Fraunhofer-CPG [46] as a tool to generate code graphs and preprocess them using networkx [19]. We use the GNN implementations from Pytorch Geometric [14] and train them on AWS EC2 g4dn instances. All experiments are conducted using 10-fold cross-validation and our VGAE consists of 2 encoder and 2 decoder layers for topology and node features respectively with a dimension of 384. We use an Adam and AdamW optimizer for the VGAE and the transformer with learning rates of 0.0005 and 0.001 respectively.

Datasets and Models. For our experimental evaluation, we use the following three datasets from recent publications around learning-based vulnerability discovery. All three datasets consist of a corpus of vulnerable and clean samples from C and C++ code repositories.

1. Chromium+Debian. The Chromium+Debian dataset consists of 1924 vulnerable and 17294 clean samples. Thus, the imbalance ratio is 10.01%. The dataset has been extracted from the Debian and Chromium bug tracker and hence contains C++ code samples [6].
2. FFmpeg+Qemu. The FFmpeg+Qemu dataset is nearly balanced with a ratio of 45.96% having 11466 and 9751 samples respectively for the clean and vulnerable class. The code was extracted using security-related keywords that have been matched against commits in the Github project repositories from Qemu and FFmpeg [55].
3. *PatchDB*. Finally, we utilize PatchDB [44], which consists of patches extracted from the national vulnerability database (NVD) for multiple C and C++ open-source projects. Vulnerable samples are labeled by their common weakness enumerations (CWEs). Overall, it has 3441 vulnerable and 30149 clean samples resulting in an imbalance ratio of 10.24%.

As ML models for the downstream vulnerability discovery task, we use ReVeal and Devign [6,55]. They both rely on GGNNs and a pooling layer followed by a feed-forward neural network prediction head. We train the transformer, VGAE and downstream classifier on the same training set and test on a disjoint separate dataset containing only real samples.

Metrics. We use two metrics recommended especially for imbalanced learning tasks to provide a comprehensive evaluation of the model's performance. By comparing these scores before and after augmentation, we can assess whether the augmentation has improved the model's performance.

1. *F1-score*. The F1-score is a commonly used metric to evaluate the performance of a classification model. It's a measure of the model's ability to correctly predict both positive and negative classes. The F1-score is calculated as the harmonic mean of precision (P) and recall (R).
2. *Balanced accuracy*. The second is balanced accuracy, which takes into account both, the true positive rate and true negative rate, and is calculated as the average of these two rates.

Baselines. To compare our method for plausibility and practicability, we benchmark against four commonly used augmentation strategies for graphs in general and vulnerability discovery models in particular.

1. *SARD enrichment*. The software assurance reference dataset (SARD) is a synthetic vulnerability corpus containing about 30k vulnerable and 30k clean samples. The vulnerable samples are pattern generated, and consequently, ML models tend to overfit [6]. We use vulnerable samples from this dataset to enrich their original dataset as proposed by Nong et al. [34].
2. *Graph Perturbation*. Borrowed from the graph domain, we can augment the dataset by randomly dropping nodes and edges. This graph perturbation

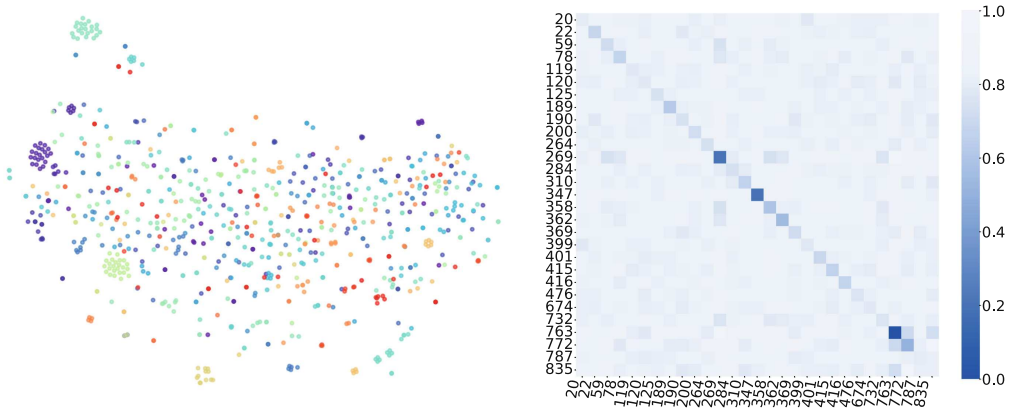


Fig. 4. Average normalized inter- and intra-cluster distance per CWE on PatchDB.

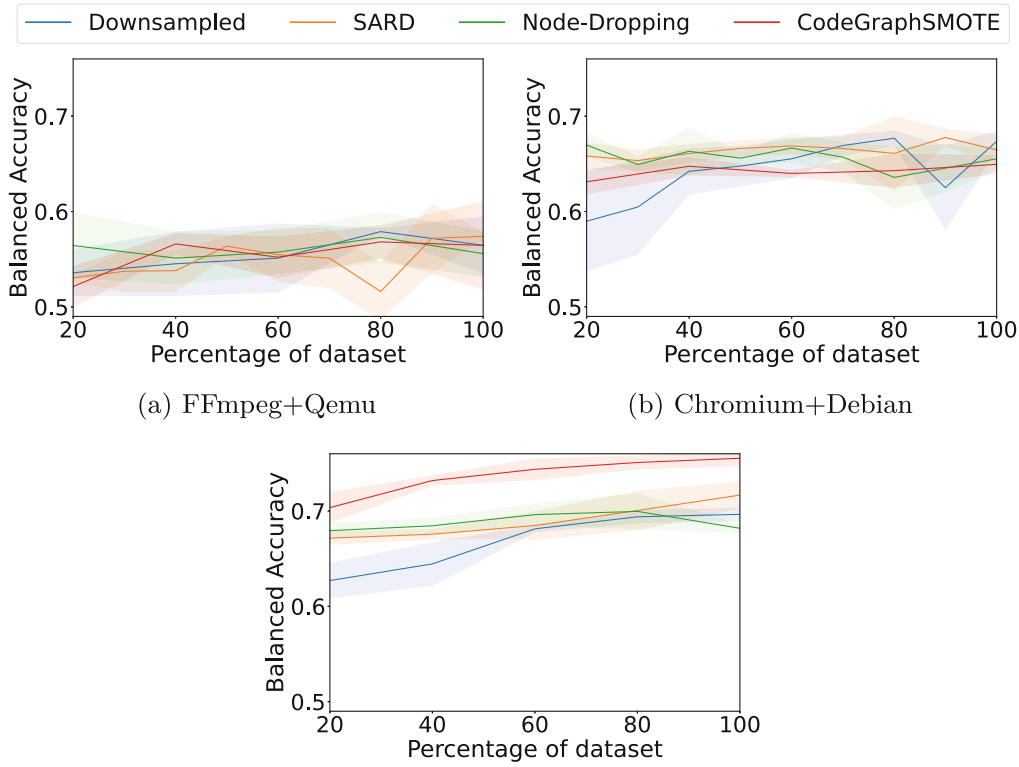
technique can be applied to vulnerable samples to augment the dataset. However, this technique will most likely break the code graph's semantics and generate unintelligible samples.

3. *Graph2Edit.* According to an empirical study [34], Graph2Edit [50] is currently state-of-the-art in neural code generation for vulnerability discovery. It uses a GGNN to learn graph and node embeddings and a LSTM network to predict edit actions on the AST. It is trained to convert ASTs of clean samples to vulnerable ones.
4. *Downsampling.* A naive approach is to downsample the majority class. That is, we remove clean samples until we have an imbalance ratio of 50%.

6.2 Results

The discussion of the experimental results is organized along the three research questions posed at the beginning of this section, which we try to answer in the following.

RQ1 — *Does CodeGraphSMOTE provide a sound latent representation?* First, we want to assess whether the learned latent space from the VGAE in CodeGraphSMOTE represents important features from the code graph and in particular for vulnerability discovery. The scatter plot on the left-hand side of Fig. 4 shows a two-dimensional t-SNE embedding of the VGAE latent representation per vulnerable code graph of the training set from PatchDB. Each sample is colored by its CWE. We can reason about the quality of the interpolated vulnerable samples since it correlates with the quality of the cluster. At least five clusters are clearly visible, including CWE-269 (improper privilege management), CWE-347 (improper verification of cryptographic signature) and CWE-763 (release of invalid pointer or reference). The right-hand side of Fig. 4 shows the average inter- and intra-cluster distances between the CWEs. The matrix is diagonal-dominant suggesting that the learned representation places samples from the



(a) FFmpeg+Qemu

(b) Chromium+Debian

(c) PatchDB

Fig. 5. Dataset augmentation strategies by replacement.

same type of vulnerability closer together. Overall, we can conclude that the latent representation encodes crucial information about the semantics of the code and vulnerability. Since SMOTE selects neighbors that are close to each other as interpolation candidates, it is safe to assume that it will automatically interpolate between vulnerabilities of the same type.

> The latent space representation learned by the VGAE clusters code graphs by their vulnerability type, making it suitable for SMOTE on vulnerability datasets.

RQ2 — *Can CodeGraphSMOTE improve detection performance when we lack data?* We evaluate our method against simple downsampling, SARD enrichment and node-dropping. We simulate smaller datasets, by removing a partition of vulnerable samples from the original FFmpeg+Qemu, Chromium+Debian and PatchDB datasets and re-balance them by augmenting the remaining. Figure 5 shows the performance of the Chromium+Debian model measured in their balanced accuracy. The x-axis denotes the percentage of real samples remaining from the original datasets, while 100% corresponds to the original

Table 1. Cross dataset evaluation presenting the F1-score. *C+D* and *F+Q* denote the Chromium+Debian and FFmpeq+Qemu datasets respectively.

Model	Training	Testing	Downsampling	Graph2Edit	SARD	CodeGraphSMOTE
ReVeal	C+D	F+Q	31.86%	36.94%	15.16%	**41.37**%
	F+Q	C+D	19.31%	**21.31**%	20.07%	18.26%
Devign	C+D	F+Q	5.25%	7.11%	5.70%	**62.97**%
	F+Q	C+D	16.83%	**19.31**%	18.58%	18.20%

dataset re-balanced using the specific method. On PatchDB, the most realistic dataset, CodeGraphSMOTE achieves an overall area under balanced accuracy score of 73.9%, compared against 68.9%, 65.5% and 63.5% for SARD, Node-Dropping and Downsampling respectively. Hence CodeGraphSMOTE yields a significantly stronger improvement compared to the other approaches with nearly 24% improvement against simple downsampling. This is particularly interesting because PatchDB has the most diverse and realistic dataset containing samples from multiple projects collected directly from the NVD. Although no augmentation strategy is a clear winner on the FFmpeg+Qemu dataset and the overall performance is only slightly above 55% as already shown by Chakraborty et al. [6] and Ganz et al. [15], we can still see that SARD is slightly worse than the other methods. Other observations are not statistically significant due to their large standard deviations.

For the Chromium+Debian datasets all augmentation strategies have less influence on the model as depicted by the large standard deviation compared to their effect on PatchDB. There is no augmentation strategy that dominates another with statistical significance. Thus, no method provides a statistically significant improvement over another except for downsampling. Downsampling is the worst method on every dataset while simple graph perturbation is the second best approach.

> Our method provides an improvement of up to 21% balanced accuracy against simple downsampling on realistic datasets and keeps the model performance constant at only 20% of the original dataset.

RQ3 — *Do the augmented datasets yield better model transferability?* Finally, we evaluate whether a pre-trained instance of CodeGraphSMOTE can be used to enhance vulnerability discovery when applied to different datasets that lack labeled or vulnerable samples. Despite recent publications showing that the Devign dataset (FFmpeg+Qemu) and model are unrealistic and underperforming [6,15], we still include both to stay comparable with Graph2Edit.

In contrast, we excluded PatchDB, which contains over 300 C and C++ projects, to demonstrate the usability of CodeGraphSMOTE on small individual projects.

Table 1 shows the average F1-score for the models ReVeal and Devign using four different augmentation strategies to re-balance the datasets. While the models have been trained and tested on disjoint datasets, our results, as shown in Table 1 and Fig. 5, indicate that training on the FFmpeg+Qemu dataset did not yield noteworthy detection capabilities. However, with F1-scores of 18.26% and 18.20%, respectively, CodeGraphSMOTE can be considered comparable to Graph2Edit with its F1-scores of 21.31% and 19.31%. Furthermore, the models trained on the Devign+Qemu dataset did not provide any transferability, highlighting the challenges associated with this dataset.

In contrast, training on the Chromium+Debian dataset reveals that CodeGraphSMOTE significantly improves detection capabilities by a factor of nearly 9 for Devign and 12% for ReVeal, as compared to the state-of-the-art method Graph2Edit. Interestingly, the Chromium+Debian dataset, with a higher degree of class imbalance than the FFmpeg+Qemu dataset, demonstrates the superior performance of CodeGraphSMOTE with increasing class imbalance.

> CodeGraphSMOTE significantly improves model transferability by up to 800% measured by the F1-score. The performance enhancement scales with increasing class imbalance.

7 Conclusion

This work introduces CodeGraphSMOTE, a novel augmentation method designed to address imbalanced attributed code graph datasets. Our approach employs a variational graph autoencoder to interpolate between code graph samples in the latent space, and a transformer model to convert these graphs back to their source code representation. On the way, we also address several common issues with graph autoencoders in general, particularly in topology reconstruction. Through experimental evaluation, we demonstrate that our method not only achieves comparable vulnerability discovery performance with fewer data but also improves the models' generalizability and transferability to new datasets.

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

A Appendix

A.1 Derivation of the Threshold Adjustment

Our goal is to adjust the threshold t in Equation (2) such that the average degree of a vertex in the reconstructed graph equals a given degree deg. Using

$\mathbb{E}(|E|) = p\,(|V|^2 - |V|)$, since we consider directed edges but no loops, this is the case if $p\,(|V|^2 - |V|) = deg\,|V|$ or

$$p = \frac{deg}{|V| - 1} \quad .$$

Now $p = P(X_i' \, X'^T_j > t)$ where X_i' and X'_j are d-dimensional latent representations of two nodes which (due to the targeted latent distribution of the VAE) we assume to be independent and identically distributed according to the standard normal distribution $\mathcal{N}(\mathbf{0}_d, \mathbf{I}_d)$. Hence the correct adjusted choice of t is given by

$$t = \mathrm{CDF}_Z^{-1}\left(1 - \frac{deg}{|V| - 1}\right) \tag{5}$$

where CDF_Z is the cumulative distribution function of the product $Z = XY$ of two i.i.d. vectors X and Y as above. By symmetry, we may assume that Y is parallel to the first coordinate axis and then X can be marginalized to this axis without affecting the inner product. Thus, we assume w.l.o.g. $d = 1$.

The density of $Z = XY = \frac{1}{4}\left((X+Y)^2 + (X-Y)^2\right)$ (a.k.a. variance gamma distribution) is known to be given by $\mathrm{PDF}_Z(z) = \frac{1}{\pi} K_0(z)$ where $K_0(z)$ is a modified Bessel function of the second kind (see [17]).

Finally, we use numerical integration to get $\mathrm{CDF}_Z(z) = \frac{1}{\pi}\int^z K_0(z)dz$ and solve numerically for t as in Equation (5).

References

1. Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. ArXiv abs/1711.00740 (2017)
2. Arp, D., et al.: Dos and don'ts of machine learning in computer security. In: 31st USENIX Security Symposium (USENIX Security 22), pp. 3971–3988, USENIX Association, Boston, MA (2022). ISBN 978-1-939133-31-1
3. Bronstein, M.M., Bruna, J., Cohen, T., Velivckovi'c, P.: Geometric deep learning: Grids, groups, graphs, geodesics, and gauges (2021)
4. Cai, T., Luo, S., Xu, K., He, D., yan Liu, T., Wang, L.: GraphNorm: a principled approach to accelerating graph neural network training (2020)
5. Cao, S., Sun, X., Bo, L., Wei, Y., Li, B.: Bgnn4vd: constructing bidirectional graph neural-network for vulnerability detection. Inf. Softw. Technol. **136**, 106576 (2021)
6. Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: are we there yet? IEEE Trans. Softw. Eng. TBD, 1 (2020)
7. Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: Smote: synthetic minority over-sampling technique. J. Artif. Int. Res. **16**(1), 321–357 (2002). ISSN 1076–9757
8. Chen, D., Lin, Y., Li, W., Li, P., Zhou, J., Sun, X.: Measuring and relieving the over-smoothing problem for graph neural networks from the topological view (2019)
9. Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L., Poshyvanyk, D., Monperrus, M.: Sequencer: sequence-to-sequence learning for end-to-end program repair. IEEE Trans. Softw. Eng. **47**(09), 1943–1959 (2021), ISSN 1939–3520

10. Cheng, X., Wang, H., Hua, J., Xu, G., Sui, Y.: DeepWukong: statically detecting software vulnerabilities using deep graph neural network. ACM Trans. Softw. Eng. Methodol. **30**(3) (2021)
11. Dablain, D., Krawczyk, B., Chawla, N.: DeepSMOTE: fusing deep learning and smote for imbalanced data. IEEE Trans. Neural Netw. Learn. Syst., 1–15 (2022). https://doi.org/10.1109/TNNLS.2021.3136503
12. Dinella, E., Dai, H., Li, Z., Naik, M., Song, L., Wang, K.: Hoppity: learning graph transformations to detect and fix bugs in programs. In: International Conference on Learning Representations (2020)
13. Do, T.H., Nguyen, D.M., Bekoulis, G., Munteanu, A., Deligiannis, N.: Graph convolutional neural networks with node transition probability-based message passing and DropNode regularization. Expert Syst. Appl. **174**, 114711 (2021)
14. Fey, M., Lenssen, J.E.: Fast graph representation learning with PyTorch Geometric. In: ICLR Workshop on Representation Learning on Graphs and Manifolds (2019)
15. Ganz, T., Härterich, M., Warnecke, A., Rieck, K.: Explaining graph neural networks for vulnerability discovery. In: Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security, pp. 145–156, AISec '21, New York, NY, USA (2021)
16. Gao, Z., Bhattacharya, S., Zhang, L., Blum, R.S., Ribeiro, A., Sadler, B.M.: Training robust graph neural networks with topology adaptive edge dropping (2021)
17. Gaunt, R.E.: Products of normal, beta and gamma random variables: stein operators and distributional theory. Brazilian J. Probab. Stat. **32**(2), 437–466 (2018)
18. Grieco, G., Grinblat, G.L., Uzal, L., Rawat, S., Feist, J., Mounier, L.: Toward large-scale vulnerability discovery using machine learning. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, pp. 85–96, CODASPY '16, New York, NY, USA (2016)
19. Hagberg, A., Swart, P., S Chult, D.: Exploring network structure, dynamics, and function using networkx (1 2008)
20. Hamilton, W.L., Ying, R., Leskovec, J.: Inductive representation learning on large graphs. In: NIPS (2017)
21. Han, X., Jiang, Z., Liu, N., Hu, X.: G-mixup: graph data augmentation for graph classification. In: Chaudhuri, K., Jegelka, S., Song, L., Szepesvari, C., Niu, G., Sabato, S. (eds.) Proceedings of the 39th International Conference on Machine Learning, Proceedings of Machine Learning Research, vol. 162, pp. 8230–8248, PMLR (17–23 Jul 2022)
22. Johnson, J.M., Khoshgoftaar, T.M.: Survey on deep learning with class imbalance. J. Big Data **6**(1), 27 (2019)
23. Kingma, D.P., Welling, M.: Auto-encoding variational Bayes. CoRR abs/1312.6114 (2014)
24. Kipf, T.N., Welling, M.: Variational graph auto-encoders. In: NIPS Workshop on Bayesian Deep Learning (2016)
25. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: International Conference on Learning Representations (ICLR) (2017)
26. Kong, K., et al.: Robust optimization as data augmentation for large-scale graphs. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 60–69 (June 2022)
27. Lewis, M., et al.: BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pp. 7871–7880, Association for Computational Linguistics, Online (Jul 2020)

28. Li, J., Li, J., Liu, Y., Yu, J., Li, Y., Cheng, H.: Deconvolutional networks on graph data. In: Beygelzimer, A., Dauphin, Y., Liang, P., Vaughan, J.W. (eds.) Advances in Neural Information Processing Systems (2021)
29. Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R.S.: Gated graph sequence neural networks. In: Bengio, Y., LeCun, Y., (eds.) 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2–4, 2016, Conference Track Proceedings (2016)
30. Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z.: Sysevr: a framework for using deep learning to detect software vulnerabilities. IEEE Trans. Depend. Secure Comput. **19**(4), 2244–2258 (2022)
31. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: Vuldeepecker: a deep learning-based system for vulnerability detection. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18–21, 2018, The Internet Society (2018)
32. Luo, Y., McThrow, M., Au, W.Y., Komikado, T., Uchino, K., Maruhashi, K., Ji, S.: Automated data augmentations for graph classification (2022)
33. Nijkamp, E., et al.: Codegen: an open large language model for code with multi-turn program synthesis. In: The Eleventh International Conference on Learning Representations (2023)
34. Nong, Y., Ou, Y., Pradel, M., Chen, F., Cai, H.: Generating realistic vulnerabilities via neural code editing: an empirical study, pp. 1097–1109, ESEC/FSE 2022, New York, NY, USA (2022)
35. NT, H., Maehara, T.: Revisiting graph neural networks: all we have is low-pass filters (2019)
36. Pewny, J., Holz, T.: Evilcoder: automated bug insertion. In: Proceedings of the 32nd Annual Conference on Computer Security Applications, pp. 214–225, ACSAC '16, New York, NY, USA (2016)
37. Rong, Y., Huang, W., Xu, T., Huang, J.: Dropedge: towards deep graph convolutional networks on node classification. In: ICLR (2020)
38. Russell, R., et al.: Automated vulnerability detection in source code using deep representation learning, pp. 757–762 (2018)
39. Sennrich, R., Haddow, B., Birch, A.: Improving neural machine translation models with monolingual data. CoRR abs/1511.06709 (2015)
40. Shorten, C., Khoshgoftaar, T.M.: A survey on image data augmentation for deep learning. J. Big Data **6**(1), 60 (Jul 2019), ISSN 2196–1115
41. Spinelli, I., Scardapane, S., Hussain, A., Uncini, A.: Biased edge dropout for enhancing fairness in graph representation learning (2021)
42. Vaswani, A., et al.: Attention is all you need. In: Guyon, I., et al., (eds.) Advances in Neural Information Processing Systems, vol. 30, Curran Associates, Inc. (2017)
43. Wang, H., et al.: 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
44. Wang, X., Wang, S., Feng, P., Sun, K., Jajodia, S.: Patchdb: a large-scale security patch dataset. In: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 149–160 (2021)
45. Wang, Y., Wang, W., Liang, Y., Cai, Y., Hooi, B.: Graphcrop: subgraph cropping for graph classification. CoRR abs/2009.10564 (2020)
46. Weiss, K., Banse, C.: A language-independent analysis platform for source code (2022)

47. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Yu, P.S.: A comprehensive survey on graph neural networks. IEEE Trans. Neural Netw. Learn. Syst. **32**(1), 4–24 (2021). https://doi.org/10.1109/tnnls.2020.2978386
48. Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? In: International Conference on Learning Representations (2019)
49. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy, pp. 590–604 (2014)
50. Yao, Z., Xu, F.F., Yin, P., Sun, H., Neubig, G.: Learning structural edits via incremental tree transformations. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3–7, 2021, OpenReview.net (2021)
51. Zhao, T., Liu, G., Gunnemann, S., Jiang, M.: Graph data augmentation for graph machine learning: a survey (2022)
52. Zhao, T., Liu, G., Wang, D., Yu, W., Jiang, M.: Counterfactual graph learning for link prediction. CoRR abs/2106.02172 (2021)
53. Zhao, T., Liu, Y., Neves, L., Woodford, O.J., Jiang, M., Shah, N.: Data augmentation for graph neural networks. In: AAAI (2021)
54. Zhao, T., Zhang, X., Wang, S.: GraphSMOTE: imbalanced node classification on graphs with graph neural networks. In: Proceedings of the 14th ACM International Conference on Web Search and Data Mining (2021)
55. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 32, Curran Associates, Inc. (2019)
56. Zhu, Y., Xu, Y., Yu, F., Liu, Q., Wu, S., Wang, L.: Graph contrastive learning with adaptive augmentation. In: Proceedings of the Web Conference 2021, ACM (2021)

D

Explaining Graph Neural Networks for Vulnerability Discovery

Explaining Graph Neural Networks for Vulnerability Discovery

Tom Ganz
tom.ganz@sap.com
SAP SE – Security Research
Germany

Martin Härterich
martin.haerterich@sap.com
SAP SE – Security Research
Germany

Alexander Warnecke
alexander.warnecke@tu-bs.de
TU Braunschweig
Germany

Konrad Rieck
konrad.rieck@tu-bs.de
TU Braunschweig
Germany

Abstract

Graph neural networks (GNNs) have proven to be an effective tool for vulnerability discovery that outperforms learning-based methods working directly on source code. Unfortunately, these neural networks are uninterpretable models, whose decision process is completely opaque to security experts, which obstructs their practical adoption. Recently, several methods have been proposed for explaining models of machine learning. However, it is unclear whether these methods are suitable for GNNs and support the task of vulnerability discovery. In this paper we present a framework for evaluating explanation methods on GNNs. We develop a set of criteria for comparing graph explanations and linking them to properties of source code. Based on these criteria, we conduct an experimental study of nine regular and three graph-specific explanation methods. Our study demonstrates that explaining GNNs is a non-trivial task and all evaluation criteria play a role in assessing their efficacy. We further show that graph-specific explanations relate better to code semantics and provide more information to a security expert than regular methods.

CCS Concepts

• **Computing methodologies** → **Neural networks**; • **Security and privacy** → **Software and application security**.

Keywords

Machine Learning, Software Security

ACM Reference Format:
Tom Ganz, Martin Härterich, Alexander Warnecke, and Konrad Rieck. 2021. Explaining Graph Neural Networks for Vulnerability Discovery. https://doi.org/10.1145/3474369.3486866

1 Introduction

Graph neural networks (GNN) belong to an emerging technology for representation learning on geometric data. GNNs have been applied successfully to a variety of challenging tasks, such as the classification of molecules [11] and protein-protein interactions [20]. Compared to other neural network architectures, GNNs can effectively make use of graph topological structures and thus constitute a versatile tool for analysis of complex data.

Because of these capabilities, GNNs have also been applied to source code to identify security vulnerabilities [39] and locate potential software defects [2]. Source code naturally exhibits graph structures, such as abstract syntax trees, control-flow structures, and program dependence graphs [10, 32], and thus is a perfect match for analysis with GNNs. Previous work could demonstrate that GNNs perform better on identifying security vulnerabilities than classical static analyzers and learning-based methods that operate directly on the source code [39]. Consequently, these neural networks are considered the basis for new and intelligent approaches in software security and engineering.

The efficacy of GNNs, however, comes at a price: neural networks are black-box models due to their deep structure and complex connectivity. While these models produce remarkable results in lab-only experiments, their decisions are opaque to security experts, which hinders their adoption in practice. Identifying security vulnerabilities is a subtle and non-trivial task. Moreover, there are even theoretical bounds as there cannot be a general approach to vulnerability detection by Rice's theorem [15], and therefore interaction with human experts is indispensable when searching for vulnerabilities. For them it is pivotal to understand the decision process behind a method to analyze its findings and decide whether a piece of code is vulnerable or not. Hence, any method for their discovery must be *interpretable*.

One promising direction to address this problem is offered by the field of *explainable machine learning*. A large body of recent work has focused on explaining the decisions of neural networks, including feed-forward, recurrent, and convolutional architectures. Similarly, some specific methods have been proposed that aim at making GNNs interpretable. Still, it is unclear whether and which of the methods from this broad field can support and track down decisions in vulnerability discovery. In this paper we address this problem and establish a link between GNNs and vulnerability discovery by posing the following research questions:

(1) *How can we evaluate and compare explanation methods for GNNs in the context of vulnerability discovery?*
(2) *Do we need graph-specific explanation methods, or are generic techniques for interpretation sufficient?*
(3) *What can we learn from explanations of GNNs generated for vulnerable and non-vulnerable code?*

To answer these questions, we present a framework for evaluating explanation methods on GNNs. In particular, we develop a set of evaluation criteria for comparing graph explanations and linking them to properties of source code. These criteria include general measures for assessing explanations adapted to graphs as well as new graph-specific criteria, such as the contrastivity and stability of edges and nodes. Based on these criteria, we are able to draw

conclusions about the quality of explanations and gain insights into the decisions made by GNNs.

To investigate the utility of our framework, we conduct an experimental study with regular and graph-specific explanation methods in vulnerability discovery. For regular approaches we focus on white-box methods, such as CAM [34] and Integrated Gradients [29], which have proven to be superior to black-box techniques in the security domain [30]. For graph-specific approaches we consider GNNExplainer [35], PGExplainer [19], and Graph-LRP [24], which all have been specifically designed to provide insights on GNNs. Our study shows that explaining GNNs is a non-trivial task and all evaluation criteria are necessary to gain insights into their efficacy. Moreover, we show that graph-specific explanations relate better to code semantics and provide more information to a security expert than regular methods.

2 Neural Networks on Code Graphs

We start by introducing the basic concepts of *code graphs*, *graph neural networks*, and their application in *vulnerability discovery*.

Code graphs. We consider directed graphs $G = (V, E)$ with vertices V and edges $E \subseteq V \times V$. Nodes and edges can have attributes, formally defined as (keyed) maps from V or E to a feature space. It is well known that source code can be modeled inherently as a directed graph [1, 2, 5], and we refer to the resulting program representation as a *code graph*. In particular, the following code graphs have been widely used for finding vulnerabilities:

AST An abstract syntax tree (AST) describes the syntactic structure of a program. The nodes of the tree correspond to symbols of the language grammar and the edges to grammar rules producing these symbols.

CFG A control flow graph (CFG) models the order in which the statements of a program are executed. Therefore, each node is a set of statements and edges are directed and labeled with flow information and conditionals.

DFG A data flow graph (DFG) models the flow of information in a program. A node denotes the use or declaration of a variable, while an edge describes the flow of data between the declaration and use of variables.

PDG The program dependence graph (PDG) proposed by Ferrante et al. [13] describes control and data dependencies in a joint graph structure. It was originally developed to slice a program into independent sub-programs.

Based on these classic representations, combined graphs have been developed for vulnerability discovery. The code property graphs (CPG) by Yamaguchi et al. [32], for example, is a combination of the AST, CFG and PDG. Likewise, the code composite graph (CCG) encodes information from the AST, DFG and CFG [7]. In the remainder, we use these two combined code graphs for our experiments, as they have proven to be effective and capture semantics from multiple representations. As an example, Figure 1 shows a CPG of a simple vulnerability.

Graph neural networks. GNNs are a model of deep learning and realize a prediction function $f\colon G(V, E) \to \mathbb{R}^d$ [23] that can be used for classification and regression. The most popular GNN types belong to so-called message passing networks (MPNs) [31] where the

```
1  #define MAXSIZE    40
2  void test(void)
3  {
4          char buf[MAXSIZE];
5          cin>>buf;
6          cout<<"result:"<<buf<<endl;
7  }
```

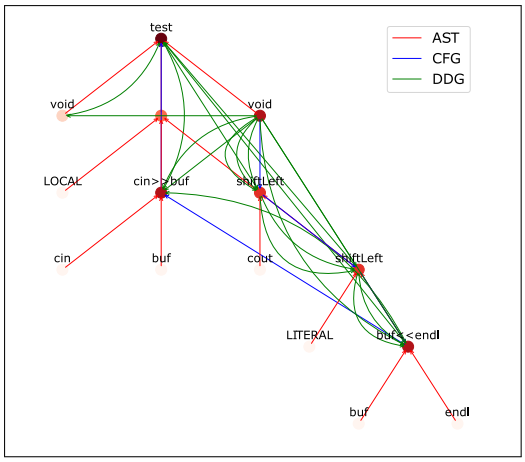


Figure 1: Source code and CPG of a simplified vulnerability. More saturated red on nodes in the CPG corresponds to more attributed relevance.

prediction function is computed by iteratively aggregating information from neighboring nodes.

In the simplest GNN layer the aggregation is the sum of the neighboring feature vectors and the update forwards the aggregated feature vector per node to a multilayer perceptron (MLP). The prediction $\hat{y}$ is then given by $\hat{y} = \hat{A}XW$, where $\hat{A}$ is the normalized adjacency matrix, X the initial feature matrix, and W a weight matrix which we seek to optimize [14]. Kipf et al. coined the term *graph convolutional network* (GCN) for this specific layer. GNNs can be further extended by incorporating other graph features as well as adding pooling and readout layers. Finally, other architectures like GRUs or LSTMs can be used to update the node embeddings, too, resulting in *gated graph neural networks* (GGNNs) [17].

GNNs for vulnerability discovery. Due to the rich semantics captured by code graphs, GNNs have been applied in a series of work for vulnerability discovery. In particular, we focus on the approaches Devign [39], ReVeal [8] and BGNN4VD [7] that can be considered state of the art and are reported to provide promising results in the respective publications.

Devign uses CPG graphs with additional edges connecting leaf nodes with their successors. These edges are called natural code sequence (NCS) and represent the natural order of the statements. The model constitutes a six-time-step GGNN. The final embedding of the sixth iteration and the initial node features are both forwarded through a novel pooling layer:

$$\sigma(\cdot) = \textsc{MaxPool}(\text{ReLu}(\textsc{Conv}(\cdot)))$$

The Conv layer is a regular 1D-convolution and is followed by a ReLu activation and max-pooling. Afterwards, the output is forwarded through an MLP. The output of both passes is multiplied pairwise and the prediction is the averaged result [39].
ReVeal is a model using the regular CPG. The pre-processing step includes a re-sampling strategy and the model consists of an eight-time-step GGNN followed by a sum aggregation and a final MLP as a prediction layer. The training involves a triplet loss incorporating binary cross-entropy, L2-regularization, and a projection loss minimizing resp. maximizing the vector distances between similar or different classes [8].
BGNN4VD differs from the other two GNN models as it uses bidirectional CCGs. The model uses an eight-time-step GGNN with the same pooling operator as Devign but followed by an MLP as a prediction layer [7].

3 Explaining Machine Learning

Vulnerability discovery using machine learning has made remarkable progress over the last years. The proposed systems, however, are opaque to practitioners and it is unclear how they arrive at their decisions. This lack of transparency obstructs their deployment in the field of security and creates a gap between research and practical demands. *Explanation methods* (EM) for machine learning have the potential to alleviate this problem and help to gain insights into the capabilities of learning-based security systems. Explainability methods turn the decision of a machine learning model into a transparent and more likely to be human interpretable result. Since all decisions depend only on the input signal of a model, conventional explainability methods try to correlate the input features to the final output. However, in the domain of GNNs, we are working with graph signals that do not only depend on features in a vector space but on discrete topological structures.

In the following, we introduce common explanation methods as well as approaches specifically designed to explain GNNs.

Graph-agnostic explanation methods. There exist a variety of general techniques for explaining learning models. For this paper, we focus on nine common approaches and adapt them for explaining GNNs. Since a node is the primitive element of code graphs, we seek explanations that indicate the relevance of nodes for the discovery of vulnerabilities. As general EMs explain only based on features, we propagate the corresponding relevance scores of edges to adjacent nodes, such that all methods yield node-level explanations.
Class Activation Maps (CAM) were originally designed for explaining Convolutional Neural Networks (CNNs). Deep layers tend to learn semantically meaningful features and CAM scales these features from the last hidden layer with the weight connecting them to the desired output node in order to generate a feature-wise explanation [38].
Linear Approximation also known as Gradient $\odot$ Input calculates for each input feature its linearized contribution to the classification output. Technically, this is the element-wise multiplication of the desired output's gradient with respect to the input node feature with the corresponding input activation [26].
GradCAM applies the idea of Linear Approximation to the intermediate activations of GNN layers instead of the input activations. This yields an advantage similar to CAM since more relevant information is aggregated. In this work, we take the GradCAM variant where activations of the last convolutional layer before the readout layer are used [25].
SmoothGrad averages the node feature gradients on multiple noisy inputs and compared to simple gradients yields noise-robust explanations [27]. We use noise sampled from a normal distribution ($\sigma = 0.15$) with 50 samples. These parameters were optimized for the descriptive accuracy of the ReVeal model.
Integrated Gradients (IG) improves the Linear Approximation by referring to a counterfactual baseline input G' and then using (1) gradients that are averaged along a straight path to the actual input G, and (2) the difference of the input activations between G and G' instead of the absolute activation [29]. We set all node features to zero for G' to achieve a very low base prediction score.
Gradient or Saliency Method simply measures the change in the prediction with respect to the change of the input by calculating the corresponding gradients. Although simple and effective it is known that the generated relevance maps are oftentimes noisy [29].
Guided Backpropagation (GB) clips the negative gradients to have a positive influence during backpropagation. This technique yields explanations that concentrate on features having an *excitatory* effect on the output prediction [28].
Layerwise Relevance Propagation (LRP) creates relevance maps by propagating the prediction back to the input such that a conservation property holds with respect to the total relevance scores of each layer. We tested the (α, β) and ϵ-rule [16] but use only the ϵ-rule since it yields better results in our experiments.
Excitation Backpropagation (EB) calculates the relative influence of the activations of the neurons in layer $l-1$ to one from layer l using backpropagation while only taking positive weights into account [37]. The gradients are normalized to sum up to 1 so that the output can be interpreted as the probability whether a neuron will fire or not given some input.

Graph-specific explanation methods. In addition to the nine general methods for explainable machine learning, we also consider three EMs that have been specifically designed for explaining GNNs. To realize a unified analysis of all methods in this paper, we adapt these graph-specific approaches, such that they also provide explanations on the level of nodes. In particular, we propagate relevance scores assigned to edges and walks to adjacent nodes in the code graphs, resulting in explanations similar to those of the graph-agnostic methods.
GNNExplainer is a black-box forward explanation technique for GNNs. Ying et al. [35] argue that general EMs fall short of incorporating graph topological properties and therefore develop this approach. For a given graph, GNNExplainer tries to maximize the *mutual information* (MI) of a prediction with respect to the prediction based on a variable discriminative subgraph S and a subset of node features. The subgraph that retains important edges is obtained by learning a mask that is applied to the adjacency matrix.
PGExplainer tackles the problem that the explanations for GNNExplainer have to be calculated for every individual graph instance. PGExplainer provides a global understanding of the inductive nature of the model by extracting relevant subgraphs S similar to

GNNExplainer. Whereas GNNExplainer is not suitable for an inductive graph learning setting (cf. [19]) PGExplainer uses a so-called explanation network on a universal embedding of the graph edges to obtain a transferable version of the EM.

Graph-LRP is a method using higher order Taylor expansions to identify relevant walks over multiple layers of a GNN where the message propagation between nodes during training is considered as walks of information flow [24]. The relevance per walk is computed using a backpropagation similar to LRP for each node in the walk. Schnake et al. [24] argue that the information contained in these walks is richer compared to explanations generated by GNNExplainer or PGExplainer.

4 Evaluating Explanations of GNNs

It is evident from the previous section that a large arsenal of methods is readily available for explaining and understanding GNNs. However, the presented explanation methods considerably differ in how they characterize the decision process and derive explanations for a given input graph. As a result, it is unclear which methods are suitable for explaining the predictions of GNNs in vulnerability discovery and how the generated relevance maps relate to the semantics of code and security flaws.

To tackle this problem, we introduce a framework for evaluating explanation methods on GNNs. In particular, we build on previous work by Yuan et al. [36] and Warnecke et al. [30] who propose criteria for comparing explanation methods in security. As their work does not account for relational information and topological structure, we extend their criteria as well as introduce new ones, specifically designed for understanding how an EM characterizes nodes in code graphs.

Descriptive accuracy. To determine whether an EM captures relevant structure in a GNN, we remove a relative amount $k\%$ of the most relevant nodes from the input graph and calculate another forward pass of the model for each graph in the test set. The *descriptive accuracy (DA)* for $k\%$ is then given by the model's drop in accuracy (with respect to its original accuracy) averaged over the test data. The larger this drop is, the more relevant nodes are identified by the EM. The area under the DA curve is a single numerical quantity that summarizes its behavior, with a large area indicating a steep rise of the curve and thus an accurate explanation of the GNN's decision process.

The DA shares similarities with the fidelity measurement by Yuan et al. [36]. This measure uses thresholds on the relevance maps to evaluate the accuracy of the explanations. Instead, we first rank the nodes by relevance and then remove a fixed percentage to obtain the DA. Since there is often a high variance in the size of code graphs, we find this measure beneficial as it allows us to obtain relevant subgraphs for vulnerability localization.

Structural robustness. To measure the robustness of an input graph for a given EM we use the *remaining agreement* (RA). We compute the 10% most relevant nodes before and after perturbing the input graph by dropping its edges with a certain probability p and define the RA as the size of the intersection of these highly relevant nodes.

Our motivation comes from the desire to understand how susceptible an EM is against manipulations where an adversary tampers with the input such that the explanation method creates an arbitrary, meaningless explanation, for example, when GNN and EMs are used to assess code of unknown origin. In this context, we interpret structural robustness as the sensitivity of the model and the EM to still label the original relevant code parts despite an attacker trying to hide certain program semantics by altering control or data flow. Although structural robustness measures the stability of the EM against noise only, it provides an upper bound of the effort a potential attacker needs to successfully attack the EM.

Contrastivity. The descriptive accuracy and structural robustness provide a general view on the quality of an explanation for a GNN, yet they do not take into account the specifics of code analysis and vulnerability discovery.

To address this issue, we propose to measure the contrastivity of an explanation. This measure is calculated by comparing relevant statements for the vulnerable and non-vulnerable class. In taint-style-analysis [32], for example, identifying vulnerabilities can be approached by traversing all CPG edges that flow from user-controlled nodes to security-critical sinks (e.g. `fopen` or `memcpy`). Hence, we expect the explanations to differ for vulnerable and non-vulnerable samples in these paths. If, for instance, critical calls are completely absent in the AST, the model does not need to further look at the DFG and CFG edges.

We calculate the contrastivity of an EM using a histogram of the AST terminal and non-terminal (type) identifier of the 10% most relevant nodes. Then, we compute the chi-square distance of the normalized node histograms of negative and positive samples:

$$\chi^2(x, y) = \frac{1}{2} \sum_{i=1}^{d} \frac{(x_i - y_i)^2}{x_i + y_i}.$$

A higher difference indicates that the model takes more different node types into account when distinguishing between vulnerable or non-vulnerable samples, providing a more diverse view on the characteristics of the code.

Graph sparsity. An explanation method must stay concise during operation, since code graphs can become too large to be manually assessed by practitioners. An explanation method that marks hundreds of nodes in a code graph as relevant yields no practical benefit. To measure the conciseness of an explanation, we adapt the mass around zero (MAZ) measure [30] to GNNs: To this end, the relevance values of the nodes are normalized to be contained in the interval $[-1, 1]$ and then a cumulative distribution function $r \mapsto \int_{-r}^{r} h(x)dx$ of their *absolute values* is calculated. The larger the area under this curve is, the more relevance values of nodes are close to 0 and hence of little influence. If all the relevance values are positive (resp. negative) then normalization is just division by the value with highest absolute value (i.e. $x_{\max}$ resp. $x_{\min}$), otherwise we use the projective transformation

$$x \mapsto \frac{(x_{\max} - x_{\min}) \cdot x}{(x_{\max} + x_{\min}) \cdot x - 2 \cdot x_{\max} \cdot x_{\min}}$$

with fixed point 0 mapping $x_{\min}$ to -1 and $x_{\max}$ to 1.

The area under the adapted MAZ provides us with a single numerical quantity that describes how concise an EM operates, where a high area indicates explanations with a sparse assignment of relevance values to nodes.

Stability. Some EMs are non-deterministic and do not provide identical results during different runs. This slight randomness can pose a problem for vulnerability discovery, where the differences between vulnerable and secure code is often nuanced and subtle. To account for this problem, we measure the stability in terms of standard deviation of the descriptive accuracy and sparsity over five runs. Note that only Smoothgrad, PGExplainer, GNNExplainer and Graph-LRP are non-deterministic, as they use randomly initialized weights or random sampling. The remaining graph-agnostic methods are deterministic by design and hence stable per definition.
Efficiency. Finally, the runtime performance of an explanation method should not drastically increase the time a security specialist needs for her traditional workflow. Especially for large and complex code graphs, it is crucial that explanations are generated in reasonable time, for instance, a few seconds. To reflect this requirement, we measure the average runtime of an EM per single graph. Note that the runtime in a practical setup also depends on details of the implementation and GNN model, and thus this criterion should only be used to provide an intuition of the performance rather than precise runtime numbers.

5 Experimental Study

After introducing our evaluation criteria, we are finally ready to empirically evaluate the performance of explanation methods on GNNs for vulnerability discovery. In particular, we consider the generic and graph-specific approaches (9+3) for explanations described in Section 3 on the three GNNs presented in Section 2.

5.1 Setup

We train Devign and ReVeal on graphs with vulnerabilities from C and C++ open-source software and BGNN4VD on a dataset containing vulnerabilities in open-source Java software. Some explainability algorithms have hyperparameters that need to be calibrated. We use Bayesian optimization to find suitable parameters for Integrated Gradients, SmoothGrad, GNNExplainer, PGExplainer and Graph-LRP. Finally, we calculate the area under curve for DA with $k \in \{1, 5, 10, 15, 30, 50, 75\}$, for the sparsity metric with interval sizes $r \in \{0.05, 0.1, 0.25, 0.5, 0.75, 1.0\}$ and for structural robustness with edge dropping probabilities $p \in \{0.005, 0.1, 0.2, 0.5, 0.75\}$. As a baseline in all experiments, we randomly generate explanations, where the relevance values for nodes are drawn independently from a uniform distribution.

Case studies. For our study, we consider three datasets and the corresponding GNNs as case studies. Each dataset consists of source code with and without security vulnerabilities. Table 1 shows an overview of the case studies and the reproduced performance of the three models. Mean and standard deviation of ten experiments, each with different stratified dataset 80/20 splits, are reported. We see a broad spectrum in the case studies' performances which is desirable, since we obtain insights in to what extent EMs depend on the underlying GNN model.

Case-Study	Accuracy	Precision	Recall	F1-Score
Devign	55.68±0.36	55.28±0.38	90.32±2.68	68.58±1.09
ReVeal	84.66±0.18	58.53±0.34	58.14±0.45	58.33±0.40
Vulas (BGNN4VD)	88.05±0.18	84.10±0.12	90.10±0.03	87.00±0.07

Table 1: Performance of all case studies for vulnerability discovery (our re-implementations).

Case study A: Devign. According to the original publication, the source code is transformed into a CPG using Joern[1] and enhanced with the NCS. Type information is label encoded. We replace the original gated graph neural network (GGNN) with six GCN message-passing networks, as this provides a slightly better model performance. We use L2-regularization during training with $\lambda_{L2} = 0.0001$ and a learning rate of 0.0001 for the Adam optimizer, since the original hyperparameters were not published. The model is originally trained to identify security vulnerabilities found in the projects FFmpeg and Qemu with excellent accuracy [39]. However, we are not able to reproduce the corresponding results and only attain a moderate F1-score.[2]

Case study B: ReVeal. In this case study, the dataset is composed of security vulnerabilities extracted from patches for the Chromium and Debian projects [8]. The code graphs are again extracted using Joern. We use L2-regularization with $\lambda_{L2} = 0.001$ and a learning rate of 0.001 for the Adam optimizer. Our accuracy (cf. Table 1) is on par with the original publication; however, we report a higher F1-score which could be due to different dataset splits.

Case study C: Vulas (BGNN4VD). As the third case study, we use a Java dataset refered to as *Vulas*[3] from Ponta et al. [21]. The dataset consists of manually curated CVEs mined from Java software repositories like Tomcat, Struts, and Spring. In contrast to memory-based vulnerabilities often found in C/C++ code, this dataset contains security issues like SQL injections, XXE vulnerabilities, directory traversals, or XSS injections. These vulnerabilities are linked to commits before and after the respective patches. We apply the BGNN4VD model and extract the CCG from each changed file in the commit both before and after the actual fix using the Fraunhofer-CPG[4]. This tools extracts graphs similar to the CCG used by Cao et al. [7]. Regarding this dataset, each Java file in a commit is merged into a single potentially disconnected graph. Furthermore, we add random Java files from the same repositories. We end up with a dataset composed of 1,000 vulnerable samples, 500 fixed samples and 500 randomly chosen benign samples. The model is trained using the Adam optimizer with $lr = 0.001$ and L2-regularization with $\lambda_{L2} = 0.0001$.

[1] https://github.com/joernio/joern
[2] Chakraborty et al. [8] report the same and, like them, we were not successful in contacting the authors.
[3] https://sabetta.com/post/vulas-dataset-released/
[4] https://github.com/Fraunhofer-AISEC/cpg

Criteria	Descriptive Accuracy			Structural Robustness			Contrastivity			Graph Sparsity		
Model	**Devign**	**ReVeal**	**Vulas**	**Devign**	**ReVeal**	**Vulas**	**Devign**	**ReVeal**	**Vulas**	**Devign**	**ReVeal**	**Vulas**
GNNExplainer	0.08 ±0.003	0.15 ±0.008	0.29 ±0.005	0.55 ±0.000	0.58 ±0.010	0.49 ±0.000	0.09 ±0.01	0.19 ±0.01	0.00 ±0.010	0.73 ±0.000	**0.73** ±0.001	**0.83** ±0.001
PGExplainer	0.09 ±0.003	0.16 ±0.002	0.22 ±0.013	0.37 ±0.000	0.57 ±0.010	0.49 ±0.010	0.10 ±0.010	0.21 ±0.030	0.12 ±0.040	**0.81** ±0.010	**0.73** ±0.001	0.77 ±0.140
Graph-LRP	0.09 ±0.002	0.10 ±0.000	0.23 ±0.014	0.13 ±0.000	0.71 ±0.000	0.22 ±0.010	**0.11** ±0.000	**0.35** ±0.010	0.19 ±0.000	0.79 ±0.000	0.14 ±0.000	0.79 ±0.000
Random	0.08 ±0.003	0.18 ±0.014	0.19 ±0.012	0.07 ±0.000	0.07 ±0.010	0.08 ±0.010	0.12 ±0.000	0.40 ±0.000	0.18 ±0.000	0.51 ±0.000	0.52 ±0.000	0.51 ±0.000
EB	0.09	0.10	0.12	0.48	0.71	0.39	0.02	0.00	0.22	0.80	0.14	0.32
GB	0.10	0.10	0.25	0.40	0.71	0.50	0.05	0.00	0.00	0.80	0.14	0.14
Gradient	0.10	0.10	0.25	0.40	0.71	0.50	0.05	0.00	0.00	0.80	0.14	0.14
LRP	0.09	0.10	0.25	0.16	0.71	0.34	0.08	0.00	0.27	0.77	0.14	0.75
CAM	0.26	**0.29**	0.12	0.45	0.49	0.49	0.01	0.07	0.21	0.48	0.14	0.69
SmoothGrad	0.08	0.10	**0.34**	0.30	0.71	0.55	0.03	0.00	**0.30**	0.77	0.15	0.78
GradCAM	0.11	0.10	0.33	0.42	0.71	0.49	0.01	0.00	0.28	0.56	0.14	0.77
Linear-Approx	0.09	0.10	0.13	0.42	0.71	0.49	0.02	0.00	0.17	0.80	0.14	0.67
IG	**0.31**	0.14	0.20	**0.71**	**0.72**	**0.72**	0.00	0.06	0.08	0.15	0.19	0.14

Table 2: AUC for descriptive accuracy (DA), sparsity (MAZ) and structural robustness (RA) and χ^2 distance for contrastivity. The standard deviation is omitted for deterministic methods as well as SmoothGrad as it is neglectable.

5.2 Results

Equipped with three case studies on vulnerability discovery, we proceed to compare the different explanations based on our evaluation criteria. These experiments are repeated five times and mean and standard deviation are reported in Table 2.

Descriptive accuracy. We find that all graph-specific methods are inferior to the graph-agnostic ones under this criterion. Overall, the best method depends on the tested model. Graph-LRP is on par with its structure-unaware counterpart LRP. Furthermore, PGExplainer performs equal or better than GNNExplainer on two out of three tasks. Some graph-agnostic methods are even worse than the random baseline for certain models. Furthermore, as seen in Figure 2, for Vulas it is sufficient to remove less than 10% of the nodes to nearly render the prediction insignificant, since a DA of $84\% - 50\% = 34\%$ corresponds to the model predicting similar to random guess for Vulas. The DA curves show different levels of the results, which are due to the different model baselines. Compared to ReVeal, if more than 40% of the relevant nodes are removed, the accuracy drops close to random for most methods, even though Vulas has a lower node median count than ReVeal. We measure the drop in the F1-score for Devign, since this model has a low accuracy score in the first place.

As expected from the values in Table 1, the explanation methods can not reveal much for Devign as the model does not predict much better than random guessing. IG works best in the Devign case study. Our observation fits with the insights from Sanchez-Lengelin et al. [23]. Just as they suggest, we see that CAM and IG are among the best candidates. Moreover, according to our experiments SmoothGrad is a winning candidate as well.

We link the bad performance of PGExplainer to a phenomenon called *Laplacian oversmoothing* [6]. For deep GNNs, the node embeddings tend to converge to a graph-wide average. Depending on the depth of the network, the node embeddings get harder to separate and the performance of the network gets worse. Chen et al. [9] measure the mean average distance (MAD) of the node embeddings and demonstrate how networks with a higher MAD perform better. In the best runs, ReVeal, Devign and Vulas have a MAD of 1.0, 0.21 and 0.88 respectively. Because PGExplainer uses node embeddings to predict an edge's existence, we argue that this phenomenon influences such explanation methods. We can link the low MAD to the low DA from Table 2.

From descriptive accuracy to visualization. Based on the DA, we can easily extract *minimal descriptive subgraphs* that contain relevant nodes and yield insights on what paths characterize a vulnerability. As an example, BGNN4VD correctly identifies the SSRF vulnerability (CVE-2019-18394[5]) from Vulas that occurred in the OpenFire software. Figure 3 shows the vulnerability. After retrieving the 10% most relevant nodes from SmoothGrad, we can construct a minimal descriptive subgraph of this vulnerability as shown in Figure 3. We can traverse the CFG and DFG edges to reproduce the vulnerability, starting from `doGet` over `getParameter(host)` and the method call `getImage(host, defaultBytes)` and ending with the `IfStatement` where we would expect an input sanitization.

Extending descriptive accuracy to edges. Besides determining relevant nodes, it is also possible to calculate the most important edges and their descriptive accuracy. Except for GNNExplainer and PGExplainer, which both compute edge relevance scores, we calculate an edge relevance score by calculating the harmonic mean of the adjacent node relevance scores for each edge for the remaining EMs. An edge is only important if both adjacent nodes are similarly important. Eventually, the relevance of the edge types can be calculated by computing the histogram of the top 10% relevant edges. For space reasons, we compare the edge type attributions of the

[5]https://nvd.nist.gov/vuln/detail/CVE-2019-18394

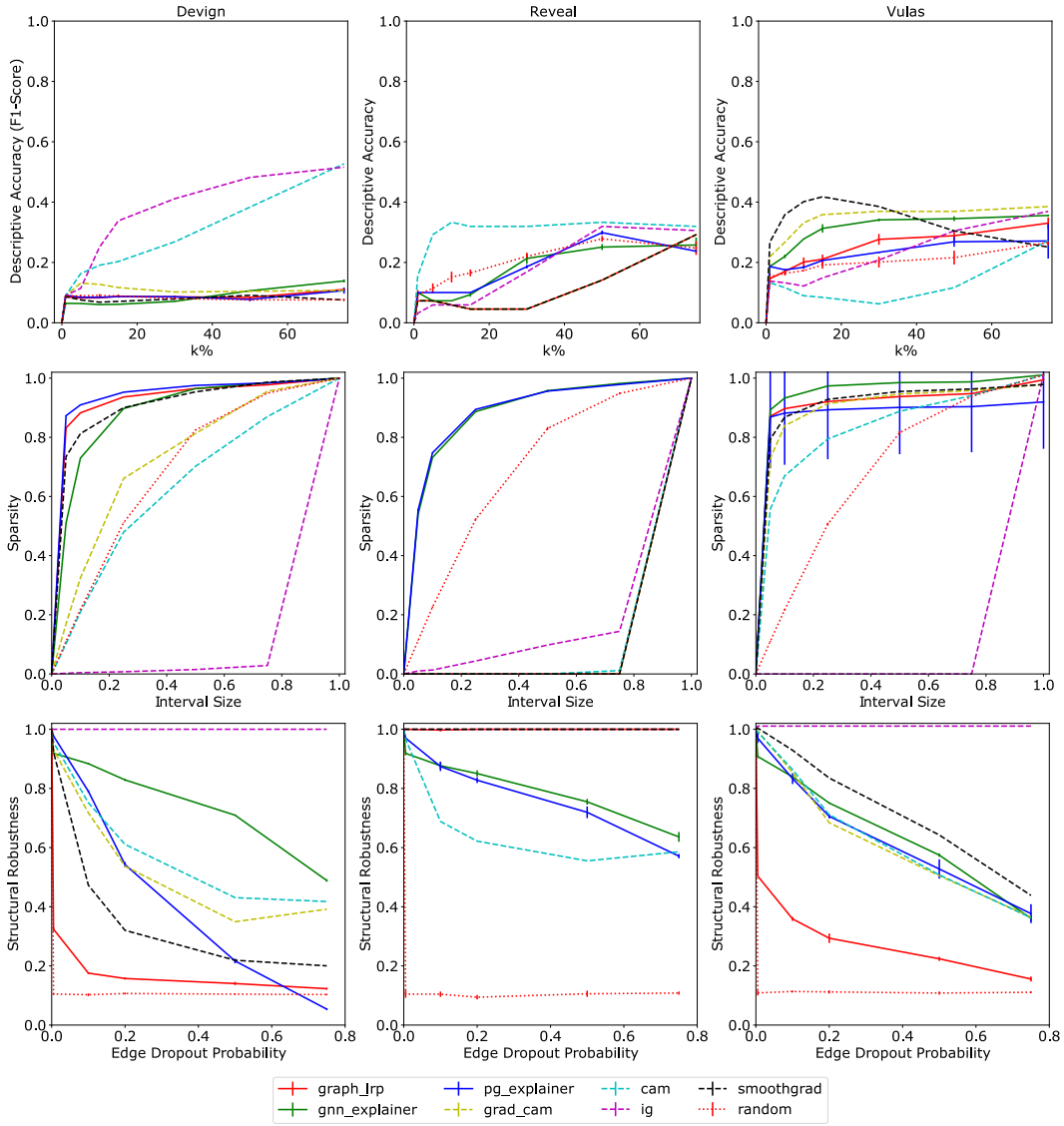


Figure 2: Descriptive accuracy (first row from top), sparsity (second row) and robustness curves (last row) for the Devign, ReVeal and Vulas case study for selected explanation methods.

graph-specific methods only with those for the generic EMs *with the best DA*.

In this setting, SmoothGrad shows the best DA for Vulas, although, it only attributes high relevance to AST edges (Figure 4). On the other hand, PGExplainer attributes a lot more relevance to semantically important edge types, although its DA is lower. It would make sense, that assuming the model correctly learns to identify security vulnerabilities, EMs should assign more relevance on semantically meaningful edges. The AST edges should not encode much information when identifying vulnerable code. For the Vulas case study, DFG seems to be important for identifying vulnerabilities, comparing the histogram with the negative and positive samples. Unfortunately, SmoothGrad also shows the same histogram, both for negative and positive samples, while PGExplainer attributes more scores to semantically interesting edge types.

Given the results for the ReVeal case study from Figure 4, the issue becomes more obvious: Most graph-agnostic methods fail to attribute relevance to semantically meaningful edges. Only GNNExplainer and PGExplainer attribute more relevance to meaningful edges when seeing positive samples. In general, CFG seems to be unimportant for positive samples. Graph-agnostic explanation methods attribute most relevance to semantically irrelevant AST and NCS edges for Devign (not shown in the plot).

```
...
public void doGet(HttpServletRequest request,
HttpServletResponse response) {
  ...
  byte[] bytes = getImage(host, defaultBytes);
  if (bytes != null) {
      writeBytesToStream(bytes, response);
  }
}
private byte[] getImage(String host,
byte[] defaultImage) {
   if (hitsCache.containsKey(host)) {
      return hitsCache.get(host);
    }
   byte[] bytes = getImage("http://" + host
                  + "/favicon.ico");
   ...
}
...
```

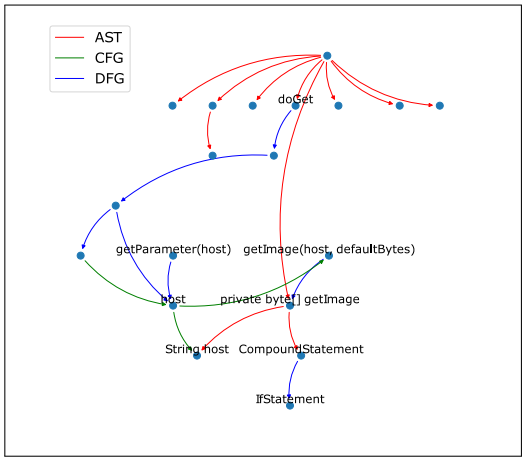


Figure 3: Minimal descriptive subgraph for the vulnerability CVE-2019-18394. The vulnerability has been detected by BGNN4VD and the graph extracted with SmoothGrad.

Structural robustness. Overall, Integrated Gradients is by far the best EM according to its robustness (cf. Table 2). By contrast, Graph-LRP is the worst method on average, which makes sense since it calculates relevant walks and therefore strongly depends on edges. In Figure 2, we can see how the remaining methods compare against each other, with random being the worst method. Devign and Vulas as opposed to ReVeal show a steeper decrease, which could mean, that the model is trained to focus on the edges instead of the nodes. The random baseline is very low, as we attribute random nodes high relevance and an intersection of relevant nodes is very unlikely. Finally, ReVeal is less affected by edge perturbations.

Contrastivity. The contrastivity is rather low for most EMs, indicating that the selection of nodes is not very diverse and there is room for improvement. Still, Graph-LRP provides the largest distance in the case studies Devign and ReVeal between vulnerable and non-vulnerable code. SmoothGrad achieves the best contrastivity score for Vulas. For Devign and Vulas, all graph-agnostic EMs are below the baseline. In general, graph-specific methods seem to be better in identifying differences between relevant node types of vulnerable vs. non-vulnerable samples.

We observe that those EMs with a very low contrastivity attribute most relevance to the root nodes, both in the CCG and CPG.

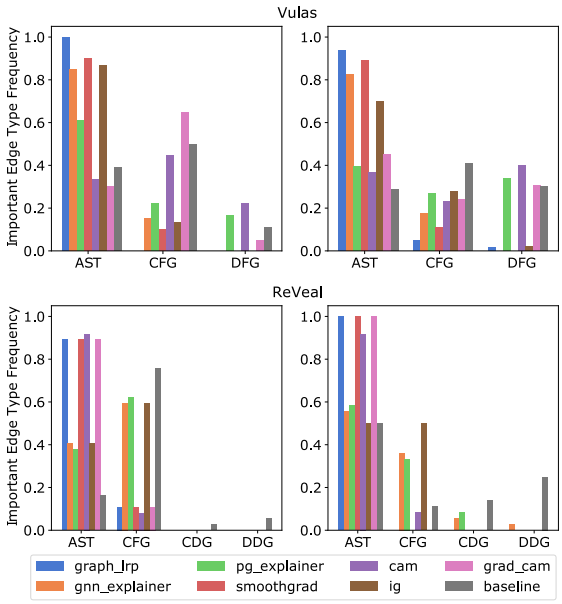


Figure 4: Important edge types for Vulas and ReVeal. Left column shows negative results and right positive.

By looking at the histogram over the most important node types (AST block identifier) labelled by graph-specific and graph-agnostic explainability methods respectively, we can clearly see a more diverse distribution for the graph-specific methods in Figure 5, although the root nodes still determine the largest attribution mass for both EM classes. Some labels are skipped to be more readable.

However, we find that the contrastivity of the graph-specific methods is influenced by the root nodes of the AST. When removing

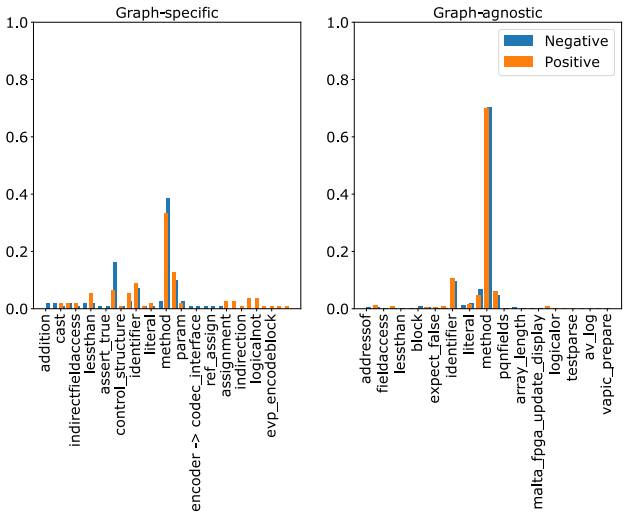


Figure 5: Important AST identifier histogram for ReVeal for negative and positive samples.

Category	DA	Sparsity	Robustness	Contrastivity	Stability	Efficiency
Graph-agnostic	● ● ●	○ ○ ○	● ● ●	● ○ ○	● ● ●	● ● ●
Graph-specific	○ ○ ○	● ● ●	○ ○ ○	● ● ○	○ ○ ○	○ ○ ○

Table 3: Final evaluation comparing graph-agnostic and graph-specific EMs. One point for a *winner* EM per model.

the root nodes and measuring the accuracy we observe only a drop of 8%, 5% and 0% for Vulas, ReVeal and Devign, respectively. This is a hint that it is not the model that focuses on top nodes but rather the explanations do. Intuitively, it is not a desired behavior that an EM distributes relevance to nodes that do not provide any useful information to an expert. However, since the root node aggregates the relevance from nodes lower in the hierarchy, it makes sense. We can see this phenomenon in Figure 1, too, where the root node has similar relevance as the node `cin >> buf`.

Graph sparsity We see in Table 2 that for all models the graph-specific EMs yield the sparsest scores. This makes perfect sense since they are optimization algorithms that seek to maximize mutual information by maximizing the prediction score and minimizing the probability of an edge between two nodes. The random baseline has an AUC of around 50% because all nodes' relevance scores are uniformly distributed. Integrated Gradients have the worst results given the Devign and Vulas results. IG, for instance, attributes around 90% of the overall importance to approximately 60% of the nodes in the ReVeal case study.

In Figure 2 the MAZ curves (sparsity) are presented for the case studies. All graph-agnostic methods give extremely dense explanations for the ReVeal case study. Overall the graph-agnostic methods seem to be inferior compared to graph-specific methods. In Figure 1, we present an explanation of a CPG showing a vulnerability[6] that is correctly classified by ReVeal. The attribution is applied using PGExplainer which correctly attributed relevance to the `cin>>buf` node. However, unimportant nodes like the root node are highlighted as well.

Stability. Table 2 shows that all graph-specific methods yield an uncertainty that differs extremely from model to model. The graph-agnostic explanation methods do not vary at all. Furthermore, in the Sparsity and the DA column we see that PGExplainer has very different score levels across multiple runs. Each run differs in the descriptiveness from the identified important nodes and the amount of relevance distributed over all nodes.

The variance in the runs of Graph-LRP correlates to the sampled walks: Depending on the dataset and the sampling strategy, there is a difference in DA and MAZ. Graph-LRP and GNNExplainer generally have a much lower MAZ AUC standard deviation than PGExplainer, i.e. there is little variation in their conciseness. On the other hand PGExplainer's MAZ AUC varies extremely and, therefore, may yield different explanations. In addition its stability depends proportionally on the median node count of the dataset which is lowest for Devign.

We connect the large standard deviation of the graph-specific EMs in Vulas concerning the DA with the low node and edge count. A low node count means removing a single different node could have a stronger effect on the model's decision. Furthermore the CCG has less edges than the CPG, since the CCG does not contain PDG edges. Hence, a single misspredicted edge in PGExplainer or GNNExplainer could lead to a vastly different classification output.

Efficiency. Graph-specific methods are almost always slower than their conventional competitors. PGExplainer is trained one time per dataset, which in turn, renders its training time for the inference negligible. Due to the fact, that the PGExplainer uses node embeddings to predict the edge probability in a graph, we see that its runtime is extremely slow for ReVeal which can be directly linked to the large node and edge count median of 333 and 1132 respectively, for this particular case study. CAM, GB, linear approximation and EB scored the best scores in terms of runtime in our experiments. Among the graph-specific methods, PGExplainer was the fastest in 2 out of 3 tasks[7]. Graph-LRP is slow as well since it calculates one LRP run for each walk. Runtime figures can be seen in the appendix.

6 Discussion

Our evaluation of the various EMs provides a comprehensive yet also complex picture of their efficacy in explaining GNNs. Depending on the evaluation criteria, the approaches differ considerably in their performance and a clear winner is not immediately apparent, as shown in Table 3. In the following, we thus analyze and structure the findings of our evaluation by returning to the three research questions posed in the introduction.

(1) *How can we evaluate and compare explanation methods for GNNs in the context of vulnerability discovery?*

We find that existing criteria for evaluating EMs are incomplete when assessing GNNs in vulnerability discovery. Our experiments show that graph-specific criteria are crucial for understanding how an approach performs in a practical application. For example, a security expert would not only focus on high accuracy of explanations but also stability, sparsity, efficiency, robustness, and contrastivity. Theoretically, a study with human experts would provide more insights. However, this would be intractable. As a trade-off, we suggest using combinations of our proposed evaluation criteria to measure the potential to be human interpretable. The interplay of these measurements is crucial and all have to be considered.

(2) *Do we need graph-specific explanation methods, or are generic techniques for interpretation sufficient?*

Our evaluation demonstrates that generic EMs often lack sparse explanations and tend to mark more nodes as relevant than needed. For a security expert, it is necessary to spot the location of vulnerabilities. Not only have graph-specific methods larger differences between negative and positive samples more often, but also do they focus on semantically more meaningful edge types. As Yamaguchi et al. [32] show only few security vulnerability types can be found

[6] Taken from https://samate.nist.gov/SARD/

[7] Measured on AWS EC2 p3.2xlarge instance.

when only taking AST edges into account and hence a more contrastive view is necessary. It turns out that generic techniques often fail to provide this perspective when analyzing GNNs.

The stability and descriptive accuracy of graph-specific explanation methods, however, is inferior to generic approaches. Consequently, the sparse and more focused explanations comes with a limited accuracy in the relevant features. This opens new directions for research and developing graph-specific methods that attain the same accuracy as generic approaches. Some possible improvements could be adding regularization to focus on semantically important nodes, using node embeddings from lower layers to overcome *Laplacian oversmoothing*, or to use the contrastivity criterion already within the generation of explanations.

(3) *What can we learn from explanations of GNNs generated for vulnerable and non-vulnerable code?*

We observe that many explanation methods focus on semantically unimportant nodes and edges, while having a large descriptive accuracy. This could be a hint that the GNNs do not actually learn to identify vulnerabilities but artifacts in the data sets, so-called spurious correlations. As this phenomena occurs over several explanation methods, it seems rooted in the learning process of GNNs and thus cannot be eliminated easily. This finding is in line with recent work on problems of deep learning in vulnerability discovery [8] that also points to the risk of learning artifacts from the data sets. Hence, there is a need for new approaches that either eliminate spurious correlations early or improve the learning process, such that more focus is put on semantically relevant structures, for example, by additionally pooling AST, CFG and DFG structures.

Moreover, we show on a real-world vulnerability that the extraction of minimal relevant subgraphs from explanations is possible and provides valuable insights. These subgraphs can be used to construct detection patterns for static-analyzers [33], to guide fuzzers [40], or to find possible attack vectors for penetration testing [12]. Hence, despite the discussed shortcomings of explanation methods and GNNs in vulnerability discovery, we finally argue that they provide a powerful tool in the interplay with a security expert. Especially, the generation of subgraphs from explanations helps to understand the decision process for a discovery and to decide whether a learning-based system spotted a promising candidate for a vulnerability in source code.

7 Related Work

The variety of methods for explaining machine learning has brought forward different approaches for evaluating and comparing their performance [e.g., 18, 29, 34, 37]. In the following, we briefly discuss this body of related work, indicating similarities and differences to our framework.

Closest to our work is the study by Warnecke et al. [30] who develop evaluation criteria for EMs in security-critical contexts. For instance, they propose variants of the descriptive accuracy, sparsity, robustness, stability, and completeness for regular explanation methods. We build on this work and adapt the criteria to graph structures, such that they do not only measure the relevance of individual features but topological structures. Furthermore, we introduce new criteria that complement the evaluation and emphasize important aspects in the context of GNNs. Baldassarre and Azizpour [4] compare different explanation methods by attributing relevance to features but do not consider the underlying graph structure. Since nodes and edges are natural building blocks of a graph, it is beneficial to focus on identifying those important topological structures. This is especially important since we represent code as graphs and relevant nodes can be directly mapped to relevant code parts.

In a different research branch, explanation methods on GNNs have been evaluated by Sanchez-Lengeling et al. [23], Baldassarre and Azizpour [4] and Pope et al. [22]. Their main contributions include the reinterpretation of classical EMs to be applicable to graph neural networks and their evaluation on GNNs such as CAM, LRP and GradCAM. However, their works fall short of introducing new graph-specific criteria that are designed to explain structures not captured in common feature vectors. Besides their lack of a thorough comprehensive assessment as we introduce in our work, they do not consider any graph-specific EM.

Furthermore, Yuan et al. [36] introduce a framework for evaluating explanation methods for GNNs. They introduce the criteria fidelity, stability, and sparsity which compute the relevance for the model's prediction, the robustness against noise, and the conciseness of the methods respectively. Their work does not consider robustness against adversaries, efficiency or contrastivity and, most importantly, lacks experimental evaluations.

Pope et al. also determine the contrastivity of an explanation method by measuring the contrast between explanations for different classes [22]. However, they do not deliver insights about robustness or efficiency in their experiments which is especially important for the security domain. We adapt their contrastivity into the context of vulnerability discovery and use it to asses how well an explanation aligns with the actual code semantics. Besides that, we want to assess how the model differentiates between vulnerable and non-vulnerable samples. We try to answer whether GNN models actually learn to identify vulnerabilities. This question aligns with different works, that critically analyze the capability of models learning to represent vulnerabilities [3, 8].

In summary, current research does not offer any comprehensive framework applicable to GNNs in security related contexts. The majority of related work measures the quality of graph explanation methods with a specific ground truth [35] or domain knowledge [24] when checking whether EMs correctly detect cycles in a synthetic dataset, for example [35]. We try to evaluate models and explanations without using ground truth for the attributions, since this information rarely exists in realistic scenarios.

8 Conclusion

We compare multiple graph-agnostic and graph-specific explanation methods on three state-of-the-art GNN models which identify security vulnerabilities. For the assessment, we introduce a framework combining the evaluation criteria stability, descriptiveness, structural robustness, efficiency, sparsity and contrastivity. Taking only the descriptive accuracy and runtime (efficiency) into account for the three GNN models under test, CAM, IG and SmoothGrad outperform all other explainability techniques. However, explanation methods for security-critical tasks, need to be thoroughly assessed using all of the above criteria. We find that all explanation methods

have shortcomings in at least two criteria and therefore hope to foster research for new explanation methods. When it comes to meaningful, contrastive and sparse explanations that emphasize the underlying graph topology we find graph-specific methods to be superior.

To actually locate security vulnerabilities given human interpretable explanations we thus suggest using GNNExplainer or PG-Explainer. Our experimental results could guide development for novel graph-specific explanation methods or to overcome current shortcomings for GNNs in identifying security vulnerabilities.

Acknowledgments

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

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to Represent Programs with Graphs. *CoRR* abs/1711.00740 (2017). arXiv:1711.00740 http://arxiv.org/abs/1711.00740

[3] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2020. Dos and Don'ts of Machine Learning in Computer Security. *CoRR* abs/2010.09470 (2020). arXiv:2010.09470 https://arxiv.org/abs/2010.09470

[4] Federico Baldassarre and Hossein Azizpour. 2019. Explainability Techniques for Graph Convolutional Networks. *CoRR* abs/1905.13686 (2019). arXiv:1905.13686 http://arxiv.org/abs/1905.13686

[5] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. *CoRR* abs/1806.07336 (2018). arXiv:1806.07336 http://arxiv.org/abs/1806.07336

[6] Chen Cai and Yusu Wang. 2020. A Note on Over-Smoothing for Graph Neural Networks. *CoRR* abs/2006.13318 (2020). arXiv:2006.13318 https://arxiv.org/abs/2006.13318

[7] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. 2021. BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection. *Information and Software Technology* 136 (2021), 106576. https://doi.org/10.1016/j.infsof.2021.106576

[8] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[9] Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. 2019. Measuring and Relieving the Over-smoothing Problem for Graph Neural Networks from the Topological View. *CoRR* abs/1909.03211 (2019). arXiv:1909.03211 http://arxiv.org/abs/1909.03211

[10] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, and Hugh Leather. 2020. ProGraML: Graph-based Deep Learning for Program Optimization and Analysis. arXiv:2003.10536 [cs.LG]

[11] David Duvenaud, Dougal Maclaurin, Jorge Aguilera-Iparraguirre, Rafael Gómez-Bombarelli, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P. Adams. 2015. Convolutional Networks on Graphs for Learning Molecular Fingerprints. *CoRR* abs/1509.09292 (2015). arXiv:1509.09292 http://arxiv.org/abs/1509.09292

[12] Mohd Ehmer and Farmeena Khan. 2012. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. *International Journal of Advanced Computer Science and Applications* 3 (06 2012). https://doi.org/10.14569/IJACSA.2012.030603

[13] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

[14] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *CoRR* abs/1609.02907 (2016). arXiv:1609.02907 http://arxiv.org/abs/1609.02907

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

[16] Sebastian Lapuschkin, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. 2015. On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation. *PLoS ONE* 10 (07 2015), e0130140. https://doi.org/10.1371/journal.pone.0130140

[17] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2017. Gated Graph Sequence Neural Networks. arXiv:1511.05493 [cs.LG]

[18] Pantelis Linardatos, Vasilis Papastefanopoulos, and Sotiris Kotsiantis. 2021. Explainable AI: A Review of Machine Learning Interpretability Methods. *Entropy* 23, 1 (2021). https://doi.org/10.3390/e23010018

[19] Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. 2020. Parameterized Explainer for Graph Neural Network. arXiv:2011.04573 [cs.LG]

[20] Niccolò Pancino, Alberto Rossi, Giorgio Ciano, Giorgia Giacomini, Simone Bonechi, Paolo Andreini, Franco Scarselli, Monica Bianchini, and Pietro Bongini. 2020. Graph Neural Networks for the Prediction of Protein-Protein Interfaces.

[21] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and C´edric Dangremont. 2019. A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software. In *Proceedings of the 16th International Conference on Mining Software Repositories.* https://arxiv.org/pdf/1902.02595.pdf

[22] P. E. Pope, S. Kolouri, M. Rostami, C. E. Martin, and H. Hoffmann. 2019. Explainability Methods for Graph Convolutional Neural Networks. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).* 10764–10773. https://doi.org/10.1109/CVPR.2019.01103

[23] Benjamin Sanchez-Lengeling, Jennifer Wei, Brian Lee, Emily Reif, Peter Wang, Wesley Qian, Kevin McCloskey, Lucy Colwell, and Alexander Wiltschko. 2020. Evaluating Attribution for Graph Neural Networks. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 5898–5910. https://proceedings.neurips.cc/paper/2020/file/417fbbf2e9d5a28a855a11894b2e795a-Paper.pdf

[24] Thomas Schnake, Oliver Eberle, Jonas Lederer, Shinichi Nakajima, Kristof T. Schütt, Klaus-Robert Müller, and Grégoire Montavon. 2020. Higher-Order Explanations of Graph Neural Networks via Relevant Walks. arXiv:2006.03589 [cs.LG]

[25] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision.* 618–626.

[26] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. 2017. Learning Important Features Through Propagating Activation Differences. *CoRR* abs/1704.02685 (2017). arXiv:1704.02685 http://arxiv.org/abs/1704.02685

[27] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda B. Viégas, and Martin Wattenberg. 2017. SmoothGrad: removing noise by adding noise. *CoRR* abs/1706.03825 (2017). arXiv:1706.03825 http://arxiv.org/abs/1706.03825

[28] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. 2015. Striving for Simplicity: The All Convolutional Net. arXiv:1412.6806 [cs.LG]

[29] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. 2017. Axiomatic Attribution for Deep Networks. arXiv:1703.01365 [cs.LG]

[30] Alexander Warnecke, Daniel Arp, Christian Wressnegger, and Konrad Rieck. 2020. Evaluating Explanation Methods for Deep Learning in Security. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P).* IEEE, Genoa, Italy, 158–174. https://doi.org/10.1109/EuroSP48549.2020.00018

[31] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A Comprehensive Survey on Graph Neural Networks. *CoRR* abs/1901.00596 (2019). arXiv:1901.00596 http://arxiv.org/abs/1901.00596

[32] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy.* 590–604. https://doi.org/10.1109/SP.2014.44

[33] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *2015 IEEE Symposium on Security and Privacy.* 797–812. https://doi.org/10.1109/SP.2015.54

[34] Wenjie Yang, Houjing Huang, Zhang Zhang, Xiaotang Chen, Kaiqi Huang, and Shu Zhang. 2019. Towards Rich Feature Discovery With Class Activation Maps Augmentation for Person Re-Identification. (2019), 1389–1398. https://doi.org/10.1109/CVPR.2019.00148

[35] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. GNN Explainer: A Tool for Post-hoc Explanation of Graph Neural Networks. *CoRR* abs/1903.03894 (2019). arXiv:1903.03894 http://arxiv.org/abs/1903.03894

[36] Hao Yuan, Haiyang Yu, Shurui Gui, and Shuiwang Ji. 2020. Explainability in Graph Neural Networks: A Taxonomic Survey. *CoRR* abs/2012.15445 (2020). arXiv:2012.15445 https://arxiv.org/abs/2012.15445

[37] Jianming Zhang, Zhe Lin, Jonathan Brandt, Xiaohui Shen, and Stan Sclaroff. 2016. Top-down Neural Attention by Excitation Backprop. *CoRR* abs/1608.00507 (2016). arXiv:1608.00507 http://arxiv.org/abs/1608.00507

[38] B. Zhou, A. Khosla, Lapedriza. A., A. Oliva, and A. Torralba. 2016. Learning Deep Features for Discriminative Localization. *CVPR* (2016).

[39] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. *CoRR* abs/1909.03496 (2019). arXiv:1909.03496 http://arxiv.org/abs/1909.03496

[40] Xiaogang Zhu, Shigang Liu, Xian Li, Sheng Wen, Jun Zhang, Seyit Ahmet Çamtepe, and Yang Xiang. 2020. DeFuzz: Deep Learning Guided Directed Fuzzing. *CoRR* abs/2010.12149 (2020). arXiv:2010.12149 https://arxiv.org/abs/2010.12149

A Runtime Evaluation

Method	Devign	ReVeal	Vulas
EB	0.11	0.17	**0.07**
GB	**0.10**	0.16	0.09
Gradient	0.10	0.16	0.091
LRP	0.14	0.207	0.12
CAM	0.12	0.17	0.09
SmoothGrad	1.66	1.72	1.79
GradCAM	0.11	0.16	0.08
Linear-Approx	0.11	**0.16**	0.09
IG	1.63	2.52	1.52
GNNExplainer	3.99	5.06	8.88
PGE-Training	4.36	50.14	3.20
PGE-Inference	1.22	47.04	3.06
Graph-LRP	22.24	33.01	19.10

Table 4: Average runtime (in s) per single graph instance.

B Model Comparison

Model	Devign	BGNN4VD	Reveal
Graph	CPG	Bidirectional CCG	CPG
Network Type	6-GGNN	8-GGNN	8-GGNN
Pooling	1D-Conv	1D-Conv	Maxpooling
Prediction Head	PEM[8]	MLP	MLP
Loss	BCE + L2-Reg	BCE	Triplet Loss

Table 5: Notable differences between the models.

C Datasets

In the next three subsections we give a more detailed introduction to the datasets.

C.1 Devign

The Devign dataset consists of manually labeled C functions gathered from Qemu and FFmpeg open source projects [39]. It consists of 6000 malicious and 6000 benign samples. All bugs have been found by scraping the commit history for certain keywords including *injection* and *DoS*. Often vulnerabilities consist of *out of bounds* or other memory related security issues. For example this[9] or that[10]. Since FFmpeg and Qemu are part of the OSS-Fuzz project and are continuously fuzzed, such vulnerabilities are oftentimes detected during that process.

[8]Pairwise Embedding Multiplication
[9]https://github.com/ffmpeg/ffmpeg/commit/06e5c791949b63555aa4305df6ce9d2ffa45ec90
[10]https://github.com/ffmpeg/ffmpeg/commit/5a2a7604da5f7a2fc498d1d5c90bd892edac9ce8

C.2 Reveal

Reveal consists of Debian security vulnerabilities taken from its tracker[11] and of Chromium vulnerabilities taken from its issue tracking tool[12]. Only bugs that are labeled *security* with a existent patch are scraped. Assuming a file has been patched, all its functions are extracted and labeled *benign*. Functions that differ from before and after fix are labeled *malicious*. Therefore, the dataset is unbalanced and consists of more benign than malicious functions.

```
1  static void eap_request(
2  eap_state *esp, u_char *inp, int id, int len) {
3  ...
4  if (vallen < 8 || vallen > len) {
5  ...
6  break;
7  }
8  /* FLAW: 'rhostname' array is vulnerable to overflow.*/
9  - if (vallen >= len + sizeof (rhostname)){
10 + if (len - vallen >= (int)sizeof (rhostname)){
11 ppp_dbglog(...);
12 MEMCPY(rhostname, inp + vallen,
13 sizeof(rhostname) - 1);
14 rhostname[sizeof(rhostname) - 1] = '\0';
15 ...
16 }
17 ...
18 }
```

Listing 1: Reveal example vulnerability CVE-2020-8597

In Listing 1 a sample vulnerability form the Reveal dataset taken from their original publication can be seen [8]. The sample shows a buffer overflow vulnerability due to a logic flaw in the *point to point protocol daemon* with the corresponding fix (line 9 and 10).

C.3 Vulas

Vulas is a collection of CVEs associated with large open source Java projects and their respective fix-commits [21]. We extract each changed function before and after the actual patch together with multiple randomly chosen functions from the same repository. The newest vulnerability in our dataset is CVE-2020-9489[13] and the oldest one CVE-2008-1728 [14]. A sample security issue can be seen in Figure 3.

[11]https://security-tracker.debian.org/tracker/
[12]https://bugs.chromium.org/p/chromium/issues/list
[13]https://www.suse.com/security/cve/CVE-2020-9489.html
[14]https://nvd.nist.gov/vuln/detail/CVE-2008-1728

E

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

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.

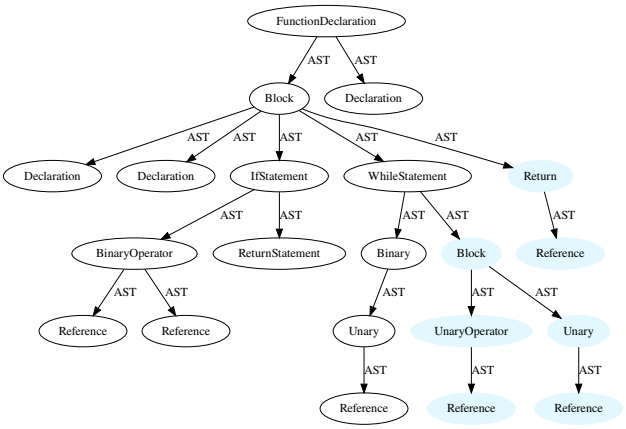


Figure 2: Explanation in the excerpt of a code graph. Relevant nodes are highlighted in blue.

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

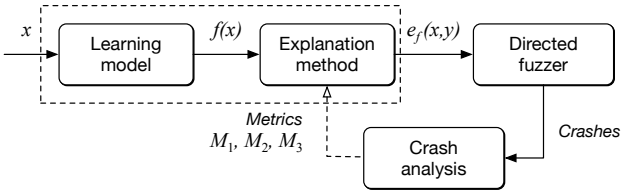


Figure 3: Overview of our approach for generating local ground-truth for explanation methods.

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 $\mathcal{X}$

repeat n **times**
  for $x \in \mathcal{X}$ **do**
    $y \leftarrow f(x)$ // *Inference on sample*
    $\mathcal{M} \leftarrow e_f(x, y)$ // *Generation of explanation*
    Highlight code lines L from $\mathcal{M}$
    Run directed fuzzer towards L
    Reproduce and analyze crashes
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 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.

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

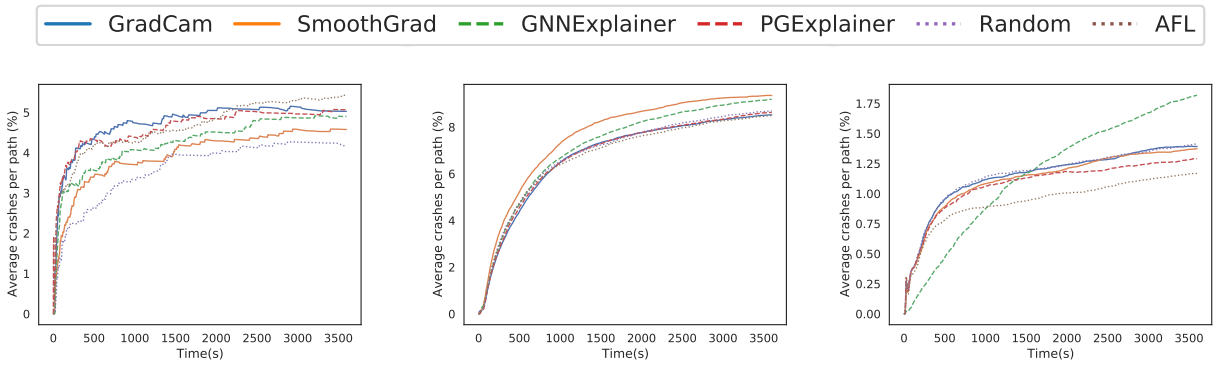
Discussion. According to Table 1, the crash reproduction is faster using the extracted lines from explanation methods compared to vanilla AFL without any targets. After re-executing the generated seeds from the directed fuzzer during the crash analysis, using the debugger, we observe that the lines were 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.

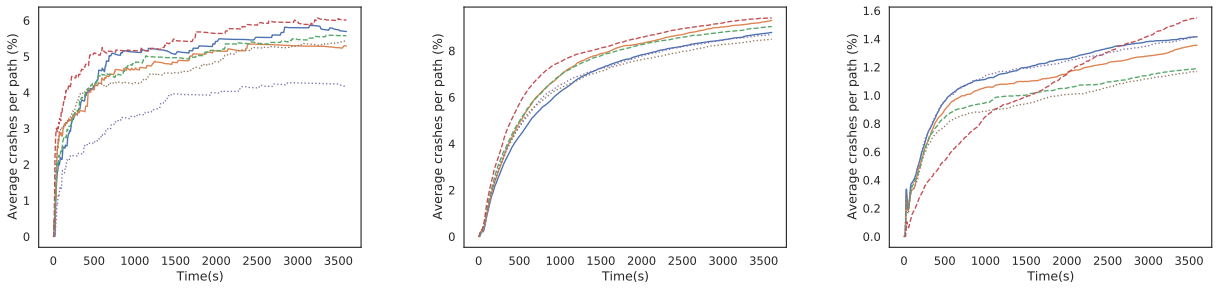
RQ2. — *Which explanation method provides the best explanations according to extrinsic criteria?* For visualization purposes, consider Figure 5 showing the average crashes per path (M_1) over the fuzzing period for five runs for Devign and ReVeal on all three PUTs. AFLGo uses *simulated annealing* [28] 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.

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



(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

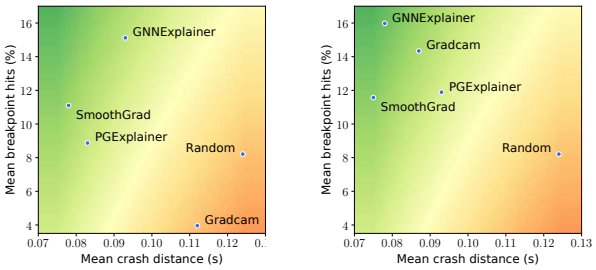
M_1 where graph-specific EMs achieve the best scores in four out of six while graph-agnostics only in two out of six experiments according to Table 2. Contrary to the empirical study of Ganz et al. [21] we find that graph-specific methods are more veracious than graph-agnostic EMs. We thus conclude that 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.

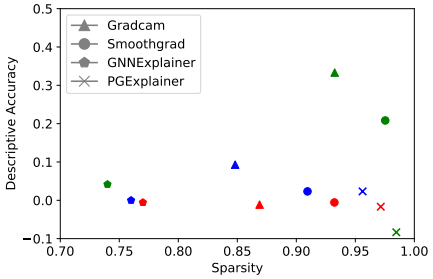
RQ3. — *How do extrinsic and intrinsic criteria differ when comparing explanation methods?* We evaluate two commonly used intrinsic criteria for ReVeal, Devign, VulDeeLocator and LineVul on the PUTs in Figure 7. We measure the sparsity (MAZ) and descriptive accuracy for each EM and PUT. The higher and 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

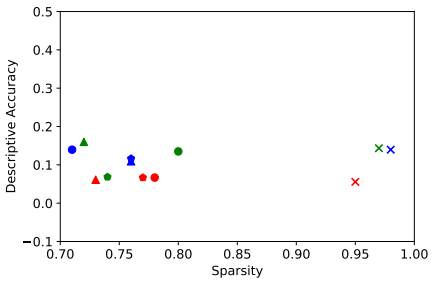


(a) Pareto map for Devign. (b) Pareto map for ReVeal.

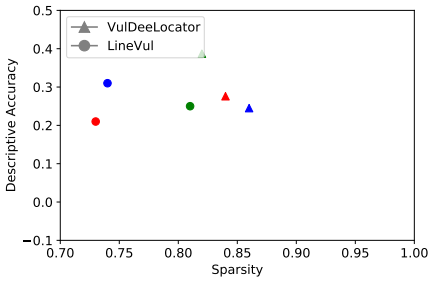
Figure 6: Mean breakpoint hits (M_2) and mean crash distance (M_3).



(a) ReVeal



(b) Devign



(c) Models with integrated EMs

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

Accuracy as LineVul, which mirrors the results from their intrinsic evaluation [20].

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.

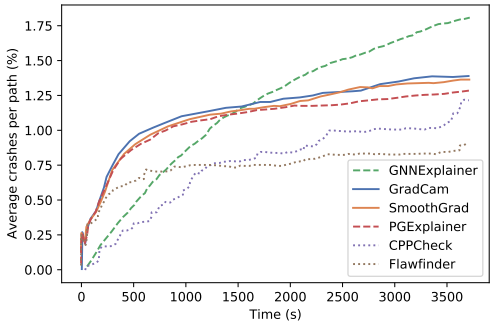
RQ4. — *How do rule-based auditing tools perform against explanation methods?* We compare Flawfinder and CPPCheck against the explanation methods in Table 2. All static analyzers are inferior in our experiments and only beat the random baseline on Giflib. Figure 8a exemplary visualizes the average crashes 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.

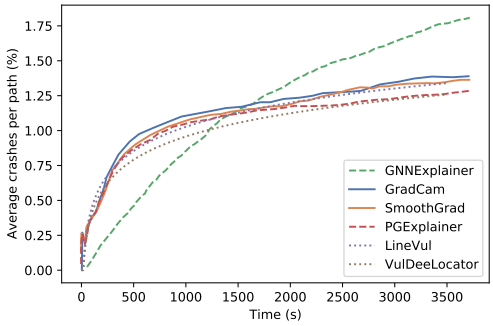
Discussion. As previously pointed out, another advantage is that our extrinsic criteria allow us to benchmark vulnerability discovery models with their EMs against classical rule-based static analyzers, which is not possible with intrinsic criteria. In our experiments given Figure 8a, we see that all 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 grey-box 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	

F

Broken Promises: Measuring Confounding Effects in Learning-based Vulnerability Discovery

Broken Promises: Measuring Confounding Effects in Learning-based Vulnerability Discovery

Erik Imgrund
erik.imgrund@sap.com
SAP Security Research
Germany

Tom Ganz
tom.ganz@sap.com
SAP Security Research
Germany

Martin Härterich
martin.haerterich@sap.com
SAP Security Research
Germany

Lukas Pirch
lukas.pirch@tu-berlin.de
Technische Universität Berlin
Germany

Niklas Risse
niklas.risse@mpi-sp.org
Max-Planck Institute
Germany

Konrad Rieck
rieck@tu-berlin.de
Technische Universität Berlin
Germany

ABSTRACT

Several learning-based vulnerability detection methods have been proposed to assist developers during the secure software development life-cycle. In particular, recent learning-based large transformer networks have shown remarkably high performance in various vulnerability detection and localization benchmarks. However, these models have also been shown to have difficulties accurately locating the root cause of flaws and generalizing to out-of-distribution samples. In this work, we investigate this problem and identify spurious correlations as the main obstacle to transferability and generalization, resulting in performance losses of up to 30% for current models. We propose a method to measure the impact of these spurious correlations on learning models and estimate their true, unbiased performance. We present several strategies to counteract the underlying confounding bias, but ultimately our work highlights the limitations of evaluations in the laboratory for complex learning tasks such as vulnerability discovery.

CCS CONCEPTS

• **Security and privacy → Software and application security**;
• **Computing methodologies → Machine learning**.

KEYWORDS

Vulnerability Discovery, Confounding Effect, Overfitting, Causal Learning, Large Language Models

ACM Reference Format:
Erik Imgrund, Tom Ganz, Martin Härterich, Lukas Pirch, Niklas Risse, and Konrad Rieck. 2023. Broken Promises: Measuring Confounding Effects in Learning-based Vulnerability Discovery. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec '23), November 30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3605764.3623915

AISec '23, November 30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0260-0/23/11.
https://doi.org/10.1145/3605764.3623915

1 INTRODUCTION

The traditional approach to finding vulnerabilities in software relies on manual code review and extensive testing. This approach is time-consuming, resource-intensive, and prone to human error. Static program analysis, on the other hand, supports developers and security professionals in automatically identifying and locating potentially flawed areas without actually running the program. Unfortunately, such static application security testing (SAST) tools often report many false positive alerts, which consequently require expensive manual triage as well.

As a remedy, methods for learning-based vulnerability detection have been proposed to automatically derive rules from historical data to increase the detection rate while at the same time pertaining to a lower false-positive rate [31, 39]. These machine learning (ML) models have been shown to outperform rule-based SAST tools under laboratory settings [15, 65]. There is a vast number of learning-based detection methods available differing mainly in their model architecture, dataset, and the preprocessing techniques they use. For instance, there exist sequence-based solutions from the NLP domain [31, 39], graph neural networks for code analysis [7, 9, 50, 65] and more recently transformer-based models [8, 15, 46].

With large language models (LLMs) being on the rise, transformer networks have been trained on many code-centric tasks, achieving remarkable results on, for instance, code clone detection [61], code completion [33], code generation [45] and code summarization [51]. Naturally, this progress has also inspired LLM-based approaches for vulnerability detection and localization. However, since training an LLM requires a vast amount of data and resources, techniques categorized as one-shot and few-shot learning have been adopted to fine-tune pre-trained models in this setting. Novel advances like adapters [22] or low-rank adaption [23] yield possibilities to optimize pre-trained LLMs for very specific learning tasks. Similarly, it has been shown that the learned representation of pre-trained LLMs beneficially supports the performance on fine-tuned downstream tasks [24].

As a consequence of this development, recent works present an astonishing performance on the discovery and localization of defects in source code using fine-tuned LLMs, beating prior learning-based and rule-based methods by far and achieving more than 90% balanced accuracy under in-lab conditions [15]. A further benefit of these transformer networks is improved vulnerability localization through the interpretation of token attention scores as a measure

of code importance [15, 40]. This success of transformer networks, however, is overshadowed by a notable weakness: The models fail to generalize to out-of-distribution samples [8]. That is, high performance is only achievable if the training and test data come from the same software project, which obviously undermines the practical utility of learning-based vulnerability discovery.

In this work, we explore the reason for this deficit and show that transformer networks suffer from spurious correlation, hindering generalization and transferability. These correlations create a confounding bias in the learning models, which impacts the detection as well as localization of vulnerabilities. The networks not only fail to identify out-of-distribution data but also hint at irrelevant code when explaining their decisions. To characterize this problem, we propose a methodology to measure the impact of spurious correlations on learning-based vulnerability discovery using the framework of causal inference. To this end, we correlate the loss in performance with semantics-preserving transformations that gradually change the appearance of the code. We find that even minor tweaks in style, control flow, or variable naming are enough to render transformer networks unusable.

We propose three techniques to counter the underlying confounding effects: First, we observe that graph neural networks (GNNs) are less susceptible to artifacts in the dataset and hence offer an alternative architecture for vulnerability discovery. Second, we can mitigate some confounders by pre-tokenizing the input for the LLMs, and finally, we propose to normalize code to a canonical representation before passing it to LLMs. The latter achieves the best overall results. Although we cannot completely eliminate spurious correlations, their impact can be reduced significantly, enabling research to avoid confounding effects.

The rest of this paper is structured as follows: After an introduction to LLMs and graph representations for vulnerability discovery in Section 2, we detail the problem setting and our methodology in Section 3. In Section 4, we present our empirical evaluation and discuss the results in Section 5, ending with related works and our conclusion in Section 6 and Section 7, respectively.

2 VULNERABILITY DISCOVERY

We start by introducing the basic concepts of large language models (LLMs) and graph neural networks (GNNs) for the task of vulnerability discovery, before exploring their limitations.

2.1 Vulnerability Discovery

A vulnerability detection method aims to derive a single score indicating the vulnerability likelihood of a program based on a particular representation of it. This is expressed in Definition 1, which defines a decision function that takes a piece of code and maps it to the probability of it being vulnerable.

Definition 1. *A method for static vulnerability discovery is a decision function* $f_\theta \colon x \mapsto P(\textsc{vulnerable} \mid x)$ *that maps a code sample* x *to its probability of being vulnerable [17].*

Learning-based methods for vulnerability discovery utilize a parameterized classification function f_θ as depicted in Definition 1, whose weights θ are optimized during training on a dataset of vulnerable and non-vulnerable code samples [18]. We denote the classes with prediction probabilities as Vulnerable and Clean, where the former denotes a sample with a code bug present and the latter denotes a sample without bugs.

2.2 Large Language Models

Prior works apply models borrowed from the natural language processing domain to vulnerability discovery [31]. This includes the interpretation of code as a natural sequence of tokens. Models like recurrent neural networks (RNNs) or long short-term memorys (LSTMs) are naturally suited for this task [31, 39, 65]. With the rise of LLMs, which are in essence large transformer models, such networks are increasingly used and fine-tuned for the task of detecting and locating code defects.

Typically, a transformer model consists of either an encoder, a decoder, or both [32]. Each part is composed of multiple blocks consisting of bidirectional multi-head self-attention mechanisms and feed-forward neural networks. Compared to RNNs, transformer models are not limited by the Markov property, where the last hidden state of an RNN needs to store the latent representation of the entire program. Instead, attention matrices produce an attention vector for each token denoting the influence of each other token in the sequence [15]. Since the self-attention mechanism is the heart of LLMs, we define it formally using the original notation by Vaswani et al. [48]:

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}V\right) \tag{1}$$

The matrix $Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$ denotes a query containing the set of representations for the current tokens, which is then multiplied with the key matrix $K \in \mathbb{R}^{d_{\text{model}} \times d_k}$. The result is scaled by the inverse square root of the embedding size d_k and finally, after a softmax, used as an index to the value matrix $V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ yielding the attention vector. d_{model} denotes the size of the vocabulary and the Q, K and V matrices can be split into multiple attention-heads to capture richer semantics. Recent methods for vulnerability discovery, such as LineVul [15], use pre-trained transformer networks, like CodeT5 [52], BERT [12] or RoBERTa [34], and fine-tune them on vulnerable code.

2.3 Graph Neural Networks

Since programs can be modeled as directed graphs [1, 5, 58], a different strain of research has explored graph representations for source code instead of flat token sequences [9, 50, 65]. We refer to the resulting program representation as a *code graph* and denote the underlying directed graphs as $G = G(V, E)$ with vertices V and edges $E \subseteq V \times V$. Moreover, nodes and edges of the graphs are attributed, that is, elements of V or E are assigned values in a feature space that characterize local properties of the code.

Different code graphs capture different syntactic and semantic features. A popular representation is the code property graph (CPG) by Yamaguchi et al. [58], which is a combination of the abstract syntax tree (AST), the control flow graph (CFG), and the program dependence graph. Other approaches use different combinations, for instance, combining the AST with the CFG and the data flow graph (DFG) [6]. Using such representations, research has started to focus on graph convolutional networks (GCNs) [65]. These networks are a class of deep learning models realizing a function

$f\colon G(V,E) \mapsto y \in \mathbb{R}^d$ that can be used for the classification of graph-structured data [40].

GCNs can be viewed as a generalization of convolutional neural networks (CNNs), just as an image can be viewed as a regular grid graph where each pixel denotes a node in the graph connected by edges to its neighboring pixels [57]. A graph convolutional network needs two mandatory input parameters, that is, an initial feature matrix $X \in \mathbb{R}^{N\times F}$, with N being the number of nodes in the graph and F the number of features per node, and the topology commonly described by the adjacency matrix $A \in [0,1]^{N\times N}$. The most popular GCN types belong to so-called message passing networks (MPNs) where the prediction function is computed by iteratively aggregating and updating information from neighboring nodes. One of the simplest MPNs is the one defined by Kipf and Welling [26]:

$$h^{(l)} = \sigma(\hat{A}\, h^{(l-1)} W^{(l-1)}) \tag{2}$$

with $h^0 = X$ [26]. Here, the intermediate representations are linearly projected and sum-wise aggregated according to the normalized adjacency matrix $\hat{A}$ with self-loops followed by a non-linear activation function. These GCNs can be stacked to learn filters w.r.t. larger neighborhoods. Other GCN layers use different aggregation and update mechanisms, for instance, instead of an multilayer perceptron (MLP), gated graph neural networks (GGNNs) use gated recurrent unit (GRU) cells to update the hidden state of nodes [29], while graph attention network (GAT) layers use attention mechanisms [49]. We refer the reader to the overview article by Wu et al. [57] that discusses GNNs in detail.

Because of the fitting premise of GCNs, they have been widely adopted for representation learning on code graphs. The graph-based approaches in recent literature outperform classical SAST tools and older sequential models, such as VulDeepecker [31] or Draper [39]. Graph-based methods like Devign [65] and ReVeal [7] are currently among the best learning-based approaches for vulnerability discovery, though with lower performance than recent methods based on LLMs.

3 METHODOLOGY

We proceed to outline the problem setting and introduce our methodology for measuring the impact of spurious correlations.

3.1 Problem Setting

Currently, many popular learning-based vulnerability detectors exist with varying efficiency. Furthermore, previous works have shown, that although these approaches provide promising results, their ability to precisely pin down the root cause of a bug is lacking. It is questionable, how a security practitioner should respond when a model classifies a function as vulnerable if the model is unable to precisely locate the bug.

Some models come with an integrated explanation mechanism, for instance, LLMs, while others can be enhanced using model-agnostic explanation mechanisms [42], such as Class Activation Maps (CAM) or SHAP [35]. These explanation methods provide a more fine-grained view of the decision of the model and can be used for line-level or even statement-level bug localization. However, these methods provide vastly differing results [54, 66] and a fair comparison is generally non-trivial [3, 16]. It has been shown that vulnerability discovery models tend to focus on irrelevant artifacts in the provided data [16] and that their measured performance may be biased with respect to practice [3, 7].

A motivating example. Let us consider the example in Figure 1. Here, LineVul [15] correctly identifies a bug in the given C function. However, the model falsely claims that the root cause lies in line 2, that is, the instantiation of a matrix on the stack. The actual cause of the vulnerability, however, is a type confusion in lines 3 and 5. The variable `var` is pulled from a hash map and then, without further checks, converted to a double[1]. The matrix plays no role in this vulnerability and thus the explanation misleads a manual investigation of the finding.

```
1 ...
2 float matrix[3][3] = {{0,0,0}, {0,0,0}, {0,0,0}};
3 if (zend_hash_index_find(Z_ARRVAL_PP(var), (j), (void **)
      &var2) == SUCCESS) {
4     SEPARATE_ZVAL(var2);
5     convert_to_double(*var2);
6     matrix[i][j] = (float)Z_DVAL_PP(var2);
7  ...
```

Figure 1: Type confusion bug in the PHP Zend engine.

Interestingly, the tokens with the greatest attention scores from LineVul in line 2 consist of "float", "}}" and "{{". In the training set, these tokens co-occur 198 times for vulnerable functions and only 67 times for non-vulnerable functions, thus creating a spurious correlation. The model incorrectly learns this correlation as an indicator for a vulnerable function. Obviously, there must be more biases present in the training dataset such as the one identified here, which makes the model concentrate on irrelevant artifacts.

The problem becomes worse when we try to slightly change the coding style and obfuscate some lines as seen in Figure 2. Although the function is semantically equivalent and only minimally changed, LineVul now classifies this function as clean, despite the original vulnerability being still present.

```
1 ...
2 float matrix[3 & 0xF][3 & 0xF] = {
3   {0o0, 0o0, 0o0}, {0o0, 0o0, 0o0}, {0o0, 0o0, 0o0}};
4 if (zend_hash_index_find(Z_ARRVAL_PP(var), (j), (void **)&
       var2) ==
5     SUCCESS) {
6   SEPARATE_ZVAL(var2);
7   convert_to_double(*var2);
8   matrix[i][j] = (float)Z_DVAL_PP(var2);
9  ...
```

Figure 2: Obfuscated version of type confusion bug.

To visualize the effect of spurious correlation, we present Figure 3 which derives a simple causal model for a vulnerability discovery function f_θ [44]. Here, X is the input data and Y is the label, being either VULNERABLE or CLEAN. The learning goal of f_θ is to find a relationship between the learned representation R to the label Y. There is a relationship $C \leftarrow X \rightarrow A$ denoting that the causal features C and trivial or biased feature patterns A both influence the final latent representation R. Missing or different artifacts in unseen data then weaken the model performance.

[1] https://cve.report/CVE-2014-2020

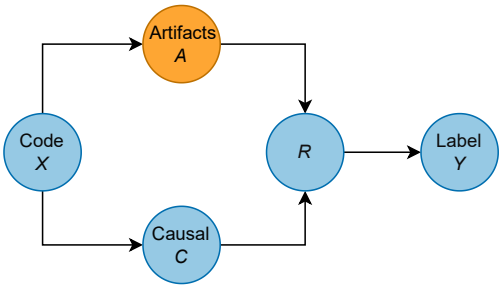


Figure 3: Simple causal model of machine learning for vulnerability discovery.

The artifacts are called *confounders* in causal learning and in our example, the confounders originate from irrelevant features that cause the model to learn biased representations. A simple example for an artifact would be a common code style in all vulnerable samples and a different style for all clean samples. The ultimate goal in this work is not only to measure f and the actual influence of A but also to remediate this influence since these learned artifacts hinder the model from generalizing well over unseen or out-of-distribution data, that is, real-world code.

To this end, we identify three sources of bias that can manifest as artifacts in program code:

(1) *Coding style.* Every collected sample has an implicit coding style. Since many open-source projects use automatic linting, it is likely that samples from one project to another differ in their styles. If there are more vulnerable samples from one project than another, the coding style correlates with Y.

(2) *Control flow.* Projects often contain different calling hierarchies or indirections due to programming patterns, for instance, object-oriented design principles. Several projects and authors prefer one pattern over another which may introduce further confounding bias.

(3) *Naming.* Different samples from different projects naturally vary in their naming conventions. Hence, vulnerable samples may potentially differ in variable naming compared to clean samples. Although it is common to mask such symbols, recent works unfortunately desist from normalizing them.

This list is non-exhaustive since, potentially, there can be an infinitely large number of artifacts. Nonetheless, we state that a model is confounded if artifacts heavily influence its decision. Since other works do not account for this, we propose a more reliable evaluation methodology.

3.2 Evaluating Models

Current models for learning-based vulnerability discovery suffer from low transferability and generalizability. Yet, they pertain to a high true positive and true negative rate on test sets aligning with the training distribution [7, 8]. How a security expert can gain insight into how well the model performs in practice is an open research question. The performance can not be truly measured on the test set, as it is of the same distribution as the training set and might lack diversity compared to real-world code samples.

Training the model on one dataset and testing on another is a possible evaluation approach; however, it remains unclear how many datasets one has to evaluate. Worse still, vulnerability datasets are scarcely available [7]. Using more datasets improves the insights gained, but increases the amount of data and computing resources necessary for the evaluation. As a consequence of this situation, we propose a method that uses only one dataset.

To motivate our evaluation scheme, we briefly introduce the concept of causal inference that reveals influencing variables. If we inspect Figure 3, it is trivial to see that the representation learned by a model f directly influences the predicted label Y. But instead of using the sample under analysis to directly influence the representation, we model it so that X has a causal and a trivial part [44]. We call the latter the confounding variable, shortcut feature, or spurious correlation [44]. As a result, we have three relationships: $A \leftarrow X \rightarrow C$, $C \rightarrow R \leftarrow A$, and $R \rightarrow Y$. To measure the true causal correlation and to remove confounding variables in causal learning, it is common to calculate the influence of one variable affecting the other by *intervention*.

This can be done using do-calculus [36], that is, we can stratify the confounder by calculating the influence of $C \rightarrow Y$ given all possible artifacts from $a \in A$ [36]:

$$P(Y|do(C)) = \sum_{a \in A} P(Y|C, A = a)P(a) \tag{3}$$

We approximate the distribution of A by calculating the estimated likelihood of the code samples $X = (x_0, \ldots, x_n)$ with our different perturbations. As using all possible artifacts is not feasible, we use a subset $A' \subset A$ and define an artifact $a \in A'$ to be a variant of the code samples $k_a(X) = (k_a(x_0), \ldots, k_a(x)_n)$ obtained by one of our perturbations $a \in A'$. Equation 3 then becomes:

$$P(Y|do(C)) \approx \sum_{a \in A'} P(Y|C, A = a)\hat{P}(a). \tag{4}$$

We estimate $\hat{P}(a) = \frac{P_\theta(a)}{\sum_{a' \in A'} P_\theta(a')} \approx P(a)$ empirically by calculating the likelihood of a particular variant of the code sample $P_\theta(a)$ utilizing a generative LLM with weights θ. We calculate the likelihood of each token of the code sample dependent on the previous tokens. Since calculating the likelihood of the entire sequence by multiplying the individual token likelihoods is numerically infeasible, we instead average the log-likelihoods over the entire sequence to obtain the approximate likelihood, similar to the calculation of the perplexity, a popular metric for generative LLMs [37].

Further, we can measure the impact of the artifacts on the model by calculating the average relative difference between the original model decision compared to the decision when every artifact is marginalized. We call this difference the *confounding effect*,

$$c = \frac{\sum_{a \in A'} P(Y|C, A = a)\hat{P}(a) - P(Y|C, A)}{P(Y|C, A)} \tag{5}$$

As an intuition, consider $c = 0$, meaning that $P(Y|C, A) = P(Y|do(C))$. However, the more c deviates from 0, the greater the influence of the artifacts and $P(Y|C, A) \neq P(Y|do(C))$.

The application of different perturbations to the code should resemble a causal intervention. A non-confounded model should perform equally on semantically equivalent but perturbed code since the decision should solely depend on the causal feature part.

Let us define the model predictions on the code samples under different perturbations as $\forall a \in A' : f(k_a(x))$. Consequently, this intervention provides insight into how the model behaves under different artifacts and yields a more robust basis for model evaluation and comparison. In a more practical sense, suppose that we have a metric $\mathcal{M}: f \to \mathbb{R}$ assessing the quality of a learning-based vulnerability discovery model, such as the accuracy. We then have

$$c = \frac{\sum_{a \in A'} \mathcal{M}(f(k_a(x))\hat{P}(a) - \mathcal{M}(f(x))}{\mathcal{M}(f(x))} \tag{6}$$

and can measure the influence of the confounder using the *confounding effect* as defined above.

3.3 Reducing Confounder

The measurement of the confounding effect of artifacts on the model is one side of the coin, but on the other side, we also want to reduce the influence of such trivial patterns on the model. We propose three methods that can be applied to either LLMs or GNNs for vulnerability discovery that mitigate the effects in practice.

LLMs with normalized code. To remove the effect of style artifacts on LLMs, one naive solution is to normalize the code. Code normalization is the modification of code so that it conforms to a given style guide, which reduces, but does not remove, the impact of the personal style on the code [21]. As a code normalization method, we apply one code style to all methods and use the uniformly formatted code as training data. This has the same effect of normalizing the coding style, and thus removing style artifacts while preserving closeness to real-world code and thus making better use of the pretraining than the next method. We abstain from normalizing the variable naming by masking the variable names during training so that the effect of the naming artifacts can be measured as part of our evaluation.

LLMs with pre-tokenized code. Another solution follows from the work of Roziere et al. [38], who propose to tokenize the code before applying the byte-pair encoding using a programming language-specific lexer. They feed the resulting tokens as space-separated plain text into the model, a process we refer to as *pre-tokenization*. We adopt the same methodology, but as our models were trained with untokenized code, we expect a performance drop from the different data distributions obtained, as the code samples now lack all newlines and other style practices common to real-world code. Retraining an LLM completely on tokenized code is impractical, as the main benefit of LLMs is the adaptability to several tasks using fine-tuning.

Causal graph learning. A more principled approach arises from the work of Sui et al. [44] in the domain of GNNs: By applying an intervention directly to the learning model, they can mitigate the impact of confounding variables. This is done by conditioning the causal input features, in this case, a code graph, per sample with all possible trivial subgraphs obtained during training.

To encode the input graph, a graph isomorphism network (GIN) layer is used followed by two MLPs to calculate a relevance score for nodes and edges. For any node $v_i \in V$ we calculate their node attention and for any pair of nodes $(v_i, v_j) \in E$ their edge attention. Furthermore, the output dimension of the MLPs is halved to perform a latent space disentanglement, where the first half will later be optimized to contain only the relevant nodes that are causal for our task and the second half will be trained to only contain the trivial part of the graph, in this case, the artifacts.

A mean readout layer is applied last as a pooling strategy followed by a final MLP with softmax activation as the prediction head returning either VULNERABLE or CLEAN. Using the attention scores we can calculate attention masks for both, the causal and trivial nodes, and causal and trivial edges. We apply these masks to the adjacency matrix and feature matrix of the input graph resulting in the causal and trivial subgraphs respectively. The causal graph can be used to explain the prediction and track the cause of a vulnerability.

To train the model in a supervised fashion, we first apply a traditional negative log-likelihood (NLL) loss to our ground truth and the latent representation of the causal graph. Then, we take the representation of the trivial subgraph and optimize the model to separate trivial and causal features by fitting the softmax distribution using the trivial graph to a uniform distribution using the Kullback-Leibler divergence (KL). Finally, to stratify the confounder, another NLL loss is calculated between the ground truth and the prediction, while the causal graph is augmented with a random trivial subgraph from another graph in the dataset. During the training procedure, the model essentially learns to ignore trivial patterns.

4 EVALUATION

In this section, we describe the experimental setup and the results of our evaluation. The experiments are devised to give answers to the following research questions. We publish our code for easy experimental reproduction [2].

- RQ1: Are confounding effects measurable?
- RQ2: How do artifacts influence vulnerability localization?
- RQ3: Can the confounding effect be reduced?

4.1 Experimental Setup

We rely on *Fraunhofer-CPG* by Weiss and Banse [55] and *networkx* [19] as tools to generate code graphs. The graph-based models are implemented using *Pytorch Geometric* [14] and trained on AWS EC2 g4dn instances. We use the transformers library [56] to fine-tune the transformer-based models. The tokenization of code is calculated using *tree-sitter* as the parser. The hyperparameters of all models are documented in Table 1.

[2]https://github.com/SAP-samples/security-research-confoundingeffects

Table 1: Hyperparameters used for considered models.

	ReVeal	StackLSTM	CGIN	CodeT5+	LineVul
Optimizer	Adam			AdamW	
Learning Rate	$5 \cdot 10^{-4}$	10^{-4}	10^{-4}	10^{-5}	10^{-5}
Epochs	70	50	14	3	10
Batch Size	128	1	2	8	8
Warmup Steps	0	0	0	50	50
Weight Decay	10^{-4}	0	0	0.01	0.01
Number of parameters	719k	1.7M	222k	223M	249M

Table 2: Function-level accuracies with different augmentations. CodeT5+$_n$ and LineVul$_n$ are trained on normalized data and CodeT5+$_t$ is trained on tokenized data.

Transformation	ReVeal	CGIN	StackLSTM	CodeT5+	CodeT5+$_n$	CodeT5+$_t$	LineVul	LineVul$_n$	LineVul$_t$
None	63.57%	61.55%	62.79%	**94.62%**	83.15%	59.51%	*93.09%*	83.62%	65.21%
Chromium	63.62%	62.24%	62.79%	59.21%	*66.46%*	59.56%	58.76%	**68.68%**	65.25%
Mozilla	63.59%	61.39%	62.84%	59.18%	*66.54%*	59.56%	57.67%	**67.63%**	65.25%
Google	63.53%	63.42%	62.79%	59.16%	*66.20%*	59.56%	58.95%	**68.39%**	65.25%
LLVM	63.43%	60.37%	62.84%	59.01%	*66.54%*	59.56%	58.73%	**68.22%**	65.25%
Uglify	**62.29%**	58.02%	61.04%	50.67%	55.62%	55.32%	50.44%	55.77%	60.30%
Uglify (-Whitespace)	**62.86%**	58.47%	*61.04%*	54.15%	57.37%	55.35%	49.51%	55.88%	61.02%
Obfuscate	50.51%	59.41%	50.74%	53.61%	59.41%	*60.73%*	52.11%	57.29%	**60.74%**
Obfuscate (+Format)	50.46%	56.12%	50.71%	37.14%	58.13%	*61.19%*	40.56%	53.60%	**61.44%**
Obfuscate (-Whitespace)	49.76%	55.28%	53.12%	52.05%	*55.79%*	55.50%	51.02%	53.68%	**60.92%**
Causal Accuracy	64.18%	61.59%	64.75%	59.09%	*65.41%*	59.80%	58.71%	**67.46%**	65.02%

Dataset. We use *Big-Vul* [13] as the underlying dataset for all of our experiments, as it is one of the biggest currently available datasets with line-level defect information. The dataset consists of 26 635 vulnerable and 352 606 non-vulnerable functions from different code repositories.

Transformations. We apply transformations on the dataset that implicitly remove artifacts to obtain different variants of the dataset for training and evaluation. These transformations can be categorized into three classes:

(1) *Styling.* We apply style formatting using clang-format [21] with different popular predefined styles. We test the predefined styles *Chromium*, *Google*, *LLVM*, and *Mozilla* for a diverse range of style choices. As previously described, applying the formatting removes style-related artifacts.

(2) *Uglification.* We test two different kinds of "uglification". The first variant consists of removing comments and renaming all variables to a string of twelve randomly chosen lowercase letters and applying style normalization. The second variant is the same except for additionally removing all unneeded whitespace. This transformation removes artifacts in code style and naming whilst also partially removing causal information, as the variable names are typically chosen deliberately and essential to detect vulnerabilities.

(3) *Obfuscation.* The obfuscation consists of randomly renaming variables and functions, removing comments, adding unneeded statements, adding function definitions, and replacing numbers with an obfuscated equivalent number. The numbers are obtained by converting them to decimal, binary, octal, or hexadecimal. Additionally, we evaluate the models on the obfuscated code after applying a predefined coding style and after removing all unneeded whitespace. The obfuscation also removes control-flow artifacts, since statements are randomly inserted.

Models. We have chosen a diverse set of models for evaluation, both graph-based and text-based. For graph-based models, we train and evaluate ReVeal as a state-of-the-art GNN for vulnerability detection [7] and causal graph isomorphism network (CGIN) [44] as a causal graph model that mitigates the effect of artifacts. For text-based models, we choose LineVul [15] and fine-tune a CodeT5+ [52] model with 220 million parameters similar to Chen et al. [8] and Thapa et al. [46]. We fine-tune the transformer models on the original code available as part of the dataset. Additionally, we train the models on normalized and tokenized code.

Delétang et al. [11] show that transformer models cannot generalize well over different-sized input token lengths. The authors show that classical LSTMs with differentiable memory provide stronger generalization performance than transformer models on increasingly complex tasks. Since the number of tokens within samples can impose another bias, we extend VulDeepecker [31], a LSTM-based vulnerability discovery model, with a differentiable stack [27]. VulDeePecker is generally inferior to the other models [7]. However, an LSTM with access to a stack has been shown to provide advantageous results for regular and context-free tasks, similar to the capabilities of a real-world parser [11].

Evaluation tasks and metrics. The models are evaluated based on two tasks: function-level binary classification with the classes VULNERABLE and CLEAN and line-level classification of known-vulnerable functions. We use the balanced accuracy, as the mean between the true positive and true negative rate, for both tasks, due to the heavily imbalanced datasets. Additionally, we measure the causal accuracy as the balanced accuracy based on causal predictions according to Equation 4. Further, we use the balanced accuracy as metric $\mathcal{M}$ for measuring the confounding effect from Equation (6). For the line-level task, we use the top-1, top-3, and top-5 accuracy as introduced by Fu and Tantithamthavorn [15].

To obtain a ranking of the lines by each model, we use model-specific explainability methods. For the graph-based models we obtain node relevance scores and then propagate these scores to all lines included in the node to arrive at a line relevance [16]. We obtain node relevance scores for ReVeal by applying Grad-CAM [41] and for CGIN by utilizing the causal node attention scores. For the transformer-based models, we calculate the relevance of each line in the same way as proposed by Fu and Tantithamthavorn [15]. The relevance of each token in the line is summed to obtain the aggregate line relevance. The token relevance is similarly obtained as the attention to each token in the first layer of the encoder.

As our transformations can change the layout of the code and thus the locations of the vulnerable lines, we need to match the original lines to the transformed lines. As the matching is non-trivial and harder with additional transformations, we only test on the original dataset and the styled variants. We match the vulnerable lines to the formatted lines if one of the lines is an exact substring of the other line without considering whitespace. As a baseline, we also measure the expected top-k accuracy with random line orderings, which can be calculated based on the expected best rank of any vulnerable line, which is distributed according to a particular *negative hypergeometric* distribution. More formally, the expected rank is $E(X)$ for $X \sim NHG(|L|, |L_0|, 1)$ with the set of all lines L and the set of non-vulnerable lines L_0. By calculating the expected rank of each code sample in our test set, we can obtain the expected top-k accuracy for a random baseline.

4.2 Results

In this section, we provide our experimental results and use the outcomes to provide answers to our research questions.

RQ1: Are confounding effects measurable? In Table 2, we present our results for the function-level prediction scores for all eight models measured in their balanced accuracy. The best performance per transformation is bold and the second best value is italic. CodeT5+ and LineVul have an initial balanced accuracy of 94.62% and 93.09% on the test set, respectively. That is an approximate increase of 50% relative to ReVeal, StackLSTM, and CGIN. Interestingly, the detection performance of the transformer models shrinks to a detection rate lower than that of the other models when the test samples are transformed using different styles. With less than 60% balanced accuracy CodeT5+ and LineVul are performing worse than the non-transformer-based models having about 63% balanced accuracy on average. This is a clear hint that the transformer models overfit on artifacts in the train and test distribution aligning with the coding style. The discrepancy between the performances with and without augmentations for the other models is negligible, with CGIN having the overall worst results with approximately 61%, followed by StackLSTM with 62% and ReVeal with 63%.

The situation becomes worse when comparing the performances of the transformer models when provided with uglified code, that is, inlining functions, and removing whitespace and tabs. While the other models pertain to a comparable performance at around 60%, the performance of LineVul and CodeT5+ is not different from random guessing. The uglifier without the removal of whitespace leaves CodeT5+ with 54%, while LineVul is still comparable to a random guesser. Using an obfuscator is the most drastic augmentation transformation, as it even changes control flow and adds superfluous function calls. All models, except CGIN, are not significantly better than random guessing in this scenario.

Furthermore, when comparing the accuracy obtained based on causal-only features, it is obvious that the initial performance of CodeT5+ and LineVul is based on artifacts instead of causal features with only 59.80% and 58.71% causal accuracy, respectively. The graph-based models ReVeal and CGIN as well as StackLSTM on the other hand, obtain a causal accuracy near their initial performance, indicating that their predictive performance is based on causal features instead of artifacts.

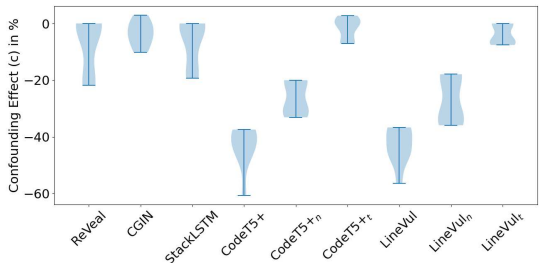


Figure 4: Confounding effect on function-level accuracy.

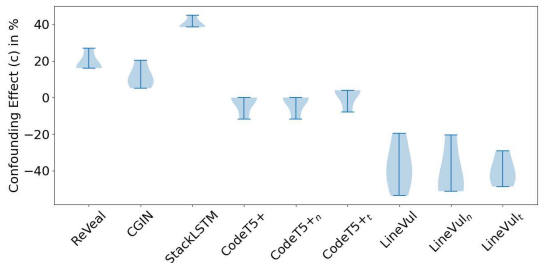


Figure 5: Confounding effect on Top-5 line-level accuracy.

We show that there are indeed artifacts encoded in the training set that negatively influence the models when transferring to semantically equivalent code. If the model already fails in predicting vulnerabilities in the test set with the same distribution as the training set but with minor changes, it is highly uncertain how it behaves on real data. The reported performance scores of the models are not suitable for a veracious comparison. While the initial highest performance is 94.62%, the true models' capabilities collectively level off at around 60%. In summary, the influence of the confounding artifacts distorts the reported performance by up to 60% measured by the discrepancy between the test and augmented test set. Figure 4 summarizes the confounding effect on different models visualizing the degrees of performance drop. The transformers are in fact the models most affected in this experiment.

RQ2: How do artifacts influence vulnerability localization? Table 3, Table 4 and Table 5 show the top-1, top-3 and top-5 line-level accuracy, respectively. The best scores per transformation are highlighted in bold. The first column denotes the performance of the random baseline. This is included as a reference for the performance of the models, as well as to show the limitations of our measurement method. Due to the fuzzy matching of formatted lines to the original lines, the number of total lines varies in the formatted code, and consequently, the expected performance changes slightly.

Although the performance of the transformer models for function-level prediction is greatly influenced by the application of code formatting, only much smaller differences for line-level localization can be observed. For LineVul, the top-1 accuracy drops from 40.33% to at most 38.67% while for CodeT5+ the performance increases from 38.67% up to 44.75%. For the top-3 accuracy, the same effect

Table 3: The Top-1 line-level Accuracy of the Transformer and Graph models with different code styles.

Codestyle	Random	ReVeal	CGIN	StackLSTM	CodeT5+	CodeT5+$_n$	CodeT5+$_t$	LineVul	LineVul$_n$	LineVul$_t$
None	26.86%	43.09%	**43.65**%	39.23%	38.67%	38.67%	38.67%	40.33%	39.78%	41.44%
Chromium	26.86%	44.20%	43.65%	**45.86**%	44.75%	44.75%	44.75%	37.02%	37.02%	37.02%
Mozilla	25.71%	43.65%	42.54%	**44.20**%	40.88%	40.88%	40.88%	38.67%	37.57%	39.23%
Google	28.57%	44.75%	44.20%	**45.30**%	44.75%	44.75%	44.75%	38.67%	37.57%	37.57%
LLVM	28.57%	44.20%	44.20%	**45.30**%	43.65%	43.65%	43.65%	35.91%	34.25%	34.81%

Table 4: The Top-3 line-level Accuracy of the Transformer and Graph models with different code styles.

Codestyle	Random	ReVeal	CGIN	StackLSTM	CodeT5+	CodeT5+$_n$	CodeT5+$_t$	LineVul	LineVul$_n$	LineVul$_t$
None	46.86%	59.67%	58.01%	58.56%	**64.64**%	**64.64**%	**64.64**%	62.43%	63.54%	64.09%
Chromium	46.86%	60.77%	57.46%	60.77%	**64.64**%	64.09%	63.54%	56.91%	55.25%	56.35%
Mozilla	45.14%	61.88%	58.56%	60.77%	**64.09**%	63.54%	63.54%	61.88%	60.77%	60.22%
Google	47.43%	61.33%	60.77%	61.88%	**64.64**%	64.09%	64.09%	55.25%	55.25%	55.25%
LLVM	46.86%	61.33%	58.56%	61.88%	**64.64**%	64.09%	64.09%	56.91%	56.35%	55.80%

Table 5: The Top-5 line-level Accuracy of the Transformer and Graph models with different code styles

Codestyle	Random	ReVeal	CGIN	StackLSTM	CodeT5+	CodeT5+$_n$	CodeT5+$_t$	LineVul	LineVul$_n$	LineVul$_t$
None	57.71%	67.96%	68.51%	66.30%	**71.82**%	**71.82**%	**71.82**%	69.06%	68.51%	69.06%
Chromium	57.71%	69.61%	69.06%	69.61%	71.82%	71.82%	**72.38**%	64.64%	64.09%	64.64%
Mozilla	57.71%	**70.72**%	69.61%	70.17%	70.17%	70.17%	**70.72**%	66.85%	66.30%	65.75%
Google	57.71%	69.61%	70.72%	69.61%	71.82%	71.82%	**72.38**%	62.98%	62.98%	63.54%
LLVM	57.71%	69.61%	69.61%	69.61%	71.82%	71.82%	**72.38**%	64.64%	62.98%	64.09%

can be seen for LineVul with a drop from 62.93% to 55.25% and at most 61.88%, while CodeT5+ does not induce any big differences. The same can be seen in the top-5 accuracy. The graph-based models CGIN and ReVeal yield smaller differences in their performance and in general improve when a code style is applied. Their results closely follow the expected performance and the variations in performance can be explained by changes in the total number of lines and the matched number of flawed lines.

All models show better than random performance, even when applying the code formatting augmentation, implying they are not relying on styling artifacts for line-level localization. For the top-1 accuracy, the graph-based models and CodeT5+ are competitive and within reach of each other depending on the code style used with no clear best approach. The performance of LineVul on the other hand is worse than all the others, which is also seen for the top-5 accuracy and with mixed results for the top-3 accuracy. For the latter, ReVeal and CGIN show similar performance with ReVeal being better in all cases, while CodeT5+ is better than all other models. CodeT5+ is also better than the other models in all cases but one when considering its top-5 accuracy.

In summary, we see only a slight effect of artifacts on vulnerability localization for CodeT5+ and the graph-based models. LineVul is definitely affected by the removal of artifacts from the data and shows an overall weaker performance. Models that are less affected by the application of code formatting show an overall greater performance for vulnerability localization, indicating that generalizing code formatting changes are helpful to vulnerability localization. It is interesting that CodeT5+ does not see a performance drop in vulnerability localization, even though it was present in function-level prediction, indicating that the model is attending to the correct parts of the code but drawing wrong conclusions. Moreover, the performance difference between LineVul and CodeT5+ cannot be satisfactorily explained, as both models are trained similarly with the only difference being base architecture, indicating that the vulnerability localization performance of transformer models trained with artifacts is unpredictable. The graph-based models on the other hand are less affected in general and more predictable. Figure 5 visualizes the confounding effect calculated as the relative change of top-5 line-level accuracy normalized by subtracting the random baseline. All models suffer under the confounding effect, while ReVeal and CGIN suffer the least.

RQ3: Can the confounding effect be reduced? Considering Table 2 again, we can see that the adjustment of the training procedure for the transformer models significantly reduces the discrepancy between the test and augmented test performance. LineVul trained on normalized code achieves the best results on different styled code of around 68%, followed by CodeT5+ trained in normalized code with around 66%, beating the balanced accuracy scores from ReVeal, StackLSTM and CGIN by $\sim$ 3%. As opposed to the technique by Roziere et al. [38], pre-tokenization of the input code may decrease the artifact overfitting effect but its performance is still inferior to the other non-transformer-based models.

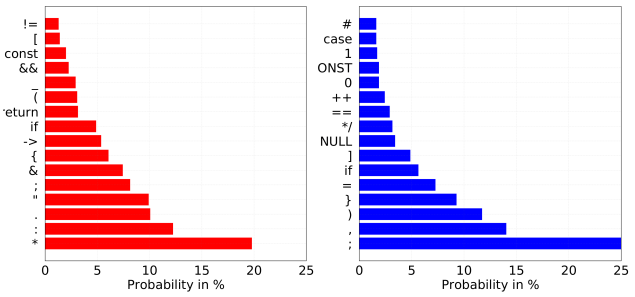
Tokenization and normalization for the LLMs also helped to reduce the bias measured by the *Uglify* transformation. However, ReVeal, StackLSTM, and CGIN outperform all transformer models. Surprisingly, reducing the confounder effect on the transformers helps to improve the performance on obfuscated code changes, slightly outperforming CGIN with up to 7%.

Moreover, tokenization and normalization also boost the causal accuracy of LineVul significantly. With a causal accuracy of 67.46%, an increase of nearly 15% from the unnormalized model, LineVul$_n$ performs best in our testing, beating even the graph-based models. CodeT5+$_n$ performs second-best with a causal accuracy of 65.41%, but CodeT5+$_t$ only achieves 59.80%. We attribute this only slight increase in performance of the tokenized model to the data distribution shift of tokenized data, which looks significantly different from the real-world data, that the model was pre-trained on, requiring further training. For LineVul$_t$ and CodeT5+$_t$ the causal accuracy is equal to the accuracy of the raw data, indicating that the artifacts initially present in the raw data are removed by tokenization.

It is also surprising that StackLSTM beats the transformer models and CGIN on uglified code. We conjecture that the StackLSTM learned to parse code despite syntactical differences. Considering Figure 6, we can see that the model learned to push opening brackets to the stack and pop closing ones from the stack if encountering them. Interestingly, StackLSTM learned to mimic a parser using the vulnerability discovery dataset.

We have shown that we can reduce the confounding influence of artifacts in the dataset on the detection models. GNNs are less influenced by code obfuscations than by control flow distortions as opposed to transformer models for which we observe the opposite effect. LSTMs are also less susceptible to style changes and CGIN is a viable approach to reduce the confounding influence. Transformer models and specifically LLMs are severely influenced by slight code transformations, however, we can mitigate this by rather normalizing the input code than tokenizing it.

In the end, LLMs prove their superiority to LSTMs and GNNs when correctly trained. Recall Figure 4, CGIN and CodeT5+ trained on pre-tokenized code reduced the confounding effect the most. While training on normalized code is also a viable strategy, in general, inferior to GNNs. The confounding effect on LineVul is



(a) Probability for pushing a token to the stack. **(b) Probability for popping a token from the stack.**

Figure 6: The learned stack policy for StackLSTM.

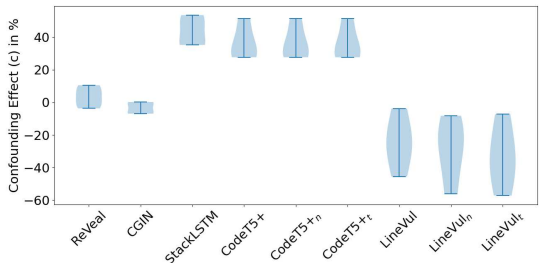


Figure 7: Confounding effect on Top-1 line-level accuracy.

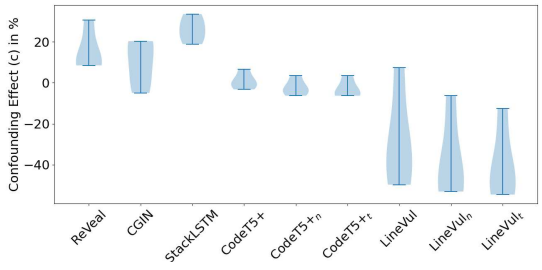


Figure 8: Confounding effect on Top-3 line-level accuracy.

largest on Top-3 line-level accuracy, considering Figure 8, while some models, especially CodeT5+, even benefit from the artifact removal as depicted by Figure 7 where they have a positive change of up to 50% detection improvement.

Interestingly, the shape and values of the confounding effect on the transformer models trained on the original data are nearly the same as those of the corresponding models trained on normalized and pre-tokenized data. As only the encoder input attention is used for generating line-level localization as per LineVul's method, this indicates that the encoder input attention is not affected by the presence of style artifacts in training while being greatly influenced by the presence of style artifacts at inference.

5 IMPACT

In Section 4, we have measured the confounding effect on both the function-level and line-level balanced accuracy for vulnerability discovery models. We have seen that their predictive performance and bug localization capabilities severely depend on artifacts present in the dataset. Furthermore, we demonstrate that we can not only measure but also reduce the confounding effect through our model or pre-processing choices. We provide here the most critical suggestions derived from our experimental study.

Tokenize the Code. The first insight is that using normalized code for fine-tuning the transformers yields the best function-level results on the augmented samples. However, there is still a measurable confounding effect of up to $c = 30\%$ for CodeT5+ and LineVul. That means that there are probably still artifacts remaining that normalizing code cannot account for. GNNs and StackLSTM provide better robustness in the first place but lose performance after

code obfuscation. CodeT5 + on tokenized code has the lowest scores but also the lowest confounding effect with around $c = 0\%$. Hence, given an LLM, pre-tokenizing the code may be the best option to receive unbiased performance results.

Compare against GNNs. ReVeal and CGIN have the best overall robustness against artifacts. CodeT5+, however, shows the strongest results in accurate line-level bug localization. Since a larger confounding effect hints that the model does not actually learn the underlying task, we argue that the graph-based models have better out-of-distribution performances and are thus better applicable to real-world cases. This is also in line with the results from Chen et al. [8]. By utilizing normalized or tokenized training data, the confounding effect of the transformer-based models is reduced such that their out-of-distribution performance is improved and competitive with GNNs.

Better Transferability. We argue that our evaluation provides a more faithful view of vulnerability discovery models. Obviously, if models tend to overfit to artifacts on one dataset, they lack generalization to another. Out-of-distribution transferability is an important property since this also mirrors the applicability to real-world cases. We tested LineVul on another dataset containing vulnerabilities from Chromium and Debian [7]. The new test set is disjoint from the original trainset. LineVul achieves 50%, tokenized LineVul 55%, and normalized LineVul 64% balanced accuracy on the unseen samples, underlining the usefulness of our evaluation and stratification approach.

6 RELATED WORK

In the following, we provide an outline of recent works that are tangent to this research area.

Vulnerability Discovery Models. There is a notable amount of research interest in developing novel vulnerability discovery models and comparing them to prior ones. With Vuddy [25] being a heuristic vulnerable clone detection model, Draper [39] and VulDeePecker [31] being one of the first token-based deep learning solutions, consecutive works started to benchmark their approaches against them. Slicing-based approaches [9, 30] and graph-learning-based approaches starting with Devign [65], have followed shortly after [7, 50, 65] reporting remarkable success even compared to traditional rule-based tools. The novel soft-attention mechanism from Vaswani et al. [48] has fostered research and approaches like VulSPG [64] and Cheng et al. [10] report even better results. Finally, LLMs like RoBERTa [34] or CodeT5 [53] have been applied to vulnerability discovery tasks [8, 15, 46] achieving currently the best performances on realistic datasets.

Explainable AI for Security. With the success of learning-based function-level vulnerability discovery models comes the problem with a lack of interpretability and defect localization [16]. Explainable AI helps to open up black boxes such as deep neural networks [4, 47]. This is even more critical to applications in a security context. Under the sheer number of explainability algorithms, finding the best suited for vulnerability discovery has been an active research question [54, 66]. Ganz et al. [16] investigate how a security practitioner can compare different localization of bugs, as a bug can be rarely pinned down to a single line [16, 64].

Fallacies in Vulnerability Discovery. Arp et al. [3] give rise to fallacies in developing ML models for security. For instance, they examine which metrics may induce a biased view of the performance, and how artifacts in the dataset can negatively impact the true performance. Chakraborty et al. [7] find that current datasets as the one from Zhou et al. [65] or Li et al. [31] are unrealistic and biased. They further show that most models lack transferability to out-of-distribution datasets by cross-evaluating popular models. Wang et al. [50] criticize datasets obtained through biased approaches like filtering commit messages by certain keywords. They propose to filter samples using a classifier identifying security-relevant patches. We leverage these insights in our experimental design as well as our metrics and dataset choice.

Code Transformations. In the experiments presented in this paper, we use specific types of transformations to investigate the confounding effects of artifacts on LLMs and graph-based models, specifically the application of predefined styles, uglification, and obfuscation. Applying transformations to code in order to investigate the limits of LLMs or graph-based models has received growing attention in the vulnerability discovery research community. Examples of such transformations that have been investigated are identifier renaming [20, 59, 60, 62, 63], insertion of unexecuted statements [20, 43, 60, 62] or replacement of code elements with equivalent elements [2, 28]. We continue this investigation by providing a novel implementation of transformations, by applying them to measure the confounding effects of artifacts, and by evaluating strategies to mitigate such effects.

7 CONCLUSION

In this work, we show that current vulnerability discovery models are severely influenced by artifacts such as code styles, variable naming, and common control flow patterns. The true performance of such models is hardly measurable and the reported ones from recent works can not be extrapolated to out-of-distribution code samples. We link the problem to spurious correlations in the dataset enabling models to shortcut decisions using unrelated information. We show that some models are less impacted by confounders and others more. Especially, large language models achieve remarkable results, but when provided with slightly modified code, their initial performance degrades. We propose three mitigations to drastically improve performance as a remedy to spurious correlation.

ACKNOWLEDGMENTS

The authors gratefully acknowledge funding from the German Federal Ministry of Education and Research (BMBF) under the grants IVAN (16KIS1165K) and BIFOLD (BIFOLD23B), from Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972, and from the European Research Council (ERC) under the consolidator grant MALFOY (101043410).

F. Broken Promises: Measuring Confounding Effects in Learning-based Vulnerability Discovery

REFERENCES

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to Represent Programs with Graphs. *CoRR* abs/1711.00740 (2017). arXiv:1711.00740 http://arxiv.org/abs/1711.00740

[2] Leonhard Applis, Annibale Panichella, and Arie van Deursen. 2021. Assessing Robustness of ML-Based Program Analysis Tools using Metamorphic Program Transformations. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1377–1381. https://doi.org/10.1109/ASE51524.2021.9678706

[3] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2020. Dos and Don'ts of Machine Learning in Computer Security. *CoRR* abs/2010.09470 (2020). arXiv:2010.09470 https://arxiv.org/abs/2010.09470

[4] Vaishak Belle and Ioannis Papantonis. 2021. Principles and Practice of Explainable Machine Learning. *Frontiers in Big Data* 4 (2021). https://doi.org/10.3389/fdata.2021.688969

[5] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. *CoRR* abs/1806.07336 (2018). arXiv:1806.07336 http://arxiv.org/abs/1806.07336

[6] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. 2021. BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection. *Inf. Softw. Technol.* 136 (2021), 106576.

[7] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[8] Yizheng Chen, Zhoujie Ding, Xinyun Chen, and David A. Wagner. 2023. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. *ArXiv* abs/2304.00409 (2023).

[9] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. *ACM Trans. Softw. Eng. Methodol.* 30, 3, Article 38 (apr 2021), 33 pages. https://doi.org/10.1145/3436877

[10] Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. 2022. Path-Sensitive Code Embedding via Contrastive Learning for Software Vulnerability Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 519–531. https://doi.org/10.1145/3533767.3534371

[11] Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A. Ortega. 2023. Neural Networks and the Chomsky Hierarchy. arXiv:2207.02098 [cs.LG]

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[13] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 508–512. https://doi.org/10.1145/3379597.3387501

[14] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[15] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE.

[16] Tom Ganz, Martin Härterich, Alexander Warnecke, and Konrad Rieck. 2021. Explaining Graph Neural Networks for Vulnerability Discovery. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security* (Virtual Event, Republic of Korea) *(AISec '21)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/3474369.3486866

[17] Tom Ganz, Philipp Rall, Martin Härterich, and Konrad Rieck. 2023. Hunting for Truth: Analyzing Explanation Methods in Learning-based Vulnerability Discovery. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. 524–541. https://doi.org/10.1109/EuroSP57164.2023.00038

[18] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward Large-Scale Vulnerability Discovery Using Machine Learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy* (New Orleans, Louisiana, USA) *(CODASPY '16)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2857705.2857720

[19] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. Exploring network structure, dynamics, and function using networkx. (1 2008).

[20] Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. 2022. Semantic Robustness of Models of Source Code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. https://doi.org/10.1109/saner53432.2022.00070

[21] Micha Horlboge, Erwin Quiring, Roland Meyer, and Konrad Rieck. 2022. I still know it's you! On Challenges in Anonymizing Source Code. arXiv:2208.12553 [cs.CR]

[22] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-Efficient Transfer Learning for NLP. arXiv:1902.00751 [cs.LG]

[23] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685 [cs.CL]

[24] Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah. 2022. Transformers in Vision: A Survey. *ACM Comput. Surv.* 54, 10s, Article 200 (sep 2022), 41 pages. https://doi.org/10.1145/3505244

[25] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. 595–614. https://doi.org/10.1109/SP.2017.62

[26] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *CoRR* abs/1609.02907 (2016). arXiv:1609.02907 http://arxiv.org/abs/1609.02907

[27] Guillaume Lample, Miguel Ballesteros, Kazuya Kawakami, Sandeep Subramanian, and Chris Dyer. 2016. Neural Architectures for Named Entity Recognition. In *Proc. NAACL-HLT*.

[28] Yaoxian Li, Shiyi Qi, Cuiyun Gao, Yun Peng, David Lo, Zenglin Xu, and Michael R. Lyu. 2022. A Closer Look into Transformer-Based Code Intelligence Through Code Transformation: Challenges and Opportunities. https://doi.org/10.48550/ARXIV.2207.04285

[29] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2017. Gated Graph Sequence Neural Networks. arXiv:1511.05493 [cs.LG]

[30] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, Zhaoxuan Chen, Sujuan Wang, and Jialai Wang. 2018. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *CoRR* abs/1807.06756 (2018). arXiv:1807.06756 http://arxiv.org/abs/1807.06756

[31] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *CoRR* abs/1801.01681 (2018). arXiv:1801.01681 http://arxiv.org/abs/1801.01681

[32] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. 2021. A Survey of Transformers. arXiv:2106.04554 [cs.LG]

[33] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2021. Multi-Task Learning Based Pre-Trained Language Model for Code Completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 473–485. https://doi.org/10.1145/3324884.3416591

[34] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692 [cs.CL]

[35] Scott Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. arXiv:1705.07874 [cs.AI]

[36] Judea Pearl. 2009. *Causality* (2 ed.). Cambridge University Press. https://doi.org/10.1017/CBO9780511803161

[37] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[38] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1730, 11 pages.

[39] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. *CoRR* abs/1807.04320 (2018). arXiv:1807.04320 http://arxiv.org/abs/1807.04320

[40] Benjamin Sanchez-Lengeling, Jennifer Wei, Brian Lee, Emily Reif, Peter Wang, Wesley Qian, Kevin McCloskey, Lucy Colwell, and Alexander Wiltschko. 2020. Evaluating Attribution for Graph Neural Networks. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 5898–5910. https://proceedings.neurips.cc/paper/2020/file/417fbbf2e9d5a28a855a11894b2e795a-Paper.pdf

[41] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*. 618–626.

[42] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. 2017. Learning Important Features Through Propagating Activation Differences. *CoRR* abs/1704.02685 (2017). arXiv:1704.02685 http://arxiv.org/abs/1704.02685

[43] Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. 2021. Generating Adversarial Computer Programs using Optimized Obfuscations. In *International Conference on Learning Representations*. https://openreview.net/forum?id=PH5PH9ZO_4

[44] Yongduo Sui, Xiang Wang, Jiancan Wu, Min Lin, Xiangnan He, and Tat-Seng Chua. 2022. Causal Attention for Interpretable and Generalizable Graph Classification. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 1696–1705. https://doi.org/10.1145/3534678.3539366

[45] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode Compose: Code Generation Using Transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1433–1443. https://doi.org/10.1145/3368089.3417058

[46] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-Based Language Models for Software Vulnerability Detection. In *Proceedings of the 38th Annual Computer Security Applications Conference* (Austin, TX, USA) *(ACSAC '22)*. Association for Computing Machinery, New York, NY, USA, 481–496. https://doi.org/10.1145/3564625.3567985

[47] Erico Tjoa and Cuntai Guan. 2019. A Survey on Explainable Artificial Intelligence (XAI): Towards Medical XAI. *CoRR* abs/1907.07374 (2019). arXiv:1907.07374 http://arxiv.org/abs/1907.07374

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. arXiv:1706.03762 [cs.CL]

[49] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. *International Conference on Learning Representations* (2018). https://openreview.net/forum?id=rJXMpikCZ

[50] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang. 2021. Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection. *IEEE Transactions on Information Forensics and Security* 16 (2021), 1943–1958. https://doi.org/10.1109/TIFS.2020.3044773

[51] Wenhua Wang, Yuqun Zhang, Zhengran Zeng, and Guandong Xu. 2020. TranSĤ: A Transformer-based Framework for Unifying Code Summarization and Code Search. arXiv:2003.03238 [cs.SE]

[52] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[53] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[54] Alexander Warnecke, Daniel Arp, Christian Wressnegger, and Konrad Rieck. 2020. Evaluating Explanation Methods for Deep Learning in Security. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, Genoa, Italy, 158–174. https://doi.org/10.1109/EuroSP48549.2020.00018

[55] Konrad Weiss and Christian Banse. 2022. A Language-Independent Analysis Platform for Source Code. arXiv:2203.08424 [cs.CR]

[56] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[57] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A Comprehensive Survey on Graph Neural Networks. *CoRR* abs/1901.00596 (2019). arXiv:1901.00596 http://arxiv.org/abs/1901.00596

[58] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604. https://doi.org/10.1109/SP.2014.44

[59] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural Attack for Pre-Trained Models of Code. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1482–1493. https://doi.org/10.1145/3510003.3510146

[60] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial Examples for Models of Code. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 162 (nov 2020), 30 pages. https://doi.org/10.1145/3428230

[61] Aiping Zhang, Liming Fang, Chunpeng Ge, Piji Li, and Zhe Liu. 2023. Efficient transformer with code token learner for code clone detection. *Journal of Systems and Software* 197 (2023), 111557. https://doi.org/10.1016/j.jss.2022.111557

[62] Huangzhao Zhang, Zhiyi Fu, Ge Li, Lei Ma, Zhehao Zhao, Hua'an Yang, Yizhe Sun, Yang Liu, and Zhi Jin. 2022. Towards Robustness of Deep Program Processing Models—Detection, Estimation, and Enhancement. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 50 (apr 2022), 40 pages. https://doi.org/10.1145/3511887

[63] Huangzhao Zhang, Zhuo Li, Ge Li, L. Ma, Yang Liu, and Zhi Jin. 2020. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. In *AAAI Conference on Artificial Intelligence*.

[64] Weining Zheng, Yuan Jiang, and Xiaohong Su. 2021. VulSPG: Vulnerability detection based on slice property graph representation learning. *CoRR* abs/2109.02527 (2021). arXiv:2109.02527 https://arxiv.org/abs/2109.02527

[65] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. *CoRR* abs/1909.03496 (2019). arXiv:1909.03496 http://arxiv.org/abs/1909.03496

[66] Deqing Zou, Yawei Zhu, Shouhuai Xu, Zhen Li, Hai Jin, and Hengkai Ye. 2021. Interpreting Deep Learning-Based Vulnerability Detector Predictions Based on Heuristic Searching. *ACM Trans. Softw. Eng. Methodol.* 30, 2, Article 23 (mar 2021), 31 pages. https://doi.org/10.1145/3429444

G

PAVUDI: Patch-based Vulnerability Discovery using Machine Learning

PAVUDI: Patch-based Vulnerability Discovery using Machine Learning

Tom Ganz
tom.ganz@sap.com
SAP SE
Germany

Erik Imgrund
erik.imgrund@sap.com
SAP SE
Germany

Martin Härterich
martin.haerterich@sap.com
SAP SE
Germany

Konrad Rieck
rieck@tu-berlin.de
Technische Universität Berlin
Germany

ABSTRACT

Machine learning has been increasingly adopted for automatic security vulnerability discovery in research and industry. The ability to automatically identify and prioritize bugs in patches is crucial to organizations seeking to defend against potential threats. Previous works, however only consider bug discovery on statement, function or file level. How one would apply them to patches in realistic scenarios remains unclear. This paper presents a novel deep learning-based approach leveraging an interprocedural patch graph representation and graph neural networks to analyze software patches for identifying and locating potential security vulnerabilities. We modify current state-of-the-art learning-based static analyzers to be applicable to patches and show that our patch-based vulnerability discovery method, a context and flow-sensitive learning-based model, has a more than 50% increased detection performance, is twice as robust against concept drift after model deployment and is particularly better suited for analyzing large patches. In comparison, other methods already lose their efficiency when a patch touches more than five methods.

CCS CONCEPTS

• **Security and privacy → Software and application security**;
• **Computing methodologies → Machine learning**.

KEYWORDS

Vulnerability Discovery, Program Representations, Patches

1 INTRODUCTION

A change to a program is typically accomplished through a patch, providing feature updates or fixes for bugs and vulnerabilities [47]. With the increasing adoption of continuous integration (CI) and continuous deployment (CD), the need to monitor and validate patches for potential bugs has become substantial [33]. To maintain a secure and reliable software development life-cycle, organizations must have a robust software quality assurance process in place to identify and mitigate security risks before a patch reaches production systems.

The traditional approach to finding bugs in software relies on manual code reviews and extensive testing. However, this approach is time-consuming, resource-intensive, and prone to human error. Static program analysis, on the other hand, supports developers to identify potentially flawed code regions without actually running the program. Unfortunately, such static application security testings (SASTs) tools often report many false positive alerts, which consequently requires expensive manual triage. The problem arises from too general detection rules and the theoretical limits of static analysis including bug and vulnerability detection [27]. In practice, developing detection rules for SAST tools is an error-prone and tedious task [30]. Hand-crafted rules are often incomplete or too sensitive, resulting in unfavorable trade-offs between high false-positive and false-negative rates. The detection deteriorates even more when these rules are project-agnostic and intended to apply to a large number of applications.

As a remedy, methods for learning-based vulnerability detection have been proposed to automatically derive rules from historical data [30, 35, 58]. Current machine learning (ML) models have been shown to beat rule-based SAST tools with a much higher detection rate while pertaining to a lower false-positive rate [14, 43, 57, 58]. However, the prevailing ML models focus exclusively on features in local code regions, such as functions [58], statements [17] or slices [29]. Moreover, these models are not context- or flow-sensitive, and thus suffer from low generalizability and transferability in realistic settings [11, 12, 32, 39].

With patches, the situation is even worse, as there is often little time to test and validate them before they are released and it remains unclear how a security expert would apply learning-based SAST tools to commits, as a patch can potentially span over multiple disjoint modules, functions, and classes. Since patches are the only atomic unit defining the evolution of software, it makes sense to adapt existing methods or develop novel techniques that infer bugs in patches rather than identifying bugs on more fine-granular levels. Yin et al. [56] state that up to 24% of patches introduce new bugs, moreover, with open-source software, most patch developers are not even familiar with the entire code base.

The main research question addressed in this work is how to effectively identify and locate bugs in patches. As current learning-based analyzers do not consider patches, we can demonstrate that their detection performance is poor. Thus, we present a novel patch-based vulnerability discovery (PAVUDI) approach. We combine graph neural networks with traditional taint analysis to identify and locate bugs in patches considering the entire application. We first formalize a new graph representation that allows security practitioners to analyze interprocedural data and control flow from

potentially attacker-controlled sources to software-critical regions. Such a single graph captures the impact of a given patch on the system's security posture. Secondly, we design a learning model specifically for this graph representation to automatically infer vulnerable or flawed paths within a patch.

To empirically validate the effectiveness of our approach, we conduct a comprehensive evaluation of the proposed method on a dataset of security patches. We show that PAVUDI outperforms state-of-the-art methods under real-world conditions. Furthermore, we find that current approaches suffer from concept drift, that is, they lose their initial detection capabilities over time. PAVUDI is less affected by this drift, providing a more stable detection performance over time.

In summary, we make the following contributions in this work:

(1) *Interprocedural code representation.* We introduce and formalize a new graph representation of code for finding security vulnerabilities. The graph is context- and flow-aware and provides security-relevant information especially for bug discovery in patches.
(2) *Explainable graph learning model.* We implement a novel model architecture that is well suited for our graph representation and provides explainable results using causal structure learning.
(3) *Extensive empirical evaluation.* We compare PAVUDI against several different state-of-the-art vulnerability discovery models and implement several detection strategies to apply these models to patches.
(4) *Real-world findings.* We test PAVUDI in realistic scenarios by detecting previously unknown bugs in open-source software. More specifically, we apply our model to two popular C libraries and find five bugs in their most recent 100 commits.

The rest of this paper is structured as follows: We begin with an introduction to vulnerabilities and graph representations in Section 2, then we detail our problem setting in Section 3 and present our methodology in Section 4. In Section 6, we discuss our empirical evaluation and end with related work and conclusions in Section 7 and Section 8, respectively.

2 VULNERABILITIES IN PATCHES

Before presenting our approach to detect vulnerabilities in patches, let us first introduce the basic concepts of static non-learning and learning-based vulnerability discovery methods.

2.1 Vulnerability Discovery

To begin, we define the task of discovering vulnerabilities in a program. We aim to derive a single score that indicates the likelihood of a program being vulnerable based on a particular representation of it. This is expressed in Definition 1, which defines a decision function that takes a piece of code and maps it to the probability of it being vulnerable. Note, that this definition does not differentiate between x being a function, statement or patch.

Definition 1. *A method for static vulnerability discovery is a decision function $f : x \mapsto P(\mathit{vulnerable} \mid x)$ that maps a piece of code x to its probability of being vulnerable [20].*

Classic rule-based SAST tools can be described directly as a function f predicting vulnerabilities, by for instance, matching function calls against known patterns or applying *taint style analysis* by tracking the flow of user-provided values, checking buffer bounds, detecting undefined behavior and more. Learning-based methods for vulnerability discovery, on the other hand, build on a function $f = f_\theta$ parameterized by model weights θ that are obtained by training on a dataset of vulnerable and non-vulnerable code [23]. 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. The primary differences among these approaches lie in the input program representation and the learning model, for instance, how the function depends on the model weights.

2.2 Vulnerabilities and Patches

There are several ways vulnerabilities can slip into program code during software development, ranging from a single patch to a series of complex and intertwined changes to a program. While there are approaches to trace a discovered bug back to the inducing changes, the other direction, namely identifying all commits sufficient for spotting an unknown vulnerability, is a hard problem in the general case. As a remedy, we focus in this work on vulnerabilities that are linked to one specific patch.

In particular, we consider a patch as vulnerability-inducing if its code changes either directly introduce the defect or are in close proximity to an existing one, so that vulnerable data passes through code changes as defined later in Definition 7. An example of such a patch is the heartbeat commit introducing a buffer-overread in CVE-2014-0160, as shown in Figure 1. Note that even though we restrict our scope to single vulnerability-inducing patches, their complexity can still be significant, covering dozens of disconnected regions across an entire code base.

2.3 Graph Representations

Since programs can be modeled as directed graphs [2, 6, 53], recent approaches make use of graph representations [14, 43, 58] for source code instead of flat token sequences [30, 35]. We refer to the resulting representation as a *code graph* and denote the underlying directed graphs as $G = G(V, E)$ with vertices V and edges $E \subseteq V \times V$. Moreover, the nodes and edges are attributed, that is, elements of V or E are assigned values in a feature space.

However, different code graphs capture different syntactic and semantic features. Recent works, for instance, rely only on syntactic features for neural code comprehension using the abstract syntax tree (AST) [2]. This is a tree representing the syntactic structure of source code.

Definition 2. *The abstract syntax tree (AST) of a function f is the result of parsing its source such that the leaf nodes in the resulting tree $G_A = G(V_A, E_A)$ are the literals and the edges E_A describe the composition of syntactic elements [1].*

The semantic attributes of a function can be captured in flow graphs for instance with the flow of control or the flow in information defined in Definition 3.

Definition 3. *The control flow graph (CFG) within a function f is $G_C = G(V_C, E_C)$ with the nodes $V_C \subset V_A$ being statements, and where the directed edges E_C describe the execution order of the statements $V_C \subset V_A$ [53]. The data flow graph (DFG) within a function f is $G_D = G(V_D, E_D)$ with the nodes $V_D \subset V_A$ being variable assignments and references, and where the directed edges E_D describe read or write access from or respectively to a variable [9].*

These graph representations allow us to reason about the order of the executed statements and the flow of information between variables. An analysis using these properties is considered flow-sensitive [32]. Finally, a call graph connects function call-sites with the function definitions as defined in Definition 4.

Definition 4. *The call graph (CG) within a program is defined as $G_{CG} = G(V_{CG}, E_{CG})$ where the nodes $V_{CG} \subset V_A$ being function call-sites and definitions, while the edges E_{CG} connect the caller with the respective function definition.*

An analysis using the call context of a program is considered context-sensitive [32]. Code graphs capture syntactic and semantic relationships between statements and expressions in programs. Based on these classical representations, combined graphs have been developed for vulnerability discovery. A popular one is the code property graph (CPG) by Yamaguchi et al. [53], which is a combination of the AST, CFG and program dependence graph. Other approaches use different combinations, for instance, combining the AST with the CFG and the DFG [9], called code composite graph (CCG) as defined in Definition 5.

Definition 5. *The CCG is a disconnected graph G_{CCG} for a program $P = \{f_1, f_2, .., f_n\}$ with $V = \bigcup_{i=1}^{n} V_A^i$ and $E = E_A \cup E_D \cup E_C$ combining the AST with the semantic information from the CFG and DFG.*

The components of a CCG are easily obtained during compilation, and capture syntactic features and information flow, which fits neatly into the definition of taint-style analysis, which we will revisit in Section 4 [54].

2.4 Graph Representation Learning

With the recent success of graph-based program representations, research has started to focus on graph convolutional networks (GCNs) [58]. These networks are a class of deep learning models realizing a function $f : G(V, E) \mapsto y \in \mathbb{R}^d$ that can be used for the classification of graph-structured data [36].

GCNs can be viewed as a generalization of convolutional neural networks (CNNs), just as an image can be viewed as a regular grid graph where each pixel denotes a node in the graph connected by edges to its neighboring pixels [51]. A graph convolutional network needs two mandatory input parameters, that is, an initial feature vector $X \in \mathbb{R}^{N \times F}$, with N being the number of nodes in the graph and F the number of features per node, and the topology commonly described by the adjacency matrix $A \in [0, 1]^{N \times N}$. The most popular GCN types belong to so-called message passing networks (MPNs) where the prediction function is computed by iteratively aggregating and updating information from neighboring nodes. One of the simplest MPNs is the one defined by Kipf and Welling [26]:

$$h^{(l)} = \sigma(\hat{A}\, h^{(l-1)} W^{(l-1)}) \tag{1}$$

with $h^0 = X$. Here, the intermediate representations are linearly projected and sum-wise aggregated according to the normalized adjacency matrix $\hat{A}$ with self-loops followed by a non-linear activation function. These GCNs can be stacked to learn filters with respect to larger neighborhoods. Other GCN layers use different aggregation and update mechanisms, for instance, instead of an multilayer perceptron (MLP), gated graph neural networks (GGNNs) use gated recurrent unit (GRU) cells to update the hidden state of nodes [28], while graph attention network (GAT) layers use attention mechanisms [42]. We refer the reader to the overview article by Wu et al. [51] for further details.

Because of the fitting premise of GCNs, they have been widely adopted for representation learning on code graphs. The graph-based approaches in recent literature outperform classical SAST tools and older sequential learning models such as VulDeepecker [30] or Draper [35]. Graph learning-based approaches like Devign [58] and ReVeal [11] can be considered state-of-the-art and provide strong results in their respective publications.

3 PROBLEM SETTING

Current research on learning-based static code analysis focuses on local code regions, for instance, functions [11, 58], slices [29], or small code gadgets [14]. However, software development revolves around changes that can span multiple files and functions. Concurrent versioning systems like Git allow software developers to track changes that may contain bug fixes or feature enhancements commonly denoted as *patch*:

Definition 6. *A patch (commit) $[P' \Rrightarrow P]$ is a transition from one program P' to another P. It consists of changed code lines and files commonly denoted as* hunk. *A patch is often associated with a Git commit and its unique identifier [47].*

Besides introducing new features or fixing bugs, a patch also potentially adds new bugs [56] which we want to detect. The decision function from Definition 1 does not specify how to apply existing methods to patches. A naive approach would be to glue together all snippets changed by a patch before applying a decision function that operates on function or statement level.

However, problems arise due to the difficulty in identifying and locating all possible changes within a commit that potentially introduces bugs. Consider the heartbleed bug (CVE-2014-0160) in the OpenSSL C library. The bug was introduced due to a feature change adding *TLS heartbeats* three years prior to the discovery of the actual security vulnerability. The commit[1] touches twelve different C files and five header files in two different packages. A static analyzer would need to check all changed functions in all changed files to find the defect causing the heartbleed vulnerability shown in Figure 1.

We can see that `memcpy` copies a buffer with the size obtained by the client. If the client proposes a buffer length larger than the target buffer, it effectively triggers a heap-based buffer over-read. Finding this bug among all the changed files seems like finding the needle in the haystack. A static analyzer would need to identify `memcpy`

[1] https://github.com/openssl/openssl/commit/481750

```
if (hbtype == TLS1_HB_REQUEST)
    {
    unsigned char *buffer, *bp;
    int r;

    /* Allocate memory for the response, size is 1 byte
     * message type, plus 2 bytes payload length, plus
     * payload, plus padding
     */
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
    bp = buffer;

    /* Enter response type, length and copy payload */
    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
```

Figure 1: Buffer over-read in dtls1_process_heartbeat().

as a critical code statement, consider `payload` user-controlled and detect that there is no sanitization in-between.

We conclude that locating bugs in patches naturally comes with several problems that need to be addressed:

(1) *Context-sensitive changes.* Patches may only touch certain functions and modules but need to be analyzed in the surrounding context. A bug typically spans over multiple modules [57], that may be associated with a patch but do not have to be directly affected by it.
(2) *Non-coherent changes.* A patch may not correspond to a single feature change but potentially touch multiple modules that do not necessarily have to be associated with each other. Applying a SAST on all changed components may significantly increase the reported false positive alerts.
(3) *Evolution of software.* The learning-based discovery of vulnerabilities has already been addressed in research [11, 47, 58], however, finding a patch that introduces a bug is non-trivial, as a program may undergo several changes before a bug actually manifests. In addition, the feature representation of the program may change over time, causing the performance of a learning-based analyzer to degrade over time as well.

4 METHODOLOGY

PAVUDI is inspired by classic taint analysis, a dynamic program analysis approach where particular statements or expressions are tainted and monitored at run-time [37]. This analysis allows security practitioners to identify, for instance, potential attacker-controlled sources flowing into sensitive program regions. Yamaguchi et al. [53] define an over-approximate static approach by tainting program parts and propagating tainted values statically along the control and data flow. We extend their original formal definition to allow us to statically find vulnerabilities in patches per Definition 7.

Definition 7. *a) A taint-style analysis for vulnerable patch detection is a 4-tuple* $(V_{SOURCE}, V_{SINK}, V_{SAN}, V_{EDIT})$ *consisting of the nodes in the CCG of a program P denoting the taint source, sink and sanitizer nodes as depicted from* V_A *[53] as well as the nodes corresponding to code that is changed or newly created in a patch* $[P' \Rightarrow P]$.
b) We say the patch $[P' \Rightarrow P]$ *contains a bug if there exists a vulnerable data or control flow between any* $v_0 \in V_{SOURCE}$ *and* $v_1 \in V_{SINK}$ *with the constraint of not reaching any defined sanitizer but intersecting with at least one node from* V_{EDIT}.

4.1 Overview

To overcome the issues described in Section 3, our method is comprised of a new graph representation, called taint graphs which considers the context of a patch, a value-set analysis and an explainable graph neural network (GNN) that learns to infer detection rules on this particular representation in a taint-style fashion as described in Definition 7.

Graph representation. We define a new interprocedural patch graph representation. Beba and Karlsen [5] have shown that taint information significantly reduces false positives for rule-based static analyzers, hence, we similarly argue that this graph representation yields more valuable context to learning-based analyzers. In contrast to current discovery models, interprocedural graph representation is arguably more beneficial, since it enables us to propagate taint information within the entire program, which is impossible with a function, local slice or file-level graph.

Value-set analysis. As another improvement over recent discovery models, we calculate a value-set analysis to track variable domains in the graphs. This assists in reasoning about potential bounds and sanitizations. More specifically, whether or not the value of a user-controlled variable or buffer length is bounded beneficially affects the model's decision.

Causal GNN model. Finally, we use graph isomorphism network (GIN) layers to train an inductive model to infer detection rules applied to taint graphs. Our model is especially suited for processing long input graphs by its skip connections and attention mechanism. The attention weights from the latter can be interpreted as relevance scores per node to achieve a fine-granular localization of bugs.

4.2 Representation

To obtain a graph representation that is appropriate for vulnerability discovery in patches, we slice from *taint graphs* that are an extension to CCGs. We can calculate them in four steps:

(1) Insert call edges
(2) Insert interprocedural data flow
(3) Perform a value-set analysis
(4) Create security-relevant slices

(1) Insert call edges. A CCG is an intraprocedural disconnected graph G_{CCG} for a program P. Each function within P has its own CCG. We can connect each of them by adding call graph edges. More concretely, we connect call-sites with function definitions per Definition 4. We eventually end up with a single connected graph for P with $V = \bigcup_{i}^{n} V_A^i$ and $E = E_A \cup E_D \cup E_C \cup E_{CG}$. Although the CCG now encompasses an over-approximate global semantic of the program P with a single connected graph, it is still hard to track interprocedural data and control flow.

(2) Insert interprocedural data flow. The CG provides some intuition about the relation between functions during execution. Yet, to keep track of user-controlled variables, it is necessary to

provide a more fine-grained view of the interprocedural data flow. Consider a function call $v_1 \rightarrow v_2$ with $(v_1, v_2) \in E_{CG}$. Since v_1 is a call statement, we can associate its accompanying argument nodes, while the same applies to the function parameters represented in the function signature of the callee v_2. We sort the argument nodes by their appearance in the function definition and connect them pairwise yielding E_{IDFG}. During the static analysis, it is hard to infer whether a variable passed by reference may be written to or only read from, thus, we model interprocedural data flow graph (IDFG) edges between pointers as a bidirectional relationship. This step leaves us with an IDFG graph formally defined in Definition 8.

Definition 8. *The interprocedural data-flow graph (IDFG) is defined as G_{IDFG} where $V_{IDFG} \subseteq \bigcup_{i=1}^{n} V_D^i$ with E_{IDFG} connecting parameters in the function call to their respective arguments in the function definition.*

(3) Perform a value-set analysis. Given G_D we can select any variable assignment $v_e \in V_D$ and find $(v_s, v_e) \in E_D$ where v_e reads from v_s. If v_s is a constant and v_e is a Boolean, Float or Integer operation, we can evaluate v_e. If v_s is not a constant, we can find $(v, v_s) \in E_D$ and repeat. This eventually boils down to constant propagation and folding. If we are able to evaluate v_e, we attach the evaluated value to the node. Otherwise, if the operation can not be evaluated because, for instance, one data flow dependent v_d of v_e relies on I/O input or external API calls, we annotate v_e with v_d.

Lastly as described by Wegman and Zadeck [50], we find all expressions within surrounding conditional blocks that may act as invariants. If within this conditional block, we run into a variable that appears in a conditional of the form `<var> <comparison> <expression>`, we annotate its bounds with its value if it could be evaluated in the previous step. As an example, in Figure 2, we can assert that `len` has a lower bound of 10 in the entire conditional block after line 4.

We can formalize this by attaching a lower-bound domain to every reference node $v \in V_D$ defined as a four-tuple semi lattice $(\mathbb{R}, \leq, \bot, \sqcap_l)$ and an upper-bound domain $(\mathbb{R}, \geq, \top, \sqcap_u)$. $\top$ and $\bot$ denote UNBOUNDED, that is, the variable has no upper or lower bound respectively, $\sqcap_u$ is the least upper bound defined as $\sqcap_u : (k_1, k_2) \rightarrow \min(k_1, k_2)$ and $\sqcap_l$, consequently, is the greatest lower bound defined as $\sqcap_l : (k_1, k_2) \rightarrow \max(k_1, k_2)$. Both are used as transfer functions at the control flow join points [3].

(4) Create security-relevant slices. At this point, we have a program P represented by an interprocedural CCG. As a first step towards the definition of taint graphs, we define taint paths. For this, we select taint sources $V_{\text{SOURCE}} \subset V$ providing user input and taint sinks $V_{\text{SINK}} \subset V$ denoting critical code regions and the subset $V_{\text{EDIT}} \subset V$ of all nodes that have been edited in a specific patch.

Definition 9. *A single* taint path *p of a patch $[P' \Rrightarrow P]$ is an oriented path with vertices $v_0, \ldots, v_b, \ldots v_e$ starting at $v_0 \in V_{SOURCE}$, passing through $v_b \in V_{EDIT}$ and ending in $v_e \in V_{SINK}$ where all the edges are in E_{IDFG}, E_{DFG} or E_{CFG}.*

To obtain taint paths we perform forward slicing from V_{EDIT} to V_{SINK} following any IDFG, DFG or CFG while neglecting the AST and CG edges. This leaves us with a set of paths that describe the changed spots within a patch potentially flowing into critical sinks. Likewise, we perform backward slices from V_{EDIT} to V_{SOURCE}. Combining both sets of slices leaves us with a set of paths describing all flows starting with user-defined inputs intersecting the patched locations and reaching the critical sinks. To provide a holistic view of a patch, we arrive at the taint graph, as defined in Definition 10, by combining all taint paths and gluing them together at their patch intersections V_{EDIT}. It is trivial to see, that this graph representation neatly fits into the definition of taint-style vulnerable patch detection from Definition 7.

Definition 10. *A taint graph (TG) of a patch $[P' \Rrightarrow P]$ is defined as G_{TG} joining its taint paths $\{p^1, p^2, \ldots, p^k\}$ at their common AST nodes, starting from V_{SOURCE} flowing through V_{EDIT} and reaching V_{SINK}.*

Originally, the term "taint graph" has been frequently used in malware analysis, where information collected by malicious applications is tracked to analyze how it flows through processes and files [55]. In our case, we are interested in how user-provided data flows through a patch and whether they may reach critical program sections. The definition of V_{SOURCE} and V_{SINK} is specific to the intended use and can be set appropriately. Furthermore, the number of paths in G_{TG} might become exponentially large, hence we suggest sub-sampling k paths at random.

4.3 Discovery

Let us consider the vulnerability in Figure 2 for CVE-2015-7497 which was introduced 12 years before it was publicly disclosed [2]. `plen` is a user-controlled variable that could trigger a buffer underflow in `name` when provided with an integer larger than `len`.

```
1 ...
2 value = *name;
3 value <<= 5;
4 if (len > 10) {
5     value += name[len - (plen + 1 + 1)];
6 ...
```

Figure 2: Buffer underflow in Libxml2.

The corresponding taint graph with $k = 4$ is depicted in Figure 3. For visualization purposes, we have omitted irrelevant node and edge labels and shortened the remaining node labels. We can see that four possible input sources flow into the array access to `name`. The common joint AST node, reached by each input source node happens to be a function call to `XMLDictLookUp()`. The critical node in this example is an array index calculation highlighted by PAVUDI.

Representation Learning. For a taint graph to be applicable for GNNs, we represent the textual code that is attached to every AST node using Word2Vec on each token and eventually take the average similar to recent works [11, 58]. We then train a causal graph isomorphism network (CGIN) [40] to infer bugs in patches using our novel graph representation. The GIN model performs state-of-the-art while relying on a simple update and aggregation mechanism similar to the GCN in Equation (1):

$$h^{(l)} = W^{(l-1)} \left((A + (1 + \epsilon) \times I) \times \text{RELU}(h^{(l-1)}) \right) \tag{2}$$

[2] https://github.com/GNOME/libxml2/commit/2fdbd3

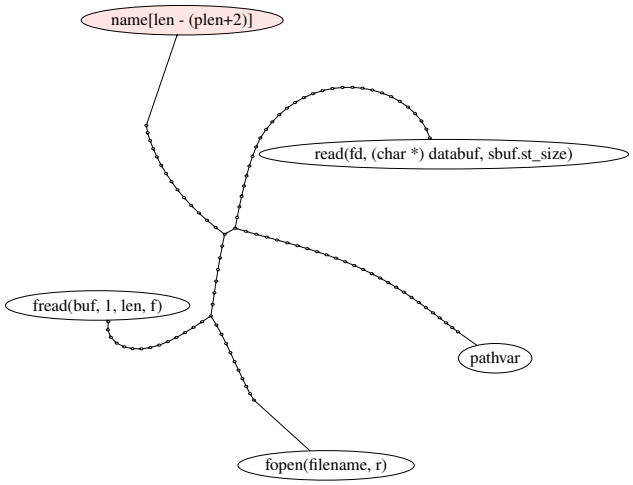


Figure 3: The G_{TG} with $k = 4$ for a Libxml2 buffer-underflow. Red denotes the relevant node according to PAVUDI.

Since, compared to classic code graphs, taint graphs are potentially longer and have a smaller average degree, we implement several design choices that help to pertain important information over large distances. We set ϵ in Equation (2) as a trainable parameter, which is particularly useful in conjunction with GINs to reduce smoothing out information from distant nodes. Furthermore, we use skip connections between the layers to help relevant information propagate across the topology.

After three graph encoding layers, we use two MLPs to calculate a relevance score for nodes and edges. For any node $v_i \in V$ we calculate their node attention as depicted in Equation (3) and for any pair of nodes $(v_i, v_j) \in E$ their edge attention as depicted in Equation (4) respectively. Furthermore, we halve the output space of the MLPs to perform a latent space disentanglement, where the first half will later be optimized to contain only the nodes causal for our task and the second half will be trained to only contain the trivial part of the graph which can be considered noise.

$$a_i^c, a_i^t = \sigma(\text{MLP}_{\text{NODE}}(h_i)) \tag{3}$$

$$b_{ij}^c, b_{ij}^t = \sigma(\text{MLP}_{\text{EDGE}}(h_i || h_j)) \tag{4}$$

A mean readout layer is applied last as a pooling strategy followed by a final MLP with softmax activation as the prediction head returning either VULNERABLE or CLEAN. Using the attention scores we can calculate attention masks $M_x, \hat{M}_x, M_a, \hat{M}_a$ respectively for the causal and trivial features, and causal and trivial edges. We apply these masks to the adjacency matrix and feature matrix of the taint graph yielding $\mathcal{G}^c$ and $\mathcal{G}^t$ for the causal and trivial subgraphs respectively. The causal taint graph can be used to explain the prediction and find the cause of a vulnerability.

To train the model in a supervised fashion we first apply a traditional NLL-loss $\mathcal{L}_{sup}$ to our ground truth and the latent representation of the causal graph h_{G_c} as depicted in Equation (5).

$$\mathcal{L}_{\text{sup}} = -\frac{1}{|\mathcal{D}|} \sum_{\mathcal{G} \in \mathcal{D}} \mathbf{y}_{\mathcal{G}}^{\top} \log\left(\sigma(\mathbf{h}_{\mathcal{G}^c})\right) \tag{5}$$

Then we take the representation of the trivial subgraph $h_{\mathcal{G}^t}$ and optimize the model to separate trivial and causal features by fitting the classifier with the trivial graph to be close to a uniform distribution using the Kullback-Leibler divergence (KL):

$$\mathcal{L}_{\text{unif}} = \frac{1}{|\mathcal{D}|} \sum_{\mathcal{G} \in \mathcal{D}} \text{KL}\left(\mathbf{y}_{\text{unif}}, \sigma(\mathbf{h}_{\mathcal{G}^t})\right) \tag{6}$$

Ganz et al. [19] observe that common explanation methods put high relevance scores to features that stem from artifacts in the dataset. To reduce such bias taking effect in our future model interpretation, Sui et al. [40] suggest applying a backdoor adjustment to take care of any confounding variable. This can be achieved by conditioning the causal graph per sample with all possible trivial graphs $\mathcal{G}^t$ with $t \in \mathcal{T}$ obtained during training. This stabilizes training and helps reduce the influence of noise and spurious correlated features in the taint graph as defined in Equation (7).

$$\mathcal{L}_{\text{caus}} = -\frac{1}{|\mathcal{D}| \cdot |\mathcal{T}|} \sum_{\mathcal{G} \in \mathcal{D}} \sum_{t \in \mathcal{T}} \mathbf{y}_{\mathcal{G}}^{\top} \log\left(\sigma\left(\mathbf{h}_{\mathcal{G}^c} + \mathbf{h}_{\mathcal{G}^t}\right)\right) \tag{7}$$

After optimizing the model by minimizing $\mathcal{L}_{\text{sup}} + \mathcal{L}_{\text{unif}} + \mathcal{L}_{\text{caus}}$ we obtain a GNN that is able to process potentially long taint graphs. We can use the causal part of the graph G^c that was relevant to the model to classify the patch VULNERABLE and to localize the bug. Even more concretely using a_i^c from Equation (3) we can rank the most important causal nodes relevant for this classification, as the attention score can be directly interpreted as relevance score [36] e.g. by calculating the top-1 important node:

$$\max_{v_i \in V} a_i^c$$

4.3.1 Data Labeling. In order for a ML model to learn whether a patch potentially introduces new bugs, we would need to have a dataset with commits that originally add bugs to the code base, however, such a dataset is non-existent and not trivial to create, since, for example, a bug may need several commits until it manifests itself. Instead, current vulnerability datasets only contain commits that are known to fix bugs. Per commit, we can check out a software project and sample taint graphs with a pre-defined maximum length. We support the decision with a study from Calder et al. [8] stating that most C and C++ open-source projects have a maximum call-stack depth of 15. For each patch commit we mark the changed files and lines and move back in time to find commits prior to the patch touching the same lines in the same file. Particularly, for any commit including the patch, we extract its TG and label it VULNERABLE if it touches the same file and lines as the patch and label it CLEAN otherwise. This leaves us with three different types of TGs: Randomly selected clean graphs, vulnerable graphs that touch pre-fixed code locations, and clean graphs patching a bug. We will publish our extraction tool for subsequent research [3].

In general, we can address the issues from Section 3, since the program-aware context-sensitive taint graphs can be regarded as a solution to the problem of *(1) context-sensitive changes*, since we are using only tainted paths we can consider the problem *(2) non-coherent changes* to be circumvented to some extent. Lastly, extracting the commits from different time steps and only if they

[3] https://github.com/SAP-samples/security-research-taintgraphs/tree/main/PAVUDI

are associated with a known bug addresses the issue *(3) evolution of software*. However, using this approach, we can't decide whether a patch introduces a bug, but whether patched code contains a bug. As long as we do not have a dataset with patches introducing bugs, but only patches that fix bugs, we argue, that this is a good compromise.

5 EVALUATION

In the following section, we first lay out our experimental setup and then provide answers to the following questions:

- RQ1 How do the individual components of PAVUDI contribute to the detection capability?
- RQ2 How do other strategies compare to PAVUDI?
- RQ3 How does the size of a commit affect the performance?
- RQ4 How does PAVUDI behave after training and deployment?

5.1 Experimental Setup

In this section, we present our experimental setup for the experimental evaluation.

5.1.1 Datasets. We use the state-of-the-art datasets from Zhou et al. [58] for comparison with related approaches. The datasets are derived from the open-source projects *FFmpeg*, a video decoding and encoding command line tool, and *QEMU*, a generic emulation software. Zhou et al. [58] curate a list of security-related keywords[4] associated with security patches that are used to crawl the Github repositories of both projects and find security-relevant commits. This has the advantage over other datasets [e.g. 17] that they have a large number of samples per project. They then proceed to extract code samples from the bug-fixing commits. We, on the other hand, use these fixing commits to extract clean taint graphs (at the time of the commit) and vulnerable taint graphs (prior to the commit) as outlined in Section 4.3.1. The FFmpeg project contains exactly 2558 vulnerable and 3037 clean or fixed taint graphs, while QEMU is slightly smaller with 1006 clean and 928 vulnerable taint graphs. We also assess PAVUDI's performance on five smaller projects, namely Libxml2, an XML parser, and Lrzip, a compression tool, since they have already been targeted by program analysis research [e.g. 7, 16, 20], cURL, a command line tool for HTTP requests, and OpenSSL, a cryptographic library, since these pose security-critical applications which have been exploited in the past and finally TinyProxy, as a relatively new untested security-relevant application.

5.1.2 Taint Graphs. Per commit, we sample at most $k = 1000$ taint paths with a maximum length of $l = 200$. For each commit, we can incrementally update the interprocedural CCG in a graph database using only the changes per patch. We choose V_{source} and V_{sink} similar to Pewny and Holz [34], by annotating *libc* functions that are related to bugs or provide user input. We refer the reader to the appendix for the complete list of tainted statements. Furthermore, we attach Boolean upper- and lower-bound information to each variable reference and assignment to provide the model with information on whether the value of the expression is bounded.

[4] https://sites.google.com/view/devign

5.1.3 Baseline Models. There are several state-of-the-art methods for learning-based static vulnerability discovery. We compare against three intraprocedural graph-learning-based models since they also rely on graph representations. These models classify bugs only on the function level.

Devign uses as its initial input graph the CPG enhanced with the natural sequence graph connecting leaf nodes in their order of evaluation [58]. The model consists of a six-step GGNN with a one-dimensional CNN as pooling.

ReVeal is similar to Devign and uses an eight-step GGNN to embed the graph structure in latent space [11]. However, they use a simple max pooling. Their original input graph is the CPG and they use a triplet loss for training.

BGNN4VD uses the bidirectional CCG instead of the CPG [9]. They use an eight-step GGNN just as ReVeal followed by a 1D convolutional pooling and a final linear layer.

Due to the fact that all three only consider local functions, their static analysis approach is not context-sensitive, however flow-sensitive since they use data and control flow edges [39]. Thus, as another set of baselines, we select two models that are able to process bugs in a limited interprocedural context.

DeepWukong is a graph-based learning model [14]. Compared to the other GNN-based approaches it uses pre-processed slices around potentially critical code locations, for instance, array indexing arithmetics, pointer usages or library calls.

SysEVR is similar to DeepWukong, as it first finds locations potentially containing bugs, for instance, pointer usages, arithmetic expressions, function calls, and array indexing [29]. However, instead of using a graph representation, they stick to the token representation of the code extracted from the slicing operation on the CPG to feed it into a bidirectional gated recurrent unit (BGRU).

DeepWukong and SySEVR extract syntactic vulnerability candidates in the source code used for positioning the slices. Their slicing operation includes function calls within the function under analysis. Both approaches are context-sensitive and flow-sensitive since they either use flow graphs or a flow-sensitive slicing approach. Lastly, we compare PAVUDI's performance against two more popular approaches.

VUDDY [25] is a non-learning-based static analyzer that detects bugs by comparing their function signatures against known CVEs and NVDs.

VulDeePecker similarly to SySEVR uses a token-based representation of intraprocedural forward and backward slices over the CPG [30]. It uses a BiLSTM for classification.

VUDDY is a non-learning-based analyzer that is not optimized for the dataset and thus a suitable baseline representative for other non-learning based SAST tools. VulDeePecker, a context-unaware analyzer, is similar to SySEVR but does not incorporate any a priori information about the slicing locations. Furthermore, VUDDY is neither context nor flow-sensitive.

5.1.4 Strategies. All baselines process code at different granularity, but to the best of our knowledge, none of them has been previously applied to patches. While some methods classify slices and others'

entire functions, it remains unclear how one would apply them to patches. Thus, we aggregate their decisions to a single score per patch using five different aggregation strategies as if we would integrate them into a software quality assurance process:

Max is a strategy where simply the maximum value of all prediction scores from slices or functions is taken.

Mean strategy averages over every prediction score from slices or functions.

Probability is a strategy where the probability of the patch being vulnerable depending on its k components, is similar to calculating a system's failure probability.

$$P = 1 - \prod_{i=0}^{k} (1 - f(p_i)) \tag{8}$$

As denoted in Equation (8), for each vulnerable component, we multiply the probability of the complement event yielding the probability that no component is vulnerable. Then we take the complementary event again and obtain the probability that at least one component is flawed.

Isotonic Probability is similar to the probability strategy. Niculescu-Mizil and Caruana [31] state that prediction scores from ML models can not be mapped to probabilities out-of-the-box. We use an isotonic regressor to calibrate the predictions before calculating the probability as in Equation (8).

Commit merges all changed code components within a patch together and feeds them into the model if applicable, instead of returning predictions per function.

The different strategies have different effects on the FPs and TPs. If the vulnerable score for the only function in the patch gets smoothed out with the Mean-Strategy, we have fewer TPs and FPs. On the other hand, the Max-Strategy may be too sensitive to functions being slightly above the threshold resulting in higher FPs and TPs resulting in a lower precision but higher recall. Note that VUDDY only returns *CLEAN* or *VULNERABLE* without any confidence score. Hence, we can only apply strategy Commit and Max. The slice-based approaches can not have merged components as input hence we can not apply the Commit-Strategy to them.

5.1.5 Performance Metrics. To assess the performance of the different models we use several performance measurements that are recommended for comparing ML models in security [4]. For the comparison against other models and in the ablation study we use the F1-Score, that is, the harmonic mean between precision and recall. Furthermore, we use the area under receiver operating characteristic curve (AUROC), particularly as a second measurement for the evaluation against the baselines. For the concept drift experiment, we choose the balanced accuracy, calculated as the arithmetic mean between the sensitivity and specificity. We repeat every experiment ten times and report the best score.

5.1.6 Implementation Details. We implement PAVUDI on top of Pytorch Geometric and Memgraph[5] for storing and transforming the taint graphs. Furthermore, we use an AWS EC2 g4dn instance for extracting the taint graphs and for training. We use PyDriller and the GitHub API for extracting patch information, such as for instance, the commit date, the number of changed functions and methods. We train every model including PAVUDI on the same dataset with an identical 70/30 random split, except for the non-learning-based static analyzer VUDDY. We use the hyperparameters from the respective original publications or reference implementations for the baselines. For PAVUDI we use an ADAM optimizer with a learning rate of 0.0001 and for the Word2Vec node embedding we use a vector of size 100 and a context window of size 3 [11].

[5] https://memgraph.com/

Table 1: Ablation study: F1-Scores measured for different settings.

Dataset	GIN	GGNN	GCN	CGIN
FFmpeg+CutOff	0.42 ± 0.14	0.48 ± 0.08	0.50 ± 0.10	0.56 ± 0.05
QEMU+CutOff	0.51 ± 0.11	0.47 ± 0.08	0.50 ± 0.03	0.61 ± 0.22
FFmpeg+TG	0.89 ± 0.02	0.89 ± 0.01	0.85 ± 0.02	0.90 ± 0.02
QEMU+TG	0.84 ± 0.01	0.79 ± 0.03	0.79 ± 0.02	0.84 ± 0.01
FFmpeg+TG+Bounds	0.90 ± 0.03	0.89 ± 0.01	0.89 ± 0.02	**0.91 ± 0.02**
QEMU+TG+Bounds	0.84 ± 0.03	0.80 ± 0.03	0.80 ± 0.02	**0.85 ± 0.21**

5.2 Evaluation

We proceed to present our experimental results to provide answers to our research questions.

How do the individual components of PAVUDI contribute to the detection capability? We present our ablation study for our CGIN model on taint graphs on the FFmpeg and QEMU datasets. We try three other popular GNN architectures, namely GIN [52], GGNN [28] and GCN [26]. Also, we evaluate the models without using the bounds information from the value-set analysis. Finally, we slice off a subgraph from the taint graphs, such that each graph only captures the immediate data flow around the patch neglecting taint information. In Table 1, we see that the cut-off taint graphs yield worse performance, hence we argue that providing interprocedural and taint information is crucial for classifying vulnerable patches. GCN is the worst architecture, while GGNN is very close to GIN. However, CGIN provides the best F1-Scores.

Table 2: F1-Score for dataset cross-evaluation.

Testset	Trainset		
	FFmpeg	QEMU	FFmpeg+QEMU
FFmpeg	**91.1** ± 1.7%	34.0 ± 1.4%	N/A
QEMU	41.3 ± 1.1%	**82.9** ± 2.7%	N/A
Libxml2	39.6 ± 1.8%	**58.0** ± 1.2%	57.0 ± 1.6%
cURL	**48.5** ± 0.3%	22.2 ± 2.2%	14.0 ± 1.2%
OpenSSL	60.9 ± 1.43%	**83.4** ± 1.6%	54.0 ± 2.4%

Using the backdoor adjustment from Sui et al. [40] our results align with their observation, that we can even reduce the out-of-distribution (OOD) problem. This can be seen in Table 2 where PAVUDI achieves noticeable detection performance measured by the F1-Scores on completely different open-source software projects that it was not trained on. It achieves an F1-Score of 58% for Libxml2 and even 83.4% for OpenSSL, although, there is an overlap between QEMU's and OpenSSL cryptographic feature implementations.

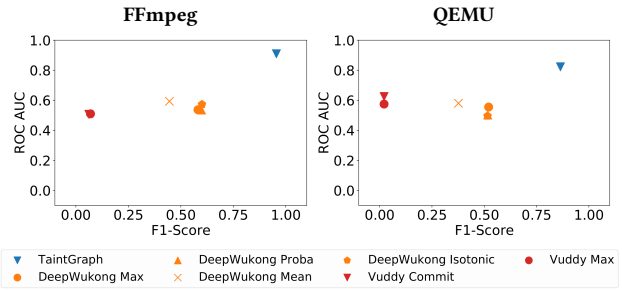


Figure 4: Performance comparison against DeepWukong and Vuddy.

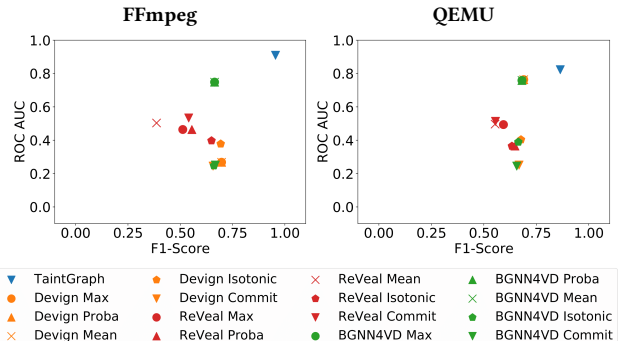


Figure 5: Performance comparison against Devign, ReVeal and BGNN4VD.

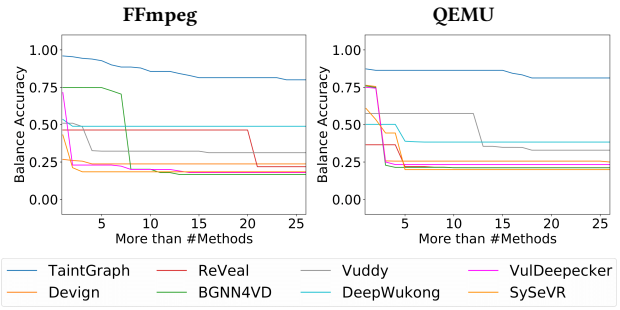


Figure 6: Performance decrease with increasing number of changed methods.

Context information contained in taint graphs and GIN layers significantly contributes to the detection performance of PAVUDI.

How do other strategies compare to PAVUDI? We compare our approach against the seven baselines with the five aggregation strategies. In Figure 4 we can see that PAVUDI has a much higher AUROC and F1-Score compared to Vuddy and DeepWukong. Since VUDDY is a deterministic method and not fine-tuned to our dataset, the bad performance is expected and is representative of other rule-based SAST tools. DeepWukong, however, is a sliced-based GNN approach and only achieves an F1-Score of 65% using the Isotonic-Strategy. In Figure 8 it is surprising that the simple token-based approach VulDeePecker with the Probability-Strategy achieves an AUROC of 79% on QEMU and Devign beating SySEVR. Furthermore, we observe an overall beneficial score using the isotonic projection for all methods. Figure 5 shows the result against other graph-based methods. Especially BGNN4VD outperforms the former token-based and slice-based methods with an AUROC of 80% and an F1-Score of 75%. In all of our experiments, using the Mean-Strategy yields the worst scores. The Max and Commit Strategies are similarly underperforming. Both increase the false positive rate too much resulting in disadvantageous F1- and AUROC-scores. That means the naive strategy, to merge all changed functions and classify this, is detrimental to the detection performance. However, calculating the failure probability using the Probability-Strategy is the best aggregation approach. The inferiority of SySEVR and DeepWukong may stem from the fact that they define slices around syntactic vulnerability candidates which may end up with too many candidates and foster false positive alerts. 70% of all statements in FFmpeg correspond to syntactic vulnerable candidates using SySEVR potentially posing a low signal-to-noise ratio.

The probabilistic aggregation is the best strategy for previous models applied to patches, still, PAVUDI provides an up to three times stronger detection performance.

How does the size of a commit affect the performance? We assume that the difficulty of detecting bugs is positively correlated with number of methods touched in a commit. Relying on the Mean-Strategy, for instance, would smooth out the result of a bug prediction with too many changed methods. Resulting in a larger amount of false negatives. Since the Probability-Strategy yields the best result in our evaluation, we proceed to measure the performance loss with commits that have an increasing number of changed methods. In Figure 6 we indeed observe a performance drop. With five changed methods the model performance already deteriorates significantly. VUDDY and BGNN4VD can pertain to their original performance only when analyzing less than 13 and 8 methods respectively. PAVUDI, however, is only slightly affected by the number of methods in a commit.

The detection performance of most baselines deteriorates after 8 changed methods within a patch. PAVUDI's performance slightly drops only after 15 methods.

How does PAVUDI behave after training and deployment? Concept drift describes the performance loss after the deployment of a model. We assume, that the features that a model is applied on might change relative to the features the model was trained on originally. We conduct another experiment to explain whether a model needs to be fine-tuned after initial training and how large its generalization effect is. We train all models, excluding VUDDY, on our dataset. The dataset is sorted by contribution date and split in half. The first half is for training and the second is for validation. In Figure 7, we see that PAVUDI has a significant drop after 300 days. DeepWukong deteriorates after 200 days on FFmpeg. The other models struggle to learn anything initially on the time-sorted datasets. On QEMU

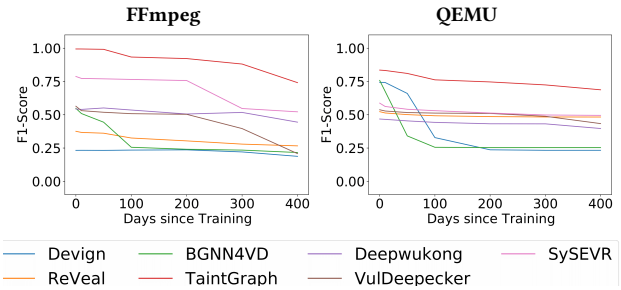


Figure 7: Performance decrease measured over time since training.

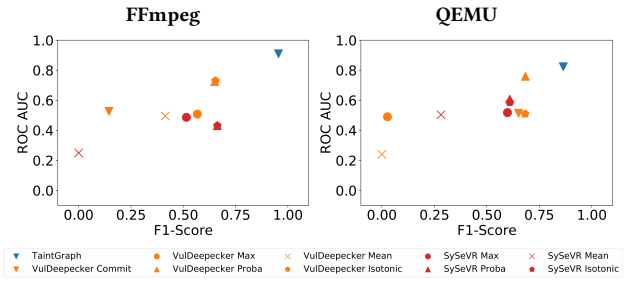


Figure 8: Performance comparison against VulDeePecker and SySEVR.

only Devign and BGNN4VD have a competitive initial F1-Score of 75%. However, their detection capabilities shrink already after respectively 25 and 60 days.

> All baseline models lose their initial performance after at most half a year from training. Only after one year, PAVUDI significantly loses its detection efficiency.

6 PRACTICAL APPLICATION

We demonstrate that PAVUDI has a notable performance benefit over other learning-based vulnerability discovery methods when applied to patches. Due to this, we propose to integrate PAVUDI in practical scenarios.

Integration. PAVUDI is specifically designed to detect patches that either introduce or touch bugs or security vulnerabilities. As already detailed in Section 4.3.1, we can detect bugs in patches by checking out a project at a specific commit and extracting a taint graph through the changed code lines with respect to the preceding commit. This effectively enables us to integrate PAVUDI into a secure software development lifecycle. For instance, PAVUDI can be triggered on every new commit within a continuous integration or deployment pipeline and analyze the changes. We report a finding if PAVUDI's confidence score for a code change exceeds a threshold. However, due to the inherent problem of concept drift in the vulnerable patch detection task which we have shown in our experiments, it is evident that PAVUDI requires a continuous learning process when deployed in software development. Therefore, PAVUDI has to be retrained or fine-tuned regularly.

Interpretation. Each reported finding should be reviewed manually. Therefore, it is essential that the results help security practitioners to find the root cause of the bug quickly. For this reason, we have added an explanation mechanism to PAVUDI that can highlight relevant nodes in the taint graph. Recall the vulnerable patch from Figure 2 and its taint graph in Figure 3 with its most important node highlighted in red according to PAVUDI. Just like Sui et al. [40], we arrive at such an explanation by considering the causal graph G^c during inference. Since the attention mask can be interpreted as relevance scores, we can rank the nodes by their attention and highlight the node with the largest attention value. This directly hints us to the node responsible for an array index calculation that causes the bug in Figure 3. However, taint graphs can become very large and cluttered, thus we suggest extracting line-level information from the highlighted relevant nodes. By simply storing the line-level information on every node, we can extract the relevant code lines from the highlighted nodes according to the explanation. This allows for precise localization of bugs and vulnerabilities and may even be integrated into integrated development environments (IDEs).

6.1 Case-Studies

To demonstrate the practicability of our tool, we apply it in a realistic scenario on two tools with their most recent 100 commits at the time of writing, namely *Tinyproxy* and *Lrzip* to find unknown bugs. We reported every finding to the respective maintainer. In particular, we proceed as outlined in the prior section: We extract the taint graphs, run PAVUDI's inference and extract line-level explanation scores.

Table 3: Detected bugs by PAVUDI.

Project	Bug Description	Found in Commit	Fixed?
Tinyproxy	Buffer-Overflow	453235	Fixed (470cc0)
TinyProxy	Undefined Behavior	64badd	Fixed (6ffd9a)
TinyProxy	Missing Format Limits	252959	Fixed (3764b8)
TinyProxy	Undefined Behavior	252959	Reported
Lrzip	Use-Of-Uninitialized Memory	09ceb8	Reported

Overall we found five bugs as depicted in Table 3. In TinyProxy we identified a buffer overread, three bugs related to undefined behavior, and one bug related to missing width limits in `sscanf` fields. In Lrzip we found a use-of-uninitialized value. We present a bug found in TinyProxy, an HTTP/HTTPS proxy written in C, as a case study in the following Section. In Figure 9, we can see the change of a config parser function. The analyzed patch should initially enhance the parsing speed, however, PAVUDI detects a bug at line 11. The pointer `q` might exceed its limit resulting in a buffer over-read. TinyProxy expects its configuration input space-separated key-value-pairs to be of the form `key value`. If a space is missing, the program crashes.

6.2 Limitations

The discovery of security vulnerabilities in software is a hard problem that is undecidable in the general case. As a result, our approach

Patch-based Vulnerability Discovery

```
...
- while (fgets (buffer, sizeof (buffer), f)) {
-         if (check_match (conf, buffer, lineno)) {
-             printf ("Syntax error on line %ld\n", lineno);
+ for (;fgets (buffer, sizeof (buffer), f);++lineno) {
+         if(buffer[0] == '#') continue;
+         p = buffer;
+         while(isspace(*p))p++;
+         if(!*p) continue;
+         q = p;
+         while(!isspace(*q))q++;
          ...
```

Figure 9: The buffer overflow in TinyProxy's config parser.

naturally comes with limitations and blind spots that we discuss in the following.

Non-taint vulnerabilities. PAVUDI relies on data and control flow between user-controlled sources and critical sinks. This consequently means it can only detect vulnerabilities that manifest in the interprocedural data or control flow. We cannot possibly find bugs that do not share this characteristic, for instance, race conditions or deadlocks.

Definition of sinks. The performance of our approach depends on the definition of appropriate sinks and sources. This may be tricky and subject to each individual project. We follow the approach by Pewny and Holz [34] as described in Section 5.1.2. While this selection of *libc* functions can miss certain vulnerabilities, such as off-by-one errors, including all pointer arithmetics and function calls, as Li et al. [29] propose, leads to computationally infeasible large graphs for our method.

Model updates. Although PAVUDI seems to be more stable than other methods, we see a performance decline after model deployment. This indicates that it is necessary to regularly update or retrain PAVUDI with new data. Therefore, we propose to combine PAVUDI with other vulnerability discovery methods, such as rule-based and dynamic analysis methods, so that each balances the blind spots of the others until updates to models and rules are available.

7 RELATED WORK

In this section, we present the literature related to research areas tangent to this work.

Learning-based vulnerability discovery. Several past works already target the problem of automatically discovering vulnerabilities and bugs. For instance, Russell et al. [35] and Li et al. [30] use a token-based approach. The combination of GNNs and code graphs has already been proven to be suited for the discovery of bugs and security vulnerabilities in software [10, 13, 44, 59]. Zhou et al. [59] introduce the first gated graph neural network on code property graphs to identify bugs in vulnerabilities collected from real-world commits. Their approach outperforms popular open-source and commercial static analyzers as well as token-based ML models. Cao et al. [10] combine data and control-flow graphs with the abstract syntax tree to the code composite graph.

Interprocedural Graphs. Li et al. [29] use interprocedural slices for the vulnerability discovery task. Zheng et al. [57] show that graphs extracted from intraprocedural function slices make it impossible for the models to learn interprocedural bugs spanning multiple functions. Although these approaches try to incorporate interprocedural information, it is insufficient since either the function call depth is limited and rather small, or neither the input source nor critical sinks are considered. Using a whole-program interprocedural graph representation would solve this problem, however, this is a nontrivial task, since programs can become rather large [41]. Our approach overcomes the issue by only selecting relevant paths. Cheng et al. [15] extract multiple interprocedural paths starting from a function under analysis until its return. They neglect sources and sinks and abstain from a whole-program perspective.

Learning on patches. Since this work focuses on patches, it is noticeable that there is a research interest around the classification of patches [46, 48] that have been introduced silently or are security relevant. In this particular field, GNNs could have been applied successfully [45]. Wang et al. [43] also classify security-relevant patches to improve the dataset preparation. Another research branch tries to detect anomalies in patches using meta-information of the specific versioning control system [18, 21, 22].

Explainable AI. Deep-Learning models tend to be black boxes, hence, there exist a large variety of explanation methods to enable us to interpret a model's decision. Guo et al. [24] introduce a black-box method specifically for security-relevant models. Selvaraju et al. [38], for instance, introduce a white-box method for image classification. Sanchez-Lengeling et al. [36] port many methods to the graph domain. Ganz et al. [19] and Warnecke et al. [49] show that explanation methods used to locate bugs tend to reveal bias and artifacts from the training procedure of a model. We address the explainability of PAVUDI in this paper using soft attention as presented by Sanchez-Lengeling et al. [36] as it has already been done in other works [17].

8 CONCLUSION

In this work, we adapt several state-of-the-art learning-based vulnerability discovery models to vulnerable patch detection and show that they have a poor performance and their detection capabilities even degrade over time after deployment. As a consequence, we present our novel patch-based vulnerability detection model, PAVUDI, which leverages interprocedural code graphs and taint-style static program analysis.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to Represent Programs with Graphs. *CoRR* abs/1711.00740 (2017). arXiv:1711.00740 http://arxiv.org/abs/1711.00740

[3] Roberto Amadini, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2020. Abstract Interpretation, Symbolic Execution and Constraints. In *Recent Developments in the Design and Implementation of Programming Languages (OpenAccess Series in Informatics (OASIcs), Vol. 86)*, Frank S. de Boer and Jacopo Mauro (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 7:1–7:19. https://doi.org/10.4230/OASIcs.Gabbrielli.7

[4] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2020. Dos and Don'ts of Machine Learning in Computer Security. *CoRR* abs/2010.09470 (2020). arXiv:2010.09470 https://arxiv.org/abs/2010.09470

[5] Sindre Beba and Magnus Melseth Karlsen. 2019. Implementation Analysis of Open-Source Static Analysis Tools for Detecting Security Vulnerabilities.

[6] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. *CoRR* abs/1806.07336 (2018). arXiv:1806.07336 http://arxiv.org/abs/1806.07336

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020

[8] Brad Calder, Dirk Grunwald, and Benjamin Zorn. 1994. Quantifying Behavioral Differences Between C and C++ Programs. *Journal of Programming Languages* 2 (02 1994).

[9] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. 2021. BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection. *Inf. Softw. Technol.* 136 (2021), 106576.

[10] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. 2021. BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection. *Information and Software Technology* 136 (2021), 106576. https://doi.org/10.1016/j.infsof.2021.106576

[11] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[12] Yizheng Chen, Zhoujie Ding, Xinyun Chen, and David A. Wagner. 2023. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. *ArXiv* abs/2304.00409 (2023).

[13] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2020. DeepWukong: Statically Detecting Software Vulnerabilities using Deep Graph Neural Network. *ACM Trans. Softw. Eng. Methodol. 1, 1, Article* 1 (2020), 32. https://doi.org/10.1145/3436877

[14] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. *ACM Trans. Softw. Eng. Methodol.* 30, 3, Article 38 (apr 2021), 33 pages. https://doi.org/10.1145/3436877

[15] Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. 2022. Path-Sensitive Code Embedding via Contrastive Learning for Software Vulnerability Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 519–531. https://doi.org/10.1145/3533767.3534371

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[17] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE.

[18] Tom Ganz, Inaam Ashraf, Martin Härterich, and Konrad Rieck. 2023. Detecting Backdoors in Collaboration Graphs of Software Repositories. In *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy* (Charlotte, NC, USA) *(CODASPY '23)*. Association for Computing Machinery, New York, NY, USA, 189–200. https://doi.org/10.1145/3577923.3583657

[19] Tom Ganz, Martin Härterich, Alexander Warnecke, and Konrad Rieck. 2021. Explaining Graph Neural Networks for Vulnerability Discovery. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security* (Virtual Event, Republic of Korea) *(AISec '21)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/3474369.3486866

[20] Tom Ganz, Philipp Rall, Martin Härterich, and Konrad Rieck. 2023. Hunting for Truth: Analyzing Explanation Methods in Learning-based Vulnerability Discovery. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. 524–541. https://doi.org/10.1109/EuroSP57164.2023.00038

[21] Izhak Golan and Ran El-Yaniv. 2018. Deep anomaly detection using geometric transformations. *arXiv preprint arXiv:1805.10917* (2018).

[22] Danielle Gonzalez, T. Zimmermann, Patrice Godefroid, and Maxine Schaefer. 2021. Anomalicious: Automated Detection of Anomalous and Potentially Malicious Commits on GitHub. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (2021), 258–267.

[23] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward Large-Scale Vulnerability Discovery Using Machine Learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy* (New Orleans, Louisiana, USA) *(CODASPY '16)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2857705.2857720

[24] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. 2018. LEMNA: Explaining Deep Learning Based Security Applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 364–379. https://doi.org/10.1145/3243734.3243792

[25] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. 595–614. https://doi.org/10.1109/SP.2017.62

[26] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *CoRR* abs/1609.02907 (2016). arXiv:1609.02907 http://arxiv.org/abs/1609.02907

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

[28] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2017. Gated Graph Sequence Neural Networks. arXiv:1511.05493 [cs.LG]

[29] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, Zhaoxuan Chen, Sujuan Wang, and Jialai Wang. 2018. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *CoRR* abs/1807.06756 (2018). arXiv:1807.06756 http://arxiv.org/abs/1807.06756

[30] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *CoRR* abs/1801.01681 (2018). arXiv:1801.01681 http://arxiv.org/abs/1801.01681

[31] Alexandru Niculescu-Mizil and Rich Caruana. 2005. Predicting good probabilities with supervised learning. *ICML 2005 - Proceedings of the 22nd International Conference on Machine Learning*, 625–632. https://doi.org/10.1145/1102351.1102430

[32] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis*. Springer Publishing Company, Incorporated.

[33] Cong Pan and Michael Pradel. 2021. Continuous Test Suite Failure Prediction. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 553–565. https://doi.org/10.1145/3460319.3464840

[34] Jannik Pewny and Thorsten Holz. 2016. EvilCoder: Automated Bug Insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (Los Angeles, California, USA) *(ACSAC '16)*. Association for Computing Machinery, New York, NY, USA, 214–225. https://doi.org/10.1145/2991079.2991103

[35] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. *CoRR* abs/1807.04320 (2018). arXiv:1807.04320 http://arxiv.org/abs/1807.04320

[36] Benjamin Sanchez-Lengeling, Jennifer Wei, Brian Lee, Emily Reif, Peter Wang, Wesley Qian, Kevin McCloskey, Lucy Colwell, and Alexander Wiltschko. 2020. Evaluating Attribution for Graph Neural Networks. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 5898–5910. https://proceedings.neurips.cc/paper/2020/file/417fbbf2e9d5a28a855a11894b2e795a-Paper.pdf

[37] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE Symposium on Security and Privacy*. 317–331. https://doi.org/10.1109/SP.2010.26

[38] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*. 618–626.

[39] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, Flow-, and Field-Sensitive Data-Flow Analysis Using Synchronized Pushdown Systems. *Proc. ACM Program. Lang.* 3, POPL, Article 48 (jan 2019), 29 pages. https://doi.org/10.1145/3290361

[40] Yongduo Sui, Xiang Wang, Jiancan Wu, Min Lin, Xiangnan He, and Tat-Seng Chua. 2022. Causal Attention for Interpretable and Generalizable Graph Classification. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 1696–1705. https://doi.org/10.1145/3534678.3539366

[41] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental Whole-Program Analysis in Datalog with Lattices. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3453483.3454026

[42] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. *International Conference on Learning Representations* (2018). https://openreview.net/forum?id=rJXMpikCZ

[43] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang. 2021. Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection. *IEEE Transactions on Information Forensics and Security* 16 (2021), 1943–1958. https://doi.org/10.1109/TIFS.2020.3044773

[44] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. 2021. Combining

Graph-Based Learning with Automated Data Collection for Code Vulnerability Detection. *IEEE Transactions on Information Forensics and Security* 16 (2021), 1943–1958. https://doi.org/10.1109/TIFS.2020.3044773

[45] Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. 2023. GraphSPD: Graph-Based Security Patch Detection with Enriched Code Semantics. In *2023 IEEE Symposium on Security and Privacy (SP)*. 604–621. https://doi.org/10.1109/SP46215.2023.00035

[46] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. 2021. PatchDB: A Large-Scale Security Patch Dataset. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 149–160. https://doi.org/10.1109/DSN48987.2021.00030

[47] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. 2021. PatchRNN: A Deep Learning-Based System for Security Patch Identification. In *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*. 595–600. https://doi.org/10.1109/MILCOM52596.2021.9652940

[48] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. 2021. PatchRNN: A Deep Learning-Based System for Security Patch Identification. In *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*. 595–600. https://doi.org/10.1109/MILCOM52596.2021.9652940

[49] Alexander Warnecke, Daniel Arp, Christian Wressnegger, and Konrad Rieck. 2020. Evaluating Explanation Methods for Deep Learning in Security. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, Genoa, Italy, 158–174. https://doi.org/10.1109/EuroSP48549.2020.00018

[50] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (apr 1991), 181–210. https://doi.org/10.1145/103135.103136

[51] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A Comprehensive Survey on Graph Neural Networks. *CoRR* abs/1901.00596 (2019). arXiv:1901.00596 http://arxiv.org/abs/1901.00596

[52] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*. https://openreview.net/forum?id=ryGs6iA5Km

[53] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604. https://doi.org/10.1109/SP.2014.44

[54] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604. https://doi.org/10.1109/SP.2014.44

[55] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing system-wide information flow for malware detection and analysis. *Proceedings of the ACM Conference on Computer and Communications Security*, 116–127. https://doi.org/10.1145/1315245.1315261

[56] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How Do Fixes Become Bugs?. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 26–36. https://doi.org/10.1145/2025113.2025121

[57] Weining Zheng, Yuan Jiang, and Xiaohong Su. 2021. VulSPG: Vulnerability detection based on slice property graph representation learning. *CoRR* abs/2109.02527 (2021). arXiv:2109.02527 https://arxiv.org/abs/2109.02527

[58] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. *CoRR* abs/1909.03496 (2019). arXiv:1909.03496 http://arxiv.org/abs/1909.03496

[59] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. *CoRR* abs/1909.03496 (2019). arXiv:1909.03496 http://arxiv.org/abs/1909.03496

APPENDIX

Table 4: Functions for V_{source}.

Function	Description
getchar/getc/getch	Reads a char from stdin
fgets	Reads a line from a stream
read	Reads from a file descriptor
fopen	Opens a file
scanf	Reads formatted input from stdin
gets	Reads input from stdin
fscanf	Reads formatted input from a stream
getenv/secure_getenv	Reads from an environment variable
fread	Reads input from a stream
poll/ppoll	Wait for event on file descriptor
recvfrom/recv/recvmsg	Receives message from socket

Table 5: Functions for V_{sink}.

Function	Description
malloc/calloc/realloc	Allocate heap memory.
memcpy	Copies memory content.
strcpy	Copies string content.
printf/snprintf/sprintf	Provides formatted output.
memset	Initializes memory.
strcat	Concatenates strings.
free	Deallocates memory.