D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis

Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang,
Jim Laredo, Alessandro Morari, Zhong Su
IBM Research
{zhengyu, burn, eae, laredoj, amorari}@us.ibm.com, {saurabh.pujar, luca.buratti1}@ibm.com,
{yangbbo, suzhong}@cn.ibm.com

***Abstract*—Static analysis tools are widely used for vulnerability detection as they understand programs with complex behavior and millions of lines of code. Despite their popularity, static analysis tools are known to generate an excess of false positives. The recent ability of Machine Learning models to understand programming languages opens new possibilities when applied to static analysis. However, existing datasets to train models for vulnerability identification suffer from multiple limitations such as limited bug context, limited size, and synthetic and unrealistic source code. We propose D2A, a differential analysis based approach to label issues reported by static analysis tools. The D2A dataset is built by analyzing version pairs from multiple open source projects. From each project, we select bug fixing commits and we run static analysis on the versions before and after such commits. If some issues detected in a before-commit version disappear in the corresponding after-commit version, they are very likely to be real bugs that got fixed by the commit. We use D2A to generate a large labeled dataset to train models for vulnerability identification. We show that the dataset can be used to build a classifier to identify possible false alarms among the issues reported by static analysis, hence helping developers prioritize and investigate potential true positives first.**

***Index Terms*—dataset, vulnerability detection, auto-labeler**

I. Introduction

The complexity and scale of modern software programs often lead to overlooked programming errors and security vulnerabilities. Research has shown that developers spend more than 50% of their time detecting and fixing bugs [1], [2]. In practice, they usually rely on automated program analysis or testing tools to audit the code and look for security vulnerabilities. Among them, static program analysis techniques have been widely used because they can understand nontrivial program behaviors, scale to millions of lines of code, and detect subtle bugs [3], [4], [5], [6]. Although static analysis has limited capacity to identify bug-triggering inputs, it can achieve better coverage and discover bugs that are missed by dynamic analysis and testing tools. In fact, static analysis can provide useful feedback and has been proven to be effective in improving software quality [7], [8].

Besides these classic usage scenarios, driven by the needs of recent AI research on source code understanding and vulnerability detection tasks [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], static analysis techniques have also been used to generate labeled datasets for model training [12]. As programs exhibit diverse and complex behaviors, training models for vulnerability detection requires large labeled datasets of buggy and non-buggy code examples. This is especially critical for advanced neural network models such as CNN, RNN, GNN, etc. However, existing datasets for vulnerability detection suffer from the following limitations:

- Almost all datasets are on a function level and do not provide context information (e.g., traces) explaining how a bug may happen. Besides, they usually do not specify the bug types and locations. In many cases, the function-level example does not even include the bug root cause.
- Some datasets (e.g. CGD [13]) are derived from confirmed bugs in NVD [19] or CVE [20]. Although they have high-quality labels, the number of such samples is limited and may be insufficient for model training.
- Synthetic datasets such as Juliet [21] and S-babi[14] can be large. However, they are generated based on a few predefined patterns and thus cannot represent the diverse behaviors observed in real-world programs.
- There are also labeling efforts based on commit messages or code diffs. Predicting code labels based on commit messages is known to produce low-quality labels [12]. Code diff based methods [15] assume all functions in a bug-fixing commit are buggy, which may not be the case in reality. More importantly, these approaches have difficulty in identifying bug types, locations, and traces.

On the other hand, static analysis can reason beyond function boundaries. It's automated and scales well enough to generate large datasets from programs in the wild. For example, Russell et. al [12] applied the Clang static analyzer, Cppcheck, and Flawfinder to generate a labeled data set of millions of functions to train deep learning models and learn features from source code. In some sense, it is the most promising labeling approach as it can additionally identify bug types and locations while using traces as context information.

Despite the popularity in these scenarios, static analysis tools are known to generate an excess of false alarms. One reason is the approximation heuristics used to reduce complexity and improve scalability. In particular, static analysis tries to model all possible execution behaviors and thus can suffer from the state-space blowup problem [10]. To handle industry-scale programs, static analysis tools aggressively approximate the analysis and sacrifice the precision for better scalability and speed. For example, the path-sensitive analysis does not

scale well on large programs, especially when modeling too many path states or reasoning about complex path constraints. Therefore, path insensitive analysis that ignores path conditions and assumes all paths are feasible is commonly used in practice, which obviously introduces false positives.

These *false positives greatly hinder the utilization of static analysis tools* as it is counterproductive for developers to go through a long list of reported issues but only find a few true positives [22], [23]. To suppress them, various methods have been proposed (as summarized in [24]). Among them, machine learning based approaches [25], [26], [27], [28], [10], [29], [30], [31], [32], [33] focus on learning the patterns of false positives from examples. However, training such models requires good labeled datasets. Most existing works manually generate such datasets by reviewing the code and bug reports. In our experience, this review process is very labor-intensive and cannot scale. Therefore, the datasets are relatively small and may not cover the diverse behaviors observed in reality.

To address these challenges, in this paper, we propose D2A, a differential analysis based approach to label issues reported by static analysis tools as ones that are *more likely to be true positives* and ones that are *more likely to be false positives*. Our goal is to generate a large labeled dataset that can be used for machine learning approaches for (1) static analyzer false positive reduction, and (2) code understanding and vulnerability detection tasks. We demonstrate how our dataset can be helpful for the false positive reduction task.

In particular, for projects with commit histories, we assume some commits are code changes that fix bugs. Instead of predicting labels based on commit messages, we run static analysis on the versions before and after such commits. If some issues detected in a before-commit version disappear in the corresponding after-commit version, they are *very likely to be real bugs* that got fixed by the commit. If we analyze a large number of consecutive version pairs and aggregate the results, some issues found in a before-commit version never disappear in an after-commit version. We say they are *not very likely to be real bugs* because they were never fixed. Then, we de-duplicate the issues found in all versions and adjust their classifications according to the commit history. Finally, we label the issues that are very likely to be real bugs as *positives* and the remaining ones as *negatives*. We name this procedure *differential analysis* and the labeling mechanism *auto-labeler*. Please note that we say the reported issues are very likely to be TPs or FPs because the static analyzer may make mistakes or a bug-fixing commit was not included for analysis. We will discuss this in more detail in Sec. III.

We run the differential analysis on thousands of selective consecutive version pairs from `OpenSSL`, `FFmpeg`, `libav`, `httpd`, `NGINX` and `libtiff`. Out of 349,373,753 issues reported by the static analyzer, after deduplication, we labeled 18,653 unique issues as positives and 1,276,970 unique issues as negatives. Given there is no ground truth, to validate the efficacy of the auto-labeler, we randomly selected and manually reviewed 57 examples. The result shows that D2A improves the label accuracy from 7.8% to 53%.

Although the D2A dataset is mainly for machine learning based vulnerability detection methods, which usually require a large number of labeled samples, in this paper, we show it can be used to help developers prioritize static analysis issues that are more likely to be true positives. We will present AI-based code understanding approaches in another paper. In particular, inspired by [10], we defined features solely from static analysis outputs and trained a static analysis false positive reduction model. The result shows that we were able to significantly reduce false alarms, allowing developers to investigate issues that are less likely to be false positives first. In summary, we make the following contributions:

- We propose a novel approach to label static analysis issues based on differential analysis and commit history heuristics.
- Given it can take several hours to analyze a single version pair (e.g. 12hrs for `FFmpeg`), we parallelized the pipeline such that we can process thousands of version pairs simultaneously in a cluster, which makes D2A a practical approach.
- We ran large-scale analyses on thousands of version pairs of real-world C/C++ programs, and created a labeled dataset of millions of samples with a hope that the dataset can be helpful to AI method on vulnerability detection tasks.
- Unlike existing function-level datasets, we derive samples from inter-procedural analysis and preserve more details such as bug types, locations, traces, and analyzer outputs.
- We demonstrated a use case of the D2A dataset. We trained a static analysis false positive reduction model, which can effectively reduce false positive rate and help developers prioritize issues that are more likely to be real bugs.
- To facilitate future research, we make the D2A dataset and its generation pipeline publicly available at https://github.com/ibm/D2A.

II. Motivation

In this section, we describe two use scenarios to show why building a good labeled dataset using static analysis can be useful for AI-based vulnerability detection methods.

A. Existing Datasets for AI on Vulnerability Detection Task

Since programs can exhibit diverse behaviors, training machine learning models for code understanding and vulnerability detection requires large datasets. However, according to a recent survey [35], lacking good and real-world datasets has become a major barrier for this field. Many existing works created self-constructed datasets based on different criteria. However, only a few fully released their datasets.

Table I summarizes the characteristics of a few popular publicly available software vulnerability datasets. We compare these datasets to highlight the contributions D2A can make.

Juliet [21], Choi et.al [34], and S-babi [14] are synthetic datasets that were generated from predefined patterns. Although their sizes are decent, the main drawback is the lack of diversity comparing to real-world programs [34].

The examples in Draper [12] are from both synthetic and real-world programs, where each example contains a function and a few labels indicating the bug types. These labels were

TABLE I
PUBLICLY AVAILABLE DATASETS FOR AI ON C/C++ VULNERABILITY DETECTION

Dataset	Example Type	Example Level	Whole Dataset Released	Bug Type	Bug Line	Bug Trace	Codebase Traceability	Compilable Example	Generation Impl. Avail.	Labelling Method
Juliet [21]	synthetic	function	✔	✔	✔	✘	–	✔	–	predefined pattern
S-Babi [14]	synthetic	function	✔	✔	✔	✘	–	✔	✔	predefined pattern
Choi et.al [34]	synthetic	function	✔	✔	✔	✘	–	✔	✔	predefined pattern
Draper [12]	mixed	function	✔	✔	✘	✘	✘	✘	✘	static analysis
Devign [15]	real-world	function	✘	✘	✘	✘	✘	✘	✘	manual + commit code diff
CDG [13]	real-world	slice	✔	✘	✘	✘	✘	✘	✔	NVD + code diff
D2A	real-world	trace	✔	✔	✔	✔	✔	✔	✔	differential static analysis

Note: To the best of our knowledge, there is no perfect dataset that is large enough and has 100% correct labels for AI-based vulnerability detection tasks. Datasets generated from manual reviews have better quality labels in general. However, limited by their nature, they are usually not large enough for model training. On the other hand, the quality of the D2A dataset is bounded by the capacity of static analysis. D2A has better labels comparing to datasets labeled solely by static analysis and complements existing high-quality datasets by the size. Please refer to Sec. II-A for details.

generated by aggregating static analysis results. Draper doesn't provide details like bug locations or traces. For real-world programs, Draper doesn't maintain the links to the original code base. If we want to further process the function-level examples to obtain more information, it's difficult to compile or analyze them without headers and compiler arguments.

The Devign [15] dataset contains real-world function examples from commits, where the labels are manually generated based on commit messages and code diffs. In particular, if a commit is believed to fix bugs, all functions patched by the commit are labeled as 1, which are not true in many cases. In addition, only a small portion of the dataset was released.

CDG [13] is derived from real-world programs. It's unique because an example is a subset of a program slice and thus not a valid program. Its label was computed based on NVD: if the slice overlaps with a bug fix, it's labeled as 1. Since the dataset is derived from confirmed bugs, the label quality is better. However, the number of such examples is limited and may not be sufficient for model training.

In fact, there is a pressing need for labeled datasets from real-world programs and encoding context information beyond the function boundary [35], [15], [13]. It has been shown that preserving inter-procedural flow in code embedding can significantly improve the model performance (e.g. 20% precision improvement in code classification task) [36]. To this end, D2A examples are generated based on inter-procedural analysis, where an example can include multiple functions in the trace. D2A also provides extra details such as the bug types, bug locations, bug traces, links to the original code base/commits, analyzer outputs, and compiler arguments that were used to compile the files having the functions. We believe they are helpful for AI for vulnerability detection in general.

B. Manual Review and False Positive Reduction

We start by running a state-of-the-art static analyzer on a large real-world program. We select bug types that may lead to security problems and manually go through each issue to confirm how many reported issues are real bugs.

The goal of this exercise is two-fold. First, we want to understand the performance of a state-of-the-art static analyzer for large real-world programs in terms of how many reported issues are real bugs. Second, by looking at the false positives, we want to explore ideas that can treat the static analyzer as a black box and suppress the false positives.

TABLE II
MANUAL REVIEW: OPENSSL `7f0a8dc`

Error Type	Reported	Manual Review		
		FP	TP	FP:TP
UNINITIALIZED_VALUE	101	101	0	–
NULL_DEREFERENCE	64	51	13	4:1
RESOURCE_LEAK	1	1	0	–
TOTAL	**166**	**153**	**13**	**12:1**

Note: 326 DEAD_STORE issues were excluded from manual review.

```
crypto/initthread.c:385: error: NULL_DEREFERENCE
  pointer `gtr` last assigned on line 382 could be null and is dereferenced
  at line 385, column 53.
Showing all 5 steps of the trace

crypto/initthread.c:377:1: start of procedure init_thread_deregister()
375.
376.   #ifndef FIPS_MODE
377. > static int init_thread_deregister(void *index, int all)
378.   {
379.       GLOBAL_TEVENT_REGISTER *gtr;
380.
crypto/initthread.c:382:5:
380.       int i;
381.
382. >     gtr = get_global_tevent_register();
383.       if (!all)
384.           CRYPTO_THREAD_write_lock(gtr->lock);
385.
...
386.
crypto/initthread.c:385:17:
383.       if (!all)
384.           CRYPTO_THREAD_write_lock(gtr->lock);
385. >     for (i = 0; i < sk_THREAD_EVENT_HANDLER_PTR_num(gtr->skhands); i++) {
386.           THREAD_EVENT_HANDLER **hands
387.               = sk_THREAD_EVENT_HANDLER_PTR_value(gtr->skhands, i);
```

Fig. 1. Infer Bug Report Example.

1) Manual Case Study: Since we are interested in large C/C++ programs, we require the static analyzer should be able to handle industrial-scale programs and detect a broad set of bug types. To the best of our knowledge, the Clang Static Analyzer [37] and Infer [38] are two state-of-the-art static analyzers that satisfy our needs. However, the Clang Static Analyzer doesn't support cross translation unit analysis such that the inter-procedural analysis may be incomplete. Therefore, we choose Infer in our experiments. We use `OpenSSL` version `7f0a8dc` as the benchmark, which has 1499 `*.c/*.h` files and 513.6k lines of C code in total.

We run Infer using its default setting and the results are summarized in Table II. Infer reported 492 issues of 4 bug types: 326 `DEAD_STORE`, 101 `UNINITIALIZED_VALUE`, 64 `NULL_DEREFERENCE`, and 1 `RESOURCE_LEAK`. Among them, `DEAD_STORE` refers to the issues where the value written to a variable is never used. Since such issues are not directly related to security vulnerabilities, they were excluded

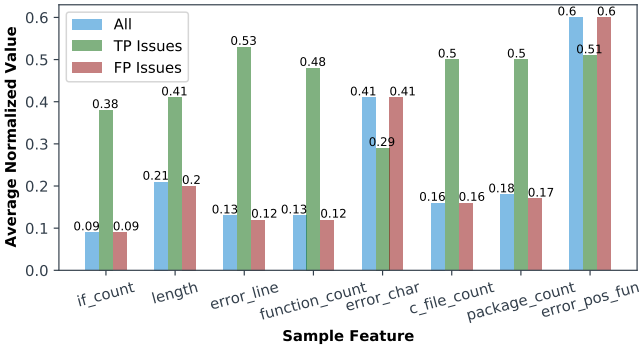

Fig. 2. Feature Exploration. We experiment with a few features that may reflect the complexity of the issues. After normalization, the averaged feature values of true positives and false positives are significantly different, which suggests a classifier may achieve good performance.

from the manual review. The remaining 166 issues may lead to security-related problems and thus were included in the study.

The manual review was performed by 8 developers who are proficient in C/C++. We started by understanding the bug reports produced by Infer. Fig. 1 shows an example of the bug report of a `NULL_DEREFERENCE` issue. It has two sections. The bug location, bug type, and a brief justification why Infer thinks the bug can happen are listed in lines 1–3. The bug explanation part can be in different formats for different bugs. In lines 6-27, the bug trace that consists of the last steps of the offending execution is listed. Fig. 1 shows 3 of the 5 steps. In each step (e.g. line 6-11), the location and 4 additional lines of code that sit before and after the highlighted line are provided.

We firstly had two rounds of manual analyses to figure out if the reported issue may be triggered. Each issue was reviewed by two reviewers. If both reviewers agreed that the reported bug can happen, we have an additional round of review and try to confirm the bug by constructing a test case. This process was very time consuming and challenging, especially when reviewing a complex program with cryptography involved.

As shown in Table II, out of 166 security vulnerability related issues, we confirmed that **13 (7.8%)** issues are true positives and **92.2%** are false positives.

2) Feature Exploration for False Positive Reduction: During the manual review, we found we can make a good guess for some issues by looking at the bug reports. Inspired by the existing false positive reduction works [27], [10], we explored the idea of predicting if the issues flagged by Infer are true positives solely based on the bug reports as shown in Fig. 1.

Existing approaches are not directly applicable as they target different languages or static analyzers. Following the intuition that complex issues are more likely to be false positives, we considered features in bug reports that may reflect the issue complexity. We explored the following 8 features that belong to 3 categories: (1) *error_line* and *error_char* denote the location (line and column number) where the bug occurs. (2) *length*, *c_file_count* and *package_count* denote the unique number of line numbers, source files and the directories respectively in the trace. (3) *if_count* and *function_count* are the numbers of the branches and functions in the trace.

We extracted the features from the bug reports of 166 issues. After normalization, we computed the average feature values of the 13 true positive issues and 153 false positive issues. As shown in Fig. 2, the average feature values of true positives and false positives are significantly different and easily separable for all 8 features, which suggests a good false positive reduction classifier can perform very well.

III. Dataset Generation

In this section, we present the differential analysis based approach that labels the issues detected by the static analyzer. Then, we show how we generate two kinds of examples for the D2A dataset based on the results obtained.

A. Overview

Fig. 3 shows the overall workflow of D2A. The input to the pipeline is a URL to a git repository. The output are examples generated purely using the static differential analysis.

As the pre-processing step, based on the commit messages only, the Commit Message Analyzer (Sec. III-B) selects a list of commits that are likely to be bug fixes. Because it can be very expensive to analyze a pair of consecutive versions, the goal of this step is to filter out commits that are not closely related to bug fixes (e.g. documentation improvement commits) and speed up the process.

For each selected commit, we obtain two sets of issues reported by the static analyzer by running the analyzer on the before-commit and the corresponding after-commit versions. The auto-labeler (Sec. III-C) compares these two sets and identifies the issues that are fixed by the commit.

After aggregating all such issues from multiple consecutive version pairs and filtering out noises based on commit history, the auto-labeler labels issues that are *very likely to be real bugs* as positives, and the issues that are never fixed by a commit as negatives because they are *very likely to be false positives*. We further extract the function bodies according to the bug traces and create the dataset.

B. Commit Message Analysis

We created the Commit Message Analyzer (CMA) to identify commits that are more likely to refer to vulnerability fixes and not documentation changes or new features. Using the NVD dataset [19], CMA learns the language of vulnerabilities and uses a hybrid approach that combines semantic similarity-based methods [39] and snippet samples-based methods [40] to identify relevant commit messages and their associated commit. Noise is reduced by eliminating meaningless tokens, names, email addresses, links, code, etc from each commit message prior to the analysis.

Based on the semantic distribution of the vulnerable mentions, CMA identifies the category of the vulnerability and ranks commits based on confidence scores.

C. Auto-labeler Examples

For each bug-fixing commit selected by CMA, we run the static analyzer on the versions before and after the commit. We evaluated several static analyzers such as CppCheck [41],

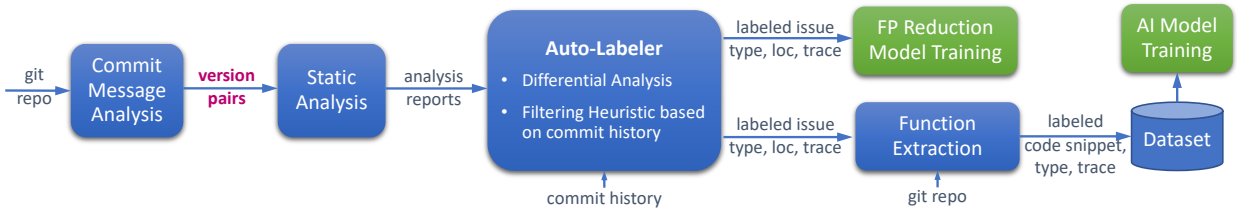

Fig. 3. The Overview of D2A Dataset Generation Pipeline.

Flawfinder [42], Clang Static Analyzer [37], and Infer [38]. We chose Infer because it can detect a nice set of security related bug types and supports cross translation unit analysis necessary for effective inter-procedural analysis. More importantly, it scales well on large programs.

Identify Fixed Issues in a Version Pair. If we denote the issues found in the before-commit version as I_{before} and the ones in the corresponding after-commit version as I_{after}, all issues can be classified into three groups: (1) the *fixed issues* ($I_{\text{before}} - I_{\text{after}}$) that are detected in the before-commit version but disappear in the after-commit version, (2) the *pre-existing issues* ($I_{\text{after}} \cap I_{\text{before}}$) that are detected in both versions, and (3) the *introduced issues* ($I_{\text{after}} - I_{\text{before}}$) that are not found in the before-commit versions but detected in the after-commit version. We are particularly interested in the *fixed issues* because they are very likely to be bugs fixed by the commit. We use the `infer-reportdiff` tool [43] to compute them.

Note that it's possible that a *fixed issue* is not a real bug as the static analyzer may make mistakes, e.g. omit an issue from the after-commit even though the code had not changed. In our experience, an important reason is that Infer can exhibit non-deterministic behaviors [44]. And the non-determinism occurs more frequently when enabling parallelization [45]. In order to minimize the impact, we have to run Infer in single-threaded mode. However, this setting brings in performance challenges and it takes several hours to analyze a version pair. For example, on an IBM POWER8 cluster, it takes 5.3 hrs and 12 hrs to analyze a version pair of `OpenSSL` and `FFmpeg`, respectively, in single-thread mode. As we will need to analyze thousands of version pairs, it's impractical to do so on a PC or a small workstation. Therefore, we addressed several technical challenges and parallelized the analysis to process more than a thousand version pairs simultaneously in a cluster.

Merge Issues Based on Commit History. After identifying fixed issues in each version pair, we merge and deduplicate the issues from all version pairs. In particular, we compute the sha1sum of the bug report after removing location-related contents (e.g, the file names, line numbers, etc) and use it as the id for deduplication. The reason why we remove location-related contents is that the same piece of code may be bumped to a different location by a commit, changing only the line numbers in the report. Then, we apply the following two heuristics to filter out the ones that are unlikely to be bugs based on the commit history.

- *Fixed-then-unfixed issues*: The same issue may appear in multiple version pairs. We sort all its occurrences based on the author date of the commit. If a fixed issue appears again in a *later* version, it's probably a false positive due to the mistake of the static analyzer. We change the labels of such cases and mark them as *negatives*.
- *Untouched issues*: We check which parts of the code base are patched by the commit. If the commit code diff doesn't overlap with any step of the bug trace at all, it's unlikely the issue is fixed by the commit but more likely to be a false positive reported by the static analyzer. We mark such cases *negatives* as well.

After applying the above filters, the remaining issues in the *fixed issues* group are labeled as positives (issues that are more likely to be buggy) and all other issues are labeled as negatives (issues that are more likely to be non-buggy). We call these *auto-labeler examples*. Because auto-labeler examples are generated based on issues reported by Infer, they all have the infer bug reports.

D. After-fix Examples

Due to the nature of vulnerabilities, the auto-labeler produces many more negatives than positives such that the dataset of auto-labeler examples is quite imbalanced. Given that the positive auto-labeler examples are assumed to be bugs fixed in the after-commit versions, extracting the corresponding fixed versions is another kind of negative examples, which we call *after-fix examples*. There are two benefits: (1) Since each negative example corresponds to a positive example, the dataset of auto-labeler positive examples and after-fix negative examples is balanced. (2) The after-fix negative examples are closely related to the positive ones so that they may help models focus on the delta parts that fixed the bugs. Note that the *after-fix* examples do not have a static analysis bug report because the issue does not appear in the after-commit version.

E. An Example in the D2A Dataset

Fig. 4 shows a D2A example, which contains bug-related information obtained from the static analyzer, the code base, and the commit meta-data.

In particular, every example has its *label* (0 or 1) and *label_source* ("auto_labeler" or "after_fix_extractor") to denote how the example was generated and if it is buggy. *bug_type*, *bug_info*, *trace* and *zipped_bug_report* are obtained from the static analyzer, which provides details about the bug types, locations, traces, and the raw bug report produced by Infer. This information can be useful to train models on bug reports.

For each step in the *trace*, if it refers to a location inside a function, we extract the function body and save it in the

```
{
  "id": "httpd_9b3a5f0ffd8ec787cf645f97902582acb3234d96_1",
  "label": 1,
  "label_source": "auto_labeler",
  "bug_type": "BUFFER_OVERRUN_U5",
  "project": "httpd",
  "bug_info": {
    "qualifier": "Offset: [0, +oo] Size: 10 by call to ...",
    "loc": "modules/proxy/mod_proxy_fcgi.c:178:31",
    "url": "https://github.com/apache/httpd/blob/..."
  },
  "versions": {
    "before": "545d85acdaa384a25ee5184a8ee671a18ef5582f",
    "after": "2c70ed756286b2adf81c55473077698d6d6d16a1"
  },
  "trace": [
    {
      "description": "Array declaration",
      "loc": "modules/proxy/mod_proxy_fcgi.c:178:31",
      "func_key": "modules/proxy/mod_proxy_fcgi.c@167:1-203:2",
    }
  ],
  "functions": {
    "modules/proxy/mod_proxy_fcgi.c@167:1-203:2": {
      "name": "fix_cgivars",
      "touched_by_commit": true,
      "code": "static void fix_cgivars(request_rec *r, ..."
    }
  },
  "commit": {
    "url": "https://github.com/apache/httpd/commit/2c70ed7",
    "changes": [
      {
        "before": "modules/proxy/mod_proxy_fcgi.c",
        "after": "modules/proxy/mod_proxy_fcgi.c",
        "changes": ["177,1^^177,5"]
      }
    ]
  },
  "compiler_args": {
    "modules/proxy/mod_proxy_fcgi.c": "-D_REENTRANT -I./server ...",
  },
  "zipped_bug_report": "..."
}
```


Fig. 4. A Simplified Example in D2A Dataset.

functions section. Therefore, an example has all functions involved in the bug trace, which can be used by function level or trace level models. Besides, we cross-check with commit code diff. If a function is patched by the commit, the *touched_by_commit* is true.

In addition, the compiler arguments used to compile the source file are saved in the *compiler_args* field. They can be useful when we want to run extra analysis that requires compilation (e.g. libclang [46] based tools).

IV. Static Analysis False Positive Reduction

Although the D2A dataset is mainly for machine learning based vulnerability detection methods, in this paper, we show the dataset can be used to train a static analysis false positive reduction model and help developers prioritize potential true positives. We will present AI-based code understanding approaches for vulnerability detection tasks in another paper.

A. Problem Statement

As observed previously [22], [23], an excessive number of false positives greatly hinders the utilization of static analyzers as developers get frustrated and do not trust the tools. To this end, we aim to device a method that can identify a subset of the reported issues that are more likely to be true positives, and *use it as a prioritization tool*. Developers may focus on the issues selected by the model first and then move to remaining issues with a higher false positive rate if they have time.

We treat the static analyzer as a black box and train a false positive reduction model solely based on the bug reports. Our goal is to achieve a balance between a large number of predicted positives and a high false positive reduction rate. We want developers to see more real bugs in the predicted positives comparing to all issues reported by the static analyzer.

B. Static Analysis Outputs/Data

a) Bug Trace description: Infer static analysis produces many output files. For our purposes, we are only interested in the bug trace text file, illustrated in Fig.1, from which we extract the features. The bug trace starts with the location where the static analyzer believes the error to have originated, and lists all the steps up to the line generating the error. Many of the bugs are inter-procedural, so the bug trace cuts across many files and functions. For each step in the flow, the trace contains 5 lines of code centered on the statement involved, the location of the file and function in the project, and a brief description of the step. At the top of the trace, the file and line of code where the bug occurred are mentioned along with the bug type (error type). There is also a short description of the bug. The bug trace is therefore a combination of different types of data like source code, natural language, numeric data like line numbers, and file paths.

b) Dataset description: As described in Sec.III-D, the original dataset has two types of negative examples, before-fix and after-fix. For these experiments, we built a dataset using the positive samples and the before-fix negative examples. We are not interested in the after-fix negative examples since these samples don't produce a bug trace. In every project, the number of negative labels is very large compared to the number of positive labels, as can be seen in Table VI.

C. Feature Engineering

Our primary assumption when coming up with features was that complex code is more likely to have bugs and/or is more likely to be classified as having bugs by a static analyzer, because it is highly probable that the developer failed to consider all possible implications of the code. Complex code is also more difficult for other developers to understand, increasing the chance of their introducing bugs.

One indication of complexity is the size of the bug trace. A long bug trace indicates that the control passes through many functions, files or packages. The location of the bug could also indicate the complexity of the code. The line number is indicative of the size of the file, and the column number indicates the length of the line of the code where the bug occurred. The depth of the line of code could indicate how entrenched the problematic code happens to be. Conditional statements cause many branches of execution to emerge and these can lead to convoluted and buggy code. One way to estimate the complexity is to count the number of times conditional statements occur and also the occurrences of OR/AND conditions. The error type is also a major feature that we consider, as well as the number of C keywords used. Table III lists our final set of features. We extract and normalize these features and save them in a features file.

TABLE III
FEATURES EXTRACTED FROM INFER BUG REPORT

Feature	Description	Feature	Description
error	Infer bug/issue type	error_line	line number of the error
error_line_len	length of error line	error_line_depth	indent for the error line text
average_error_line_depth	average indent of code lines	max_error_line_depth	max indent of code lines
error_pos_fun	position of error within function	average_code_line_length	average length of lines in flow
max_code_line_length	max length of lines in flow	length	the number of lines of code
code_line_count	the number of flow lines	alias_count	the number of address assignment lines
arithmetic_count	average operators / step	assignment_count	fraction of Assignment steps
call_count	fraction of `call` steps	cfile_count	the number of different .c files
for_count	the number of `for` loops in report	infinity_count	fraction of +00 steps
keywords_count	the number of C keywords	package_count	the number of different directories
question_count	fraction of '??' steps	return_count	average branches / step
size_calculating_count	average size calculations / step	parameter_count	fraction of parameter steps
offset_added	the number of "offset added"s in report		

D. Model Selection

We experimented with 13 well-known machine learning models: namely, Decision Trees, K-means, Random Forest, Extra-trees, Gradient Boosting, Ada Boost, XGBoost, Catboost, LightGBM, Linear Classifiers with Stochastic Gradient Descent, Gaussian Naive Bayes, Multinomial Naive Bayes, and Complement Naive Bayes. We ranked them based on both their AUC and F1 scores and selected the four best models. These were based on an ensemble of decision trees, Random Forest and Extra Trees for bagging methods, LightGBM, and Catboost for boosting.

Random Forest is made of many weak learners (single decision trees), which are fed with random samples of the data and trained independently using a random sample of the features. Each inner decision tree is grown by using the features which offer the best split at every step. The randomness, which makes every single tree different from the rest of the forest, associated with a high number of learners, makes the model quite robust against overfitting. A slightly different variation of Random Forest is Extra-Trees, also known as Extremely Randomized Trees. The only difference is how the data is sampled to create the input and how the splits are chosen randomly making the forest more diversified. Differently from bagging methods, where learners are trained independently, with gradient boosting methods each tree improves over the predictions made by the previous ones. Boosting techniques are known to work successfully with imbalanced data, but might suffer more from overfitting - to mitigate this effect typically many trees are used together. LightGBM and Catboost are different frameworks to implement this kind of ensemble: the former creates an imbalanced tree while the latter creates a balanced one.

E. Evaluation Metrics

In order to evaluate the different models, because of the imbalance in the dataset, we used the Area Under the Curve (AUC score, Fig. 5), a threshold-invariant metric that visualizes the trade-off when we want to reduce the false positive rate while maintaining a good true positive rate. Since the main task is to reduce the number of False Positives, we calculate the percentage reduction in False Positives on the test set. Relying too much on this metric can bias towards models, which make very few accurate predictions. To make sure this is not the case, we also calculate the total percentage of True Positives which are predicted by the model. An ideal model would have a very high AUC Score, low False Positive rate, and high True Positives rate: to choose the threshold for the AUC, we select that point which minimizes the distance from the top-left corner (all true positives and no false positives). Once this point is chosen we also present F1-score as the average of each class F1-score since our goal is to reduce the number of false positives while preserving the real ones.

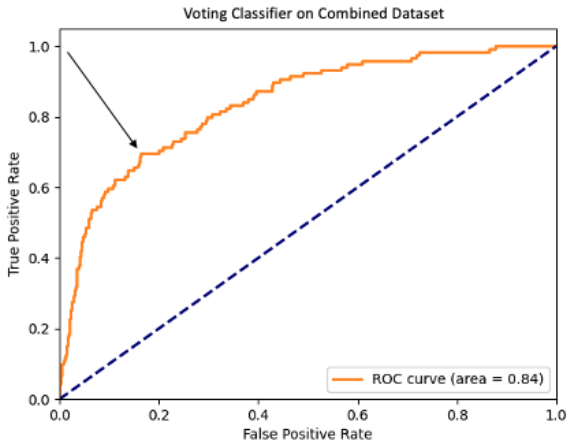

Fig. 5. The ROC curve tracks the performance at all classification thresholds, while the Area under the Curve (AUC) provides an aggregation of performance across all possible classification thresholds. The point on the curve that minimizes the distance from the top-left corner is the one which provides the best compromise between false positive rate and true positive rate.

F. Voting

Real-world datasets present a high imbalance between real bugs and false positives. Also, the projects used to derived the datasets proposed in this work vary in size yielding different dataset sizes. Therefore, it's not easy to choose the model which does the best on all the datasets. While on a specific dataset a model can perform greatly, it could work poorly in another: to mitigate such a problem we applied a soft-voting strategy, which combines the scores of each classifier, which should guarantee a more stable behavior across datasets.

V. EVALUATION

In this section, we present the result of D2A dataset generation and label evaluation. In addition, we show the evaluation

results of the AI-based static analysis false positive reduction as a use case to demonstrate how D2A dataset can be helpful.

A. Dataset Generation Results

1) Dataset Statistics: The dataset generation pipeline is written in python and runs on a POWER8 cluster, where each node has 160 CPU cores and 512GB RAM. We analyzed 6 open-source programs (namely, `OpenSSL`, `FFmpeg`, `httpd`, `NGINX`, `libtiff`, and `libav`) and generated the initial version of the D2A dataset. In particular, Infer can detect more than 150 types of issues in C/C++/Objective-C/Java programs [47]. However, some issues are not ready for production and thus disabled by default. In the pipeline, we additionally enabled all issue types related to buffer overflows, integer overflows, and memory/resource leaks, even some of them may not be production-ready.

Table IV summarizes the dataset generation results. The column *CMA Version Pairs* shows the number of bug-fixing commits selected by the commit message analyzer (Sec. III-B). For each selected commit, we run Infer on both the before-commit and after-commit versions. We drop a commit if Infer failed to analyze either the before-commit version or the after-commit version. Column *Infer* shows the number of commits or version pairs Infer successfully analyzed. For auto-labeler examples (Sec. III-C), column *Issues Reported* and *unique auto-labeler examples - all* shows the number issues Infer detected in the before-commit versions before and after deduplication, which will be labeled as positives and negatives as shown in column *Positives* and *Negatives*. For after-fix examples (Sec. III-D), column *Negatives* shows the number of examples generated based on the auto-labeler positive examples. In total, we processed 11,846 consecutive versions pairs. Based on the results, we generated 1,295,623 unique auto-labeler examples and 18,653 unique after-fix examples.

2) Manual Label Validation: As there is no ground truth, to evaluate the label quality we randomly selected 57 examples (41 positives, 16 negatives) with a focus on positives. We gave more weights to positive examples because they are more important for our purpose. As mentioned in Sec. IV-A, labeling a non-buggy example as buggy is against the goal of false positive reduction. But it's acceptable if we miss some of the real bugs. If we select examples according to the overall dataset distribution, we will have too few positive examples. Each example was independently reviewed by 2 reviewers.

Table V shows the label validation results. On this biased sample set, the accuracy with and without the auto-labeler is 53% and 35% respectively. Note the accuracy on an unbiased sample set is expected to be higher as there should be more negative examples. Take the `OpenSSL` study in Sec. II-B as an example. Without auto-labeler, the accuracy was only 7.8% on the set of 166 security-related examples.

B. False Positive Reduction Results

1) Dataset: To facilitate reproducibility, we defined and plan to release a split for each project. In particular, we drop bug types without any positive examples and split each project's data into *train:dev:test* sets (80:10:10) while maintaining the distribution of bug types. We use the same split in this experiment. The model will be trained on the *train* + *dev* sets and tested on the *test* set.

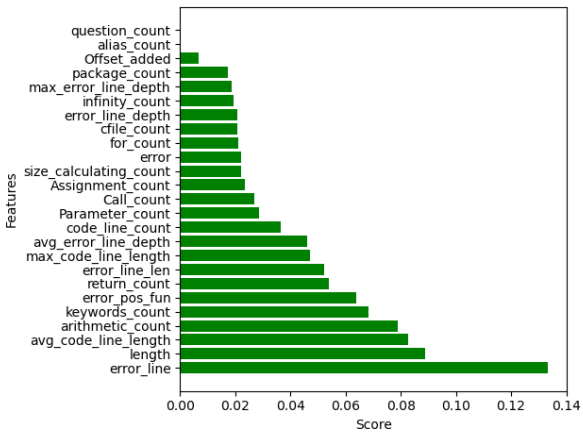

Fig. 6. Feature Importance of Random Forest algorithm trained on OpenSSL

We observed that some `FFmpeg` and `libav` examples are quite similar as `libav` was forked from `FFmpeg` [48]. We dropped `FFmpeg` examples so that the all-data combined experiment would be fair. `FFmpeg` examples are more imbalanced compared to `libav` and we leave it for future work.

Although we collect examples generated for many bug types that are not production-ready and are disabled by default in Infer, in this experiment, we consider just the 18 *security-related bug types* that are enabled by default. Table VI shows the statistics of the data used in the experiment.

2) Feature Importance: We used feature importance ranking when selecting the final set of 25 features. Figure 6 shows the features and their relative importance for one of the models. Two features that were important for many models are the line number of the error in the file and the number of lines of code in the bug report, perhaps suggesting that large files and complex bug reports distinguish real errors.

3) FP Reduction Model: We trained Random Forest using 1000 estimators, Extra Trees with 500. For the Boosting algorithms, we used 500 estimators, learning rate 0.03, importance type *gain* for the LGBM classifier, and the same number of estimators for Catboost. We define False Positive Reduction Rate (FPRR) = $(\text{FP}_{\text{infer}} - \text{FP}_{\text{predict}})/\text{FP}_{\text{infer}} * 100$. Because all issues are positive according to Infer, FP_{infer} is just the number of negative examples.

As shown in Table VII, all models can effectively reduce the false positives for each project. In most cases, the FPRR is above 70% for every model, without penalizing too much the reduction of true bugs. As expected, it's hard to find one best model across all the projects. However, it's encouraging that the voting can outperform the single models in the `combined` experiment, letting us believe that the more data we have, the better the voting system could perform.

VI. Related Work

Datasets for AI-based Vulnerability Detection. Juliet [21], Choi et.al [34], and S-babi [14] are synthetic datasets. They are

TABLE IV
DATASET GENERATION RESULTS

Project	Version Pairs		Issues Reported	Unique Auto-labeler Examples			Unique After-fix Examples
	CMA	Infer		All	Negatives	Positives	Negatives
OpenSSL	3,011	2,643	42,151,595	351,170	343,148	8,022	8,022
FFmpeg	5,932	4,930	215,662,372	659,717	654,891	4,826	4,826
httpd	1,168	542	1,681,692	12,692	12,475	217	217
NGINX	785	635	3,283,202	18,366	17,945	421	421
libtiff	144	144	525,360	12,649	12,096	553	553
libav	3,407	2,952	86,069,532	241,029	236,415	4,614	4,614
Total	**14,447**	**11,846**	**349,373,753**	**1,295,623**	**1,276,970**	**18,653**	**18,653**

- **CMA:** The number of bug-fixing commits identified by the commit message analyzer.
- **Infer:** The number of version pairs successfully analyzed by Infer.
- **Issues Reported:** The number of issues detected in the before-commit versions before deduplication.

TABLE V
AUTO-LABELER MANUAL VALIDATION RESULTS

	Positives			Negatives			All		
	#	A	D	#	A	D	#	A	D
BUFFER_OVERRUN_L1	2	0	2	1	1	0	3	1	2
BUFFER_OVERRUN_L2	3	1	2	1	1	0	4	2	2
BUFFER_OVERRUN_L3	6	1	5	4	4	0	10	5	5
BUFFER_OVERRUN_S2	0	0	0	1	0	1	1	0	1
INTEGER_OVERFLOW_L1	3	2	1	1	1	0	4	3	1
INTEGER_OVERFLOW_L2	13	6	7	3	3	0	16	9	7
INTEGER_OVERFLOW_R2	1	1	0	0	0	0	0	1	0
MEMORY_LEAK	1	1	0	1	1	0	2	2	0
NULL_DEREFERENCE	2	1	1	1	0	1	3	1	2
RESOURCE_LEAK	1	1	0	1	1	0	2	2	0
UNINITIALIZED_VALUE	9	3	6	1	1	0	10	4	6
USE_AFTER_FREE	0	0	0	1	1	0	1	1	0
ALL	41 100%	17 41%	24 59%	16 100%	13 81%	3 19%	57 100%	30 53%	27 47%

- **#:** the issue count; **A/D:** manual review agrees/disagrees with the auto-labeler label

TABLE VI
PRODUCTION-READY SECURITY RELATED ERROR TYPES FILTERING

	All Errors			Prod-ready Sec Errs		
	Negatives	Positives	N:P	Negatives	Positives	N:P
OpenSSL	341,625	7,916	43:1	27,227	797	34:1
libav	235,369	4,585	51:1	14,954	280	53:1
NGINX	1,7829	417	43:1	1,446	36	40:1
libtiff	11,720	552	21:1	1,185	27	44:1
httpd	11,511	208	55:1	174	11	16:1

generated based on predefined patterns and cannot represent real-world program behaviors. Draper [12], Devign [15] and CDG [13] were generated from real-world programs. However, as discussed in Sec. II, they suffer from labeling or source limitations. In fact, lacking good real-world datasets has become a major barrier for this field [35]. D2A is automated and scales well on large real-world programs. It can produce more bug related information. We believe it can help to bridge the gap.

AI-based Static Analysis FP Reduction. Static analysis is known to produce a lot of false positives. To suppress them, several machine learning based approaches [25], [26], [27], [28], [10], [29], [30], [31], [49], [32], [33] have been proposed. Because they either target different languages or different static analyzers, they are not directly applicable. Inspired by their approaches, we designed and implemented a false positive reduction model for Infer as a use case for the D2A dataset.

VII. CONCLUSION

In this paper, we propose D2A, a novel approach to label static analysis issues based on differential analysis and build a labeled dataset from real-world programs for AI-based vulnerability detection methods. We ran D2A on 6 large programs and generated a labeled dataset of more than 1.3M examples with detailed bug related information obtained from the interprocedural static analysis, the code base, and the commit history. By manually validating randomly selected samples, we show D2A significantly improves the label quality compared to static analysis alone. We train a static analysis false positive reduction model as a use case for the D2A dataset, which can effectively suppress false positives and help developers prioritize and investigate potential true positives first.

TABLE VII
FALSE POSITIVE REDUCTION RESULTS

	Model	GP	P	TP	GN	N	TN	F1	FPRR	AUC
OpenSSL	cb	81	858	62	2711	1934	1915	0.48	70.6%	0.79
	lgbm	81	827	65	2711	1965	1949	0.49	71.9%	0.82
	rf	81	591	59	2711	2201	2179	0.53	80.4%	0.83
	etc	81	616	52	2711	2176	2147	0.51	79.2%	0.78
	voting	81	506	58	2711	2286	2263	0.55	83.5%	0.83
libav	cb	28	256	22	1495	1266	1260	0.53	84.3%	0.89
	lgbm	28	220	21	1495	1303	1296	0.55	86.7%	0.91
	rf	28	287	22	1495	1236	1230	0.52	82.3%	0.87
	etc	28	54	13	1495	1469	1454	0.65	97.3%	0.70
	voting	28	254	21	1495	1269	1262	0.53	84.4%	0.89
NGINX	cb	5	27	3	145	123	121	0.55	83.4%	0.85
	lgbm	5	47	4	145	103	102	0.49	70.3%	0.86
	rf	5	46	3	145	104	102	0.47	70.3%	0.75
	etc	5	60	3	145	90	88	0.42	60.7%	0.67
	voting	5	54	4	145	96	95	0.46	65.5%	0.78
libtiff	cb	3	17	2	118	104	103	0.56	87.3%	0.92
	lgbm	3	5	1	118	116	114	0.61	96.6%	0.72
	rf	3	7	2	118	114	113	0.69	95.8%	0.98
	etc	3	8	2	118	113	112	0.67	94.9%	0.97
	voting	3	7	2	118	114	113	0.69	95.8%	0.97
httpd	cb	2	5	1	17	14	13	0.56	76.5%	0.88
	lgbm	2	3	1	17	16	15	0.65	88.2%	0.94
	rf	2	9	1	17	10	9	0.42	52.9%	0.77
	etc	2	6	1	17	13	12	0.53	70.6%	0.85
	voting	2	6	1	17	13	12	0.53	70.6%	0.85
combined	cb	119	1403	95	4486	3202	3178	0.48	70.8%	0.82
	lgbm	119	1274	93	4486	3331	3305	0.49	73.7%	0.83
	rf	119	1063	86	4486	3542	3509	0.50	78.2%	0.84
	etc	119	1053	74	4486	3552	3507	0.50	78.2%	0.74
	voting	119	814	82	4486	3791	3754	0.54	83.7%	0.84

- The released dataset split defines train/dev/test for each project. For combined, its train/dev/test sets are the union of corresponding sets of all project
- The models are trained on train + dev sets and tested on the test set.
- **GP/P/TP** Ground-truth/Predicted/True Positives; **GN/N/TN** defined similarly.
- **FPRR**: False Positive Reduction Rate = **(GN - FP) / GN * 100**
- **cb**: Catboost, **lgbm**: LightGBM, **rf**: Random Forest, **etc**: Extra-Trees

REFERENCES

[1] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in *Proceedings of the 28th International Conference on Software Engineering.*, 2006.

[2] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design space of bug fixes and how developers navigate it," *IEEE Transactions on Software Engineering*, vol. 41, no. 1, pp. 65–81, 2015.

[3] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008.

[4] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Using findbugs on production software," in *OOPSLA'07*, 2007.

[5] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *2015 IEEE Symposium on Security and Privacy*, 2015.

[6] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang, "Smoke: Scalable path-sensitive memory leak detection for millions of lines of code," in *ICSE'19*, 2019.

[7] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th Conference on USENIX Security Symposium*, 2005.

[8] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the world wide web from vulnerable javascript," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA'11*, 2011.

[9] U. Yüksel and H. Sözer, "Automated classification of static code analysis alerts: A case study," in *ICSM'13*, 2013.

[10] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin, "ALETHEIA: Improving the Usability of Static Security Analysis," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 762—-774.

[11] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter, "Learning a classifier for false positive error reports emitted by static code analysis tools," in *MAPL'17*, 2017.

[12] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, "Automated vulnerability detection in source code using deep representation learning," in *ICMLA'18*, 2018.

[13] 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," in *25th Annual Network and Distributed System Security Symposium, NDSS'18*, 2018.

[14] C. D. Sestili, W. S. Snavely, and N. M. VanHoudnos, "Towards security defect prediction with AI," *CoRR*, vol. abs/1808.09897, 2018. [Online]. Available: http://arxiv.org/abs/1808.09897

[15] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *NeurIPS'19*, 2019.

[16] L. Buratti, S. Pujar, M. Bornea, J. S. McCarley, Y. Zheng, G. Rossiello, A. Morari, J. Laredo, V. Thost, Y. Zhuang, and G. Domeniconi, "Exploring software naturalness through neural language models," *CoRR*, vol. abs/2006.12641, 2020.

[17] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, and A. Morari, "Learning to map source code to software vulnerability using code-as-a-graph," *CoRR*, vol. abs/2006.08614, 2020.

[18] R. Paletov, P. Tsankov, V. Raychev, and M. Vechev, "Inferring crypto api rules from code changes," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018, 2018, pp. 450—464.

[19] NIST, "National vulnerability database," https://nvd.nist.gov/.

[20] MITRE, "Common vulnerabilities and exposures," https://cve.mitre.org/index.html.

[21] NIST, "Juliet test suite for c/c++ version 1.3," https://samate.nist.gov/SRD/testsuite.php.

[22] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *ICSE'13*, 2013, p. 672–681.

[23] T. B. Muske, A. Baid, and T. Sanas, "Review efforts reduction by partitioning of static analysis warnings," in *13th International Working Conference on Source Code Analysis and Manipulation*, 2013.

[24] T. Muske and A. Serebrenik, "Survey of approaches for handling static analysis alarms," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2016, pp. 157–166.

[25] T. Kremenek and D. R. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," in *Static Analysis, 10th International Symposium, SAS 2003*, R. Cousot, Ed., 2003.

[26] Y. Jung, J. Kim, J. Shin, and K. Yi, "Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis," in *Proceedings of the 12th International Conference on Static Analysis*, ser. SAS'05, 2005, p. 203–217.

[27] U. Yüksel and H. Sözer, "Automated classification of static code analysis alerts: A case study," in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 532–535.

[28] Q. Hanam, L. Tan, R. Holmes, and P. Lam, "Finding patterns in static analysis alerts: Improving actionable alert ranking," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014, 2014, p. 152–161.

[29] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter, "Learning a classifier for false positive error reports emitted by static code analysis tools," in *MAPL'17*, 2017, p. 35–42.

[30] X. Zhang, X. Si, and M. Naik, "Combining the logical and the probabilistic in program analysis," in *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2017, 2017, p. 27–34.

[31] Z. P. Reynolds, A. B. Jayanth, U. Koc, A. A. Porter, R. R. Raje, and J. H. Hill, "Identifying and documenting false positive patterns generated by static code analysis tools," in *4th International Workshop on Software Engineering Research and Industrial Practice*, 2017.

[32] M. Raghothaman, S. Kulkarni, K. Heo, and M. Naik, "User-guided program reasoning using bayesian inference," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018, 2018, p. 722–735.

[33] U. Koc, S. Wei, J. S. Foster, M. Carpuat, and A. A. Porter, "An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 288–299.

[34] M.-J. Choi, S. Jeong, H. Oh, and J. Choo, "End-to-end prediction of buffer overruns from raw source code via neural memory networks," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, ser. IJCAI'17, 2017.

[35] G. Lin, S. Wen, Q. L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1825–1848, 2020.

[36] Y. Sui, X. Cheng, G. Zhang, and H. Wang, "Flow2vec: Value-flow-based precise code embedding," *OOPSLA*, 2020.

[37] LLVM, "The clang static analyzer," https://clang-analyzer.llvm.org/.

[38] Facebook, "Infer static analyzer," https://fbinfer.com/.

[39] D. Chandrasekaran and V. Mago, "Evolution of semantic similarity - A survey," *CoRR*, vol. abs/2004.13820, 2020. [Online]. Available: https://arxiv.org/abs/2004.13820

[40] M. Sahami and T. D. Heilman, "A web-based kernel function for measuring the similarity of short text snippets," in *WWW '06*, 2006.

[41] Cppcheck-team, "Cppcheck," http://cppcheck.sourceforge.net/.

[42] D. A. Wheeler, "Flawfinder," https://dwheeler.com/flawfinder/.

[43] Facebook, "infer reportdiff," https://fbinfer.com/docs/man-infer-reportdiff.

[44] J. Villard, "Infer is not deterministic, infer issue #1110," https://github.com/facebook/infer/issues/1110.

[45] Y. Zheng, "Parallelism gives inconsistent results, infer issue #1239," https://github.com/facebook/infer/issues/1239.

[46] Clang, "Clang tooling," https://clang.llvm.org/docs/Tooling.html.

[47] Infer, "Infer issue types," https://github.com/facebook/infer/blob/ea4f7cf/infer/man/man1/infer.txt#L370.

[48] Wiki, "Libav," https://en.wikipedia.org/wiki/Libav#Fork_from_FFmpeg.

[49] W. S. Lori Flynn, Zachary Kurtz, "Test suites as a source of training data for static analysis alert classifiers," *SEI Blog*, 2018. [Online]. Available: https://insights.sei.cmu.edu/sei_blog/2018/04/static-analysis-alert-test-suites-as-a-source-of-training-data-for-alert-classifiers.html