



Analyzing source code vulnerabilities in the D2A dataset with ML ensembles and C-BERT

Saurabh Pujar¹ · Yunhui Zheng¹ · Luca Buratti¹ · Burn Lewis¹ ·
Yunchung Chen² · Jim Laredo¹ · Alessandro Morari¹ · Edward Epstein¹ ·
Tsunghan Lin² · Bo Yang³ · Zhong Su³

Accepted: 28 September 2023 / Published online: 22 February 2024
© The Author(s) 2024

Abstract

Static analysis tools are widely used for vulnerability detection as they can analyze 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 learn from programming language data opens new possibilities of reducing false positives 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 Differential Dataset Analysis or D2A, a differential analysis based approach to label issues reported by static analysis tools. The dataset built with this approach is called the D2A dataset. 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. We then train both classic machine learning models and deep learning models for vulnerability identification using the D2A dataset. 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. To facilitate future research and contribute to the community, we make the dataset generation pipeline and the dataset publicly available. We have also created a leaderboard based on the D2A dataset, which has already attracted attention and participation from the community.

Keywords Dataset · Vulnerability detection · AI · Bert · Leaderboard · D2A

Communicated by: Sigrid Eldh, Davide Falessi and Burak Turhan

This article belongs to the Topical Collection: *Special Issue on Software Engineering in Practice*.

✉ Saurabh Pujar
saurabh.pujar@ibm.com

Extended author information available on the last page of the article

1 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 (LaToza et al. 2006; Murphy-Hill et al. 2015). 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 (Ayewah et al. 2008, 2007; Yamaguchi et al. 2015; Fan et al. 2019). 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 (Livshits and Lam 2005; Guarnieri et al. 2011).

Besides these classic usage scenarios, driven by the needs of recent AI research on source code understanding and vulnerability detection tasks (Yüksel and Sözer 2013a; Tripp et al. 2014; Koc et al. 2017b; Russell et al. 2018; Li et al. 2018; Sestili et al. 2018; Zhou et al. 2019; Buratti et al. 2020; Suneja et al. 2020; Paletov et al. 2018), static analysis techniques have also been used to generate labeled datasets for model training (Russell et al. 2018). 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 Li et al. 2018) are derived from confirmed bugs in NVD (NIST 2023a). 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 (NIST 2023b) and S-babi (Sestili et al. 2018) 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 (Russell et al. 2018). Code diff based methods (Zhou et al. 2019) 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. (2018) applied the Clang static analyzer (LLVM 2018), Cppcheck (Cppcheck-team 2023), and Flawfinder (Wheeler 2023) 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 use of approximation heuristics 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 (Tripp et al. 2014). 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. False positives (FP) are issues reported by static analyzers which can be ignored as they are not real bugs or vulnerabilities. Analogously, true positives (TP) are issues reported by Static Analysis tools which are genuine bugs or vulnerabilities.

These false positives greatly hinder the utilization of static analysis tools like Infer (Facebook 2023a) as it is inefficient for developers to go through a long list of reported issues to find only a few true positives (Johnson et al. 2013; Muske et al. 2013).

The analysis required to separate false positives from true positives is time consuming and often requires expert developers. Suppressing false positives is a great way to improve developer productivity, increase adoption of static analyzers while at the same time improving software security. To suppress false positives, various methods have been proposed (as summarized in Muske and Serebrenik 2016). Among them, machine learning based approaches (Kremenek and Engler 2003; Jung et al. 2005; Yüksel and Sözer 2013b; Hanam et al. 2014; Tripp et al. 2014; Koc et al. 2017a; Zhang et al. 2017; Reynolds et al. 2017; Raghothaman et al. 2018; Koc et al. 2019) focus on learning to identify the patterns of false positives from examples. However, training such machine learning models requires good labeled datasets. Most existing works manually generate such datasets by reviewing the analyzed code and the output of the static analysis, also called bug report.

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 the challenges of creating a large labelled vulnerability dataset, we propose D2A, an approach to automatically label issues as true or false positives. This approach is based on differential analysis and it generates labels to distinguish the issues that are *more likely to be true positives* from the ones that are *more likely to be false positives*. Our goal is to generate a large labelled dataset that can be used to train machine learning models for (1) static analyzer false positive prediction, and (2) vulnerability detection tasks. We demonstrate how such a dataset containing bug reports and code snippets can be helpful for the false positive prediction task. With the D2A Leaderboard (Section 5) we show how different pieces of information from the dataset can be used, either individually or combined, for the broader task of vulnerability detection.

The differential analysis approach is described as follows: (i) We select projects with commit histories and focus on commits which appear to be bug fixing code changes, as opposed to code refactoring or new features. (ii) Instead of predicting labels based on commit messages, we run static analysis on the versions before and after such commits. (iii) If any issue detected in a before-commit version disappears in the corresponding after-commit version, it is *very likely to constitute a real bug* 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 consider these issues *not very likely to be real bugs* because they were never fixed. After the differential analysis process, 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 the labeling mechanism *auto-labeler*.

We ran the Infer static analyzer on thousands of selected consecutive version pairs from OpenSSL, FFmpeg, libav, httpd, NGINX and libtiff, well known open source

projects. 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% observed in the manual case study without our technique (Section 2.2.1) to 53% based on the manual label validation of randomly selected samples in Table 4.

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. In particular, inspired by Tripp et al. (2014), we defined features from static analysis outputs and source code and trained a static analysis false positive prediction 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 that 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 the 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 both classic machine learning models and a deep learning model for the static program analysis false positive reduction task, which can effectively help developers prioritize issues that are more likely to be real bugs.
- We created a leaderboard based on the D2A dataset and made it public. It has already attracted community attention and participation. Using the leaderboard, researchers can compare their model performance on D2A with other models. The leaderboard can be found at <https://ibm.github.io/D2A>.
- To facilitate future research, we make the D2A dataset and its generation pipeline publicly available at <https://github.com/ibm/D2A>.

In Section 2 we discuss the motivation for the paper in more depth. We describe the generation of the D2A Dataset in Section 3. In Section 4 we define the false positive prediction problem and how we plan to use the D2A Dataset to tackle this problem with ML and actually improve static analyzer output. To solve the false positive prediction problem we try two approaches, feature engineering with Classical ML models (Section 4.3) and Deep Learning (Section 4.4). In order to encourage community participation, we developed a leaderboard which is discussed in Section 5. The results for Dataset Generation (Section 3.6), FP Prediction (Section 4.5) and Leaderboard (Section 5.4) are discussed in their respective sections. Related Work in different technologies and domains is covered in Section 6. We end with the Conclusion in Section 8 and Threats to Validity in Section 7.

2 Motivation

This section, describes the motivations behind this work. First, we show how existing datasets to train ML models for vulnerability detection are not sufficient, and second, we show the manual case study performed as a basis for the approach followed in this paper.

2.1 Existing Datasets for AI on Vulnerability Detection Task

Training ML models for code understanding and vulnerability detection requires large high-quality datasets. Indeed, according to a recent survey (Lin et al. 2020), the lack of large 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 1 summarizes the characteristics of a few popular publicly available software vulnerability datasets. We evaluate these datasets and highlight the contributions D2A can make.

Juliet (NIST 2023b), Choi et al. (2017), and S-babi (Sestili et al. 2018) 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 (Choi et al. 2017).

The examples in Draper (Russell et al. 2018) 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 generated by aggregating static analysis results. Draper does not provide details like bug locations or traces. For real-world programs, Draper does not 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.

Table 1 Publicly available datasets for AI on C/C++ vulnerability detection

Dataset	Type	Level	WDR	Bug Type	Bug Line	Bug Trace	CT	CE	G.A.	Labelling method
Juliet	synthetic	function	✓	✓	✓	✗	–	✓	–	predefined pattern
S-Babi	synthetic	function	✓	✓	✓	✗	–	✓	✓	predefined pattern
Choi et.al	synthetic	function	✓	✓	✓	✗	–	✓	✓	predefined pattern
Draper	mixed	function	✓	✓	✗	✗	✗	✗	✗	static analysis
Devign	real-world	function	✗	✗	✗	✗	✗	✗	✗	manual + code diff
CDG	real-world	slice	✓	✗	✗	✗	✗	✗	✓	NVD + code diff
D2A	real-world	trace	✓	✓	✓	✓	✓	✓	✓	differential analysis

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

WDR: Whole Dataset Released.

CT: Traceability to the code base, i.e. the commit version and location.

CE: Compilable example. i.e. possibility of compiling the file with bug with provided compilation arguments.

GA: Dataset generation implementation available

The Devign (Zhou et al. 2019) 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 (Li et al. 2018) 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 (Lin et al. 2020; Zhou et al. 2019; Li et al. 2018). 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) (Sui et al. 2020). 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.

2.2 Manual Review and False Positive Reduction

We perform a manual review of the static analyzer output in order to evaluate their quality. This review motivates our approach and in particular the use of ML algorithms to solve this problem.

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. We include Infer bug types based on the vulnerabilities reported as CWE (MITRETop25 (MITRETop25)) like `NULL_DEREFERENCE` (CWE476 (CWE476)), `UNINITIALIZED_VALUE` (CWE457 (2023)) and `RESOURCE_LEAK` (CWE400 (2023)).

The manual review allows us 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. This motivates the use of a ML approach that can improve the static analyzer output by reducing the number of false positives.

2.2.1 Manual Case Study

Since we are interested in large C/C++ programs, we require that 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 (LLVM 2018) and Infer (Facebook 2023a) are two state-of-the-art static analyzers that satisfy our needs. However, the Clang Static Analyzer does not support cross translation unit analysis such that the inter-procedural analysis may be incomplete. Therefore, we choose Infer in our experiments. We chose OpenSSL as a benchmark because of its importance in the open-source security ecosystem and its long commit history. We use OpenSSL version 7f0a8dc 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 2. 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 issues where the value written to a variable is never used. Since such issues are not security vulnerabilities and are in fact often intended to overwrite data such as passwords to avoid leakage of sensitive data, their removal might create a security issue so were excluded 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. Figure 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. Figure 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. If the two reviewers disagree, then a third reviewer is assigned to act as a tie breaker. This process was very time consuming and challenging, especially when reviewing a complex program with cryptography involved.

As shown in Table 2, out of 166 security vulnerability related issues, we confirmed that **13 (7.8%)** issues are true positives and **92.2%** are false positives.

2.2.2 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 (Yüksel and Sözer 2013b; Tripp et al. 2014), 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 observation from Tripp et al. (2014) and Du et al. (2019) that issues with complex code are more likely to be false positives, we considered features in bug reports and source code that may reflect code 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

Table 2 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

326 DEAD_STORE issues were excluded from manual review

```

01. crypto/initthread.c:385: error: NULL_DEREFERENCE
02. pointer 'gtr' last assigned on line 382 could be null and is dereferenced
03. at line 385, column 53.
04. Showing all 5 steps of the trace
05.
06. crypto/initthread.c:377:1: start of procedure init_thread_deregister()
07. 375.
08. 376. #ifndef FIPS_MODE
09. 377. > static int init_thread_deregister(void *index, int all)
10. 378. {
11. 379.     GLOBAL_TEVENT_REGISTER *gtr;
12.
13. crypto/initthread.c:382:5:
14. 380.     int i;
15. 381.
16. 382. >     gtr = get_global_tevent_register();
17. 383.     if (!all)
18. 384.         CRYPTO_THREAD_write_lock(gtr->lock);
19.
20. ...
21.
22. crypto/initthread.c:385:17:
23. 383.     if (!all)
24. 384.         CRYPTO_THREAD_write_lock(gtr->lock);
25. 385. >     for (i = 0; i < sk_THREAD_EVENT_HANDLER_PTR_num(gtr->skhands); i++) {
26. 386.         THREAD_EVENT_HANDLER **hands
27. 387.             = sk_THREAD_EVENT_HANDLER_PTR_value(gtr->skhands, i);

```

Fig. 1 Infer bug report example

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.

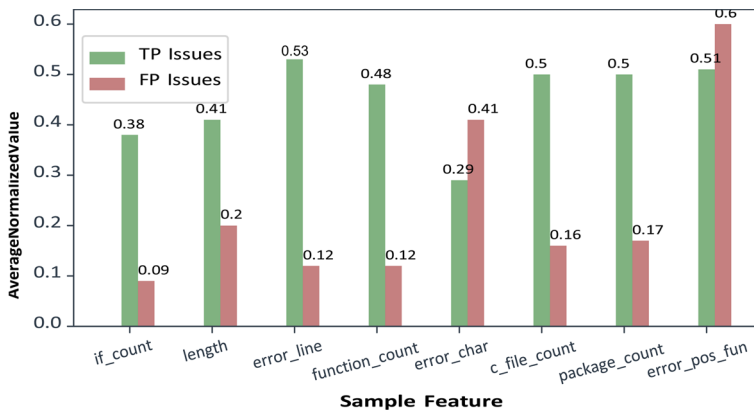


Fig. 2 Feature Exploration. We experiment with a few features that may reflect the complexity of the issues. After normalization, the averages of these 8 feature values for true positives and false positives are significantly different, which suggests a classifier may achieve good performance

3 D2A 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.

3.1 Overview

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

As the pre-processing step, based on the commit messages only, the Commit Message Analyzer (Section 3.2) 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 (Section 3.3) 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.

3.2 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 (NIST 2023a), CMA can learn the language used in commits that fix vulnerabilities. A pure ML approach would require a larger dataset than NVD, hence the CMA uses a hybrid approach. This approach combines semantic similarity-based methods (Chandrasekaran and Mago 2020) and snippet samples-based methods (Sahami and Heilman 2006) to identify relevant commit messages and their associated commits. From the commit messages, we first filter out word-level noises such as code snippets, meaningless tokens, names, email addresses, links, etc. Then, we identify commits that successfully fixed the vulnerabilities in the NVD database. We extract the commit messages that describe the vulnerabilities as well as the fixes. Then, we trained a model to learn the common keywords that can be used as indicators to help recognize bug-fixing commits. Based on the semantic



Fig. 3 The overview of D2A dataset generation pipeline

distribution of the vulnerable mentions, CMA identifies the category of the vulnerability and ranks commits based on confidence scores, with a threshold set to 0.8.

Like any ML based technique, the quality of CMA model prediction is bounded by the dataset, which in this case, refers to the NVD and commit messages. It's possible some bugs are fixed silently and the commits don't have meaningful messages. The CMA will find it hard to identify such fixes since the messages are uninformative and don't contain any useful signal that the model can exploit. We try to mitigate the number of uninformative commits by excluding messages that are too short, however, this approach will not remove all uninformative messages. The CMA will still produce useful results because NVD label quality is relatively high and most of the commit messages in our experience are informative.

3.3 Auto-labeler

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 (Cppcheck-team 2023), Flawfinder (Wheeler 2023), Clang Static Analyzer (LLVM 2018), and Infer (Facebook 2023a). We chose Infer because it's arguably the state-of-the-art inter-procedural static analysis engine that can detect non-trivial security related bug types. Infer's inter-procedural static analysis, as its name indicates, considers the interactions between different procedures or methods, offering a more comprehensive view of a program's behavior. It is enhanced by the use of separation logic (O'Hearn et al. 2001) and bi-abduction (Calcagno et al. 2011). Separation logic is about the ability to reason on disjoint parts of memory, simplifying the analysis of intricate data structures, particularly those involving pointers. Bi-abduction, an inference rule in separation logic, further aids in deducing the best explanation for the state changes in memory across procedures by breaking the large analysis of a large program in small independent analyses of its procedures. This gives Infer the ability to scale independently of the size of the analyzed code.

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 (Facebook 2023b) 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 (Villard, 2023). And the non-determinism occurs more frequently when enabling parallelization (Zheng 2023). 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 parallelized the analysis to process more than a thousand version pairs simultaneously in a cluster. The improvement in performance depends on the availability of computation resources like CPUs and RAMs.

Deduplicate Infer Issues Based on Commit History After identifying fixed and pre-existing (not-fixed) issues in each version pair, we deduplicate the issues that occur in multiple version pairs. In particular, we compute a checksum of the bug report after removing location-related contents (e.g, file names, line numbers, etc) and use it as the id for deduplication. The reason we remove location-related contents is that another commit may have made changes earlier in the file, changing the line numbers in the bug report. The expected time-based sequence for a real bug is for it to appear in a number of commit pairs, disappear from the after-commit of a later pair, and never reappear. We sort all occurrences of the fixed issues based on the author date of the commits and apply the following heuristics to eliminate any false positives.

- *Fixed-then-unfixed issues*: Because of some randomness in the way Infer selects code to analyze, it may accidentally omit a bug from the after-commit version, falsely suggesting that it was fixed by that commit. If a fixed issue reappears in a *later* commit pair, we assume that it is a false positive caused by an error in the static analyzer. We change the label of such cases and mark them as *negative*. (Note that this sequence could happen if the suspect code was removed and then later re-introduced.)
- *Untouched issues*: For each fixed issue we check which parts of the code are patched by the commit. If the commit code diff does not 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 static analyzer error. We mark such cases *negative* 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.

After-fix Examples Due to the nature of the vulnerability detection task, 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 produces another kind of auto-labeler negative examples, which we call *after-fix examples*.

In particular, the auto-labeler works on two consecutive versions that are *before* and *after* a commit. Given a particular issue detected in the *before-commit* version does not occur in the *after-commit* version, the commit may have changed something that fixed the issue. Therefore, given an issue that detected in the before version but disappeared in the after version, if we know the commit changed some code in the trace reported by static analyzer, extracting the corresponding parts from the *after-commit* version may reveal how the issue is fixed and thus gives us a negative example. More details and a concrete example could be found at <https://github.com/IBM/D2A/blob/main/README.md#sample-types>.

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.

3.4 Infer's Bug Trace

Infer static analysis produces many output files. For our purposes, the bug trace text file is particularly interesting. As illustrated in Fig. 1, features reflecting the static analysis logic and rules can be extracted from the bug trace file. 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.

3.5 An Example in the D2A Dataset

Figure 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 *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 Clang 2023 based tools).

3.6 Dataset Generation Results

3.6.1 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 (Infer Infer). However, some issues detectors are not ready for production and thus disabled by default. In the pipeline, we additionally enabled the detection of all issue types related to buffer overflows, integer overflows, and memory/resource leaks, even though some of them may not be production-ready.

Table 3 summarizes the dataset generation results. The column *CMA Version Pairs* shows the number of bug-fixing commits selected by the commit message analyzer (Section 3.2). For each selected commit, we run Infer on both the before-commit and after-commit versions. We

```

01. {
02.   "id": "httpd_9b3a5f0ffd8ec787cf645f97902582acb3234d96_1",
03.   "label": 1,
04.   "label_source": "auto_labeler",
05.   "bug_type": "BUFFER_OVERRUN_US",
06.   "project": "httpd",
07.   "bug_info": {
08.     "qualifier": "Offset: [0, +oo] Size: 10 by call to ...",
09.     "loc": "modules/proxy/mod_proxy_fcgi.c:178:31",
10.     "url": "https://github.com/apache/httpd/blob/..."
11.   },
12.   "versions": {
13.     "before": "545d85acdaa384a25ee5184a8ee671a18ef5582f",
14.     "after": "2c70ed756286b2adf81c55473077698d6d6d16a1"
15.   },
16.   "trace": [
17.     {
18.       "description": "Array declaration",
19.       "loc": "modules/proxy/mod_proxy_fcgi.c:178:31",
20.       "func_key": "modules/proxy/mod_proxy_fcgi.c@167:1-203:2",
21.     }
22.   ],
23.   "functions": {
24.     "modules/proxy/mod_proxy_fcgi.c@167:1-203:2": {
25.       "name": "fix_cgivars",
26.       "touched_by_commit": true,
27.       "code": "static void fix_cgivars(request_rec *r, ..."
28.     }
29.   },
30.   "commit": {
31.     "url": "https://github.com/apache/httpd/commit/2c70ed7",
32.     "changes": [
33.       {
34.         "before": "modules/proxy/mod_proxy_fcgi.c",
35.         "after": "modules/proxy/mod_proxy_fcgi.c",
36.         "changes": ["177,1^^177,5"]
37.       }
38.     ]
39.   },
40.   "compiler_args": {
41.     "modules/proxy/mod_proxy_fcgi.c": "-D_REENTRANT -I./server ...",
42.   },
43.   "zipped_bug_report": "...
44. }

```

Fig. 4 A simplified example in D2A dataset

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 (Section 3.3), 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 (Section 3.3), 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.

The data described in Table 3 is further processed before used for training. The training dataset is described in Section 4.5.1 and training data statistics, including statistics for *security-related bug types* can be found in Table 7.

Table 3 Dataset generation results

Project	Version pairs		Issues reported	Unique auto-labeler examples			Unique after-fix Negatives
	CMA	Infer		All	Negatives	Positives	
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 in the before-commit versions before deduplication

3.6.2 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 Section 4.1, 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 4 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

Table 4 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	17	24	16	13	3	57	30	27
ALL	100%	41%	59%	100%	81%	19%	100%	53%	47%

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

sample set is expected to be higher as there should be more negative examples. Take the OpenSSL study in Section 2.2 as an example. Without auto-labeler, the accuracy was only 7.8% observed in the manual case study of a set of 166 security-related examples (Section 2.2.1).

4 Using D2A for FP Prediction

The D2A dataset can be used to train ML models, which can operate on code directly or which can be used to improve static analyzer output. In this work we present Augmented Static Analysis, which is one way to improve the static analyzer output by using ML models trained on D2A data to predict false positives. We chose this task because we felt this would be the fastest way to improve developer productivity and experience as static analyzers are already part of their workflow. Our approach, uses the D2A dataset to train ML models and then uses one of the Infer output files called the Bug Trace, for inference. In the following subsections, we define the augmented static analysis problem (Section 4.1) and then provide some details about the Infer Static Analyzer output (Section 4.2), which becomes input to our trained ML models. ML models are discussed in depth in Sections 4.3 and 4.4. Then in Section 4.5 we discuss the results of various models on the FP prediction problem.

4.1 Problem Statement

As observed previously (Johnson et al. 2013; Muske et al. 2013), 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 define a process that can identify a subset of the reported issues that are more likely to be true positives, and *use it as a prioritization tool*. The model will rank issues based on the increasing likelihood of being false positives or decreasing likelihood of being true positives. Developers may focus on the issues likely to be true positives first and then move to remaining issues which are likely to be false positive, if they have the time.

We treat the static analyzer as a black box and train a false positive prediction model based on the bug reports and source code. 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 at the start of the prioritized list of issues. An example of this is shown in our libtiff case study in Table 9.

4.2 Problem Dataset

As described in Section 3.3, 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 have bug traces produced by static analysis. In every project, the number of negative labels is very large compared to the number of positive labels, as can be seen in Table 7.

4.3 FP Prediction with Classical Machine Learning

In this section, we discuss how we identify useful features from bug trace and source code, and then train machine learning models for the false positive prediction task.

4.3.1 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. We extract features based on this assumption and taking inspiration from Flynn (2016) and Du et al. (2019).

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 5 lists the features we extract from bug reports.

Although bug reports contain some code and we can extract features from them, the context is generally limited to only a few lines. In order to better capture code complexity, we decided to design features that used more of the surrounding context of the lines of code that appear in bug reports. Apart from bug reports, D2A includes the full source code of each function that occurs in the bug report, including the buggy function identified by the static analyzer. The URL of the file which contains the buggy function is also available and can be easily downloaded. The source code from functions and the file provides additional context which can be used to design features.

Table 6 shows the features, which are extracted from source code. This includes features extracted the source code of all the functions in the bug trace (“_functions” suffix), from the source code of the buggy function (“_bug_function” suffix), from the source code of the file which contains the buggy function (“_bug_code” suffix). There are 4 code features, for each of these 3 code subsets of functions, bug function and bug file. The Cyclomatic Complexity of pure code data (no bug reports) was calculated using the Lizard (Yin 2019) tool. All the bug report and source code features are extracted, normalized and saved in the features file. We analyze the relative importance of these features in Section 4.5.3.

The features are designed for Infer bug report and/or C/C++ source code. Generalizing these features to other static analyzers and programming languages would require careful examination of each feature and would be highly dependent on the characteristics of the static analyzer and the programming language.

4.3.2 Model Selection

We experimented with 13 well-known machine learning models. Namely the Scikit-learn (Pedregosa et al. 2011) implementations of Decision Trees, K-means, Random Forest, Extra-trees, Gradient Boosting, Ada Boost, Linear Classifiers, Gaussian Naive Bayes, Multinomial

Table 5 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 <code>call</code> steps	cfile_count	the number of different .c files
for_count	the number of <code>for</code> 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	max_if_AND_count	Max logical ANDs in an if statement
max_if_OR_count	Max logical ORs in an if statement	avg_if_AND_count	Avg logical ANDs in an if statement
avg_if_OR_count	Avg logical ORs in an if statement	error_char	char number of the error in trace
variable_changed	Number of variables that changed value		

Naive Bayes, and Complement Naive Bayes. And also XGBoost (Chen and Guestrin 2016), Catboost (Dorogush et al. 2018) and LightGBM (Ke et al. 2017).

We ranked these models based on both their AUC and F1 scores and selected the four best models for Voting and Stacking ensembles, discussed later. The four best models were, Random Forest, Extra Trees, LightGBM and Catboost.

4.3.3 Evaluation Metrics

In order to evaluate 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

Table 6 Features extracted from source code

Feature	Description	Feature	Description
CC_bug_code	Cyclomatic complexity of buggy file	params_bug_code	Number of parameters in buggy file
loop_num_bug_code	Number of loops in buggy file	IFwoELSE_bug_code	Number of if without else statements in buggy file
CC_functions	Cyclomatic complexity of trace functions	params_functions	Number of parameters in trace functions
loop_num_functions	Number of loops in trace functions	IFwoELSE_functions	Number of if without else statements in trace functions
CC_bug_function	Cyclomatic complexity of buggy function	params_bug_function	Number of parameters in buggy function
loop_num_bug_function	Number of loops in buggy function	IFwoELSE_bug_function	Number of if without else statements in buggy function

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. One choice for the threshold is the point which minimizes the distance from the top-left corner (all true positives and no false positives). Once this threshold-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.

In this work, we deviate from Zheng et al. (2021) and change the threshold-point to be where the False Positive Rate (FPR) is 5%. The 5% point was chosen arbitrarily, based on

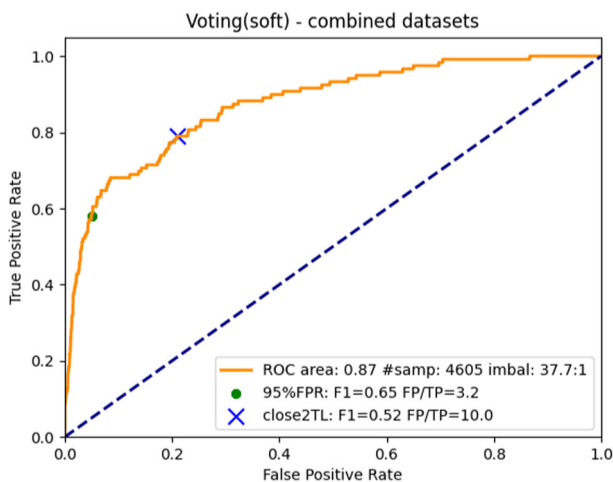


Fig. 5 The blue X marks the point on the ROC curve that minimizes the distance from the top-left corner. The green dot indicates the point where the False Positive Reduction Rate is 95%. This figure is for the Classical ML model described in Section 4.3

our assumption of how much false positives the users would be willing to tolerate. This point is indicated by the green dot in Fig. 5 while the blue cross indicates the previous threshold-point, closest to the top-left corner. The False Positive Reduction Rate (FPRR) at the new threshold-point (green dot) is 95% ($100 - \text{FPR}$). The F1 score is also calculated based on this point. Setting the threshold-point to FPRR 95% sets a standard for models and makes the comparison between them easier. The models also become more selective causing a reduction in the number of positive predictions and consequently a lower True Positive Rate (TPR) but also a much lower FPR. The lower TPR can be mitigated by model improvement.

4.3.4 Voting

Real-world datasets present a high imbalance between real bugs and false positives. Also, the projects used to derive 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, based on Scikit-learn, which averages the scores of each classifier, which should produce a more stable behavior across datasets.

4.3.5 Stacking Ensemble

Voting ensemble gives us good results but it is better suited for combining the predictions of a group of classifiers which perform equally well. If there is a difference in the performance of classifiers, or certain classifiers outperform others, it's better to use an ensemble that can learn from the predictive behavior of individual classifiers. For this purpose, we also used a stacking ensemble (Wolpert 1991) which has a logistic regression classifier that is trained with base model predictions as features.

Figure 6 shows the latest ensemble architecture we use to make predictions. Trace features, combined with Bug function features, trace functions features and bug file features are fed into the base models. Base model predictions are then combined using the scikit-learn Voting(soft) or Stacking classifiers, and scored with their `predict_proba` function. `predict_proba` is a function supported by scikit-learn models which returns the probability of different labels.

4.4 FP Prediction with Deep Learning

Zheng et al. (2021) showed that feature engineering for source code analysis can be an effective way to predict false positives. Deep learning is a powerful and a more generalizable technique that has been applied to source code related tasks (Feng et al. 2020; Guo et al. 2021; Wang et al. 2021). Both approaches come with their limitations and trade-offs. While ml ensembles are light weight and give very good results, they rely on hand crafted features which require expertise and time to develop, and are not easily generalizable. On the other hand, deep learning models are more generalizable, but have greater hardware requirements, limits on input size and may not be able to learn strong features like cyclomatic complexity. In Section 5 we show how different models perform on different types of data.

In this section, we discuss training BERT, which is a deep learning model, on both source code and static analyzer bug reports for the FP prediction task. BERT does not require hand crafted features and hence is more generalizable to other static analyzer outputs and

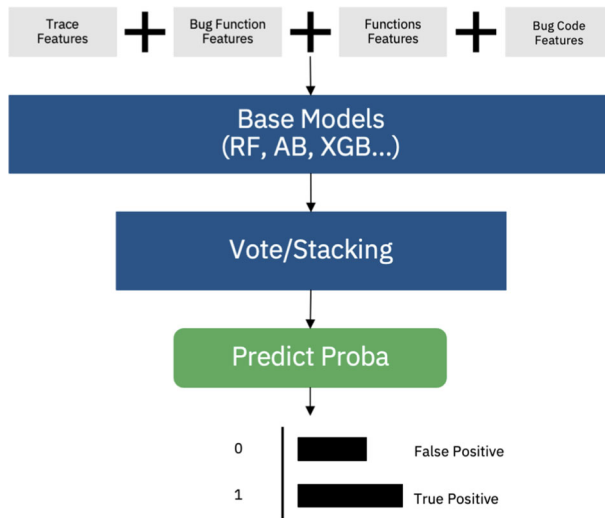


Fig. 6 Stacking and Voting Ensemble architecture with different Tree, Boosting base models. Features from different types of data can be taken as input

programming languages. Related work, Section 6, has more details about BERT (Devlin et al. (2019)) and self-supervised learning.

BERT has proven to be very successful in NLP classification tasks. The bidirectional nature of BERT suits defect detection because both front and back contexts are important in defect analysis. Since D2A contains C/C++ data only, we decided to use C-BERT (Puri et al. 2021; Buratti et al. 2020) which is a BERT-based model trained only on C source code.

Any BERT model, including C-BERT, can be trained in two phases. The first phase is unsupervised and involves training with a large quantity of text data. In case of C-BERT the text data was C source code. More details about this process can be found in Devlin et al. (2019), Buratti et al. (2020) and Puri et al. (2021). The second phase, fine-tuning, is supervised and task dependent. We use the D2A dataset in the fine-tuning phase.

Fine-tuning is performed on one single Nvidia V100 GPU. We fine-tune for 10 epochs, using a small learning rate of 0.000008 to avoid destroying the knowledge acquired during pre-training. Batch size is set to 16 and the typical fine-tuning time ranges between 30 minutes to a couple of hours (for bigger datasets such as OpenSSL and libav).

Like all transformer-based models, C-BERT has a limitation on the input size, which is the maximum size of the input text, measured in tokens, that can be fed into the model. The maximum size of input text that can be fed into the C-BERT model is 512 tokens. In other words, the input context window size of C-BERT is 512 tokens. If the input code snippet or bug report, after preprocessing with Sentencepiece (Kudo and Richardson 2018), is longer than 512 tokens, it will be truncated.

Due to the same input size limitation, we can not fit all data into the context window. Therefore, as shown in Fig. 7, we explored multiple options to combine several types of the data in the D2A dataset and evaluated their performances:

- Trace only: the bug trace produced by Infer only (e.g., Fig. 1). It's a mixture of source code and natural language.
- Single bug function: the particular function body where the bug occurs per the Infer report. It's only source code.

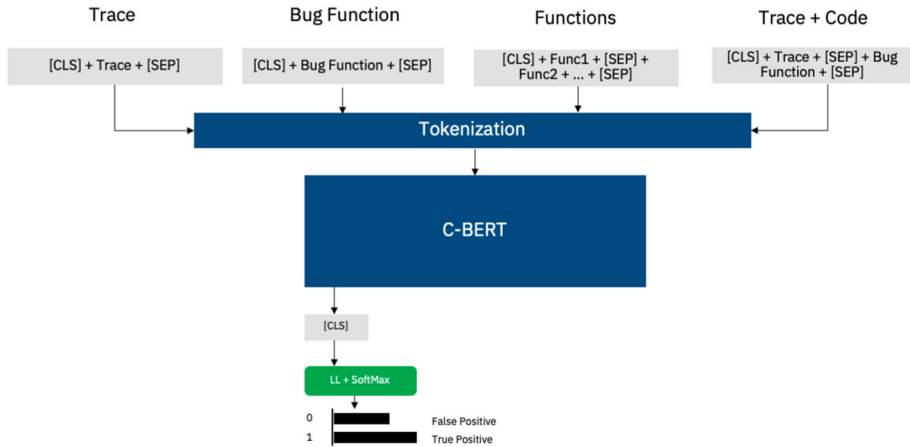


Fig. 7 Due to input size restriction for fine-tuning, we have to pick just a subset of the available data in the D2A dataset. We tried to combine all the information that are useful for a BERT-like model in different ways and evaluated their performance. For bug trace and bug function, we had two special tokens ([CLS] and [SEP]) to let the model know where the input starts and where it ends. For list of functions, since it's a concatenated list, we separate each of them with the special token [SEP] and we do the same to concatenate the bug trace and the bug function. After the input is built with the special tokens, it is tokenized and sent to the model. As designed by the original BERT model, the special token [CLS] is sent to a standard linear layer that will perform the final binary classification. It should be noted that the different alternatives in input at the tokenization block are never given to the model together, since it can see only one modality at a time

- **Functions:** concatenating all the functions that appeared in the Infer trace. It's only source code.
- **Trace + Bug Function:** besides the Infer trace, we additionally include the function body where the bug occurs per the Infer report. It's a mixture of source code and natural language.

While a detailed evaluation of C-BERT performances can be found in Section 4.5, we highlight here a few interesting findings. We observed that option of Trace+Bug Function consistently outperformed other options, which confirms our intuition that C-BERT can leverage the bi-modal data of natural language and code. Another interesting result is that only modeling the list of functions in trace usually gave the worst results, which is expected since concatenating multiple functions could be too long for the limited input size.

4.5 False Positive Prediction Results

In this section, we show the evaluation results of the AI-based static analysis false positive prediction as a use case to demonstrate how D2A dataset can be helpful.

4.5.1 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.

We observed that some FFmpeg and libav examples are quite similar as libav was forked from FFmpeg (Wiki 2023). 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, for the initial experiments, we considered just the 18 *security-related bug types* that are enabled by default. These results are in Table 8. When trained and tested on all bug types, not just security related, the classical ml models and the original set of bug report features described in Zheng et al. (2021) did not perform very well on the large OpenSSL and libav datasets, as can be seen in Table 9. With C-BERT and improved ensemble models with new code features, the results improved on the full dataset with all error types. Table 7 shows the statistics of the full *train:dev:test* data used in the experiments.

4.5.2 Results

The results show the impact of different ML techniques and different features on the FP prediction problem. We try three models, a voting ensemble trained with old set of features described in Zheng et al. (2021) (*vote*), the C-BERT model which extracts features automatically from bug reports and source code (*c-bert*) and finally a voting ensemble trained with new features (*vote-new*). The new set of features are the full set of bug report features as well features extracted directly from source code.

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. For C-BERT and Vote ensemble on new features, we fix the threshold-point to 0.05 FPR. Because this point has changed compared to Vote ensemble applied on initial set of trace features, we omit FPRR and F1 columns from the results Tables 8 and 9. We also omit Predicted Negatives (N) and Correct Negative (TN) columns since they can be calculated from other metrics. Since voting ensemble improvement reflects improvement in base models, we omit the base model results as well.

C-BERT outperforms vote ensemble on datasets with all error types. This could be because large datasets require more features to correctly classify samples. C-BERT also outperforms vote on small project datasets with default security error types, but since these datasets are small the results are not too reliable. In both the tables we can see that vote-new AUC scores are much better than vote. This improvement is because of new features. The new features

Table 7 Production-ready security related error types filtering

	All errors	Positives	N:P	Prod-ready sec Errs		
	Negatives			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

Table 8 Default security error types false positive classification results

	Model	D2A Positives	Predicted Positives	Correct Positives	D2A Negatives	AUC
OpenSSL	vote	81	506	58	2711	0.83
	c-bert	81	170	36	2711	0.80
	vote-new	81	168	43	2711	0.86
libav	vote	28	254	21	1495	0.89
	c-bert	28	53	20	1495	0.87
	vote-new	28	35	21	1495	0.91
NGINX	vote	5	54	4	145	0.78
	c-bert	5	9	3	145	0.82
	vote-new	5	6	2	145	0.89
libtiff	vote	3	7	2	118	0.97
	c-bert	3	7	2	118	0.96
	vote-new	3	4	2	118	0.98
httpd	vote	2	6	1	17	0.85
	c-bert	2	2	2	17	1
	vote-new	2	2	1	17	1
combined	vote	119	814	82	4486	0.84
	c-bert	119	224	63	4486	0.83
	vote-new	119	291	70	4486	0.87

The released dataset has train/dev/test splits for each project. The combined files are the union of corresponding sets of all projects. The models are trained on train + dev sets and tested on the test set. The D2A positive/negative labels are derived from differential analysis of commit pairs.

vote: voting ML ensemble with initial features.

vote-new: voting ML ensemble with initial features and new code-focused features.

c-bert: c-bert finetuning with Code + Trace configuration.

also help the voting ensemble outperform the more sophisticated C-BERT model across the board.

4.5.3 Feature Importance

In order to better understand the impact of features on model performance we rank the features according to their importance in Fig. 8. The initial set of bug report features from Zheng et al. (2021) are shown in green and the new bug report and source code features are shown in blue. Line number of the error in the file and the number of lines of code in the bug report and average length of line of code are significant bug report features perhaps suggesting that large files and complex bug reports distinguish real errors. The new code based features have a significant impact as they account for 4 of the top 5 features. The “If without else” count in trace functions and Cyclomatic complexity of the buggy file are the most important features after error line number. Features based on the buggy file and functions in bug reports

Table 9 All error types false positive classification results

	Model	D2A Positives	Predicted Positives	Correct Positives	D2A Negatives	AUC
OpenSSL	vote	793	34251	792	34149	0.69
	c-bert	793	2034	333	34149	0.75
	vote-new	793	1854	146	34149	0.73
libav	vote	458	8	8	23536	0.61
	c-bert	458	1297	171	23536	0.68
	vote-new	458	1294	143	23536	0.73
NGINX	vote	42	315	29	1783	0.77
	c-bert	42	106	26	1783	0.89
	vote-new	42	90	33	1783	0.93
libtiff	vote	58	198	44	1171	0.89
	c-bert	58	98	41	1171	0.94
	vote-new	58	111	54	1171	0.98
httpd	vote	20	263	13	1150	0.77
	c-bert	20	43	10	1150	0.82
	vote-new	20	64	12	1150	0.90

The released dataset has train/dev/test splits for each project. The models are trained on train + dev sets and tested on the test set. The D2A-truth labels are derived from differential analysis of commit pairs. Models as defined in Table 8

seem to have a bigger impact than features based on buggy function alone. This highlights the importance of additional context.

4.5.4 Libtiff Results Analysis

An interesting project is `libtiff` for which all models achieve very good AUC. To analyze this result further, in Fig. 9 we plot the cost of finding each True Positive in terms of False Positives for `libtiff` with the Vote-soft ensemble on all error types. The X-axis shows True Positives in decreasing order of model confidence. The Y-axis plots the count of new False Positives since the last True Positive. The purple line represents the cumulative number of False Positives, while the dotted purple line parallel to the x-axis indicates the 95% FP Reduction line or 5% FP rate, indicating that 95% of False Positives lie above this line. As mentioned before, this arbitrary point is our guess of how much false positives users would be willing to tolerate. The plot indicates that the model is confident in its prediction of True Positives to a considerable degree. The first 27 highly ranked samples are all TPs. This analysis is useful because it justifies providing a prioritized list of static analyzer output to developers so that they can focus first on those samples which a model confidently thinks are TP.

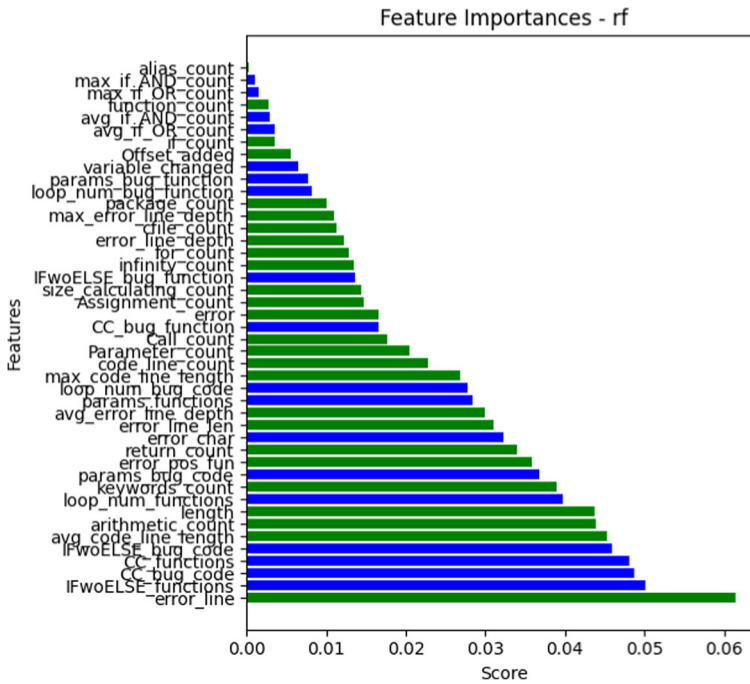


Fig. 8 Feature Importance of Random Forest algorithm trained on OpenSSL Default Security errors after including new features (shown in blue)

5 D2A Leaderboard

With Stacking ensemble, C-BERT and Voting Ensemble we had multiple models for identifying False Positives, each with its own strength. In order to compare the model performance

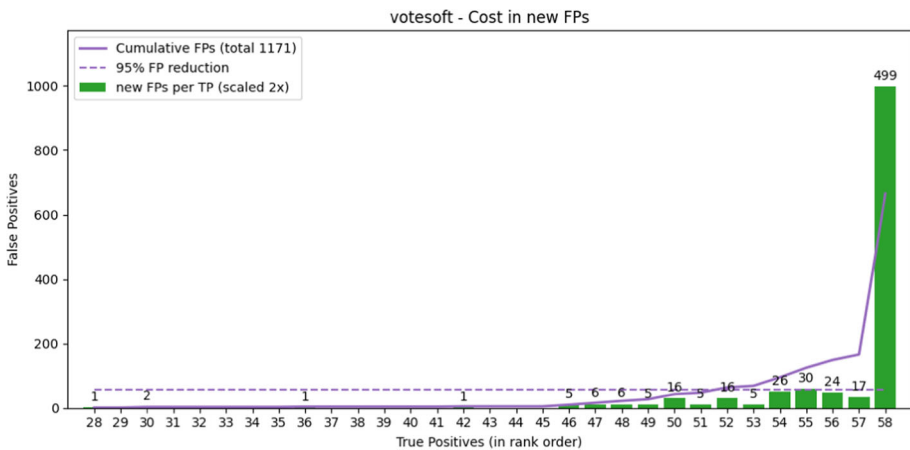


Fig. 9 Cost of each TP in new FPs

on different types of input data from D2A, we decided to create a leaderboard (<https://github.com/ibm/D2A>). Generally, a leaderboard shows the top scoring competitors in a tournament. This idea has been adopted in the AI domain, where a leaderboard will show the top performing models and teams for a given dataset. With the D2A leaderboard we could see how different models perform on tasks which cater to different types of D2A data. We have also made this leaderboard public so that others can participate and compare their model performance on D2A with other models. Table 10 summarizes the leaderboard tasks and available data.

The leaderboard is meant to demonstrate how the D2A dataset can be used both in terms of the different types of D2A data and alternative but related tasks to FP Reduction, like Vulnerability Detection. It is conceivable that the dataset can be used for more code related tasks like Bug Localization and Bug Fixing, which we do not explore in this work.

5.1 Data

All the leaderboard data is derived from the original D2A dataset which contains different program elements to help with program analysis. In order to understand the impact of different program elements on predictions, we use them in isolation and in combination. The D2A dataset contains labels derived from two sources, the auto-labeler and the after-fix extractor. The leaderboard makes use of both kinds of labels where each label indicates if a sample represents a real (1) or false (0) report of a bug.

Details of different kinds of data and the corresponding labels are given below.

- Infer Bug Reports (Trace): This dataset consists of Infer bug reports, which are a combination of English language and C Programming language text.
- Bug function source code (Function)
- Bug function source code, trace functions source code and bug function file URL (Code)

The D2A dataset contains a lot more information which we did not use in the leaderboard tasks.

5.2 Tasks

Tasks require the use some of the different program and program analysis elements that D2A contains.

- Trace: Bug trace or a bug report contains both natural language and code. The code is limited to code snippets from different functions and files. Models are expected to work with a combination of natural language and code snippets to make the prediction.

Table 10 Tasks and data summary for D2A leaderboard

Task	Metrics	Total samples	Train / dev / test	N:P
Code + Trace	AUROC, F1-5%FPR	45,957	36,719 / 4,634 / 4,604	39:1
Trace	AUROC, F1-5%FPR	45,957	36,719 / 4,634 / 4,604	39:1
Code	AUROC, F1-5%FPR	45,957	36,719 / 4,634 / 4,604	39:1
Function	Accuracy	5,857	4,643 / 596 / 618	0.9:1

- Code: Models can use source code from bug function, all the bug trace functions and the file in which the bug function occurs to make the prediction. The file pointed to by `bug_url` must be downloaded in order to be used.
- Trace + Code: Models can use all the files from the previous 2 tasks to make the prediction.
- Function: Models can use only the source code from the bug function to make the prediction. The functions have been derived from a different subset of the full D2A dataset chosen to achieve a more balanced dataset.

Most publicly available defect detection datasets contain only a single function. We include a single function task, but also include multiple functions for cross functional defects and bug reports. Different types of models may be suited for different tasks and the goal is to highlight this and encourage models that perform well across different tasks.

5.3 Metrics

The datasets for the Code + Trace, Trace and Code tasks are derived from the auto-labeler generated samples, and are quite unbalanced with a 0:1 ratio of about 40:1. The Function datasets contain functions marked '1' by the auto-labeler plus the matching after-fix '0' functions, so are well balanced with a 0:1 ratio of about 8:9. Because of these different distributions we use different metrics to measure the performance of models on different tasks.

- Balanced Data: For the balanced dataset we use Accuracy to measure model performance.
- Unbalanced Data: Because the dataset is so heavily unbalanced, we cannot use Accuracy since the model predicting only 0 would have a 98% accuracy. Instead we use the two metrics described below.
 - AUROC: Many open source project datasets are huge with hundreds of thousands of examples and thousands of positive examples. The cost associated with verifying every label is high, which is why it is important to rank the models in the order of their overall model confidence. We use AUROC percentage for this purpose.
 - F1 - 5% FPR: The macro-average F1 score is generally considered a good metric for unbalanced datasets. We want the AUROC curve to peak as early as possible so we calculate the macro-average F1-score percentage at 5% FPR point indicated in Fig. 5.
- Overall: To get the overall model performance, we calculate the simple average percentage of all the scores across all the tasks.

5.4 Leaderboard Results

Table 11 shows the result of the Stacking ensemble (*stacking*), C-BERT (*c-bert*), and the Voting ensemble (*vote-new*) on the 4 Leaderboard tasks. Both ensemble models are trained on the full set of features including bug report features and source code features. Overall Stacking and C-BERT models perform the best, with Voting scoring less than both the models. The C-BERT model performs very well on trace and single function tasks. This could be because the function and trace data fit within the context window of C-BERT, while the full list of functions and code does not. Also, as indicated by Fig. 8, some important features are based on the buggy file and its functions, which are not input to C-BERT because of the

Table 11 D2A Leaderboard

	Code + Trace		Trace		Code		Function Accuracy	Overall score Average
	F1	AUC	F1	AUC	F1	AUC		
stacking	63.4	83.6	61.1	81.2	65.8	85.2	55.2	70.8
c-bert	66.1	81.7	62.4	80.4	62.4	80.2	60.2	70.5
vote-new	64.3	85.0	61.3	80.2	65.2	85.7	45.6	69.6

stacking indicates stacking ensemble with base models trained on new features. Other models as defined in Table 8. Over all tasks stacking is the best performing model

context window limitation. Ensembles do better than C-BERT for Code and Code + trace because they can use features from all sources and are not restricted by context window size.

6 Related Work

Datasets for AI-based Vulnerability Detection Juliet (NIST 2023b), Choi et al. (2017), and S-babi (Sestili et al. 2018) are synthetic datasets. They are generated based on predefined patterns and cannot represent real-world program behaviors. Draper (Russell et al. 2018), Devign (Zhou et al. 2019) and CDG (Li et al. 2018) were generated from real-world programs. However, as discussed in Section 2, they suffer from labeling or source limitations. In fact, lacking good real-world datasets has become a major barrier for this field (Lin et al. 2020). 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 (Kremenek and Engler 2003; Jung et al. 2005; Yüksel and Sözer 2013b; Hanam et al. 2014; Tripp et al. 2014; Koc et al. 2017a; Zhang et al. 2017; Reynolds et al. 2017; Flynn et al. 2018; Raghothaman et al. 2018; Koc et al. 2019) have been proposed. Because they either target different languages or different static analyzers/model, 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. Besides the classic machine learning models presented in Section 4.3 that leverage the manually picked features, we further explored the direction of using deep learning model to learn from both the code and text in the bug report for the static analysis false positive reduction task. To the best of our knowledge, this is the first attempt along this direction.

Defect Detection Leaderboard The only other defect detection leaderboard is CodeXGLUE (Lu et al. 2021) Defect Detection (Code-Code) based on the Devign dataset. Since the samples in this dataset are limited to a single function, it is ideally suited for transformer based models and we see this reflected in the top performing models on the leaderboard, which includes C-BERT. D2A Leaderboard contains abundant information of various types of program and program analysis elements, which makes it is more challenging and levels the playing field for different machine learning approaches.

Features for Vulnerability Detection Du et al. (2019) suggest some interesting features that they extract using Joern (Yamaguchi et al. 2014). We extract some similar features but using either simple string matching or using the Lizard (Yin 2019) tool. Inspired by their

findings, we additionally included other relevant features that were extracted from both the source code and the static analysis report.

Contextual Representation with BERT BERT (Devlin et al. 2019) is a bidirectional encoder based on the transformer architecture (Vaswani et al. (2017)) that can create a meaningful representation for every token by looking at both left and right context. As a result, every token will have a contextual representation of its neighbors such that the final representation of a word changes if its context changes. The learning phase of BERT can be divided into two phases, pre-training and fine-tuning:

- Pre-training is a self-supervised process, with a goal to build a general language representation that is not connected to specific tasks. To learn the statistical properties of source code, parts of the input are masked and then the model is asked to predict them back (Feng et al. 2020; Kanade et al. 2020).
- Fine-tuning is a supervised phase and requires task-related labels, so every downstream task, like vulnerability detection, needs a specific fine-tuning dataset.

Self-supervised Learning for Code Existing works (Hindle et al. 2012; Allamanis et al. 2018; Ray et al. 2016) have shown that source code is natural. Similar to speech and natural language, source code tends to be repetitive and predictable. This naturalness can be exploited such that the statistical distribution of source code can be learned from a huge amount of data available on sites like GitHub. Compared to feature engineering based learning, which requires labeled data, the most appealing advantage of self-supervised deep learning is that the models can extract features automatically from unlabelled data. Once the model has been trained on generic source code, it only needs a small amount of labeled data to be fine-tuned for a specific task.

Self-supervised learning based language models (Devlin et al. 2019; Liu et al. 2019; Vaswani et al. 2017) have already been successful in the field of natural language processing for tasks like text classification, question & answering, text completion, etc. And now, it is possible to see the same trend for code-related tasks, as shown by the popular leaderboard CodeXGlue (Lu et al. 2021), where most of the top performing models are based on such architectures but trained specifically on source code.

7 Threats to Validity

Initial Manual Analysis The initial manual analysis of Infer output was done on one project, OpenSSL, for only 4 bug types. The reason is the high amount of time needed from expert developers - it took around 10 hours for each of the 8 developers to label 166 candidate vulnerabilities in total. Ideally, such manual analysis should be applied to more projects and more bug types. However, while we cannot always expect the same label accuracy improvement across different projects (from 7.8% in initial manual analysis to 53% in manual label validation of randomly selected samples), we think the jump is wide enough that the tool remains useful to save developers' time.

Commit Message Analyzer Like all ML models, the quality of the Commit Message Analyzer is bounded by the dataset, which in our case, are the NVDs and their commit messages. It is possible that in any given repository, some bugs are fixed silently and the commits

don't have meaningful messages. Some of these commits can be some real fixes with poorly worded commit messages. The CMA will not be able to identify such fixes.

Reliance on NVDs and the inability to recognize silent fixes or fixes from uninformative commit messages also introduces bias in the CMA. However, due to the lack of an oracle that can cover all possible scenarios (which is true for both traditional rule-based or ML-based detectors), it's unlikely to have a perfect bug detection dataset. Our goal is to create a reasonably large-scale dataset that can facilitate sophisticated model exploration and adoption and CMA helps us create such a dataset.

Result Manual Analysis Because manual analysis is an expensive process, D2A result validation was done for only 57 samples. Although we have only a very small number of curated labels, the improvements we report in the predictions of D2A labels should be useful in practice.

Data Splitting We have partitioned the dataset into train:val:test splits randomly. However, to correctly align with real life use case we should partition the data according to time, which would mean the test set consists of latest issues and train would consist of older issues. Nie et al. (2021) talk about the impact of choosing different evaluation methodologies on code summarization and show that not considering time to split the data will inflate or underestimate the result. They experiment with a strategy similar to our balanced random splits, which they call mixed project evaluation and they show that this approach inflates the results. We wish to perform a similar study for vulnerability analysis using D2A dataset. With D2A, the number of samples we generate from a repository depends on the number of commits in the repository and the number of fixes. To generate time based data splits, we will have to regenerate data by identifying a point in time and use it to partition between train and test. However this may result in too few or too many samples in train or test and we will need multiple generation attempts to arrive at the right balance. Once done, we will have to rerun all the experiments on multiple repositories for all three models. This is an interesting and ideal approach which we hope to explore in the future.

8 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, called the D2A dataset, of more than 1.3M examples with detailed bug related information obtained from the inter-procedural 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. This dataset contains varied data which can be used for Vulnerability Analysis.

We define the false positive reduction problem to showcase the utility of the dataset in improving developer productivity. We train both classic machine learning models as well as a deep learning model for the static program analysis false positive prediction task on the D2A dataset, which can effectively help developers prioritize and investigate potential true positives first. By further analyzing the results of `libtiff`, we show how a prioritized list of static analyzer issues can be helpful for developers.

By combining hand-crafted features from bug reports and source code, we show that machine learning ensembles can perform well on default security errors and on relatively smaller projects like `libtiff`, `httpd` and `NGINX`. More importantly, we show that adding more features leads to model performance improvement on harder data from relatively large projects like `OpenSSL` and `libav` with all error types. We show how C-BERT, a transformer based deep learning model can be used to train with D2A data. Deep learning models are more generalizable than hand crafted features and show good performance on D2A data.

The D2A dataset can also be leveraged in different ways to tackle the more general problem of vulnerability detection. In order to show how this can be done and to encourage community participation, we created a leaderboard based on the D2A dataset and made it publicly available. The leaderboard can be used to compare the performance of different models.

Data availability The datasets generated during and/or analysed during the current study are available in the IBM Data repository, <https://developer.ibm.com/exchanges/data/all/d2a/>.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Allamanis M, Barr ET, Devanbu P, Sutton C (2018) A survey of machine learning for big code and naturalness
- Ayewah N, Pugh W, Hovemeyer D, Morgenthaler JD, Penix J (2008) Using static analysis to find bugs. *IEEE Softw* 25(5):22–29
- Ayewah N, Pugh W, Morgenthaler JD, Penix J, Zhou YQ (2007) Using find bugs on production software. In *OOPSLA'07*
- Buratti L, Pujar S, Bornea M, McCarley JS, Zheng Y, Rossiello G, Morari A, Laredo J, Thost V, Zhuang Y, Domeniconi G (2020) Exploring software naturalness through neural language models. *CoRR*, abs/2006.12641
- Calcagno C, Distefano D, O'Hearn PW, Yang H (2011) Compositional shape analysis by means of bi-abduction. *J ACM* 58(26):1–66
- Chandrasekaran D, Mago V (2020) Evolution of semantic similarity - a survey. *CoRR*, abs/2004.13820. <https://arxiv.org/abs/2004.13820>
- Chen T, Guestrin C (2016) XGBoost: a scalable tree boosting system. In: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, KDD'16*. ACM, New York, pp 785–794. ISBN 978-1-4503-4232-2. <http://doi.acm.org/10.1145/2939672.2939785>
- Choi M-J, Jeong S, Oh H, Choo J (2017) 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, IJCAI'17*
- Clang (2023). Clang tooling. <https://clang.llvm.org/docs/Tooling.html>
- Cppcheck-team (2023). Cppcheck. <http://cppcheck.sourceforge.net/>
- CWE400 (2023). Cwe-400: uncontrolled resource consumption. <https://cwe.mitre.org/data/definitions/400.html>
- CWE457 (2023) Cwe-457: use of uninitialized variable. <https://cwe.mitre.org/data/definitions/457.html>
- CWE476 Cwe-476: null pointer dereference. <https://cwe.mitre.org/data/definitions/476.html>
- Devlin J, Chang M-W, Lee K, Toutanova K (2019) BERT: pre-training of deep bidirectional transformers for language understanding. In: *Proceedings of the 2019 conference of the north American chapter of the association for computational linguistics: human language technologies, vol 1 (Long and Short Papers)*, pp 4171–4186

- Dorogush AV, Ershov V, Gulin A (2018) Catboost: gradient boosting with categorical features support. [arXiv:1810.11363](https://arxiv.org/abs/1810.11363)
- Du X, Chen B, Li Y, Guo J, Zhou Y, Liu Y, Jiang Y (2019) Leopard: identifying vulnerable code for vulnerability assessment through program metrics. In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE). IEEE, pp 60–71
- Facebook (2023a) Infer static analyzer. <https://fbinfer.com/>
- Facebook (2023b) Infer reportdiff. <https://fbinfer.com/docs/man-infer-reportdiff>
- Fan G, Wu R, Shi Q, Xiao X, Zhou J, Zhang C (2019) Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In ICSE'19
- Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M (2020) CodeBERT: a pre-trained model for programming and natural languages. In: Findings of the association for computational linguistics: EMNLP 2020, pp 1536–1547. <https://doi.org/10.18653/v1/2020.findings-emnlp.139>
- Flynn L (2016) Prioritizing alerts from static analysis to find and fix code flaws. <http://insights.sei.cmu.edu/blog/prioritizing-alerts-from-static-analysis-to-find-and-fix-code-flaws/>
- Guarnieri S, Pistoia M, Tripp O, Dolby J, Teilhet S, Berg R (2011) Saving the world wide web from vulnerable Javascript. In: Proceedings of the 2011 international symposium on software testing and analysis, ISSTA'11
- Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, Zhou L, Duan N, Svyatkovskiy A, Fu S, Tufano M, Deng SK, Clement C, Drain D, Sundaresan N, Yin J, Jiang D, Zhou M (2021) Graphcodebert: pre-training code representations with data flow. In: International conference on learning representations
- Hanam Q, Tan L, Holmes R, Lam P (2014) Finding patterns in static analysis alerts: improving actionable alert ranking. In: Proceedings of the 11th working conference on mining software repositories, MSR 2014, pp 152–161
- Hindle A, Barr ET, Su Z, Gabel M, Devanbu P (2012) On the naturalness of software. In: 2012 34th international conference on software engineering (ICSE), pp 837–847. <https://doi.org/10.1109/ICSE.2012.6227135>
- Infer. Infer issue types. <https://github.com/facebook/infer/blob/ea47cf/infer/man/man1/infer.txt#L370>
- Johnson B, Song Y, Murphy-Hill E, Bowdidge R (2013) Why don't software developers use static analysis tools to find bugs? In ICSE'13, pp 672–681
- Jung Y, Kim J, Shin J, Yi K (2013) 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, SAS'05, pp 203–217
- Kanade A, Maniatis P, Balakrishnan G, Shi K (2020) Learning and evaluating contextual embedding of source code. In: International conference on machine learning. PMLR, pp 5110–5121
- Ke G, Meng Q, Finley T, Wang T, Chen W, Ma W, Ye Q, Liu T-Y (2017) Lightgbm: a highly efficient gradient boosting decision tree. *Adv Neural Inf Process Syst* 30:3146–3154
- Ke G, Meng Q, Finley T, Wang T, Chen W, Ma W, Ye Q, Liu T-Y (2017) Lightgbm: a highly efficient gradient boosting decision tree. *Adv Neural Inf Process Syst* 30:3146–3154
- Koc U, Saadatpanah P, Foster JS, Porter AA (2017a) Learning a classifier for false positive error reports emitted by static code analysis tools. In MAPL'17, pp 35–42
- Koc U, Saadatpanah P, Foster JS, Porter AA (2017b) Learning a classifier for false positive error reports emitted by static code analysis tools. In MAPL'17
- Kremenek T, Engler DR (2003) Z-ranking: using statistical analysis to counter the impact of static analysis approximations. In: Cousot R (ed), Static analysis, 10th international symposium, SAS 2003
- Kudo T, Richardson J (2018) Sentencepiece: a simple and language independent subword tokenizer and detokenizer for neural text processing. In EMNLP
- LaToza TD, Venolia G, DeLine R (2006) Maintaining mental models: a study of developer work habits. In: Proceedings of the 28th international conference on software engineering
- Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong Y (2018) Vuldeepecker: a deep learning-based system for vulnerability detection. In: 25th annual network and distributed system security symposium, NDSS'18
- Lin G, Wen S, Han QL, Zhang J, Xiang Y (2020) Software vulnerability detection using deep neural networks: a survey. *Proc IEEE* 108(10):1825–1848
- Liu Y, Ott M, Goyal N, Du J, Joshi M, Chen D, Levy O, Lewis M, Zettlemoyer L, Stoyanov V (2019) Roberta: a robustly optimized bert pretraining approach. *ArXiv, abs/1907.11692*
- Livshits VB, Lam MS (2005) Finding security vulnerabilities in java applications with static analysis. In: Proceedings of the 14th conference on USENIX security symposium
- LLVM. The clang static analyzer. <https://clang-analyzer.llvm.org/>

- Flynn L, Snaveley W, Kurtz Z (2018) Test suites as a source of training data for static analysis alert classifiers. SEI Blog. https://insights.sei.cmu.edu/sei_blog/2018/04/static-analysis-alert-test-suites-as-a-source-of-training-data-for-alert-classifiers.html
- Lu S, Guo D, Ren S, Huang J, Svyatkovskiy A, Blanco A, Clement C, Drain D, Jiang D, Tang D, Li G, Zhou L, Shou L, Zhou L, Tufano M, Gong M, Zhou M, Duan N, Sundaresan N, Deng SK, Fu S, Liu S (2021) Codexglue: a machine learning benchmark dataset for code understanding and generation. ArXiv, abs/2102.04664
- MITRETop25. Cwe top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html
- Murphy-Hill E, Zimmermann T, Bird C, Nagappan N (2015) The design space of bug fixes and how developers navigate it. IEEE Trans Software Eng 41(1):65–81
- Muske T, Serebrenik A (2016) Survey of approaches for handling static analysis alarms. In: 2016 IEEE 16th international working conference on source code analysis and manipulation (SCAM), pp 157–166
- Muske TB, Baid A, Sanas T (2013) Review efforts reduction by partitioning of static analysis warnings. In: 13th international working conference on source code analysis and manipulation
- Nie P, Zhang J, Li JJ, Mooney RJ, Gligoric M (2021) Impact of evaluation methodologies on code summarization. arXiv:2108.09619
- NIST (2023a) National vulnerability database. <https://nvd.nist.gov/>
- NIST (2023b) Juliet test suite for c/c++ version 1.3. <https://samate.nist.gov/SRD/testsuite.php>
- O'Hearn P, Reynolds J, Yang H (2001) Local reasoning about programs that alter data structures. LNCS 2142
- Paletov R, Tsankov P, Raychev V, Vechev M (2018) Inferring crypto api rules from code changes. In: Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation, PLDI 2018, pp 450–464
- Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay É (2011) Scikit-learn: machine learning in python. J Mach Learn Res 12(85):2825–2830. <http://jmlr.org/papers/v12/pedregosa11a.html>
- Puri R, Kung DS, Janssen G, Zhang W, Domeniconi G, Zolotov V, Dolby J, Chen J, Choudhury MR, Decker L, Thost V, Buratti L, Pujar S, Finkler U (2021) Project codenet: a large-scale AI for code dataset for learning a diversity of coding tasks. ArXiv, abs/2105.12655
- Raghothaman M, Kulkarni S, Heo K, Naik M (2018) User-guided program reasoning using bayesian inference. In Proceedings of the 39th ACM SIGPLAN conference on Programming Language Design and Implementation, PLDI 2018, pp 722–735
- Ray B, Hellendoorn V, Godhane S, Tu Z, Bacchelli A, Devanbu P (2016) On the naturalness of buggy code. ICSE '16, pp 428–439
- Reynolds ZP, Jayanth AB, Koc U, Porter AA, Raje RR, Hill JH (2017) Identifying and documenting false positive patterns generated by static code analysis tools. In: 4th international workshop on software engineering research and industrial practice
- Russell RL, Kim LY, Hamilton LH, Lazovich T, Harer J, Ozdemir O, Ellingwood PM, McConley MW (2018) Automated vulnerability detection in source code using deep representation learning. In ICMLA'18
- Sahami M, Heilman TD (2006) A web-based kernel function for measuring the similarity of short text snippets. In WWW '06
- Sestili CD, Snaveley WS, VanHoudnos NM (2018) Towards security defect prediction with AI. CoRR, abs/1808.09897. <http://arxiv.org/abs/1808.09897>
- Sui Y, Cheng X, Zhang G, Wang H (2020) Flow2vec: value-flow-based precise code embedding. OOPSLA
- Suneja S, Zheng Y, Zhuang Y, Laredo J, Morari A (2020) Learning to map source code to software vulnerability using code-as-a-graph. CoRR, abs/2006.08614
- Tripp O, Guarnieri S, Pistoia M, Aravkin A (2014) ALETHEIA: improving the usability of static security analysis. In: Proceedings of the 2014 ACM SIGSAC conference on computer and communications security, pp 762–774
- Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. In: Advances in neural information processing systems
- Villard J (2023). Infer is not deterministic, infer issue #1110. <https://github.com/facebook/infer/issues/1110>
- Wang Y, Wang W, Joty S, Hoi SCH (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, pp 8696–8708. Association for computational linguistics, November 2021. <https://doi.org/10.18653/v1/2021.emnlp-main.685>
- Wheeler DA (2023). Flawfinder. <https://dwheeler.com/flawfinder/>
- Wiki (2023). Libav. https://en.wikipedia.org/wiki/Libav#Fork_from_FFmpeg
- Wolpert DH (1992) Stacked generalization. Neural Netw 5(2):241–259

- Yamaguchi F, Golde N, Arp D, Rieck K (2014) Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE symposium on security and privacy, pp 590–604. <https://doi.org/10.1109/SP.2014.44>
- Yamaguchi F, Maier A, Gascon H, Rieck K (2015) Automatic inference of search patterns for taint-style vulnerabilities. In: 2015 IEEE symposium on security and privacy
- Yin T (2019) Lizard: an extensible cyclomatic complexity analyzer
- Yüksel U, Sözer H (2013a) Automated classification of static code analysis alerts: a case study. In: ICSM'13
- Yüksel U, Sözer H (2013b) Automated classification of static code analysis alerts: a case study. In: 2013 IEEE international conference on software maintenance, pp 532–535
- Zhang X, Si X, Naik M (2017) Combining the logical and the probabilistic in program analysis. In: Proceedings of the 1st ACM SIGPLAN international workshop on machine learning and programming languages, MAPL 2017, pp 27–34
- Zheng Y (2023). Parallelism gives inconsistent results, infer issue #1239. <https://github.com/facebook/infer/issues/1239>
- Zheng Y, Pujar S, Lewis B, Buratti L, Epstein E, Yang B, Laredo J, Morari A, Su Z (2021) D2a: a dataset built for ai-based vulnerability detection methods using differential analysis. In: 2021 IEEE/ACM 43rd international conference on software engineering: software engineering in practice (ICSE-SEIP). IEEE, pp 111–120
- Zhou Y, Liu S, Siow JK, Du X, Liu Y (2019) Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In NeurIPS'19

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Saurabh Pujar is a Senior Software Engineer at IBM Research. His research is primarily focused on AI and NLP applications like health care, software security and more recently, source code generation. His interests are at the intersection of AI, NLP and Software Engineering. Before moving to IBM Research, he worked as a full stack engineer in several companies. He has a Bachelor's in Engineering from Mumbai University and a Masters in Computer Science from Courant Institute of Mathematical Science, New York University.



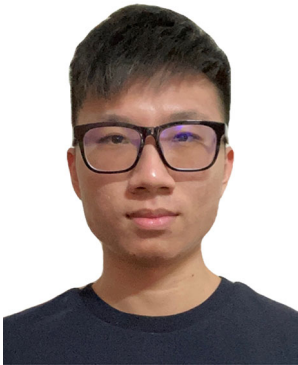
Yunhui Zheng was a Research Staff Member at IBM Research, working on projects related to program analysis, language-based security, formal verification, AI for code, etc. He received his PhD from Purdue University.



Luca Buratti I am a Research Software Engineer at IBM Research. My research interest spans various applications and domains of Artificial intelligence. In the past few years, I have mainly worked on AI4Code, which is about applying AI models for source code understanding and generation. I have contributed to different projects in this domain, from datasets creation (Project CodeNet, D2A) to models for code understanding (C-BERT, DISCO, CONCORD) and for code generation (IBM watsonx Code Assistant for Red Hat Ansible Lightspeed). I received M.Sc. and B.Sc. degrees in Computer Engineering from University of Bologna in 2018 and 2016, respectively.



Burn Lewis has worked at IBM Research for many years on speech recognition, unstructured information analysis, the Watson Jeopardy challenge, electronic medical record analysis, and AI for code generation.



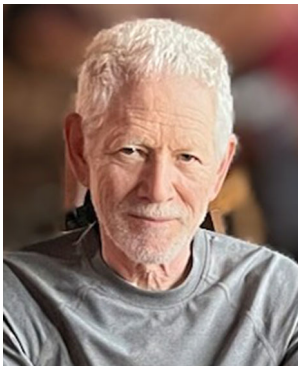
Yunchung Chen received his BS degree in computer science and information engineering from National Taiwan University, Taipei, Taiwan, in 2019. He is currently working toward his Doctoral program in Cybersecurity, Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan. His research interests include cybersecurity, machine learning, and mobile security.



Jim Laredo is an IBM Distinguished Engineer in the AI Group at IBM Research, in Yorktown Heights, New York. He leads the research agenda for Automation and Integration, infusing AI to enable business users to automate their work. He is also working on applying AI towards Software Engineering tasks, in particular to detect Security Vulnerabilities. His interests extend to BPM research, SaaS platforms and Cloud native development. Earlier in his career he was on the early stages of startups Vitria Technology and Transarc. He graduated from Universidad Simon Bolivar in Caracas, Venezuela and has a MSc. in Computer Science from University of Toronto in Canada. He holds 40 patents and over 40 publications.



Alessandro Morari Senior researcher with experience in High Performance Computing and Machine Learning. His current research focuses on the scalability of deep learning systems, and on the use of deep learning for the analysis and generation of source code. He has taught a graduate course at New York University on how to leverage HPC for Machine Learning, and he has authored multiple academic publications in international conferences and journals. He holds a Ph.D. from the University of Catalunya, Spain.



Edward Epstein received a B.E.E. degree from Georgia Tech in 1971 and subsequently created hardware and/or software in several diverse areas, including: the world's first automated white blood cell differential system based on flow-through cytochemical analysis, the Hemalog-D white cell differential system, at Technicon Instruments Corp; the world's first real-time large vocabulary automatic speech recognition system, the Tangora speech recognition system, at IBM Research; and the IBM Watson system which successfully competed against humans in "Jeopardy! The IBM Challenge". He is currently contributing to development of cloud software frameworks in the Hybrid Cloud department at IBM Research.



Tsungnan Lin received his B.S. degree from National Taiwan University, Taipei, Taiwan, in 1989 and his M.S. and Ph.D. degrees from Princeton University, Princeton, NJ, USA, in 1993 and 1996, respectively. He then joined EPSON Research and Development, Inc., San Jose, CA, USA, and EMC Corporation, Hopkinton, MA, USA. Since February 2002, he has been with the Department of Electrical Engineering and the Graduate Institute of Communication Engineering, National Taiwan University. He had been the Director of the Division of Network Management of Computer and Information Networking Center, National Taiwan University and Vice President and Director General of Cybersecurity Technology Institute, Institute for Information Industry. Dr. Lin is a member of the Phi Tau Phi Scholastic Honor Society.




Bo Yang Technical Director in AIFUTURE. He has more than 15 years of R&D experience in cloud computing, artificial intelligence, privacy protection and security compliance. Focusing on large model technology and providing intelligent and secure data services. He has published dozens of papers in core journals and patents in the field of innovation. He once served as a staff researcher member at IBM China Research and an architect at JD.com's Technology Innovation Department.



Zhong Su is a Senior Director in Alibaba Cloud and leading research activities around technology strategy and new industry innovation driven by technical innovation. Before joining Alibaba group, Dr. Su was leading AI research efforts in IBM research-China. During his research work in IBM, he has been awarded the Technical Accomplishment of IBM research for many times and Outstanding Technical Accomplishment of IBM research in 2014 and 2016. He was IBM Academy of Technology member and was awarded IBM Master Inventor in 2007 and 2013. He has published more than 80 papers in top international conferences & journals (google citations > 9000), with more than 100 patents or patents pending. He got ACM SigKDD Test of Time award in 2020. Dr. Su was adjunct professor of NanKai University, guest professor of CS department, Shanghai JiaoTong University. Currently, he is a senior member of Artificial Intelligence and Pattern Recognition Expert Committee in China Computer Federation, committee member of Chinese Information Society.

Authors and Affiliations

Saurabh Pujar¹  · Yunhui Zheng¹ · Luca Buratti¹ · Burn Lewis¹ ·
Yunchung Chen² · Jim Laredo¹ · Alessandro Morari¹ · Edward Epstein¹ ·
Tsungnan Lin² · Bo Yang³ · Zhong Su³

Luca Buratti
luca.buratti2@ibm.com

Burn Lewis
burn@us.ibm.com

Yunchung Chen
f08921a01@ntu.edu.tw

Jim Laredo
laredoj@us.ibm.com

Alessandro Morari
amorari@us.ibm.com

Edward Epstein
eae@us.ibm.com

Tsungnan Lin
tsungnan@ntu.edu.tw

¹ IBM T. J. Watson Research Center, Yorktown Heights, NY, USA

² National Taiwan University, Taipei City, Taiwan

³ IBM Research, Beijing, China