An Analysis of C/C++ Datasets for Machine Learning-Assisted Software Vulnerability Detection

Daniel Grahn *and Junjie Zhang

Wright State University

Abstract

As machine learning-assisted vulnerability detection research matures, it is critical to understand the datasets being used by existing papers. In this paper, we explore 7 C/C++ datasets and evaluate their suitability for machine learning-assisted vulnerability detection. We also present a new dataset, named *Wild C*, containing over 10.3 million individual open-source C/C++ files – a sufficiently large sample to be reasonably considered representative of typical C/C++ code. To facilitate comparison, we tokenize all of the datasets and perform the analysis at this level. We make three primary contributions. First, while all the datasets differ from our *Wild C* dataset, some do so to a greater degree. This includes divergence in file lengths and token usage frequency. Additionally, none of the datasets contain the entirety of the C/C++ vocabulary. These missing tokens account for up to 11% of all token usage. Second, we find all the datasets contain duplication with some containing a significant amount. In the *Juliet* dataset, we describe augmentations of test cases making the dataset susceptible to data leakage. This augmentation occurs with such frequency that a random 80/20 split has roughly 58% overlap of the test with the training data. Finally, we collect and process a large dataset of C code, named *Wild C*. This dataset is designed to serve as a representative sample of all C/C++ code and is the basis for our analyses.

1 Introduction

As long as software has been written, it has contained vulnerabilities. Many vulnerabilities are introduced and removed without exploitation. Some are far more consequential. To avoid exploitation and the accompanying costs and embarrassment of an exposed vulnerability, organizations have adopted a range of risk management measures. You can't fix a vulnerability you don't know exists. Thus, a cornerstone of any cybersecurity risk management is vulnerability detection.

Vulnerability detection has traditionally been a hands-on and resource-intensive process. Manual code reviews divert programmers from fixing bugs, adding new features, and performing other tasks. Source code analysis is prone to false positives that still require manual review. More advanced solutions, such as fuzzing or dynamic analysis can be difficult to set up and similarly resource-intensive. Frequently, 3rd party software is accepted as safe without detailed review.

As with many fields, machine learning offers the tantalizing hope of a future where vulnerability detection is performed, in large part, by intelligent artificial agents. While not yet realized, this promise has motivated large amounts of research in the area. Much of this research is based on a small collection of vulnerability detection datasets within the C/C++ language family. Despite the critical nature of data in machine learning, little attention has been given to the datasets themselves.

*dan.grahn@wright.edu

Proceedings of the Conference on Applied Machine Learning for Information Security, 2021

Table 1: Selected C/C++ Vulnerability Datasets

Dataset	License	Granularity	Compiles?	Cases	# of Vulns	Ref
Big-Vul	MIT	Functions	✗	348 Projects	3,754	[19]
Draper VDISC	CC-BY 4.0	Functions	✗	1.27M Functions	87,804	[54]
IntroClass	BSD	Scripts	✓	6 Assignments	998	[33]
Juliet 1.3	CC0 1.0	Scripts	✓	64,099 Cases	64,099	[9]
ManyBugs	BSD	Projects	✓	5.9M Lines	185	[33]
SVCP4C	GPLv3	Files	✗	2378 Files	9,983	
Taxonomy...	MIT	Scripts	✓	1,164 Cases	873	[29]
Wild C/C++	CC-BY 4.0	Files	✗	12.1M Files	Unknown	

✓ – Yes. ✗ – No.

Vulnerability detection datasets are quite different from other types of machine learning datasets because they require cybersecurity experts to provide labels. Thus, datasets cannot be easily crowd-sourced using tools such as Mechanical Turk and are far more expensive to produce. Many dataset producers have found ways to avoid this problem, but their methods run the risk of introducing biases into the data.

These biases may result in a model that fails to generalize. If the datasets portray a limited view of how C/C++ code is written, they may not understand the full diversity of the language. For example, a natural-language model trained only on the collected works of Dr. Seuss would not be expected to perform well on Shakespeare, Twitter, or any other number of sources. It is these biases and any additional shortcomings that we seek to uncover.

In this paper, we explore 7 vulnerability datasets in the C/C++ language family. These datasets were selected based on their usage and to provide a variety of perspectives on machine learning-assisted vulnerability detection. The datasets can be categorized along two dimensions. The first is granularity or the level at which the information is sampled: functions, files, scripts, and projects. Function-level datasets contain only the signatures and contents of functions. File-level contain the contents of a single file. Unless the file happens to be independent, they are typically not compilable. Scripts are single- or multi-file programs with a single purpose, such as demonstrating a vulnerability. Projects contain the entirety of an application derived from a publicly accessible repository. The second dimension is whether the contents are compilable. Functions and files are typically not compilable while scripts and projects are.

Our paper makes three contributions. First, we analyze the representivity of each of the datasets. We find that datasets drawn from existing code-bases are more representative than hand-crafted datasets. Second, we analyze the duplicativeness of the datasets. We find that all of them contain duplication with some containing a significant amount. Finally, we collect and process a large dataset of C code, named *Wild C*. This dataset is designed to serve as a representative sample of all C/C++ code and is the basis for our analyses.

2 Datasets

2.1 Big-Vul

Big-Vul, published in Fan et al. [19], is available as a repository of scripts and CSV files [19]. The dataset was collected by crawling the Common Vulnerabilities and Exposures (CVE) database [46] and linking the CVEs with open-source GitHub projects. Using commit information, the authors extracted code changes related to the CVE. The resulting CSV files contain extracted

Table 2: Work Referencing Each Dataset

Dataset	**Used By**
Big-Vul	[12, 40, 45]
Draper VDISC	[7, 65, 69, 6, 63]
IntroClass	[76, 74, 48, 31, 3, 28, 32, 27, 60, 24]
Juliet 1.3	[37, 35, 70, 17, 58, 14, 16, 22, 34, 1, 78, 2, 26, 72, 30, 61, 62, 56]
	[13, 41, 25, 47, 42, 20, 75, 36, 66, 4, 38, 73, 63, 5, 57, 15, 53, 67]
ManyBugs	[43, 55, 68, 39, 74, 10, 44, 52, 49, 21, 71]
SVCP4C	[51]
Taxonomy...	[18, 11]

Table 3: Dataset Metrics
A comparison of the datasets to *Wild C*. *Hist. Dist.* contains the energy distance between distributions of file length for each dataset and *Wild C*. Lower values indicate closer distributions. *%* contains the percentage of tokens/bigrams which are missing by count. *Use %* contains the percentage of token/bigrams which are missing weighted by their usage in *Wild C*.

Dataset	**Hist. Dist.**	**Missing Tokens**			**Missing Bigrams**			**Token % Diff**		**Bigram % Diff**	
		Count	%	Use %	Count	%	Use %	Median	Mean	Median	Mean
Big-Vul	0.958	8	6.1	0.002	6,063	74.0	0.055	34.5	48.1	56.3	244.1
Draper VDISC	0.878	2	1.5	0.001	4,788	58.4	0.054	41.5	49.0	68.2	226.3
IntroClass	1.115	92	70.8	11.547	8,066	98.4	42.051	81.4	316.1	94.7	734.9
Juliet	0.651	43	33.1	0.317	7,637	93.2	4.651	82.9	612.2	92.8	1,341.1
ManyBugs	0.219	11	8.5	0.018	5,408	66.0	0.106	50.0	86.0	89.6	2,363.6
SVCP4C	0.459	23	17.7	0.061	6,654	81.2	0.320	40.9	59.7	72.3	498.0
Taxonomy...	1.198	74	56.9	4.954	7,989	97.5	19.326	93.9	432.8	92.4	635.6
		130	**Total Tokens**		8,274	**Total Bigrams in *Wild C***					

functions before and after the commit that fixed the vulnerability. The scripts are included for reproducibility of this process, but we were unable to get them to execute properly. Thankfully, a 10GB CSV containing all of the processed data is available for download.

2.2 SonarCloud Vulnerable Code Prospector for C (SVCP4C)

Raducu et al. [51] take a different approach to collecting vulnerable code. Instead of relying on the existing datasets provided by the NIST or CVE database, it draws from open-source projects whose code is processed using the SonarCloud vulnerability scanner [59]. This is performed directly through the SonarCloud API which allows public access to scrape-friendly vulnerability data. SVCP4C is technically a *tool* for collecting data. However, the authors do provide a dataset in the paper. This is the data that we review. All files in the dataset contain vulnerabilities and comments detailing the vulnerable lines.

2.3 Juliet 1.3

Juliet is the largest hand-created[1] C/C++ vulnerability dataset with entire programs [9]. The dataset is available in C/C++, C#, and Java variants. Each has a large number of test cases, but C/C++ is the largest with 64,099. The test cases are divided by CWE, although some cases contain multiple CWEs. Each test case can be compiled into a separate program or combined into a monolithic binary. Compilation options allow the test cases to be compiled into safe or vulnerable versions with minimal code changes. Some test cases are only compilable on Windows machines, but the majority are cross-platform.

In a brief survey, we found at least 23 papers that used the Juliet dataset directly. Additionally, *Juliet* is a major component of the National Institute of Standards and Technology (NIST) Software Assurance Reference Dataset (SARD) [8]. When large datasets are drawn from the SARD, they are likely relying upon *Juliet* in some way. Because of this prevalence, *Juliet* deserves an extra level of scrutiny.

2.4 ManyBugs & IntroClass

ManyBugs and *IntroClass* are a pair of datasets presented by Le Goues et al. [33]. These datasets are designed to be a benchmark for automated repair methods. *ManyBugs* contains 185 defects across 9 open-source programs. These defects were collected from version control. In total, it has 5.9 million lines of code and 10,000+ test cases. *IntroClass* consists of 998 defects from student submissions of six programming assignments. It includes input/output test cases for each programming assignment.

2.5 A Taxonomy of Buffer Overflows

A Taxonomy of Buffer Overflows is unique because it attempts to create a structured taxonomy of buffer overflows based on 22 attributes. The result is 291 different buffer overflows. For each type, three flawed examples (overflow just outside, somewhat outside, and far outside) and a non-vulnerable version are included. This results in a total of 873 vulnerabilities. Due to the diversity of vulnerabilities in this dataset, it provides a distinctive opportunity for testing a vulnerability detection method against a full range of possibilities. *Taxonomy* is included as part of the NIST SARD.

2.6 Draper Vulnerability Detection in Source Code (VDISC)

The Draper VDISC dataset was produced as part of the Defense Advanced Research Projects Agency's (DARPA) Mining and Understanding Software Enclaves (MUSE) project [54]. To build the dataset the authors collected code from the Debian Linux distribution and public Git repositories from GitHub. They split the code into functions and using a custom minimal lexer then removed duplicate functions. The strict process used by the authors for removing duplicates resulted in only 10.8% of the collected functions being included in the dataset.

The authors labeled the remaining functions by using three open-source static source-code analyzers: Clang, Cppcheck, and Flawfinder. Because each of these tools has disparate outputs, the authors mapped the results into their corresponding CWEs. Despite including code from the Juliet dataset in their internal dataset, the authors do not include it in the publicly released version.

[1] *Juliet is* generated using custom software, but the test cases have been created by hand. The software is not publicly available.

3 Wild C

For this paper, we want to compare the datasets to realistic C/C++ source code. It is beyond the scope of this (or any) paper to collect all C/C++ source code. Instead, we created a dataset named *Wild C* from GitHub repositories [23].

To collect these repositories, we made use of GitHub's public search API using a simple scraping algorithm. At the time of writing, GitHub had limitations on their API that made collection challenging. First, the `search` endpoint we used is rate-limited to $5,000$ requests per hour. This limits the queries to one every 0.72 seconds on average. Because cloning repositories takes some time, we did not encounter this problem in practice. However, a simple solution would be to perform rate-limiting on the client side.

Second, GitHub will only return $1,000$ results per search query. This means that our search queries must be limited to under $1,000$ results. We accomplished this by searching for repositories with less than or equal to a certain number of stars and sorting the results by the number of stars (descending). We then iterate over the search results until we encounter a page ending with a repository starred fewer times than our current search maximum. Instead of requesting another page, we change the search to lower the maximum number of stars.

Using this method, we were able to collect $36,568$ repositories with at least 10 stars each. While there are many repositories with less than 10 stars, we found that they contained far less code and were likely to have a "spike" of commits followed by little-to-no activity. This indicates that most of these repositories are likely to be one-off projects, programming assignments, and similar.

The collected repositories contain $9,068,351$ C and $3,098,624$ C++ files for a total of $12,166,975$ source code files. In addition to using 10 stars as a cutoff to prevent diminishing returns, we also use it as a soft metric to assess approval by outside reviewers. The code collected is effectively a sample of C/C++ that is present in public repositories with some degree of community acceptance. There are a few areas where *Wild C* may not be entirely representative. First, it may favor code that complies with community standards which are strongly encouraged on GitHub. Second, it may favor less buggy code as many of the projects may have active communities. Finally, code in private repositories may differ from public repositories due to the code's functionality being necessarily private or due to the intrinsic privacy of the code. No one sees the hidden bad practices. Despite these potential areas of divergence, we believe that the collection methods and the size of the dataset indicate it is sufficiently close to a truly representative sample of C/C++ for our purposes.

We next extracted tokens from each file. For ease of use, the C/C++ and tokens were packaged into a collection of parquet files. While the dataset is licensed as CC-BY 4.0, the individual source files are licensed under their original repositories. We have released this dataset for public consumption. To the best of our knowledge, it is the first public dataset of C/C++ code and paired tokens of this size.

4 Preprocessing

Comparing the datasets requires that they be in a consistent format. This is a difficult task since they are not available in a standard format. Some datasets contain whole software projects, others single files, others individual functions. Ideally, we would be able to compile all of the files. With compiled files, we could compare their source, assembly, and binary format. However, only a few of the datasets are compilable. Thus, we will limit our comparison to source code.

As the first step, we downloaded the datasets and extracted all code into C-files. This worked best when the datasets already contained whole projects or whole files. When the datasets contained functions, we extracted each function into a separate file. While this results in invalid C-files, it allows us to trace later steps directly to the function.

Table 4: Token CSV Columns

Column	Description
`uuid`	Generated UUID for referencing purposes
`dataset`	Source dataset
`file_name`	Source filename
`token_num`	Index of the token in the file
`char_start`	Character at which the token starts, relative to file start
`char_end`	Character at which the token ends, relative to file start
`token_text`	Raw text of the token
`token_type`	Type of token as specified by the grammar
`channel`	ANTLR internal for handling different categories of input
`line`	Line on which the token starts
`line_char`	Character on which the token starts, relative to line start

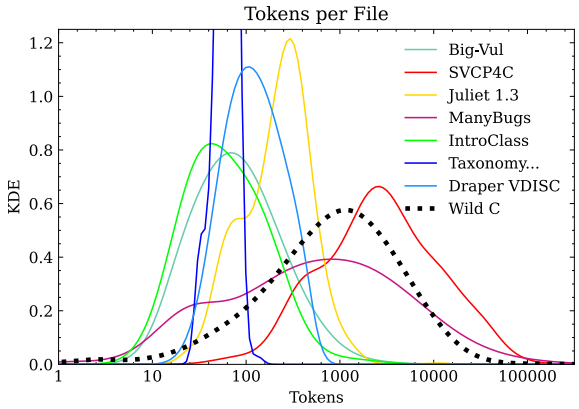

Figure 1: Histogram of tokens per file broken down by dataset using a kernel density estimate. Tokens are present on a log-scale.

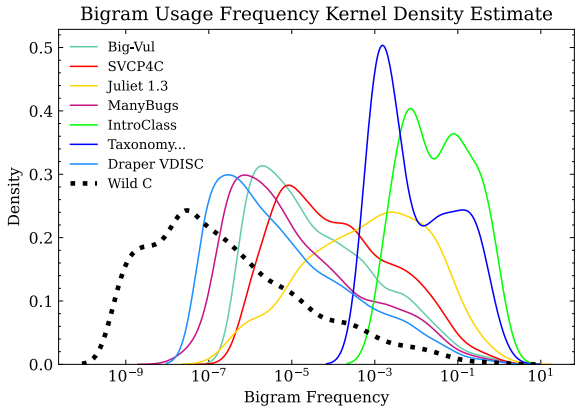

Figure 2: Kernel density estimates for each dataset's bigram usage frequency. Missing bigrams are excluded.

With all the code in C-files, we tokenize the source using *ANother Tool for Language Recognition*, also known as ANTLR [50]. ANTLR is a generic parser generator that has an existing context-free grammar for C. Each of the C-files was converted to tokens in a CSV format. The CSV files contain columns listed in Table 4. These CSV files are the basis for all of the comparisons.

5 Results

5.1 Number of Tokens Per File

For a machine learning model to generalize, the distribution of data should remain consistent from training to inference. The further the distance between these distributions, the less likely the model is to generalize. Our first comparison is the number of tokens per file aggregated for each dataset. In other words, this allows us to compare the file lengths across different datasets. Figure 1 plots the kernel density estimate for each dataset. The x-axis is the number of tokens in a given file and the Y-axis is the estimated density of files that contain the specific number of

Table 5: Top Token Usage Outliers Compared to *Wild C*

	Big-Vul		**Draper VDISC**		**IntroClass**		**Juliet**	
	Type	% Diff	Type	% Diff	Type	% Diff	Type	% Diff
0	`explicit`	425.1	`register`	255.9	%	3200.8	`wchar_t`	34435.5
1	`char16_t`	219.5	`this`	196.0	`AndAnd`	2505.7	`namespace`	3233.5
2	`register`	212.1	`delete`	140.6	/	1203.3	`delete`	2744.1
3	`static_cast`	179.4	`double`	111.1	`else`	645.2	`using`	2041.2
4	::	152.9	`inline`	−100.0	<=	606.5	`Directive`	766.6
5	`const_cast`	115.7	`constexpr`	−99.4	<	528.3	`short`	736.5
6	`new`	100.1	`static_assert`	−99.2	==	450.9	`CharLiteral`	709.6
7	`throw`	−97.9	`decltype`	−97.5	+=	810.3	`do`	704.6
8	`namespace`	−97.5	`char16_t`	−97.0	>=	327.2	`char`	635.7
9	`operator`	−97.3	`protected`	−96.6	`while`	288.4	`new`	571.5
	ManyBugs		**SVCP4C**		**Taxonomy...**		**Merged**	
	Type	% Diff	Type	% Diff	Type	% Diff	Type	% Diff
0	`extern`	2010.3	`CharLiteral`	406.0	`do`	5711.3	`extern`	1434.1
1	`typedef`	574.2	`register`	279.5	`char`	2373.4	`wchar_t`	926.7
2	`wchar_t`	332.0	`char`	190.7	<=	2293.0	`typedef`	397.2
3	`CharLiteral`	322.0	`AndAnd`	185.5	`CharLiteral`	2185.5	`CharLiteral`	284.5
4	<=	298.5	`alignof`	162.3	`while`	1843.8	`register`	253.9
5	`signed`	295.3	!=	160.5	`int`	1176.4	<=	220.6
6	`register`	262.2	`export`	158.1	`typedef`	1054.6	`signed`	199.8
7	`double`	234.3	==	143.4	+=	810.3	`double`	188.1
8	...	191.2	`wchar_t`	142.1	]	695.0	`char`	155.4
9	`char`	165.1	`OrOr`	135.4	[	695.0	...	145.6

tokens. As is evident, the vulnerability datasets are quite different from *Wild C*. We quantified this using energy distance[64] between each histogram and the histogram for *Wild C* and present the results in Table 3 (Hist. Dist. column). While one could hope for better agreement, the results are expected. *ManyBugs* is collected from several large open-source projects, similar to *Wild C*, and has the closest agreement. Conversely, *Taxonomy of Buffer Overflows* contains minimalist examples of buffer overflows which causes a spike in the KDE around 100 tokens per file and the maximum energy distance. While *Juliet* has a better than average distance, it lacks some of the longer files found in *Wild C*.

5.2 Token Usage by Dataset

Next, we compare the total usage of each token by dataset to its usage in *Wild C*. This measures how frequently each dataset uses the tokens. One of the most important observations is that each dataset is missing some of the token types. While *Draper VDISC* misses only two uncommon tokens (`alignas`, `noexcept`), *Taxonomy of Buffer Overflows* misses tokens such as `Not`, `/`, and `this`. *Juliet* misses tokens such as `+=`, `continue`, and `enum`. As shown in Table 3, the datasets such as *Juliet* and *SVCP4C* have a significant number of missing tokens. But these tokens are not frequently used in *Wild C* and thus don't cause a large increase in the *Use %*. To account

for the disparate lengths of the files, the token frequencies were normalized by the most-frequent token. This was `Identifier` for all of the datasets.

Usage of tokens is subject to extreme outliers as shown in Table 5. *IntroClass* uses the `%` token 3,200% more than *Wild C*, *ManyBugs* uses `extern` 2,010% more, and *Taxonomy of Buffer Overflows* uses `do` 5,711% more. However, the furthest outlier belongs to *Juliet* which uses `wchar_t` an astounding 34,435% more than *Wild C*. The `wchar_t` data type is found in 29,264 test cases. A review of *Juliet* indicates that the dramatic increase is likely due to how *Juliet* creates test cases. *Juliet* has many test cases that are near-identical with slight tweaks to their relevant data types. This is explored further in Section 5.4

5.3 Bigram Usage by Dataset

Extending the analysis of token types, we next compare the frequency of usage for bigrams of tokens. Bigrams are commonly used in natural language processing to provide context that individual tokens lack. We continue to normalize by the most frequent bigram per dataset. An upper bound on the total number of bigrams, derived from the 130 tokens, is 16,900. Because many of those bigrams would be invalid in C/C++, we do not have an exact total for the number of *possible* bigrams. In our datasets, we observe 8,195 unique bigrams. These results are shown in Table 3.

The number of bigrams present in *Wild C* that are missing in the datasets is far larger than the number of tokens. *Draper VDISC*, the dataset with the most bigrams, contains only 42.6% of the bigrams in *Wild C*. However, that still represents only 0.054% of bigram usage. Two datasets stand out when compared to each other. *SVCP4C* is missing 81.2% of the bigrams, but these are only used 0.320% of the time. *Juliet* slightly increases the number of missing bigrams to 93.2% but drastically increased the missing usage percentage to 4.651%.

Figure 2 shows the kernel density estimate for the bigram usage of each dataset. From this perspective, we can see a strong dividing line between collected and generated datasets. *Juliet*, *IntroClass*, and *Taxonomy of Buffer Overflows* are all created specifically for vulnerability detection or bug fixing. They have no less than 4.6% missing bigrams and are separated into a cluster of three furthest away from the *Wild C* distribution. *ManyBugs*, *SVCP4C*, *Draper VDISC*, and *Big-Vul* all represent source code drawn from open-source codebases. Each is missing less than 0.5% of bigrams and is closer to the overall distribution of *Wild C*.

Notably, the proximity to *Wild C* appears to be correlated to the size of the dataset. *Draper VDISC* contains 1,274,466 files and is most similar to *Wild C*. It is followed by *ManyBugs* with 223,052 files, *Big-Vul* with 142,937 files, and *SVCP4C* with 11,376 files respectively.

5.4 Juliet Data Leakage Analysis

The augmentation of test cases in *Juliet* has implications for using it or the SARD, of which *Juliet* is a significant subset, as a source for training and test data. Randomly splitting the *Juliet* dataset, regardless of whether stratified by CWE, will introduce data leakage between training and test sets. Consider the task of detecting faces in an image. If the dataset was augmented by changing the hair or eye color of faces, splitting the dataset randomly would cause near-duplicate images to be placed in the test and train datasets. Data leakage of this type could lead to significantly inflated test performance, a failure to generalize, and more.

To evaluate the extent of this potential data leakage, we first identified the test groups. Figure 3 shows the augmentation which was used to build *Juliet*. While there are 100,883 files in *Juliet*, there are approximately 61,000 unique test groups. On average, each test group has 1.64 augmentations (ranging between 1 and 5). The majority of files exist in test groups that contain two files.

With these test groups identified, we performed 500 random splits of the *Juliet* files using a standard 80/20 ratio. For each of these splits, we determined how many files from the test set had

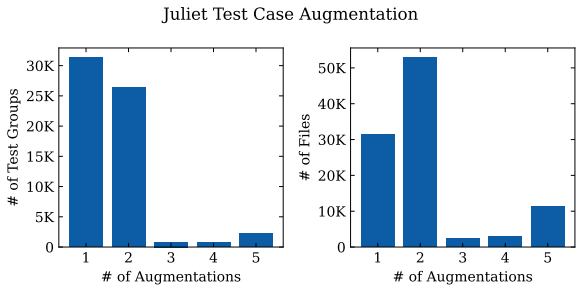

Figure 3: Distribution of augmentations by number of test cases and number of files. Augmentations made pre-train/test split are likely to produce data-leakage.

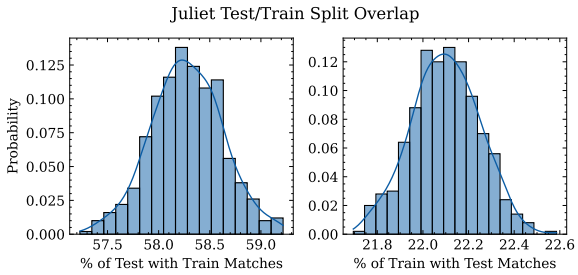

Figure 4: Test and train files with matches in the opposite set. This data leakage is likely to cause overfitting if a model is trained on the *Juliet* dataset without mitigation.

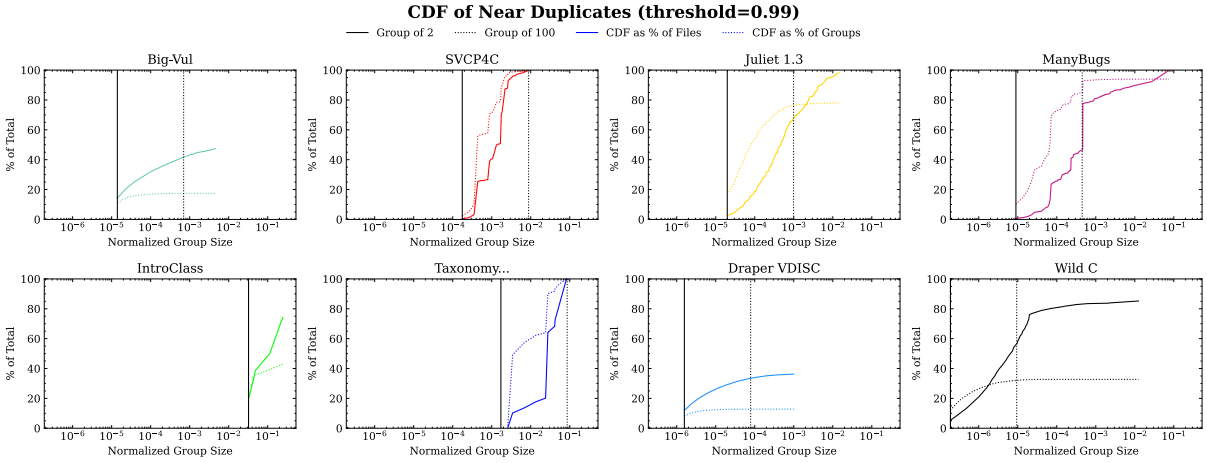

Figure 5: Histogram of near-duplicates based on the number of files in each size group of near-duplicates with vertical bars indicating groups of 2 and 100. *Draper VDISC* and *Big-Vul* exhibit deduplication. Strong duplication is seen in *Juliet*, *ManyBugs*, *SVCP4C*, and *Taxonomy of Buffer Overflows*.

augmentations that existed in the training set and vice versa. Figure 4 shows the distribution of these numbers. Without accounting for the augmentation, splitting the *Juliet* files results in a mean of 58.3% overlap of the test split with the train split and 22.1% of the train split with the test split.

5.5 Near-Duplicate Files

As a final analysis, we measured the number of near-duplicate files in each dataset. While the most precise method for finding near-duplicates would be based on the raw contents of the file, that may miss semantically similar files. To account for this, we again based our near-duplicate detection on the token types. We used MinHash with Locality-Sensitive Hashing from the datasketch library to find near-duplicates with a Jacquard similarity threshold of 0.99 [77].

Near-duplicates come in groups. Most of the time it's not two files that are identical to each other. Rather, there is a group of files that share similar attributes. Analyzing these groups is somewhat complicated. A group of 10 duplicates is far more consequential for a dataset with 100 files than a dataset of 1 million files. However, normalizing by the total number of files in the dataset makes it difficult to determine how many files are in any given group. We attempt to balance these tensions in Figure 5. This figure shows the cumulative density functions (CDF) for the percent of files and percent of groups over as the group size increases. The X-axes contain

Table 6: Near-Duplicate File Information
All datasets exhibit duplication and suffer from data leakage between random test/train splits.

Dataset	Unique Groups	% of Dataset	Test Split	% Test w/Match	% Train w/Match
Big-Vul	91,300	63.87%	0.10	45.84%	23.01%
Draper VDISC	931,804	73.12%	0.01	36.10%	5.29%
IntroClass	28	45.16%	0.20	70.14%	43.27%
Juliet	7,933	7.84%	0.10	98.00%	82.60%
ManyBugs	8,197	3.67%	0.10	99.70%	91.19%
SVCP4C	1,104	9.71%	0.20	99.77%	86.05%
Taxonomy...	61	5.24%	0.20	99.91%	93.66%
Wild C	2,343,364	21.97%	0.01	85.16%	36.25%

the group size presented on a log scale and normalized by the total number of files. Each dataset has the same axes limits and two vertical bars. The solid vertical bar indicates where a group with 2 files would be placed on the X-axis. Similarly, the dotted vertical bar indicates where a group of 100 files would be placed.

Starting with *Wild C*, we can see that there is some amount of duplication in wild source code. This is logical for at least two reasons: (1) programmers frequently share source code that gets copied and remixed; (2) discrete sections of source code are likely to repeat tasks. The largest group of near-duplicate files in *Wild C* has 132,389 files which are constant-definition files. The next largest group only has *25,751* and group sizes reduce quickly from there.

The plot of the number of files for *Draper VDISC* is similar to the number of groups for *Wild C*. This is a clear indication of the efficacy of their duplicate removal process. The difference in raw numbers can be explained by their slightly stricter approach to detecting duplicates. Of note, the plot for *Big-Vul* exhibits similar evidence of deduplication. This deduplication is not mentioned in the paper.

Figure 5 also shows that nearly all of the files from *SVCP4C*, *Juliet*, *ManyBugs*, and *Taxonomy*, and *IntroClass* have at least one duplicate. Our analysis revealed the root cause of this duplication:

- *IntroClass* draws from a limited number of assignments.
- *Taxonomy* is hand generated with intentional duplication to demonstrate good/bad code.
- *ManyBugs* includes multiple copies of the same applications.
- *Juliet 1.3* includes augmentations for many vulnerability examples.

Because *SVCP4C* uses *SonarCloud* vulnerability detection to label their dataset, some amount of duplication is expected. The algorithms used by *SonarCloud* are likely to pick up common patterns within source-code whether they are true or false positives. *Draper VDISC* likely suffered a similar problem with duplication before their duplicate removal process.

Table 6 reverses the analysis and provides the number of "near-unique" files and that number as a percentage of the original dataset. This is analogous to the number of groups of near-duplicates for each dataset. It also provides the mean percentage of test samples with a near-duplicate in the training data and the mean percentage of training samples with a near-duplicate in the test data for 500 random splits of the indicated split size. It's important to note that while the method of calculating the metric is the same as for Section 5.4, the means of identifying duplicates are different. This leads to a metric for *Juliet* than previously provided.

6 Conclusion

6.1 Contributions

Our work makes three significant contributions: (1) analysis of representivity, (2) analysis of duplicativeness, and (3) availability of *Wild C*. Our work shows that there are significant differences between the selected datasets and wild C/C++ code. As a result, some of the datasets may have limited usefulness for machine learning-assisted software vulnerability detection. The *IntroClass* and *Taxonomy of Buffer Overflows* datasets are not well suited for this task. They have over 97% of bigrams missing with a significant portion of those being in common usage. Because of this, it would be possible to have high performance on these datasets while learning less than 68% of the C/C++ language. They also exhibit significant differences in length, token usage, and bigram usage. Based on our analysis, we assess that they do not contain enough diversity to be a thorough test set nor size to be a training set.

Conversely, *Big-Vul*, *SVCP4C*, and *ManyBugs* proved to be reasonably close to *Wild C*. They had among the fewest missing tokens & bigrams and lowest token & bigram usage difference. However, all three had a high degree of duplication and different drawbacks. *Big-Vul* contains only 3,754 vulnerabilities and is not compilable because it only contains functions. While it appears to have been deduplicated, a significant amount of near-duplicates remain. It may be a suitable test dataset if the method uses file- or function-level information, pending further analysis of deduplication.

SVCP4C has only 1,104 unique groups after deduplication, a reduction of 90.29%. The collection method also means that any model trained on *SVCP4C* will be learning to emulate *SonarCloud* rather than learning the ground truth of vulnerabilities. For these two reasons, we recommend future work using *SVCP4C* address duplication and collection biases before usage.

ManyBugs does have a slight edge over *SVCP4C* and *Big-Vul* because it contains entire projects that are compilable. Despite this, it had the most duplication with unique groups making up only 3.67% of the original dataset. A model trained on the dataset as provided may learn to "spot the difference" between projects rather than identifying vulnerabilities. However, *ManyBugs* has potential as a test dataset because it contains large, real-world projects. We recommend that before using *ManyBugs*, the duplication be addressed.

Based on our evaluation of the metrics, *Draper VDISC* appears to be a promising dataset for training and testing machine learning models. It has a permissive license, contains 87,804 vulnerabilities, and has 1.27 million functions. Unfortunately, it is not compilable and is not able to be used with methods that require intermediate, assembly, or binary representations. We have two outstanding concerns regarding the use of this dataset. First, the collection method is similar to *SVCP4C*. In this case, the authors used multiple static analysis tools to identify vulnerabilities and combined the results. Analysis of *SVCP4C* showed that the code identified by *SonarCloud* was very similar. Because the authors of *Draper VDISC* deduplicated their dataset before releasing the dataset, we were unable to analyze the similarity of the dataset before deduplication. It is possible that using the intersection of static analysis tools led to a higher level of duplication. Additionally, any model trained on *Draper VDISC* is ultimately learning the tools rather than the underlying ground truth. Second, our near-duplicate detection identified 26.88% of the dataset as near-duplicates despite the authors performing deduplication. As the authors detail, their deduplication strategy was strict. A near-duplicate detection strategy may lead to a more useful dataset. While we assess that *Draper VDISC* has strong potential, we recommend future work address the above-mentioned concerns.

A significant contribution in this paper is the discussion of test case augmentation within *Juliet*. While others have stated their concerns [65], we believe this is the first empirical analysis of the drawbacks of using *Juliet* as a training and/or test set. Many of the papers making use of *Juliet* or the *NIST SARD* (it's parent dataset) do not address this augmentation or describe steps to remove it [63, 57, 5, 15, 53, 41, 25, 47, 42, 20, 75, 36, 66, 4, 38, 73, 67]. Because of

the high potential for data leakage if augmentations are not removed, we believe the evidence supports using caution when reviewing metrics based on *Juliet* as they may not be an accurate reflection of their accuracy on real-world code. For future work using *Juliet* as a training and/or test dataset, we recommend that appropriate measures be taken to mitigate the potential for data leakage and that those measures be clearly stated to avoid ambiguity.

Finally, we are pleased to provide the *Wild C* dataset to the public. There are a wide variety of potential uses for this dataset. Due to its size and composition, it is suitable as a representative sample of the overall distribution of C/C++ source code. This is a critical factor for our analysis and enables the dataset to be used as a precursor for additional tasks. With some processing, it is possible to extract any file- or function-level information and build a task-specific dataset. Potential tasks include, but are not limited to: comment prediction, function name recommendation, code completion, and variable name recommendation. There is also potential for automatic bug insertion to provide an expanded vulnerability detection dataset. Wild C is available at https://github.com/mla-vd/wild-c.

6.2 Future Work

The are many areas where this work could be expanded. First, we only considered C/C++ datasets. This is the most commonly used language family for machine learning-assisted software vulnerability detection, but it is not the only one. Datasets from other languages exist and deserve similar analysis.

Second, we only compared the datasets in their entirety. Further analysis may compare the difference between the safe and vulnerable subsets of the datasets with each other and wild C/C++ code. While this has the potential to elucidate useful differences between safe and vulnerable code, more likely it will further highlight the problems with the existing datasets. Additionally, further work is needed to determine how much deduplication of *Big-Vul*, *SVCP4C*, and *ManyBugs* would reduce the number of vulnerabilities each.

Perhaps the most pressing need from future research is the creation of vulnerability-detection benchmarks. *Juliet* has been used for this purpose in previous papers, but our analysis brings that usage into question. Given the diversity of the dataset types among those selected (e.g., files, functions, programs) it is unlikely that a single dataset could serve as a universal *training* dataset similar to those available for computer vision tasks. This does not mean that a benchmark is infeasible. Such a benchmark should meet at least five requirements. (1) It must be drawn from real-world code. As illustrated in Section 5.3, there is a distinct and quantifiable difference between synthetic and natural code. The barriers to labeling real-world code are likely far lower than bringing synthetic code into the real-world distribution of usage. (2) It must be compilable. This will enable it to support methods that work on assembly, binaries, or otherwise require compilable code. (3) It should exercise a sufficient diversity of C/C++. This will allow the dataset to avoid issues with missing tokens/bigrams and ensure that the model understands the language. Further testing is needed to determine how much diversity is necessary. (4) It should be difficult enough to act as a viable benchmark. A benchmark that is too easy will quickly outlive its usefulness. This difficulty should not only include the depth and likelihood of a vulnerability but the amount of code “noise" surrounding the vulnerabilities. (5). It should be deduplicated. As shown in Section 5.5, even *Wild C* is subject to a large degree of duplication. This should be removed to ensure that the model isn't biased towards the features present in duplicates.

Given the limited datasets that exist today, much work is needed in the field of machine learning-assisted vulnerability detection. While the methods being applied to the datasets are promising, we assess that the limiting factor may be the datasets themselves. However, many machine learning tasks seemed out of reach just a few years ago. Machine learning researchers have performed astounding tasks in many areas and we expect to count this as one in the future.

References

[1] E. Alikhashashneh, R. Raje, and J. Hill. Using software engineering metrics to evaluate the quality of static code analysis tools. In *2018 1st International Conference on Data Intelligence and Security (ICDIS)*, pages 65–72. IEEE, 2018.

[2] R. Amankwah, J. Chen, A. A. Amponsah, P. K. Kudjo, V. Ocran, and C. O. Anang. Fast bug detection algorithm for identifying potential vulnerabilities in juliet test cases. In *2020 IEEE 8th International Conference on Smart City and Informatization (iSCI)*, pages 89–94, 2020. doi: 10.1109/iSCI50694.2020.00021.

[3] L. A. Amorim, M. F. Freitas, A. Dantas, E. F. de Souza, C. G. Camilo-Junior, and W. S. Martins. A new word embedding approach to evaluate potential fixes for automated program repair. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2018.

[4] S. Arakelyan, C. Hauser, E. Kline, and A. Galstyan. Towards learning representations of binary executable files for security tasks. *arXiv:2002.03388 [cs, stat]*, 2020.

[5] S. Arakelyan, S. Arasteh, C. Hauser, E. Kline, and A. Galstyan. Bin2vec: learning representations of binary executable programs for security tasks. *Cybersecurity*, 4(1):1–14, 2021.

[6] Z. Bilgin. Code2image: Intelligent code analysis by computer vision techniques and application to vulnerability prediction. *arXiv preprint arXiv:2105.03131*, 2021.

[7] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay. Vulnerability prediction from source code using machine learning. *IEEE Access*, 8:150672–150684, 2020.

[8] P. Black. A software assurance reference dataset: Thousands of programs with known bugs, 2018.

[9] P. E. Black and P. E. Black. *Juliet 1.3 Test Suite: Changes From 1.2*. US Department of Commerce, National Institute of Standards and Technology, 2018.

[10] S. Chakraborty, Y. Li, M. Irvine, R. Saha, and B. Ray. Entropy guided spectrum based bug localization using statistical language model. *arXiv preprint arXiv:1802.06947*, 2018.

[11] J. Chen and X. Mao. Bodhi: Detecting buffer overflows with a game. In *2012 IEEE Sixth International Conference on Software Security and Reliability Companion*, pages 168–173, 2012. doi: 10.1109/SERE-C.2012.35.

[12] Z. Chen, S. Kommrusch, and M. Monperrus. Neural transfer learning for repairing security vulnerabilities in c code. *arXiv preprint arXiv:2104.08308*, 2021.

[13] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Trans. Softw. Eng. Methodol.*, 30 (3), Apr. 2021. ISSN 1049-331X. doi: 10.1145/3436877. URL `https://doi-org.ezproxy.libraries.wright.edu/10.1145/3436877`.

[14] M.-j. Choi, S. Jeong, H. Oh, and J. Choo. End-to-end prediction of buffer overruns from raw source code via neural memory networks. *arXiv preprint arXiv:1703.02458*, 2017.

[15] R. Croft, D. Newlands, Z. Chen, and M. A. Babar. An empirical study of rule-based and learning-based approaches for static application security testing. *arXiv preprint arXiv:2107.01921*, 2021.

[16] B. H. Dang. A practical approach for ranking software warnings from multiple static code analysis reports. In *2020 SoutheastCon*, volume 2, pages 1–7. IEEE, 2020.

[17] R. Demidov and A. Pechenkin. Application of siamese neural networks for fast vulnerability detection in mips executable code. In *Proceedings of the Future Technologies Conference*, pages 454–466. Springer, 2019.

[18] S. Dhumbumroong and K. Piromsopa. Boundwarden: Thread-enforced spatial memory safety through compile-time transformations. *Science of Computer Programming*, 198, 2020.

[19] J. Fan, Y. Li, S. Wang, and T. N. Nguyen. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 508–512, 2020.

[20] H. Feng, X. Fu, H. Sun, H. Wang, and Y. Zhang. Efficient vulnerability detection based on abstract syntax tree and deep learning. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 722–727, 2020. doi: 10.1109/INFOCOMWKSHPS50562.2020.9163061.

[21] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–27, 2021.

[22] Y. Gao, L. Chen, G. Shi, and F. Zhang. A comprehensive detection of memory corruption vulnerabilities for c/c++ programs. In *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pages 354–360. IEEE, 2018.

[23] github. Github, 2020. URL `https://github.com/`.

[24] T. Helmuth. *General program synthesis from examples using genetic programming with parent selection based on random lexicographic orderings of test cases*. PhD thesis, University of Massachusetts Amherst, 2015.

[25] S. Jeon and H. K. Kim. Autovas: An automated vulnerability analysis system with a deep learning approach. *Computers & Security*, 106:102308, 2021.

[26] J. Kang and J. H. Park. A secure-coding and vulnerability check system based on smart-fuzzing and exploit. *Neurocomputing*, 256:23–34, 2017. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2015.11.139. URL `https://www.sciencedirect.com/science/article/pii/S0925231217304113`. Fuzzy Neuro Theory and Technologies for Cloud Computing.

[27] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 295–306. IEEE, 2015.

[28] A. Koyuncu. *Boosting Automated Program Repair for Adoption By Practitioners*. PhD thesis, University of Luxembourg, Luxembourg, 2020.

[29] K. Kratkiewicz and R. Lippmann. A taxonomy of buffer overflows for evaluating static and dynamic software testing tools. In *Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics*, volume 500, pages 44–51, 2006.

[30] J. A. Kupsch, E. Heymann, B. Miller, and V. Basupalli. Bad and good news about using software assurance tools. *Software: Practice and Experience*, 47(1):143–156, 2017. doi: https://doi.org/10.1002/spe.2401. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2401`.

[31] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 593–604, 2017.

[32] X. B. D. Le, F. Thung, D. Lo, and C. Le Goues. Overfitting in semantics-based automated program repair. *Empirical Software Engineering*, 23(5):3007–3033, 2018.

[33] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

[34] J. Lee, S. Hong, and H. Oh. Memfix: static analysis-based repair of memory deallocation errors for c. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 95–106, 2018.

[35] Y. Lee, H. Kwon, S.-H. Choi, S.-H. Lim, S. H. Baek, and K.-W. Park. Instruction2vec: efficient preprocessor of assembly code to detect software weakness with cnn. *Applied Sciences*, 9(19):4086, 2019.

[36] Y. J. Lee, S.-H. Choi, C. Kim, S.-H. Lim, and K.-W. Park. Learning binary code with deep learning to detect software weakness. In *KSII The 9th International Conference on Internet (ICONI) 2017 Symposium*, 2017.

[37] H. Li, H. Kwon, J. Kwon, and H. Lee. A scalable approach for vulnerability discovery based on security patches. In *International Conference on Applications and Techniques in Information Security*, pages 109–122. Springer, 2014.

[38] Y. Li, S. Ji, C. Lv, Y. Chen, J. Chen, Q. Gu, and C. Wu. V-fuzz: Vulnerability-oriented evolutionary fuzzing. *arXiv:1901.01142 [cs]*, 2019.

[39] Y. Li, S. Wang, and T. N. Nguyen. Fault localization with code coverage representation learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 661–673. IEEE, 2021.

[40] Y. Li, S. Wang, and T. N. Nguyen. Vulnerability detection with fine-grained interpretations. *arXiv preprint arXiv:2106.10478*, 2021.

[41] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[42] G. Lin, W. Xiao, J. Zhang, and Y. Xiang. Deep learning-based vulnerable function detection: A benchmark. In *International Conference on Information and Communications Security*, pages 219–232. Springer, 2019.

[43] T. Lutellier. *Machine Learning for Software Dependability.* PhD thesis, University of Waterloo, 2020.

[44] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114, 2020.

[45] M. J. Michl. *Analyse sicherheitsrelevanter Designfehler in Software hinsichtlich einer Detektion mittels Künstlicher Intelligenz*. PhD thesis, Technische Hochschule, 2021.

[46] C. Mitre. Common vulnerabilities and exposures, 2005.

[47] H. N. Nguyen, S. Teerakanok, A. Inomata, and T. Uehara. The comparison of word embedding techniques in rnns for vulnerability detection. *ICISSP 2021*, 2021.

[48] V. P. L. Oliveira, E. F. Souza, C. Le Goues, and C. G. Camilo-Junior. Improved crossover operators for genetic programming for program repair. In *International Symposium on Search Based Software Engineering*, pages 112–127. Springer, 2016.

[49] V. P. L. Oliveira, E. F. de Souza, C. Le Goues, and C. G. Camilo-Junior. Improved representation and genetic operators for linear genetic programming for automated program repair. *Empirical Software Engineering*, 23(5):2980–3006, 2018.

[50] T. Parr. Antlr, 2021. URL `https://www.antlr.org/`.

[51] R. Raducu, G. Esteban, F. J. Rodríguez Lera, and C. Fernández. Collecting vulnerable source code from open-source repositories for dataset generation. *Applied Sciences*, 10(4): 1270, 2020.

[52] J. Renzullo, W. Weimer, and S. Forrest. Multiplicative weights algorithms for parallel automated software repair. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 984–993. IEEE, 2021.

[53] A. Ribeiro, P. Meirelles, N. Lago, and F. Kon. Ranking warnings from multiple source code static analyzers via ensemble learning. In *Proceedings of the 15th International Symposium on Open Collaboration*, pages 1–10, 2019.

[54] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE, 2018.

[55] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 648–659. IEEE, 2017.

[56] M. Saletta and C. Ferretti. A neural embedding for source code: Security analysis and cwe lists. In *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, pages 523–530, 2020. doi: 10.1109/DASC-PICom-CBDCom-CyberSciTech49142.2020.00095.

[57] A. Savchenko, O. Fokin, A. Chernousov, O. Sinelnikova, and S. Osadchyi. Deedp: vulnerability detection and patching based on deep learning. *Theoretical and Applied Cybersecurity*, 2, 2020.

[58] C. D. Sestili, W. S. Snavely, and N. M. VanHoudnos. Towards security defect prediction with ai. *arXiv preprint arXiv:1808.09897*, 2018.

[59] SonarSource. Sonarcloud, 2008. URL `https://sonarcloud.io/`.

[60] E. Soremekun, L. Kirschner, M. Böhme, and A. Zeller. Locating faults with program slicing: an empirical analysis. *Empirical Software Engineering*, 26(3):1–45, 2021.

[61] G. Stergiopoulos, P. Petsanas, P. Katsaros, and D. Gritzalis. Automated exploit detection using path profiling: The disposition should matter, not the position. In *2015 12th International Joint Conference on e-Business and Telecommunications (ICETE)*, volume 04, pages 100–111, 2015.

[62] G. Stergiopoulos, P. Katsaros, and D. Gritzalis. Execution path classification for vulnerability analysis and detection. *E-Business and Telecommunications. ICETE 2015. Communications in Computer and Information Science*, 585, 2016.

[63] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, and A. Morari. Learning to map source code to software vulnerability using code-as-a-graph. *arXiv preprint arXiv:2006.08614*, 2020.

[64] G. Szekely. E-statistics: The energy of statistical samples. *Preprint*, 01 2003.

[65] A. Tanwar, K. Sundaresan, P. Ashwath, P. Ganesan, S. K. Chandrasekaran, and S. Ravi. Predicting vulnerability in large codebases with deep code representation. *arXiv preprint arXiv:2004.12783*, 2020.

[66] A. Tanwar, H. Manikandan, K. Sundaresan, P. Ganesan, Sathish, and S. Ravi. Multi-context attention fusion neural network for software vulnerability identification. *arXiv pre-print server*, 2021. doi: Nonearxiv:2104.09225. URL `https://arxiv.org/abs/2104.09225`.

[67] J. Tian, J. Zhang, and F. Liu. Bbreglocator: A vulnerability detection system based on bounding box regression. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 93–100. IEEE, 2021.

[68] L. Trujillo, O. M. Villanueva, and D. E. Hernandez. A novel approach for search-based program repair. *IEEE Software*, 38(4):36–42, 2021.

[69] J.-A. Văduva, I. Culi, A. RADOVICI, R. RUGHINIS, and S. DASCALU. Vulnerability analysis pipeline using compiler based source to source translation and deep learning. *eLearning & Software for Education*, 1, 2020.

[70] N. Visalli, L. Deng, A. Al-Suwaida, Z. Brown, M. Joshi, and B. Wei. Towards automated security vulnerability and software defect localization. In *2019 IEEE 17th International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 90–93. IEEE, 2019.

[71] W. Weimer, J. Davidson, S. Forrest, C. Le Goues, P. Pal, and E. Smith. Trusted and resilient mission operations. Technical report, University of Michigan Ann Arbor United States, 2020.

[72] E. C. Wikman. Static analysis tools for detecting stack-based buffer overflows. Master's thesis, Naval Postgraduate School, 2020.

[73] Y. Wu, J. Lu, Y. Zhang, and S. Jin. Vulnerability detection in c/c++ source code with graph representation learning. In *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 1519–1524, 2021. doi: 10.1109/CCWC51732.2021.9376145.

[74] Y. Xu, B. Huang, X. Zou, and L. Kong. Predicting effectiveness of generate-and-validate patch generation systems using random forest. *Wuhan University Journal of Natural Sciences*, 23(6):525–534, 2018.

[75] H. Yan, S. Luo, L. Pan, and Y. Zhang. Han-bsvd: a hierarchical attention network for binary software vulnerability detection. *Computers & Security*, page 102286, 2021. ISSN 0167-4048. doi: 10.1016/j.cose.2021.102286. URL `https://dx.doi.org/10.1016/j.cose.2021.102286`.

[76] G. Yang, Y. Jeong, K. Min, J.-w. Lee, and B. Lee. Applying genetic programming with similar bug fix information to automatic fault repair. *Symmetry*, 10(4):92, 2018.

[77] E. Zhu, V. Markovtsev, aastafiev, W. Łukasiewicz, ae foster, J. Martin, Ekevoo, K. Mann, K. Joshi, S. Thakur, S. Ortolani, Titusz, V. Letal, Z. Bentley, and fpug. ekzhu/datasketch: Improved performance for MinHash and MinHashLSH, Dec. 2020. URL `https://doi.org/10.5281/zenodo.4323502`.

[78] K. Zhu, Y. Lu, and H. Huang. Scalable static detection of use-after-free vulnerabilities in binary code. *IEEE Access*, 8:78713–78725, 2020.