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

Daniel Grahn  
*Wright State University - Main Campus*, dan.grahn@wright.edu

Junjie Zhang  
*Wright State University - Main Campus*, junjie.zhang@wright.edu

Follow this and additional works at: https://corescholar.libraries.wright.edu/cse

.repository_citation

https://corescholar.libraries.wright.edu/cse/605

This Article is brought to you for free and open access by Wright State University's CORE Scholar. It has been accepted for inclusion in Computer Science and Engineering Faculty Publications by an authorized administrator of CORE Scholar. For more information, please contact library-corescholar@wright.edu.
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

Copyright © 2021 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
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/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 [16] 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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Used By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-Vul</td>
<td>[12, 10, 15]</td>
</tr>
<tr>
<td>Draper VDISC</td>
<td>[7, 65, 69, 6, 63]</td>
</tr>
<tr>
<td>IntroClass</td>
<td>[76, 74, 48, 31, 3, 28, 32, 27, 60, 24]</td>
</tr>
<tr>
<td>Juliet 1.3</td>
<td>[37, 35, 70, 17, 58, 14, 16, 22, 34, 1, 78, 2, 26, 72, 30, 61, 62, 56]</td>
</tr>
<tr>
<td>ManyBugs</td>
<td>[83, 68, 39, 74, 10, 44, 52, 49, 21, 71]</td>
</tr>
<tr>
<td>SVCP4C</td>
<td>[51]</td>
</tr>
<tr>
<td>Taxonomy...</td>
<td>[18, 11]</td>
</tr>
</tbody>
</table>

Table 3: Dataset Metrics

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Hist. Dist.</th>
<th>Missing Tokens</th>
<th>Missing Bigrams</th>
<th>Token % Diff</th>
<th>Bigram % Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Count</td>
<td>%</td>
<td>Use %</td>
<td>Count</td>
</tr>
<tr>
<td>Big-Vul</td>
<td>0.958</td>
<td>8</td>
<td>6.1</td>
<td>0.002</td>
<td>6,063</td>
</tr>
<tr>
<td>Draper VDISC</td>
<td>0.878</td>
<td>2</td>
<td>1.5</td>
<td>0.001</td>
<td>4,788</td>
</tr>
<tr>
<td>IntroClass</td>
<td>1.115</td>
<td>92</td>
<td>70.8</td>
<td>11.547</td>
<td>8,066</td>
</tr>
<tr>
<td>Juliet</td>
<td>0.651</td>
<td>43</td>
<td>33.1</td>
<td>0.317</td>
<td>7,637</td>
</tr>
<tr>
<td>ManyBugs</td>
<td>0.219</td>
<td>11</td>
<td>8.5</td>
<td>0.018</td>
<td>5,408</td>
</tr>
<tr>
<td>SVCP4C</td>
<td>0.459</td>
<td>23</td>
<td>17.7</td>
<td>0.061</td>
<td>6,654</td>
</tr>
<tr>
<td>Taxonomy...</td>
<td>1.198</td>
<td>74</td>
<td>56.9</td>
<td>4.954</td>
<td>7,989</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Count</th>
<th>Total Tokens</th>
<th>Total Bigrams in Wild C</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVCP4C</td>
<td>130</td>
<td>8,274</td>
<td>2,363.6</td>
</tr>
</tbody>
</table>

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 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

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

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
tokens. As is evident, the vulnerability datasets are quite different from Wild C. We quantified
this using energy distance\cite{64} between each histogram and the histogram for Wild C and present
the results in Table 2 (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

<table>
<thead>
<tr>
<th>Big-Vul</th>
<th>Draper VDISC</th>
<th>IntroClass</th>
<th>Juliet</th>
</tr>
</thead>
<tbody>
<tr>
<td>explicit</td>
<td>425.1</td>
<td>register</td>
<td>255.9</td>
</tr>
<tr>
<td>char16_t</td>
<td>219.5</td>
<td>this</td>
<td>196.0</td>
</tr>
<tr>
<td>register</td>
<td>212.1</td>
<td>delete</td>
<td>140.6</td>
</tr>
<tr>
<td>static_cast</td>
<td>179.4</td>
<td>double</td>
<td>111.1</td>
</tr>
<tr>
<td>::</td>
<td>152.9</td>
<td>inline</td>
<td>-100.0</td>
</tr>
<tr>
<td>const_cast</td>
<td>115.7</td>
<td>constexpr</td>
<td>-99.4</td>
</tr>
<tr>
<td>new</td>
<td>100.1</td>
<td>static_assert</td>
<td>-99.2</td>
</tr>
<tr>
<td>throw</td>
<td>-97.9</td>
<td>decltype</td>
<td>-97.5</td>
</tr>
<tr>
<td>namespace</td>
<td>-97.5</td>
<td>char16_t</td>
<td>-97.0</td>
</tr>
<tr>
<td>operator</td>
<td>-97.3</td>
<td>protected</td>
<td>-96.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ManyBugs</th>
<th>SVCP4C</th>
<th>Taxonomy...</th>
<th>Merged</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern</td>
<td>2010.3</td>
<td>CharLiteral</td>
<td>406.0</td>
</tr>
<tr>
<td>typedef</td>
<td>574.2</td>
<td>register</td>
<td>279.5</td>
</tr>
<tr>
<td>wchar_t</td>
<td>332.0</td>
<td>char</td>
<td>190.7</td>
</tr>
<tr>
<td>CharLiteral</td>
<td>322.0</td>
<td>AndAnd</td>
<td>185.5</td>
</tr>
<tr>
<td>&lt;=</td>
<td>298.5</td>
<td>alignof</td>
<td>162.3</td>
</tr>
<tr>
<td>signed</td>
<td>295.3</td>
<td>!=</td>
<td>160.5</td>
</tr>
<tr>
<td>register</td>
<td>262.2</td>
<td>export</td>
<td>158.1</td>
</tr>
<tr>
<td>double</td>
<td>234.3</td>
<td>==</td>
<td>143.4</td>
</tr>
<tr>
<td>...</td>
<td>191.2</td>
<td>wchar_t</td>
<td>142.1</td>
</tr>
<tr>
<td>char</td>
<td>165.1</td>
<td>OrOr</td>
<td>135.4</td>
</tr>
</tbody>
</table>

Table 5: Top Token Usage Outliers Compared to Wild C
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
An Analysis of C/C++ Datasets for MLAVD

Daniel Grahn, Junjie Zhang

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.

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 [17].

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.

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

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 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, 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.

An analysis of C/C++ datasets for MLAVD Daniel Grahn, Junjie Zhang
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


