Evaluating Biased Synthetic Data Effects on Large Language Model-Based Software Vulnerability Detection

Lucas B. Germano[a], Lincoln Q. Vieira[b], Ronaldo R. Goldschmidt[c], Julio Cesar Duarte[d] and Ricardo Choren[e]

Military Institute of Engineering, Brazil

Keywords: Data Preprocessing, Deep Learning, Large Language Models, Synthetic Vulnerability Dataset, Vulnerability Detection.

Abstract: Software security ensures data privacy and system reliability. Vulnerabilities in the development cycle can lead to privilege escalation, causing data exfiltration or denial of service attacks. Static code analyzers, based on predefined rules, often fail to detect errors beyond these patterns and suffer from high false positive rates, making rule creation labor-intensive. Machine learning offers a flexible alternative, which can use extensive datasets of real and synthetic vulnerability data. This study examines the impact of bias in synthetic datasets on model training. Using CodeBERT for C/C++ vulnerability classification, we compare models trained on biased and unbiased data, incorporating overlooked preprocessing steps to remove biases. Results show that the unbiased model achieves 98.5% accuracy, compared to 63.0% for the biased model, emphasizing the critical need to address dataset biases in training.

1 INTRODUCTION

Maintaining software security is crucial for ensuring data privacy and system reliability. Vulnerabilities introduced during the software development life cycle can enable intruders to escalate privileges, leading to data breaches and service disruptions for companies and public agencies. Static and dynamic code analysis tools have been developed to address these security challenges. However, static analyzers rely on predefined rules, which often fail to detect vulnerabilities that deviate from expected patterns and generate high false-positive rates. Additionally, as noted by (Li et al., 2018), defining these rules requires extensive manual effort, prone to errors due to the complexity of language syntax and library behavior changes.

Machine learning offers greater flexibility in error detection but remains under development for achieving satisfactory performance levels. The recent rise of Large Language Models (LLMs), powered by the Transformer architecture (Vaswani et al., 2017), has sparked interest in their application to software vulnerability detection, particularly for C and C++.

To train vulnerability detection models, various datasets have been developed, incorporating both real vulnerability data and artificial test cases. One prominent dataset in this field is the Software Assurance Reference Dataset (SARD) (NIST, 2021), which contains synthetic test cases. Several studies have utilized SARD to develop and validate their models (Li et al., 2018; Li et al., 2022; Lin et al., 2022; Zeng et al., 2023; Huang et al., 2022; Nong et al., 2024).

Although SARD is a widely used dataset, its synthetic nature introduces certain patterns from the algorithms used to generate the data. These patterns are often subtle and not immediately noticeable. The purpose of this work is to show how such patterns may actually compromise model performance, potentially leading to skewed predictions.

(Mehrabi et al., 2021) defines the "User to Data" bias, which arises when data sources are user-generated, reflecting inherent user biases. Similarly, when algorithms are used to create synthetic test cases, biases embedded in those algorithms can further influence the data generation process. (Mehrabi et al., 2021) notes that when training data are biased, models trained on them tend to internalize and propagate these biases in their predictions.

[a] https://orcid.org/0009-0007-1607-4863
[b] https://orcid.org/0009-0002-7959-6064
[c] https://orcid.org/0000-0003-1688-0586
[d] https://orcid.org/0000-0001-6656-1247
[e] https://orcid.org/0000-0003-4081-2647

Germano, L. B., Vieira, L. Q., Goldschmidt, R. R., Duarte, J. C. and Choren, R.
Evaluating Biased Synthetic Data Effects on Large Language Model-Based Software Vulnerability Detection.
DOI: 10.5220/0013156800003890

In *Proceedings of the 17th International Conference on Agents and Artificial Intelligence (ICAART 2025) - Volume 3*, pages 504-511
ISBN: 978-989-758-737-5; ISSN: 2184-433X

It is clear that, as a synthetic vulnerability dataset, SARD may contain biases introduced by the algorithms used to generate its test cases. (Barbierato et al., 2022) defines bias as the influence that certain data elements or variables may exert on other elements in a given dataset. However, they also indicate that a more specific definition can vary depending on the context in which the bias is being analyzed.

In this work, a biased synthetic dataset for vulnerability detection is a collection of training data containing specific patterns or keywords that unintentionally guide the model to rely on superficial cues instead of understanding the logic behind vulnerabilities. For example, if function names like "bad" or "good" appear in vulnerable code, the model may associate these terms with vulnerability presence rather than analyzing the actual code logic. Such biases result in incorrect predictions when these patterns are absent in real-world code, reducing the model's ability to generalize and effectively detect vulnerabilities.

In this context, this work has two objectives: (i) to identify and show some biases present in the SARD dataset, and; (ii) to create a new dataset derived from the SARD dataset, which has been adjusted to eliminate these biases. The main contributions of this work are:

1. Specification of a list of existing biases in the SARD dataset;
2. Development of an approach for processing data available in SARD to remove the identified biases;
3. Execution of experiments comparing the training results of an LLM using the original SARD dataset versus the bias-free version, showcasing improvements in vulnerability detection; and
4. Provision of a bias-free SARD dataset [1].

The experiment conducted in this work utilizes the Juliet C/C++ 1.3.1 project, available through SARD, alongside the LLM CodeBERT (Feng et al., 2020) to classify source code for the presence of vulnerabilities in C/C++ languages. The results show that a model trained on the biased dataset performed significantly worse when tested on the bias-free dataset, achieving an accuracy of only 63.0%. In contrast, the model trained on the bias-free dataset achieved a much higher accuracy of 98.6% when tested on the biased dataset, indicating that this model effectively learned to detect vulnerabilities without relying on superficial cues.

The remainder of this paper is organized as follows. Section 2 describes the proposed approach for data preprocessing. Section 3 identifies the different biases found in the dataset, while Section 4 discusses the experiment and results of the comparison between biased and unbiased models. Section 5 reviews related work in the area. Finally, Section 6 concludes the paper.

2 DATA PREPROCESSING APPROACH

SARD, developed by the National Institute of Standards and Technology (NIST), is a vulnerability dataset containing collections in C, C++, C#, Java, and PHP. This work uses the Juliet C/C++ 1.3 dataset, which includes 64,099 vulnerabilities and their fixes, totaling 128,198 files. The dataset, available online[2], covers 116 types of Common Weakness Enumeration (CWE) vulnerabilities in C and C++. It is perfectly class-balanced, with half of the cases labeled as vulnerable and the other half as non-vulnerable.

This section outlines biases present in SARD that may significantly impact vulnerability detection models and distort their results. The data processing workflow is illustrated in Figure 1.

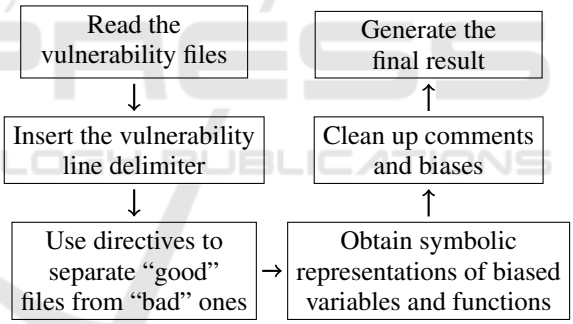

Figure 1: Data preprocessing flowchart.

Read the Vulnerability Files. Each test case in the dataset includes both an implementation containing a vulnerability and the corresponding corrected version, as depicted in Figure 1.

Insert the Vulnerability Line Delimiter. CodeBERT's 512-token context window limits its ability to process longer files in a single inference. To address this, files are divided into 512-token segments, or "chunks." As the dataset specifies the line containing the vulnerability, a delimiter is inserted to mark its location. This approach enabled the identification of the specific chunk that contains the vulnerability when the code is divided.

[1] https://github.com/lucasg1/sard_dataset_without_bias

[2] https://samate.nist.gov/SARD/test-suites/112

```
#ifndef OMITBAD

/* bad function declaration */
void CWE121_Stack_Based_Buffer_Overflow_dest_char
      _alloca_cat_51b_badSink(char * data);

void CWE121_Stack_Based_Buffer_Overflow_dest_char
      _alloca_cat_51_bad ()
{
    char * data;
    char * dataBadBuffer = (char *)ALLOCA(50*sizeof(char));
    char * dataGoodBuffer = (char *)ALLOCA(100*sizeof(char)
         );
    /* FLAW: Set a pointer to a "small" buffer. This buffer
          will be used in the sinks as a destination
     * buffer in various memory copying functions using a "
          large" source buffer. */
    data = dataBadBuffer;
    data[0] = '\0'; /* null terminate */
    CWE121_Stack_Based_Buffer_Overflow_dest_char
         _alloca_cat_51b_badSink (data);
}

#endif /* OMITBAD */
```

```
#ifndef OMITGOOD

/* good function declarations */
void CWE121_Stack_Based_Buffer_Overflow_dest_char
      _alloca_cat_51b_goodG2BSink (char * data);

/* goodG2B uses the GoodSource with the BadSink */
static void goodG2B()
{
    char * data;
    char * dataBadBuffer = (char *)ALLOCA(50*sizeof(char));
    char * dataGoodBuffer = (char *)ALLOCA(100*sizeof(char)
         );
    /* FIX: Set a pointer to a "large" buffer, thus
         avoiding buffer overflows in the sinks. */
    data = dataGoodBuffer;
    data[0] = '\0'; /* null terminate */
    CWE121_Stack_Based_Buffer_Overflow__dest_char
         _alloca_cat_51b_goodG2BSink(data);
}

void CWE121_Stack_Based_Buffer_Overflow__dest_char
      _alloca_cat_51_good()
{
    goodG2B();
}

#endif /* OMITGOOD */
```

Listing 1: Example of original code from the Juliet dataset, with excerpts taken from the same file.

Use Directives to Separate "Good" Files from "Bad" Files. SARD data uses directives to differentiate between "good" (non-vulnerable) and "bad" (vulnerable) files. Listing 1 shows examples of these directives, which indicate where code is vulnerable and where it is fixed. Such directives are used to create two files for each test case: one containing the vulnerability, marked with `#ifndef OMITBAD`, and another without it, marked with `#ifndef OMITGOOD`.

Obtain Symbolic Representations of Biased Variables and Functions. Biased variables and functions are code elements with names that provide unintended clues, potentially skewing the learning process. In the SARD dataset, as shown in Listing 1, names like "good", "bad", and "cwe" indicate vulnerability presence or absence. For example, "cwe" may appear in variables named after their CWE classification, offering hints about vulnerabilities in the code.

To address this, biased functions and variables are renamed using the formats `FUN#` and `VAR#`, respectively, while the rest of the code remains unchanged. The renaming process was automated using the Python library clang.

Clean up Comments and Biases. The comments are removed because they explicitly indicate where the vulnerabilities exist, as shown in Listing 1. Furthermore, this work detected several patterns that previous works overlooked. These patterns introduce biases that interfere with the learning process of models that use this dataset. The first pattern involves using `static void` functions, which appear in 99.7% of non-vulnerable files but only 8.3% of vulnerable ones.

An example of this pattern is shown in Listing 2. The second observed pattern becomes more apparent only after the symbolic representation is performed. This pattern, referred to as "cascade" for ease of reference in this work, is illustrated in red in Listing 3. As previously defined, the `FUN#` functions are symbolic representations of previously biased functions, meaning they contain information regarding the presence or absence of vulnerabilities within the files.

```
/* bad function declaration */
void CWE121_Buffer_Overflow_badSink(char* data);
void CWE121_Stack_Based_Buffer_Overflow_bad()

/* good function declarations */
void CWE121_Buffer_Overflow_goodSink(char* data);
/* goodG2B uses the GoodSource with the BadSink */
static void goodG2B()
```

Listing 2: Examples of biased function names and comments.

```
            ...
            }
            printLine(dest);
      }
}
void FUN2(){
       FUN0();
       FUN1();
}
```

Listing 3: "Cascade" pattern, in red, detected in files without vulnerability.

This pattern occurs in 99.6% of non-vulnerable files but only 0.01% of vulnerable ones. As a result, the model can rely on this pattern to identify 99.6% of non-vulnerable cases and 99.9% of vulnerable cases. However, in real-world scenarios, the model may fail

to detect the actual code responsible for the vulnerability.

Generate the Final Result. Listing 4 shows an example of a comparison between a vulnerable sample and a non-vulnerable sample after data sanitization. It is not possible to perceive any pattern that would bias the model's learning other than the presence or absence of vulnerability. It is worth noting that the vulnerability delimiters in the code shown in Listing 4, indicated by the markers $< START >$ and $< END >$, are used solely to label the code segments and are removed in the step before model training.

After this treatment was applied, 128,198 samples were obtained in the C/C++ language, comprising an equal number of 64,099 samples with and without vulnerabilities.

3 IDENTIFIED BIASES IN THE DATASET

Comments and variable names influence the model's judgment, as it may rely on keywords such as "bad" to classify code as vulnerable within this particular dataset. However, additional biases, such as the static function and cascade patterns, discussed in the previous section, may not be as easily identifiable.

These patterns originate from the algorithms used by specialists to build the test cases. Since SARD is a synthetic dataset, these biases were likely introduced unintentionally, resulting in patterns that can mislead the model during training. This study demonstrates that models trained on biased datasets struggle to generalize effectively, while those trained on bias-free datasets consistently identify vulnerabilities across scenarios. In summary, the following biases must be addressed and removed from SARD before training:

1. Biased function and variable names, such as those containing the words "good" and "bad;"
2. Comments that explicitly indicate a vulnerability;
3. Overrepresentation of specific function types, such as "static void", in non-vulnerable files; and
4. the cascade pattern as shown in Listing 3.

Previous studies utilizing the SARD dataset, as detailed in Section 5, have primarily focused on addressing only the first two items. The static functions and cascade pattern issues, to our knowledge, have never been identified or mitigated in any previous work.

Although other biases may exist in the dataset, our manual investigation did not reveal any evidence of their presence. This process involved a side-by-side comparison of sanitized vulnerable and non-vulnerable samples, as illustrated in Listing 4.

4 EXPERIMENTS AND RESULTS

The LLM CodeBERT was selected for vulnerability detection in the sanitized dataset due to its extensive training on a large programming language corpus. As an encoder-only model, it excels at tasks requiring a deep understanding of input, making it ideal for vulnerability classification.

Following dataset sanitization, CodeBERT's tokenizer was fine-tuned on the SARD dataset, producing a dictionary of approximately 4,100 tokens. With a context window limited to 512 tokens, code samples were divided into chunks of this size. To minimize context loss between adjacent chunks, a sliding window of 384 tokens was employed, ensuring 128 tokens of overlapping context.

Before discussing quantitative results, it is helpful to illustrate CodeBERT's capabilities with examples. Listing 4 compares a vulnerable sample (left) and its non-vulnerable counterpart (right), both correctly classified by CodeBERT. The vulnerable code contains a buffer overflow caused by using a smaller buffer of size 50 and attempting to access the 100th memory slot, leading to improper memory handling. In contrast, the non-vulnerable version allocates a buffer of size 100, avoiding overflow. This example highlights how the model, trained on a bias-free dataset, accurately differentiates between vulnerable and non-vulnerable code based on meaningful patterns.

Figure 2 illustrates the training and inference workflow for detecting software vulnerabilities using biased and bias-free datasets. Two datasets were derived from SARD: a fully sanitized, bias-free dataset (green) and a biased dataset (red) that retained only the static function and cascade patterns. This was done intentionally to simulate related work that does not remove the biases, which have been highlighted in this work.

These datasets were used to train two distinct CodeBERT-based models. The "biased" model was trained on the biased dataset, while the "bias-free" model was trained on the bias-free dataset. Once trained, each model performs inference not only on the dataset it was trained on but also on the opposite dataset. This cross-inference demonstrates the models' ability (or inability) to generalize beyond the spe-

```
void FUN0(char * data);
void FUN1()
{
    char * data;
    char * VAR0 = (char *)ALLOCA(50*sizeof(char));
    char * VAR1 = (char *)ALLOCA(100*sizeof(char));
    data = VAR0;
    data[0] = '\0';
    FUN0(data);
}
void FUN0(char * data)
{
    {
        char source[100];
        memset(source, 'C', 100-1);
        source[100-1] = '\0';
<START>
        strcat(data, source);
<END>
        printLine(data);
    }
}
```

```
void FUN0(char * data);
void FUN1()
{
    char * data;
    char * VAR0 = (char *)ALLOCA(50*sizeof(char));
    char * VAR1 = (char *)ALLOCA(100*sizeof(char));
    data = VAR1;
    data[0] = '\0';
    FUN0(data);
}
void FUN0(char * data)
{
    {
        char source[100];
        memset(source, 'C', 100-1);
        source[100-1] = '\0';
        strcat(data, source);
        printLine(data);
    }
}
```

Listing 4: Example of sanitized samples: the vulnerable sample is displayed on the left, while the non-vulnerable sample is shown on the right.

cific biases found in their respective training datasets, providing insight into the impact of data biases on model performance.

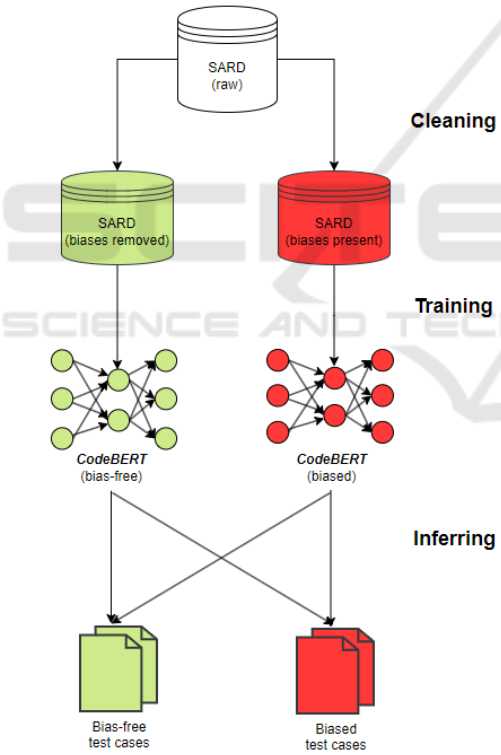

Figure 2: Training and inference workflow.

In this study, the performance metrics reported are calculated as simple averages due to the balanced nature of the dataset, which includes an equal number of vulnerable and non-vulnerable samples. However, since a holdout validation strategy is employed without guaranteed stratification, class distribution variations in the validation set may introduce minor differences in performance metrics. To complement these averages, a detailed confusion matrix is provided for deeper analysis, offering a more granular view of the model's performance across classes.

Table 1 presents the performance results of the models on the bias-free dataset. The bias-free model was validated using the holdout technique, where inference was performed on 20% of the dataset after training on the remaining 80%. In contrast, the biased model, having been trained on the biased dataset, did not require a cross-validation approach and was validated using the entire bias-free dataset.

Table 1: Results of accuracy (Acc), precision (Prec), f1-score (F1), and recall (Rec) metrics from both bias-free and biased models on the bias-free dataset. All reported metrics are calculated as simple averages, given that the dataset is perfectly balanced.

	Bias-free dataset			
Model	Acc	Prec	F1	Rec
Bias-free	**98.5 %**	**98.2 %**	**98.3 %**	98.4 %
Biased	63.0 %	54.7 %	70.6 %	**99.6 %**

For the biased model, inferring on the unbiased dataset, it can be seen in Table 1 that this is the only case where good metrics were not obtained, with the only good result being a recall of 99.6%, which may be misleading. To explain this result, it is necessary to revisit the statistics reported in Section 2, where it was made clear that the biased patterns found only occur in files without vulnerabilities. When the model performs the inference on a file that does not have these patterns, such as in the unbiased dataset, it infers that most files have vulnerabilities, since it does not find the patterns for which it was trained. Since the recall formula is given by $Recall = \frac{VP}{VP+FN}$, it measures how many of all the positive class situations (vulnerability) as the expected results are correct. Thus, it is possible to see that the biased model predominantly infers as though the majority of files contain vulnerabilities,

achieving an almost perfect recall, but at the expense of all other metrics.

This behavior is further illustrated by the confusion matrix in Figure 3, which highlights the excessive false positives generated by the model. The matrix shows the model's tendency to classify a significant number of non-vulnerable files as vulnerable, inflating the recall but severely degrading other performance metrics.

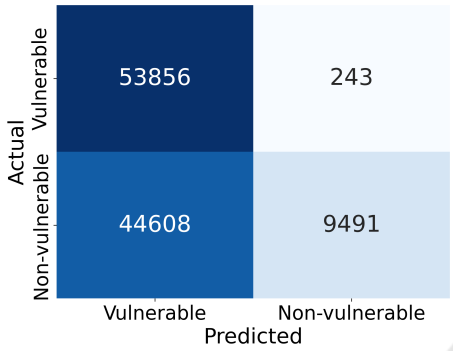

Figure 3: Confusion matrix of the biased model's inference on the bias-free dataset.

Similarly, Table 2 presents the performance results of the models on the biased dataset. Just like the bias-free model, the holdout technique was performed to train and validate the biased model in the biased dataset. The inference of the bias-free model on the biased dataset was done using the entire dataset.

Table 2: Results of Acc, Prec, f1-score, and Rec metrics from both bias-free and biased models on the biased dataset.

	Biased dataset			
Model	Acc	Prec	F1	Rec
Bias-free	98.6 %	98.4 %	98.5 %	98.5 %
Biased	**99.6 %**	**99.5 %**	**99.5 %**	**99.5 %**

For the bias-free model, inferring on the biased dataset, excellent metrics are obtained, such as an accuracy of 98.6% and a f1-score of 98.5%. This shows that, in this case, the model has effectively learned to identify the vulnerabilities instead of relying on the superficial patterns presented in the data.

5 RELATED WORK

This section covers relevant studies in two areas: debiasing vulnerability datasets and vulnerability detection using SARD. However, existing works primarily focus on improving model performance through different machine learning techniques, with little attention given to debiasing synthetic datasets.

Debiasing Vulnerability Datasets. A search was conducted to identify any existing works specifically focused on addressing and removing bias from the SARD dataset. While some studies have performed a degree of bias removal, such as deleting comments and applying symbolic representations to variables and functions, no prior research was found that thoroughly examines other subtle patterns that may skew model training, such as the static pattern or the cascade pattern, as discussed in this study. This gap in the literature suggests that current approaches may overlook important factors that could compromise model performance.

Vulnerability Detection Using SARD. Several studies have utilized the SARD Juliet dataset for vulnerability detection to classify vulnerabilities in C/C++ source code, with varying degrees of success.

(Jeon and Kim, 2021) used SARD and National Vulnerability Database (NVD) datasets to train Recurrent Neural Networks (RNNs) such as LSTM, GRU, BLSTM, and BGRU. The best results were achieved with BGRU, with an f1-score of 96.11%. Program slicing and symbolic representation were employed to reduce noise in the input data. However, the study did not explore other potential biases in code structures that could influence model learning, a gap this paper addresses.

(Lin et al., 2022) compared pre-trained contextualized models (e.g., CodeBERT) and non-contextualized models trained on synthetic SARD and real-world samples. Fine-tuning was performed using synthetic data, achieving precisions up to 86% and recalls up to 60%. Similar to (Jeon and Kim, 2021), symbolic representation was applied, but the authors did not address potential biases introduced by specific code patterns or structures.

The work by (Zeng et al., 2023) encapsulated CodeBERT with transfer learning to detect vulnerabilities in C code. Due to the scarcity of real-world data, the authors relied on combining synthetic SARD samples and real-world data to balance classes. Despite reporting an overall accuracy of 57%, the study lacked details on data preprocessing, including the removal of biased patterns or symbolic representations.

(Li et al., 2018) automated feature extraction using code gadgets, evaluating their system on synthetic and real-world datasets. Symbolic representation and comment removal were performed, but the study focused on only two CWE types without addressing broader structural biases.

(Li et al., 2022) proposed an approach inspired by region proposal techniques in image processing, extracting syntactic and semantic features for vulnera-

Table 3: Summary of related work.

Work	Symbolic Representation	Real Data	Synthetic Data	Addressed Code Biases
(Jeon and Kim, 2021)	✓	✓	✓	✗
(Lin et al., 2022)	✓	✓	✓	✗
(Zeng et al., 2023)	✗	✓	✓	✗
(Li et al., 2018)	✓	✓	✓	✗
(Li et al., 2022)	✓	✓	✓	✗
(Cheng et al., 2021)	✓	✓	✓	✗
(Li et al., 2021)	✓	✗	✓	✗

bility detection. The study achieved an accuracy of 96% using BGRU but, like previous works, did not investigate the impact of code biases, such as those highlighted in this paper.

Graph Neural Networks (GNNs) have also been applied to vulnerability detection, as demonstrated by (Cheng et al., 2021). Their system achieved f1-scores between 94.0% and 98.8% using SARD and real-world data, employing slicing and symbolic representation. However, preprocessing steps, such as comment removal, were not explicitly detailed.

Finally, (Li et al., 2021) used Hybrid Neural Networks trained solely on synthetic SARD data, reporting a high f1-score of 98.6%. While they used slicing and symbolic representation, no steps were taken to address structural biases in the code.

Table 3 summarizes related work in vulnerability detection, highlighting key aspects such as symbolic representation, using real and synthetic data, and whether code biases identified in this paper were addressed. While many studies utilized symbolic representation and combined real and synthetic data, none tackled the specific code biases this work identifies, highlighting a critical gap in existing research and underscoring the originality of this approach.

In comparison to previous studies, our method achieved highly accurate performance on the SARD dataset, with an f1-score of 98.3%. This underscores the effectiveness of the bias-free dataset preprocessing, which enhances the model's ability to generalize and detect vulnerabilities more accurately. However, it is important to acknowledge that detecting vulnerabilities in synthetic datasets like SARD is inherently easier than in real-world scenarios. Synthetic datasets often contain patterns and structures that simplify the learning process, whereas real-world data presents greater variability and complexity, lacking such consistent patterns. This disparity suggests that while our model performs exceptionally well on SARD, further validation and refinement are necessary for real-world applicability.

The main contribution of this study, beyond its strong performance in detecting vulnerabilities as evidenced by the obtained metrics, lies in the fact that this is the only one to identify and remove these patterns that can skew a model's generalization capabilities. Rather than depending on superficial patterns or cues, the pre-processing steps enhance model performance by allowing the model to focus on learning the underlying logic and structure of vulnerabilities.

6 CONCLUSIONS

This study assessed the importance of proper data processing when using synthetic vulnerability datasets such as SARD's Juliet. To achieve this, the performance of the CodeBERT model was compared, using a properly processed dataset and another that exhibited clear class-related biases.

The results demonstrated that a model trained on an unbiased dataset achieved consistently high performance, with an F1-score of 98.3% across all tested scenarios. In contrast, the biased model performed significantly worse, with an F1-score of only 70.6%. These findings underline the detrimental impact of biases on model training, showing that if datasets like SARD are not sanitized, the resulting models may learn to exploit superficial patterns rather than accurately detect vulnerabilities. Additionally, dataset biases can mask the true performance of trained models, as these patterns are easier for models to learn than the vulnerabilities themselves. Given SARD's extensive size compared to real-world datasets, biases in SARD could also distort the performance metrics of models trained on combined datasets, especially since most studies do not report individual dataset results.

This study emphasizes the importance of not relying solely on synthetic datasets for training, as the algorithms generating these datasets may unintentionally introduce learnable patterns that skew the model's predictions. To mitigate this risk, combining synthetic datasets with real-world data is recommended. This approach ensures that models learn to generalize effectively, reducing the risk of overfitting to artificial biases and improving their performance in real-world applications.

Limitations. While the debiasing approach in this work significantly improved model performance using the Juliet C/C++ 1.3 dataset, it is tailored specifically to this dataset. The identified biases, such as the static function and cascade patterns, are unique to the synthetic nature of SARD's Juliet project. Consequently, this method may not generalize to other datasets with different biases. Moreover, the process of identifying these patterns is manual and highly dependent on the dataset being analyzed, limiting its scalability.

One other limitation of this work is the potential for overfitting to the bias-free dataset. While the model performs exceptionally well on the sanitized version of the SARD dataset, there is a risk that it has learned to recognize specific patterns or cues inherent to the cleaned synthetic data rather than developing a broader understanding of vulnerability detection. Given that SARD is a synthetic dataset, it might still have nuances or hidden clues that human researchers might overlook, but that the model could use to forecast outcomes. This could result in an overestimation of the model's actual capability, as real-world datasets lack the artificial patterns introduced by test case generation algorithms, presenting a more complex and noisy environment for vulnerability detection.

Future Work. Future research should focus on automating the detection of biases in synthetic datasets or ensuring greater care in dataset creation to reduce the introduction of skewed patterns. Additionally, models trained on synthetic datasets should be rigorously evaluated on real-world datasets to better assess their generalization capabilities. While synthetic datasets like Juliet provide extensive test cases, real-world data introduces greater complexity and diversity in vulnerabilities, making it essential for robust and practical model evaluation.

REFERENCES

Barbierato, E., Vedova, M. L. D., Tessera, D., Toti, D., and Vanoli, N. (2022). A methodology for controlling bias and fairness in synthetic data generation. *Applied Sciences*, 12(9).

Cheng, X., Wang, H., Hua, J., Xu, G., and Sui, Y. (2021). Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Trans. Softw. Eng. Methodol.*, 30(3).

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547. Association for Computational Linguistics.

Huang, W., Lin, S., and Li, C. (2022). Bbvd: A bert-based method for vulnerability detection. *International Journal of Advanced Computer Science and Applications*, 13(12):890–898.

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

Li, X., Wang, L., Xin, Y., Yang, Y., Tang, Q., and Chen, Y. (2021). Automated software vulnerability detection based on hybrid neural network. *Applied Sciences*.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., and Chen, Z. (2022). Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258.

Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., and Zhong, Y. (2018). Vuldeepecker: A deep learning-based system for vulnerability detection. In *Proceedings 2018 Network and Distributed System Security Symposium*, NDSS 2018, San Diego, CA, USA. Internet Society.

Lin, G., Jia, H., and Wu, D. (2022). Distilled and contextualized neural models benchmarked for vulnerable function detection. *Mathematics*, 10(23):1–24.

Mehrabi, N., Morstatter, F., Saxena, N., Lerman, K., and Galstyan, A. (2021). A survey on bias and fairness in machine learning. *ACM Comput. Surv.*, 54(6).

NIST (2021). Software assurance reference dataset. https://samate.nist.gov/SARD/. Accessed: 2024-06-28.

Nong, Y., Aldeen, M., Cheng, L., Hu, H., Chen, F., and Cai, H. (2024). Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA. Curran Associates Inc.

Zeng, P., Lin, G., Zhang, J., and Zhang, Y. (2023). Intelligent detection of vulnerable functions in software through neural embedding-based code analysis. *International Journal of Network Management*, 33(3):e2198.