How Good is Openly Available Code Snippets Containing Software Vulnerabilities to Train Machine Learning Algorithms?

Kaan Oguzhan
T RDA CST SEL-DE
Siemens AG
Munich, Germany
email: kaan.oguzhan@siemens.com

Tiago Espinha Gasiba
T RDA CST SEL-DE
Siemens AG
Munich, Germany
email: tiago.gasiba@siemens.com

Akram Louati
T RDA CST SEL-DE
Siemens AG
Munich, Germany
email: akram.louati@siemens.com

***Abstract*—Machine learning has been gaining more and more attention over the last years. One of the recent areas where machine learning has been applied is secure software development to identify software vulnerabilities. The algorithms depend on the amount and quality of data used for training. Although many studies are emerging on machine learning algorithms, one must enquire about the data used to train these algorithms. This paper addresses this question by investigating and analyzing freely available vulnerable code snippets. We investigate their quantity and quality in terms of the existing categorization of security vulnerabilities used in industrial environments. Furthermore, we investigate these aspects in dependency on several different programming languages. In addition, we provide the database containing the collected vulnerable code snippets for further research. Our results show that, while a large number of training data is available for some programming languages, this is not the case for every language. Our results can be used by researchers and industry practitioners working on machine learning and applying these algorithms to improve software security.**

Keywords–machine learning; deep learning; industry; software; vulnerabilities.

I. INTRODUCTION

The rapid increase of digitalization and the fast-changing technology landscape are continuously increasing the potential attack surfaces for organizations [1]. According to AV-TEST [2], the number of cybersecurity incidents has been steadily increasing swiftly. One possible root cause of these incidents is poor software development that results in vulnerable code. A cybersecurity incident occurs when vulnerable code is attacked by malware. In 2013, the total number of known malware was about 182 million, whereas, in 2021, this number increased to 1313 million - an increase by more than sixfold compared to preceding years.

One way to address vulnerabilities in software, which is widely used in the industry, is by detecting vulnerabilities during the software development lifecycle through static code analysis. The traditional method of developing a new vulnerability detection algorithm starts from mathematical formulas or known vulnerable patterns and turns them into computer code that follows the exact mathematical formula or matches the exact pattern [3]–[7]. There are already many examples of static code analysis tools as both open source, such as [8] and commercial, such as [9]. The biggest drawback in developing such tools is the need for handcrafting new formulas or patterns for every type of vulnerability or, in the best case,

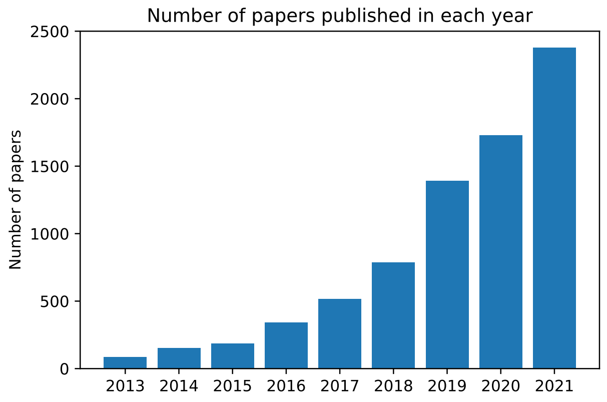

Figure 1. Appearance of both keywords ”Cybersecurity” and ”Machine Learning” in Academic Papers according to Scopus

adapting the old patterns to capture the new vulnerabilities. This process is time-consuming and requires much expertise in cybersecurity, which is not always available and can incur high costs.

It is not surprising that the traditional methods have started failing to keep up with the pace of new vulnerabilities, which are being introduced at an ever-increasing rate. It is beneficial to introduce new vulnerability detection methods to keep pace with the increasing number of vulnerabilities.

Machine learning is a sub-field of artificial intelligence; in contrast to the traditional methods, it does not require handcrafted formulas by experts. It is based on the idea that a neural network, which is essentially a vastly complex model designed after the human brain, can learn to make accurate decisions from large amounts of data through optimization processes like Gradient Descent [10] with minimal human intervention.

Although relatively new, the field of machine learning is expanding at an accelerating pace. Its expansion is driven by the explosion of Big Data [11], [12] and the ever-increasing pace of growth in accelerator computing power [13]. The implementation of these algorithms can be very efficient through the usage of highly optimized computations, such as Coppersmith–Winograd algorithms [14] and the use of specialized primitives [15].

Machine learning and deep learning algorithms can potentially significantly transform the cybersecurity field, e.g., by

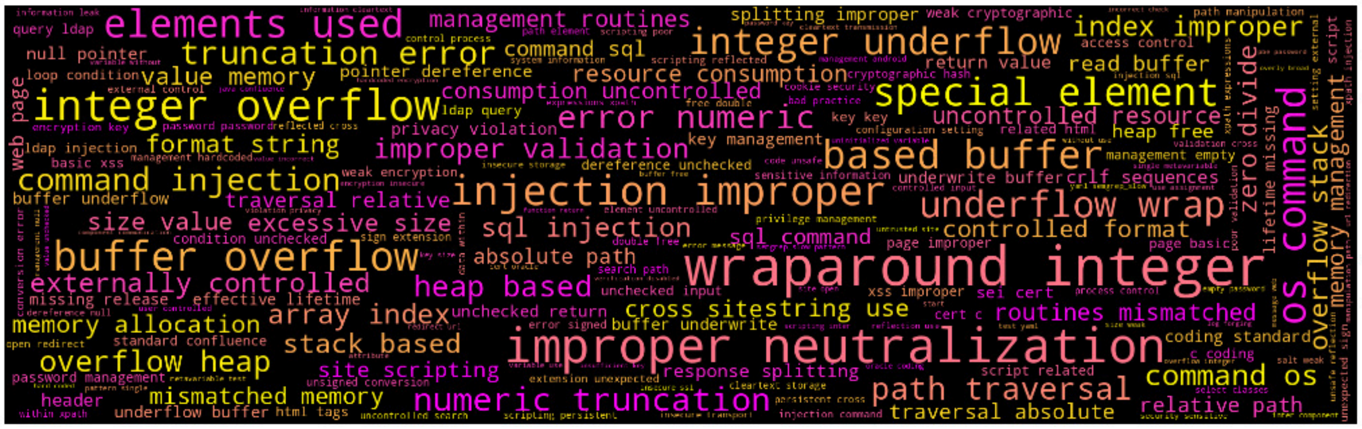

Figure 2. Word Cloud generated from 2-gram's over all the snippet titles

allowing adaptation to changes at a much faster pace and detecting very complex patterns in the data. Through the literature, we observe a significant uptrend in the number of work-related to deep learning applications in cybersecurity over the last decade (see Figure 1). We expect to see a growth in the number of static code analysis tools both open source and commercial following the trend by harnessing the power of machine learning for source-code [16] or byte-code analysis [17]–[19].

Deep learning has the potential to make cybersecurity simpler and more effective. However, it can only do so if the underlying data has enough samples and contain a distinct variety of information. In other words, deep learning models can only be as good as the data in which it has been trained. Unfortunately, when it comes to cybersecurity, that data is often lacking.

Secure coding is vital in the industry because it helps to ensure that software is free from vulnerabilities that attackers could exploit to gain access to systems and data. By ensuring that the code is secure, organizations can reduce the risk of data breaches and other cybersecurity incidents and improve their compliance with regulations, such as the General Data Protection Regulation (GDPR) [20], Payment Card Industry Data Security Standard (PCI-DSS) [21] and IEC 62.443 from the International Electrotechnical Commission.

There are several ways to improve code security, e.g., through following secure coding guidelines, conducting code reviews, and using static analysis tools. Until recently, commercial and open-sourced static code analysis tools existed using non machine learning techniques. These tools are built by handcrafting rules and using pattern matching techniques to detect vulnerabilities [3]–[7] However, following the recent trend in deep learning getting into many areas of Information Technologies, we expect to see the same up trend in cybersecurity applications, such as [22]–[24] and more specifically, into the field of static code analysis tools, such as [25].

The large amounts of data required to train deep learning algorithms are difficult to acquire. A possible way to obtain such a large amount of training data is to inspect existing open-source projects and secure programming language standards. After vulnerabilities in open-source projects are publicly disclosed, it is generally followed by the project maintainers providing a fix. Therefore, the code before and after the fix would be one way to train deep learning algorithms, as many such code examples are available. According to GitHub-2020-Digital-Insight-Report [26], as of 2020, there are over 1.05 million open-source projects across all of GitHub. This number should be more than enough for most machine learning tasks. However, most open-source projects do not use standardized vulnerability classifications, such as Common Weakness Enumeration (CWE), for their code fixes. As such, the task of collecting vulnerable code snippets directly from GitHub repositories remains very challenging.

Our work focuses on known vulnerabilities with a standardized classification that aligns with industrial security standards. We explore existing and openly available sources that contain vulnerable codes. In addition, we look at the possibility of using these sources to create static code analysis tools using machine learning techniques, thus improving overall security. Our work contributes by analyzing the quality of openly available data sources and the amount of data they can provide. Our work also assists further research in this young discipline by pointing to these data sources and also unveils possible obstacles that further work in this field can experience - both by researchers and practitioners alike. Furthermore, our work presents an overview of the state of secure coding knowledge of individual programming languages. Industry practitioners can use our results to motivate their choice of a programming language so that they can fulfill the requirements of industrial security standards.

Our work is structured as follows. In Section II, we discuss previous and related work. In Section III, we describe our methodology and experiment. In Section IV, we provide the significant contribution of the present work through a critical discussion of the results obtained in the experiment. Finally, in Section V, we finalize the paper through a brief overview and consideration of further work.

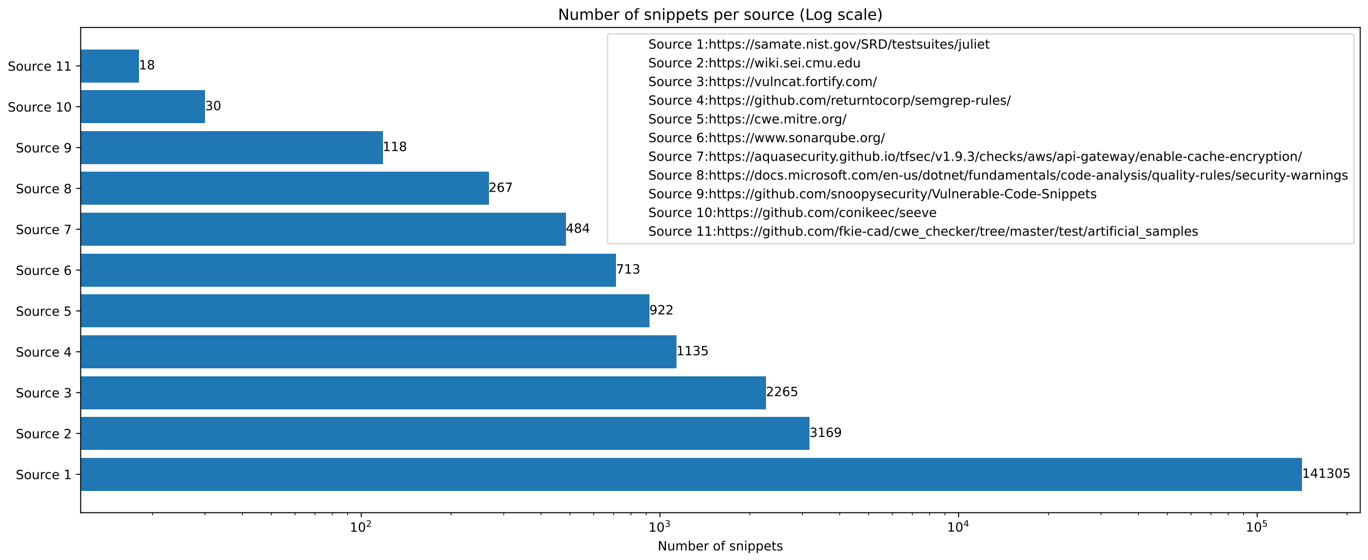

Figure 3. Number of snippets taken from their respective sources

II. RELATED WORK

As the quantity and quality of Big Data increases, many organizations have inevitably more and more demand for secure software that can protect the data of their customers and themselves. Due to the ever-increasing complexity of developing standardization, we can see more and more standardization and categorization being done by organizations such as, the PCI-DSS [21], International Standard Organization ISO/IEC 27000 series [27], Computer Emergency Response Team CERT [28], CWE [29], and Open Web Application Security Project OWASP Top 10 [30].

Producers such as, the International Organization for Standardization (ISO) [31] and the International Electrotechnical Commission (IEC) [32] have created a whole new standard for data security, ISO/IEC 27000 family, due to the need for a new standard ensuring the security of Big Data.

The ISO/IEC 27000 is a family of standards for Information Security Management Systems (ISMS) used by organizations of all sizes and industry sectors to protect their data. The standard is based on a risk management approach and provides a framework for organizations to identify, assess and manage the risks to their Information security. The standards cover various topics, including security policy, risk assessment, security controls, incident management, and business continuity.

IEC 62.443 is a family of standards for industrial automation and control systems security. The standards cover various topics, including risk assessment, security controls, incident management, and business continuity.

The ISO/IEC 27000 family of standards and the IEC 62.443 family of standards are complementary to each other. The ISO/IEC 27000 family of standards provides a general framework for Information security management, while the IEC 62.443 family of standards provides specific guidance for industrial automation and control systems.

Static code analysis is a process where the source code of a software application is analyzed without executing it. Static code analysis aims to find errors and vulnerabilities in the code that attackers can exploit. By finding these vulnerabilities, software developers can fix them before the application reaches end customers. Static code analysis tools can be used to find a wide range of security vulnerabilities in software. The classical static code analysis tools use a set of rules and patterns to find vulnerabilities in the code; however, they require much manual work to create those rules and patterns. Another disadvantage of static code analysis is that it can be time-consuming to run on large codebases. In addition, those classic static code analysis tools can produce many false positives and negatives, making it difficult to identify real security issues while highlighting the fact that additional processes must be used to write secure code.

With the rise of Big Data and machine learning, it is inevitable to think about other potentially promising and attractive solutions to the problem of finding security vulnerabilities in software. By applying machine learning techniques, we can accelerate the process of static code analysis and potentially even reduce the number of false positives. These methods require data to train the algorithms.

In our work, we look at vulnerable code snippets from those that are openly available, e.g., the Juliet Test Suite from the National Institute for Standards and Technology, the Common Weakness Enumeration [29] from the MITRE foundation, the Software Engineering Institute of the Computer Emergency Response Team (SEI CERT) [28] from Carnegie Mellon, and openly available data sets from commercial providers such as, SonarQube [9] and Fortify [33].

The CWE list from the MITRE foundation constitutes a standardized means to classify software vulnerabilities. In the documentation of each vulnerability are vulnerable code snippets for several programming languages.

The Juliet Test Suite contains a large and well-known collection of vulnerable snippets for the C, C#, C++, and Java programming languages. Although this data set is extensive, it exists for only a few programming languages. Many providers of static code analysis use this data set as a means of benchmarking their tool.

SEI CERT provides a secure coding standard for C, C++, Perl, Java, and Android. The standard contains code snippets as examples for each described vulnerability.

Fortify is a commercial company that does not focus on categorizing vulnerabilities; nevertheless, they provide snippet examples for each vulnerability they document. SonarQube is a commercial company selling a static code analysis tool; their openly available documentation provides several examples of code vulnerabilities and their corresponding CWE number.

Despite deep learning being still in its infancy in terms of popularity, it has already performed well in solving several problems in the cybersecurity field. It is worth noting that although there are many examples in academia such as, [34] most of these applications are not yet made into fully functional and production-grade software.

Deep learning models optimize to predict certain outputs given an input and is only as good as the data it has been trained on. Thus dumping sheer amounts of data might not always produce good results, after all the data quality is of importance. Withing a dataset there is always some variance, which can be minimized by increasing the number of samples, but the noise in the data might never go away. Another form of noise is the bias, which is a more structural problem and it represents the difference between an algorithm's expected output on a dataset and its actual output. When not dealt with properly, both of the problems would cripple the models performance. This concept is called the Bias-Variance trade-off.

III. EXPERIMENT

This section briefly discusses the methodology used to collect openly available vulnerable code snippets. We also present the results of the experiment together with an analysis thereof.

A. METHODOLOGY

To collect data for our experiment, we considered the following types of sources: (1) secure coding standards used in the industry, (2) official databases, (3) open documentation of available static code analysis tools, and (4) miscellaneous.

The following methods were employed in our work for each source type to obtain vulnerable code snippets :

- **(1) Standards**, and **(3) Documentation** - the web page where the standard or documentation is hosted was parsed, and the code snippets were extracted employing web crawling,
- **(2) Official Databases** - in this case, the code snippets were already provided in individual files,
- **(3) Miscellaneous** - depending on the repository, web crawling was used, or the code snippets were copied from individual files.

One crucial aspect that was also carried out was the creation of an SQL database, which contains one entry for each of the vulnerable code snippets, with a corresponding classification on the programming language and the standard secure coding rule number.

Figure 4 shows the number of code snippets gathered from each source. In [35], we provide the entire database to assist further researchers and practitioners.

B. RESULTS

For our analysis, we have gathered data from the online websites that provide vulnerable code snippets and their standardized categories. We analyzed snippets from 11 sources and then sorted these by programming language and vulnerability types. In Figure 3, we have listed all the external sources we have used and how many snippet examples they have publicly provided, as discussed in the methodology sub-section.

When categorized by the programming language Figure 4, we have found that four languages dominate the number of snippets by a considerable margin. We have found more than 45.000 code snippets for the C programming language, about 37.000 for Java, about 34.000 for C#, and 29.000 for C++. This large number is primarily due to the Julie Set. The number of available code snippets sees a sudden drop starting with 437 for Python. These results indicate that the number of vulnerable code snippets openly found on the internet is very language-specific and thus raises concerns about the performance of machine learning models trained with minimal data for these programming languages. Moreover, we have 355 standard vulnerability categories for Python, with an average of around six code snippets per category. We note that some categories overlap between different standards. To our knowledge, it is almost impossible to train a machine learning model with such limited data. Nevertheless, we expect to see models trained solely on the publicly available data for languages C, C#, C++, and Java as an extensive data set exists to train the respective algorithms.

In our experiment, we have also analysed the standardised vulnerability categories. Our main focus are on two categories, namely *OWASP TOP 10* (see Figure 5) and *PCI-DSS* (see Figure 6).

For the OWASP TOP 10, we found and collected openly available code snippets from 2004 to 2021. We have observed that the number of publicly available snippets has increased steadily over the years from approx. 1500 to approx. 2000 (see Figure 5). However, when investigating their categories, it is clear that the snippets are not equally distributed among all the ten categories (see Figure 5). To emphasize the difference, the category with the highest snippet count, ”2017-A03 Sensitive Data Exposure,” has 721 snippets, whereas ”2017-A04 Server-Side Request Forgery” has only eight snippets. It is also important to note that the number of publicly available snippets is not necessarily representative of the number of vulnerabilities that exist in the real world. There may be more real-world vulnerabilities in category 2012:A10 than in category 2017:A3.

These results highlight the importance of analyzing the sub-dimension of a data set before deciding on a machine learning

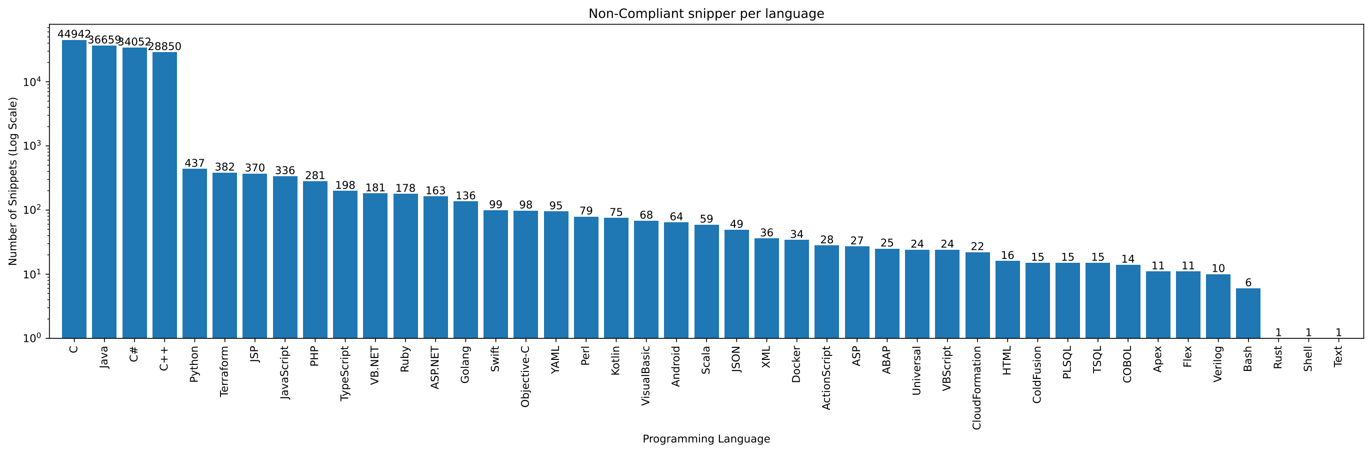

Figure 4. Number of non-compliant snippets per language

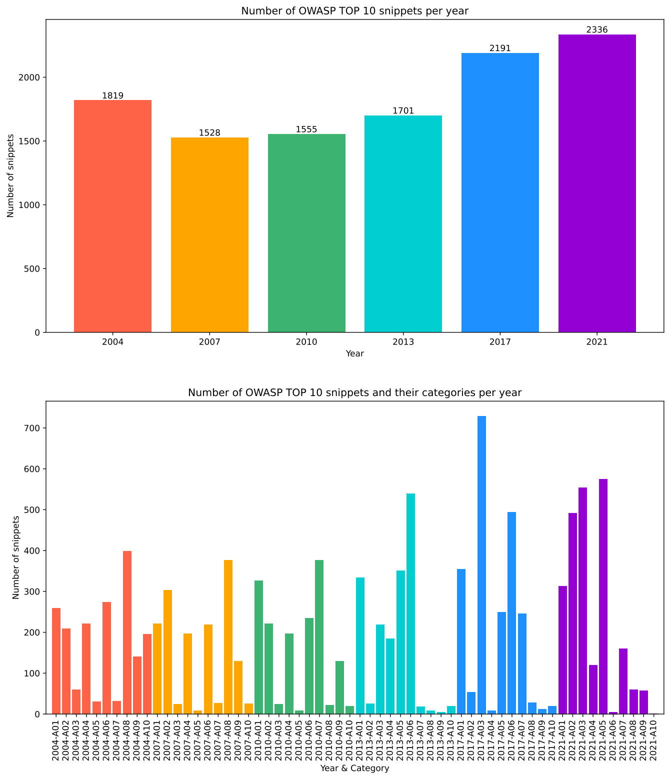

Figure 5. OWASP TOP-10 Snippet count for each Year as well as snippet count for sub-categories for each year

model. In the case of OWASP TOP 10, it is clear that if one looks at the uniformity amongst years and directly trains a machine learning model on them, the trained model will not be able to generalize well for all categories. The reason is that the model will be over-fitted on the categories with many snippets and will not work for categories with only a few snippets. In order to avoid such problems, one has to investigate the sub-categories further and explore possible sub-dimensions of the data to ensure that they are uniform.

For the case of PCI-DSS, we have snippets from main categories (V X) V1, V2, and V3 from the years 2005, 2010, and 2015. We did not observe an uptrend in the number of available snippets between versions, as was the case for OWASP TOP 10 (see Figure 5). However, by looking into categories (V X.X), we can observe that the number of snippets is uniformly distributed among them. It already looks much better for a machine learning model to train on than OWASP TOP 10. On the other hand if we go more finer into sub-categories(V X.X-X), see Figure 6. We again see an uneven distribution of snippets. For example, ”V1.2-02” has only one snippet compared to ”V1.2-06,” which has 2782 snippets.

Again, the number of snippets is not equally distributed among the PCI-DSS sub-categories. To the best of our knowledge, if one trains a machine learning model based solely on categories, where the numbers look uniformly distributed, one will end up with a model trained only on a few sub-categories without realizing it. In the best case, the model will be accurate for those categories with abundant snippets, but it will surely be heavily biased towards detecting them and will detect categories with a very sparse number of snippets, see Figure 6.

For our subsequent analysis, we wanted to understand the most commonly used wordings among the vulnerabilities to see if we could observe some patterns. For the analysis, we have gathered all titles given to snippets by their respective source. In the case of no title, we refer to their standard category and get the name given to the category. If the snippet has no title but has a standard category ”OWASP TOP 10 2017-A03,” then we assume it has the title ”Sensitive Data Exposure” and add it to our corpus. After we had gathered all the titles and then calculated 2,3,4-grams as well as made a word cloud out of the 2-grams, the word cloud can be seen in Figure 2. The Top-5 n-gram can be seen for each category in Table. Table I. It is important to note that the stop-words are only removed for the 2-gram and the word cloud, but for the 3-gram and 4-gram, they are kept as they are important to keep the semantics of the word groups intact. However, after the n-gram calculations, we made an elimination for selecting the Top-5; in case the n-gram starts or ends with a stop-word, we ignore it.

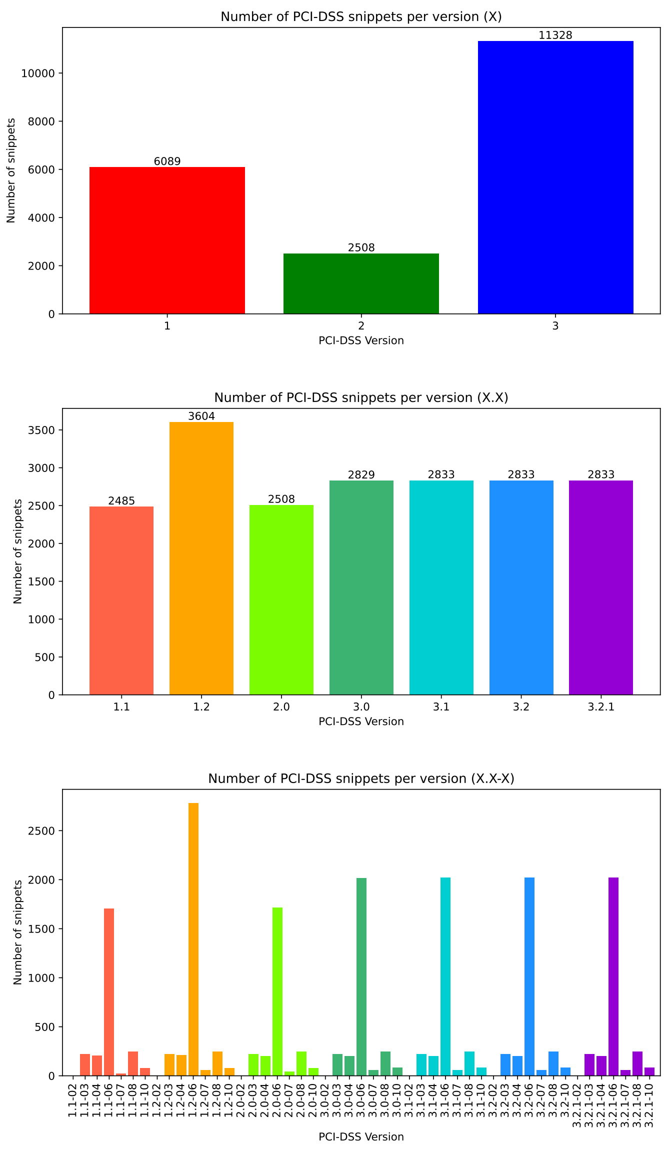

Figure 6. Number of snippets per PCI-DSS Version, going into finer categories from top to bottom

TABLE I
TOP-5 N-GRAMS

Top-5 | 2-gram
(improper, neutralization)
(integer, overflow)
(buffer, overflow)
(special, elements)
(integer, underflow)
Top-5 | 3-gram
(overflow, or, wraparound)
(neutralization, of, special)
(special, elements, used)
(integer, underflow, wrap)
(numeric, truncation, error)
Top-5 | 4-gram
(integer, overflow, or, wraparound)
(neutralization, of, special, elements)
(improper, neutralization, of, special)
(command, os, command, injection)
(improper, validation, of, array)
Top-5 | 5-gram
(improper, neutralization, of, special, elements)
(integer, underflow, wrap, or, wraparound)
(os, command, os, command, injection)
(improper, validation, of, array, index)
(use, of, externally-controlled, format, string)

For the last analysis, we wanted to see if we could observe some uniform patterns or detect unevenness for one of the most common and comprehensive data sets, the Juliet Test Suite. Many static code analysis tools use the Juliet Test Suite to benchmark their software as well as due to the sheer number of vulnerable snippets it includes, it is a tempting target for training deep learning models. This data set has been used in several previous studies [36]–[38]. We have used the same techniques as in the previous analysis and focused on the standardized vulnerability categorization CWE.

We again observed that the number of snippets is very unevenly distributed among the CWE categories. Some categories have a whooping number of snippets compared to others. For example, ”CWE-121” has 5906 snippets compared to ”CWE-561,” which has only two snippets, see Table II. The difference is again significant, and if one were to train a machine learning model directly on the Juliet Test Suite as the data set, one would end up with a model that is heavily biased towards detecting top categories such as, ”CWE-121”, ”CWE-78”, ”CWE-190”. Meanwhile, detecting other categories such as, ”CWE-561” would be very unlikely.

Before concluding our analysis, albeit not directly related to vulnerability categories or classes, it is worth mentioning that Juliet Test Suite snippets are usually very long. This fact contrasts with our previous sources, where snippets were relatively short and contained only a few lines of code. Such a difference can potentially significantly impact the performance of deep learning models. The Juliet Test Suite can test and benchmark static code analysis tools. Therefore, it contains many snippets that are very similar but have slight variations that make them unique. Those minor differences allow the Juliet Test Suite to cover the vulnerabilities from many angles. An oversimplified example would be ”if(True)...” vs ”if(1==1)...” vs ”if(varTrue)...”.

Due to the Juliet Test Suite’s previously mentioned goal

TABLE II
JULIET DATA SET SNIPPET COUNT PER CWE ID

C		Java		C#		C++	
ID	*Snippet count*	*ID*	*Snippet count*	*ID*	*Snippet count*	*ID*	*Snippet count*
CWE 121	5906	CWE 190	6555	CWE 197	7695	CWE 762	5180
CWE 78	5600	CWE 191	5244	CWE 190	5643	CWE 122	4948
CWE 190	5040	CWE 129	4104	CWE 191	3762	CWE 36	3500
...		...		...		...	
CWE 674	2	CWE 499	1	CWE 397	1	CWE 562	1
CWE 562	2	CWE 248	1	CWE 366	1	CWE 468	1
CWE 561	2	CWE 111	1	CWE 248	1	CWE 440	1

of coverage completeness, it is easy to find snippets designed to reproduce the same vulnerability from 50 different approaches, where the difference between each approach is very tiny such as, 15 lines in a 450-line of code. (CWE122_Heap_Based_Buffer_Overflow snippets). In this case, the common acceptance of ”the more data, the better” for machine learning is misleading. Having such a high similarity between the data means no good; if not handled correctly, those similarities can easily lead to overfitting, or worse, the model will learn the similarities and ignore the difference where the actual vulnerability is. Another likely pitfall is that if Juliet Test Suite alone is used for training, validation, and testing, the validation-split and test-split will be very similar to the training-split as they would share the same ”common” parts of the vulnerable snippets. Therefore, using Juliet Test Suite alone for training the model and evaluating its performance will not be a good representation of real-world vulnerabilities, thus leading to a poor representation of the model’s generalization capabilities.

IV. DISCUSSION

Throughout our experiment, we have tried to analyze publicly available vulnerable code snippets across programming languages and tried to understand their fitness for training popular deep learning models. As our first finding, we have seen a big difference in the number of publicly available vulnerable code snippets across programming languages, which to the best of our knowledge, has not been addressed in the literature. There are so few snippets available for some languages that it is impossible to train a model to detect vulnerabilities in that language, see Figure 4. For others, we have found that the sheer number of available snippets seems sufficient for training a model, but just the number of snippets is not enough to justify that a deep learning model trained directly on the publicly available code snippets would generalize well.

For a deep learning model to generalize well, there needs to be a certain amount of diversity in the snippets to make sure that the model is not just memorizing the available code snippets, as well as a uniformity among different vulnerabilities to make sure that the model will not be biased towards some specific vulnerabilities. A model lacking these two properties cannot perform well enough to be deployed in production environments.

We also looked at the distribution of snippets vs. standardized vulnerability categories and tried to draw some patterns to pin down possible pitfalls. We have found that one needs to be very careful during inspection of the data as the distribution of snippets might look uniform from a higher perspective, like the year for OWASP TOP 10 (e.g., 2007 vs. 2010 vs. 2013 (see Figure 5, Graph-1), but there could be a significant underlying bias in the sub-categorizations, like the case of OWASP-2017-A03 vs. OWASP-2017-A04 (see Figure 5, Graph-2). Such a bias will ultimately make the model biased towards specific vulnerabilities and lead to a bad performance. Depending on the severity of the bias, the model might not learn to detect some sub-categories.

We acknowledge that during the generation of our data set, although we tried our best not to miss anything, we could not have included some sources. The results of our analysis are solely based on the snippets we have collected. However, we believe that our data set is large enough and includes the most common public sources to give some insight into the distribution of vulnerable code snippets that are publicly available. Moreover, the results of our analysis can be used to evaluate machine learning models that are trained on publicly available vulnerable code snippets to see if they are indeed biased towards some vulnerabilities or vulnerability categories.

We also acknowledge that the static code analysis can either be done on source code or the byte/machine code; throughout our analysis, we have not delved into the task of analyzing the available byte code examples as we limited ourselves only to the publicly available and reviewed code snippets.

Practitioners and researchers can use our work as a source of information on where to find openly available data to perform machine learning experiments. We also highlight the limitations of the currently existing data. In particular, we observe that many vulnerable code snippets exist for the C, C++, Java, and C# programming language, while other programming languages, e.g., Python and Rust lack a good number of vulnerable samples to carry out meaningful experiments.

V. CONCLUSION AND FUTURE WORK

In cybersecurity and secure software development, novel methods that improve the security of applications are highly desirable. Securing software development is an essential topic for the industry as several cybersecurity standards require it. One possible way to achieve such compliance is using machine learning algorithms to identify vulnerabilities in software, as academia has demonstrated.

We expect that static code analysis using machine learning will become an essential part of future software development, in particular, using deep learning algorithms. We foresee that the developed tools will be integrated into existing Continuous Integration / Continuous Deliver (CI/CD) pipelines.

In this work, we explore freely available only databases containing vulnerable code snippets for different programming languages, as a means to train our machine learning algorithm. In particular, we focus on databases where the code snippets are categorized by a standard vulnerability classification - the common weakness enumeration from MITRE.

It is known that a large amount of high-quality data is needed to train these algorithms, in order to achieve a good detection rate. Our work shows that the number of freely available code snippets that can be used to train machine learning algorithms strongly depends on the programming language. Even for programming languages for which a large amount of snippets exist, these are not evenly distributed depending on the type of vulnerability classification. This presents a huge challenge to the field, as a non-uniform distribution certainly causes a bias in the trained model. Our results are in line with the authors' expectation and experience in secure coding field, and with standardized secure coding guidelines.

Furthermore, we observed that exploring a data set from different perspectives is essential, particularly in understanding the amount of existing data and its quality. Exploring the data set allows understanding it from different angles and avoiding possible pitfalls when training machine learning algorithms.

In further work, we will intend to implement a machine learning algorithm to detect vulnerabilities in C# code. We will aim to study the performance of such algorithm in relation to existing open-source static application security testing tools. The authors would also like to explore more simple criteria than the vulnerability type, in particular on whether the algorithm is simply vulnerable or not.

References

[1] The MITRE Corporation, "Common Vulnerabilities and Enumeration Security Vulnerability Database." https://www.cvedetails.com. [retrieved July, 2022].

[2] AV-TEST, "Malware statistics and trends report: AV-TEST." https://www.av-test.org/en/statistics/malware/. [retrieved July, 2022].

[3] C. Flanagan *et al.*, "Extended static checking for java," in ACM SIGPLAN Notices, vol. 37, 05 2002.

[4] Y. Xie and A. Aiken, "Context- and path-sensitive memory leak detection," in ACM Sigsoft Software Engineering Notes, vol. 30, pp. 115–125, Association for Computing Machinery, 09 2005.

[5] K. Karakaya and E. Bodden, "Sootfx: A static code feature extraction tool for java and android," in 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 181–186, 2021.

[6] E. Ersoy and H. Sözer, "Extending static code analysis with application-specific rules by analyzing runtime execution traces," in Computer and Information Sciences (T. Czachórski, E. Gelenbe, K. Grochla, and R. Lent, eds.), pp. 30–38, Springer International Publishing, 2016.

[7] J. Vanegue, S. Heelan, and R. Rolles, "SMT solvers in software security," in 6th USENIX Workshop on Offensive Technologies (WOOT 12), USENIX Association, 08 2012.

[8] AquaSecurity, "tfsec." https://github.com/aquasecurity/tfsec. [retrieved July, 2022].

[9] SonarSource, "Sonarqube." https://www.sonarqube.org. [retrieved March, 2022].

[10] H. Robbins and S. Monro, "A stochastic approximation method," The Annals of Mathematical Statistics, vol. 22, no. 3, pp. 400–407, 1951.

[11] D. Fasel and A. Meier, *Big Data: Grundlagen, Systeme und Nutzungspotenziale, in English: Big Data: Fundamentals, Systems and Potential Uses*. Springer-Verlag, 01 2016.

[12] J. Tang, T. Ma, and Q. Luo, "Trends prediction of big data: A case study based on fusion data," Procedia Computer Science, vol. 174, pp. 181–190, 2020. 2019 International Conference on Identification, Information and Knowledge in the Internet of Things.

[13] Y. Sun, N. B. Agostini, S. Dong, and D. R. Kaeli, "Summarizing CPU and GPU design trends with product data," CoRR, vol. abs/1911.11313, 2019.

[14] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," Journal of Symbolic Computation, vol. 9, no. 3, pp. 251–280, 1990.

[15] S. Chetlur *et al.*, "cudnn: Efficient primitives for deep learning," 2014.

[16] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning," in 5th USENIX Workshop on Offensive Technologies (WOOT 11), USENIX Association, 08 2011.

[17] H. Xue, S. Sun, G. Venkataramani, and T. Lan, "Machine learning-based analysis of program binaries: A comprehensive study," IEEE Access, vol. 7, pp. 65889–65912, 2019.

[18] W. Niu *et al.*, "Opcode-level function call graph based android malware classification using deep learning," Sensors, vol. 20, no. 13, p. 3645, 2020.

[19] A. Drewek-Ossowicka, M. Pietrołaj, and J. Rumiński, "A survey of neural networks usage for intrusion detection systems," Journal of Ambient Intelligence and Humanized Computing, vol. 12, pp. 497–514, 2020.

[20] Intersoft Consulting, "General data protection regulation GDPR." https://gdpr-info.eu. [retrieved July, 2022].

[21] Payment Card Industry, "Payment Card Industry Data Security Standard PCI-DSS." https://listings.pcisecuritystandards.org/documents/PCI-DSS-v4_0.pdf. [retrieved July, 2022].

[22] T. Guzella and W. Caminhas, "A review of machine learning approaches to spam filtering," Expert Systems with Applications, vol. 36, pp. 10206–10222, 09 2009.

[23] J. Z. Kolter and M. A. Maloof, "Learning to detect and classify malicious executables in the wild," Journal of Machine Learning Research, vol. 7, pp. 2721–2744, dec 2006.

[24] X.-S. Gan, J.-S. Duanmu, J.-F. Wang, and W. Cong, "Anomaly intrusion detection based on PLS feature extraction and core vector machine," Knowledge-Based Systems, vol. 40, pp. 1–6, 2013.

[25] G. Tang *et al.*, "A Comparative Study of Neural Network Techniques for Automatic Software Vulnerability Detection," CoRR, vol. abs/2104.14978, 2021.

[26] GitHub Inc., "Github 2020 digital insight report." [retrieved July, 2022].

[27] G. Disterer, "ISO/IEC 27000, 27001 and 27002 for information security management," Journal of Information Security, vol. 04 No.02, p. 9, 2013.

[28] Carnegie Mellon University, "SEI external Wiki." https://wiki.sei.cmu.edu. [retrieved July, 2022].

[29] MITRE, "Common weakness enumeration CWE." https://cwe.mitre.org. [retrieved July, 2022].

[30] Open Web Application Security Project, "OWASP Top Ten." https://owasp.org/www-project-top-ten. [retrieved July, 2022].

[31] International Organization for Standardization. https://www.iso.org. [retrieved July, 2022].

[32] International Electrotechnical Commission. https://iec.ch. [retrieved July, 2022].
[33] Micro Focus, "Fortify Taxonomy: Software Security Errors." https://vulncat.fortify.com. [retrieved March, 2022].
[34] R. Jusoh *et al.*, "Malware detection using static analysis in android: a review of feco (features, classification, and obfuscation)," PeerJ Computer Science, vol. 7, p. 522, 2021.
[35] K. Oguzhan, T. Gasiba, and A. Louati, "List Vulnerable Code Snippets." https://zenodo.org/record/7019175. Online, Accessed 24 August 2022.
[36] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability Detection with Deep Learning," in 2017 3rd IEEE International Conference on Computer and Communications (ICCC), pp. 1298–1302, 2017.
[37] N. Ziems and S. Wu, "Security Vulnerability Detection Using Deep Learning Natural Language Processing," CoRR, vol. abs/2105.02388, 2021.
[38] Z. Li *et al.*, "Vuldeepecker: A deep learning-based system for vulnerability detection," CoRR, vol. abs/1801.01681, 2018.