Shallow or Deep? An Empirical Study on Detecting Vulnerabilities using Deep Learning

Alejandro Mazuera-Rozo[†1,2] , Anamaria Mojica-Hanke[†2], Mario Linares-Vásquez[2], Gabriele Bavota[1]
[1]*SEART @ Software Institute, Università della Svizzera italiana, Lugano, Switzerland*
[2]*Universidad de los Andes, Bogotá, Colombia*

***Abstract*—Deep learning (DL) techniques are on the rise in the software engineering research community. More and more approaches have been developed on top of DL models, also due to the unprecedented amount of software-related data that can be used to train these models. One of the recent applications of DL in the software engineering domain concerns the automatic detection of software vulnerabilities. While several DL models have been developed to approach this problem, there is still limited empirical evidence concerning their actual effectiveness especially when compared with shallow machine learning techniques. In this paper, we partially fill this gap by presenting a large-scale empirical study using three vulnerability datasets and five different source code representations (*i.e.,* the format in which the code is provided to the classifiers to assess whether it is vulnerable or not) to compare the effectiveness of two widely used DL-based models and of one shallow machine learning model in (i) classifying code functions as vulnerable or non-vulnerable (*i.e.,* binary classification), and (ii) classifying code functions based on the specific type of vulnerability they contain (or "clean", if no vulnerability is there). As a baseline we include in our study the AutoML utility provided by the Google Cloud Platform. Our results show that the experimented models are still far from ensuring reliable vulnerability detection, and that a shallow learning classifier represents a competitive baseline for the newest DL-based models.**

***Index Terms*—Vulnerability detection, empirical study**

I. Introduction

DL models have been used to support software-related tasks such as bug localization [1], automated bug fixing [2]–[5], clone detection [6], code search [7], and code summarization [8]–[10]. DL models have also been used to automatically detect software vulnerabilities [11]–[18]. While several of these studies present a thorough evaluation of the technique they propose, there is still limited empirical evidence about the advantages brought by DL models over shallow machine learning algorithms. We present a large-scale study comparing two widely used deep learning models, namely Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN), with the Random Forest classifier, one of the most popular "shallow learning" models. The comparison of these three learning models is conducted across three different datasets including vulnerable and non-vulnerable C/C++ functions. Two of the employed datasets have been previously used in the literature to evaluate the effectiveness of DL-based vulnerability detection techniques [11], [19]. The first one [19] includes synthetic code examples (*i.e.,* the vulnerabilities have been manually injected) of 118 types of software vulnerabilities. The second [11] includes functions from the Debian Linux distribution and GitHub public repositories that have been labeled as vulnerable or non-vulnerable by using static analysis tools (*i.e.,* Clang [20], Cppcheck [21], and Flawfinder [22]).

Since the first dataset [19] is composed of artificial instances, and the second one [11], while including real code, is **highly unbalanced** (>90% of its functions are non-vulnerable), we built a third **balanced dataset** composed of real instances mined from GitHub. Our work builds on top of the seminal paper by Russell *et al.* [11], by testing/investigating DL-based vulnerability detection techniques on an additional dataset that is balanced and composed of real code vulnerability instances, making it more suitable for experiments involving learning techniques. Overall, our empirical study includes a total of $\sim$ 1.8M C/C++ functions.

Besides using different datasets, we also experiment the effectiveness of different code representations (*i.e.,* the format in which the source code is provided as input to the learning models). We consider five different representations inspired by the work of Tufano *et al.* [2] and using different levels of source code abstraction. Finally, we executed the experiments also with the AutoML utility provided by Google Cloud Platform. The study of different code representations as well as the usage of AutoML as a baseline for DL methods represent, besides the novel dataset we contribute to the community, additional points of novelty for our work as compared to the most related studies [11], [15].

Our work does not aim to replicate or verify the reproducibility of previous approaches presented in the literature for DL-based vulnerability detection. Indeed, many factors come into play when implementing, training, and customizing the developed models. Our goal is to empirically evaluate our own implementations of the DL and shallow learning models using different datasets and code representations, to assess the advantages (if any) brought by the DL-models over the shallow one. The material used in our study is publicly available [23].

The achieved results show that (i) on the dataset featuring synthetic examples of vulnerabilities [19], all experimented models behave almost as perfect predictors, questioning the usefulness of this dataset for assessing vulnerability predictors; (ii) on the other datasets, as expected, the models struggle to achieve very high performance, showing that important margins for improvement are possible; and (iii) the shallow model represents a very competitive approach for the DL-based models we tested.

[†] Both authors contributed equally to this manuscript.

II. Related Work

There are several studies describing and analyzing vulnerabilities and their impact in software projects [24]–[33]. Given the goal of our work, we focus our discussion on approaches to detect software vulnerabilities in code. The latter can be roughly classified into three groups: (i) approaches using static and dynamic code analyses with detection rules to identify vulnerabilities; (ii) program analysis combined with shallow learning techniques (*i.e.,* traditional machine learning classifiers/predictors); and (iii) more recent approaches for learning semantic and syntactic features of vulnerable code using deep learning. Given the very extensive literature in this field, we focus our discussion on a limited number of representative examples for each of these three categories.

Approaches based on static and dynamic code analysis are the most diffused. Those relying on static analysis examine the code without executing the program. A representative example of this family of techniques is the work by Pengfei *et al.* [34] that uses pattern-based matching to detect double fetch vulnerabilities in the Linux kernel; other examples of available tools for static analysis are Clang [20], Flawfinder [22], Checkmarx [35] and Cppcheck [21]. Dynamic analysis-based detectors assess a program by injecting data in real-time or simulating conditions that could trigger states leading to vulnerabilities (see *e.g.,* AddressSanitizer [36], SANTE [37], DR. CHECKER [38], OPIA [39], and DroidForensics [40]). Moreover, hybrid software techniques have also been proposed: Concolic testing approaches use dynamic symbolic execution to reach a trade off between the costs and benefits of dynamic and static analysis. Tools such as FUZZBUSTER [41], VUzzer [42] and QSYM [43] are examples of this trend.

Concerning techniques using shallow learning, a representative example is the work by Hovsepyan *et al.* [44], in which the authors use a support vector machine (SVM) to train a vulnerabilities detector; Hovsepyan *et al.* use a tokenized representation of Java source code to predict whether a Java file is vulnerable or not. Another representative approach is SOFIA [45], which is a programming-language and source-code independent Security Oracle; the approach is unsupervised and uses clustering to group SQL statements and detect anomalies possibly representing security issues. Lastly, Perl *et al.* [46] presented VCCFinder, an approach to find potentially dangerous code in software repositories; VCCFinder works on code snippets to detect suspicious commits by using SVM.

The deep learning (DL) approaches also use classification as a way to categorize code as vulnerable or not, similarly to what shallow learning techniques do. However, in the case of DL, compositional representations are automatically learnt. When using DL, semantic and syntactic features can be automatically extracted from the source code into vectors representing the code and can be fed into a neural network. Such a flexibility has led to the application of DL to many software engineering tasks (see Section I for a list of works applying DL in the software engineering domain). One of these tasks is the detection of security vulnerabilities. Some representative works in this area are discussed in the following.

The interested reader can refer to the work by Lin *et al.* [47] for a complete overview of the relevant literature in the field. Most of the DL-based approaches for detecting vulnerabilities have used Convolutional Neural Network (CNN) [11]–[14], Recurrent Neural Network (RNN) [11], [13], [15], [18] or improved variants of these models [12], [13], [15]–[17]. Table I summarizes those works.

For instance, Russell *et al.* [11] use CNN with custom lexed representations of C/C++ source code, working with a dataset of over 1.2 million curated functions. Li *et al.* [15] propose VulDeePecker, a deep learning-based system for vulnerability detection using BLSTM; it uses their own fine-grained representation of C/C++ code called *code gadget*, which is a set of lines of code that are semantically related. VulDeePecker was validated with two types of vulnerabilities, buffer error vulnerabilities (CWE-119) and resource management error vulnerabilities (CWE-399). Li *et al.* [13] have presented a new approach outperforming VulDeePecker, by implementing a Bidirectional Gated Recurrent Unit (BGRU) model which overcomes the results of other state-of-the-art detectors and uses a new representation of C/C++ source code that considers syntactic and semantic information; this model uses 126 types of vulnerabilities aggregated in 4 categories: issues related to library/API function calls, arrays, pointers and improper arithmetic expressions. Zou *et al.* [16] presented μVulDeePecker, a deep learning-based system for multi-class vulnerability detection, considering a total of 40 different types of software vulnerabilities.

In our work, we experiment with implementations of an RNN-based and a CNN-based model as representative of the work done in the field of DL-based vulnerability detection, and a Random Forest classifier as representative of shallow learning techniques.

III. Empirical Study

The *goal* of this study is to assess the effectiveness of deep/shallow learning techniques for detecting software vulnerabilities at function-level granularity when using different models and source code abstractions. We conduct experiments with three different models (two deep and one shallow). In particular, we experiment with: (i) Random Forest (RF), (ii) a Convolutional Neural Network (CNN), and (iii) a Recurrent Neural Network (RNN), with the first being representative of shallow classifiers and the last two of deep learning models. We chose RF due to its popularity in the software engineering domain (see *e.g.,* [48]–[50]). Concerning the two DL models, they have been used, with different variations, in previous studies on the automatic detection of software vulnerabilities: CNN [11]–[14] and RNN [11], [13], [15], [18]. We also exploit as baseline for our experiments an automated machine learning (AutoML) approach, which is a solution to build DL systems without human intervention and not relying on human expertise. AutoML has been used in Natural Language Processing (NLP) and it is provided by Google Cloud Platform (GCP). AutoML eases the hyper-parameter tuning and feature selection using Neural Architecture Search (NAS) and transfer learning [51]–[53].

Table I: Previous work using deep learning for vulnerability detection.

Study	Data origin	Lang.	Assessed Artefact	Representation	Classifiers	Classification	Vulnerabilities
Russell *et al.* [11]	SATE IV, Github & Debian open repositories	C/C++	∼1.3M Functions	Relevant meaning of critical tokens (*e.g.,* keywords, operators and separators)	BoW+RF, RNN, CNN, RNN+RF, CNN+RF	Multiple Binary classification	CWE-120, CWE-119, CWE-476, CWE-469, CWE-Others
Wu *et al.* [12]	Binary programs in "/src/bin" and "/usr/sbin/"	C	∼10K Sequences of function calls	Events regarding sequential calls within the program augmented with its arguments	CNN, LSTM, CNN-LSTM, MLP	Binary classification	None
Li *et al.* [13]	NVD open source software programs & SARD	C/C++	∼420k SeVCs	Sliced program representations that can accommodate syntax and semantic information pertinent to vulnerabilities	CNN, DBN, LSTM, GRU, BLSTM, BGRU	multi-class classification and Binary classification	4 Kinds (*i.e.,* issues related to library/API function calls, arrays, pointers and improper arithmetic expressions)
Harer *et al.* [14]	Packages distributed with the Debian Linux distribution and functions pulled from public Git repositories on Github	C/C++	∼900k Functions	Parsed code and categorized elements into different bins (*e.g.,* string literals, numbers, operators)	Word2vec+ CNN, CNN+ET, BOW+ET	Binary classification	None
Li *et al.* [15]	NVD open source software programs & SARD	C/C++	∼61k Code gadgets	Program slices represented as lines of code that are semantically related to each other	BLSTM	Multiple Binary classification	CWE-119, CWE-399
Zou *et al.* [16]	NVD open source software programs & SARD	C/C++	Code attention & Code gadgets extracted from ∼33k Programs	Multiple program semantically related statements in a piece of code being discrete aspects of information	Enhanced BLSTM	Multi-class classification	40 types of vulnerabilities (*e.g.,* CWE-119)
Wang *et al.* [17]	SARD, GitHub & Exploit-DB	C/C++	Execution paths retrieved from ∼56k binary programs	Vector representation retaining original semantic information of the execution path	LSTM	Binary classification	None
Lin *et al.* [18]	SARD & Real-world Open source projects (*e.g.,* LibTIFF, LibPNG)	C/C++	∼169k Functions	Vectors representing sequences computed from the source code and ASTs	RNN (Bi-LSTM)	Binary classification	None

Table II: Number of functions in each subject dataset.

	GH-DS	J-DS	R-DS
Vulnerable	315,777	28,446	46,335
Non-vulnerable	315,777	62,475	1,072,513

Since we are experimenting with different source code representations, datasets and type of classification (binary and multi-class), we run a total of 30 different experiments for the AutoML-based baseline approach and 90 experiments for the deep/shallow approaches. The explanation and details of such a variety of experiments is provided later on in this section.

The *context* of the study is represented by three datasets of C/C++ code reporting software vulnerabilities at the function granularity level, for a total of 1,841,323 functions, of which 390,558 are vulnerable ones. Our study addresses the following research question (RQ):

> *What is the effectiveness of different combinations of classifiers and code representations to identify functions affected by software vulnerabilities?*

We answer this RQ in two steps. First, we create binary classifiers able to discriminate between vulnerable and non-vulnerable functions, without reporting the specific type of vulnerability affecting the code. This scenario is relevant for practitioners/researchers who are only interested in identifying potentially vulnerable code for inspection/investigation. Second, we experiment the same models in the more challenging scenario of classifying functions as *clean* (*i.e.,* do not affected by any vulnerability) or as affected by specific vulnerabilities.

A. Data Collection

We relied on three datasets (Table II) composed by C/C++ functions and information about the vulnerabilities affecting them. The datasets are described in the following.

GitHub Archive Dataset (GH-DS). We built GH-DS starting from GitHub Archive [54], a dataset containing every public GitHub event (*e.g.,* commits, opening of pull requests) from March 2011.

Since the events generated before 2015 were stored using a deprecated data format, we focused on public events going from January 2015 up to November 2018, when we started the building of GH-DS. We mined from this list of events all vulnerability-fixing commits accompanied by a message containing the exact (i) name of a vulnerability as reported in the CWE dictionary (*e.g.,* Use After Free) and/or (ii) id of a CWE vulnerability in the format CWE-416. We assume that in these commits developers are fixing the vulnerability described in the commit message. However, as we are aware of the fact that commit messages might imprecisely identify bug-fixing commits [55], [56] and, as a consequence, vulnerability-fixing commits, two authors independently analyzed a statistically significant sample (95% confidence level $\pm$5% confidence interval, for a total size of 384) of identified commits to check whether they were actually vulnerability fixes. After solving 45 cases of disagreement, they concluded that 90.3% of the identified vulnerability-fixing commits were true positives (additional details in our replication package [23]). For each vulnerability-fixing commit, we extracted the source code before and after the fix using the GitHub Compare API [57]. This allowed us to collect the vulnerable (pre-commit) and the non-vulnerable (post-commit) code.

We discarded all commits not modifying at least one C/C++ file and those not available anymore on GitHub. We also discarded commits modifying more than one file, to be sure about the "context" of the vulnerability fix. This reduces the likelihood of including tangled commits [58] comprising a vulnerability fix as well as other changes (*e.g.,* some refactoring operations). Finally, we used GumTreeDiff [59] to identify the list of edit actions performed between the vulnerable and the fixed C/C++ files in each commit. In this way, we identified the functions modified to fix the vulnerability, and we built pairs of functions (f_v, f_nv) where f_v represents the vulnerable version of function f (pre-commit) and f_nv represents the fixed (non-vulnerable) version of f (post-commit).

The final GH-DS dataset is composed of ∼315k pairs of vulnerable and non-vulnerable C/C++ functions related to the fixing of 70 different types of vulnerabilities across ∼29k GitHub projects. We release this dataset to the research community [23]. While we use it to experiment vulnerability detection techniques, GH-DS can also be used for the development of approaches to automatically fix vulnerabilities. Indeed, having available the version of the same function before and after the fix of a vulnerability could allow the usage of deep learning techniques such as Neural Machine Translation (NMT) to learn the code changes needed to "translate" a vulnerable function into its non-vulnerable version.

STATE IV Juliet Test Suite Dataset (J-DS) [19]. This existing dataset has been used in previous studies on automated vulnerability detection [11] and includes synthetic code examples (*i.e.,* the vulnerabilities have been manually injected) of 118 types of CWE vulnerabilities. For each example, both the vulnerable and the non-vulnerable versions of the code are available. We did not use the dataset as provided but performed a few cleaning steps aimed at excluding (i) vulnerabilities that are not at function-level (*e.g.,* `CWE401_Memory_Leak__destructor_01_bad.cpp`) and (ii) single vulnerability instances that are spread across more than one function, since in this study we want to investigate the potential of vulnerability detection techniques in the simplest 1-to-1 scenario: One function is clean or it is affected by a single type of vulnerability **and** a vulnerability instance does only affect a single function.

Russell *et al.* Dataset (R-DS). This dataset is available as part of the work by Russell *et al.* [11], and was created by using functions from the Debian Linux distribution and GitHub public repositories. The authors collected ∼1.3M functions and used three static analysis tools (*i.e.,* Clang [20], Cppcheck [21], and Flawfinder [22]) to tag them with a label identifying the type of vulnerability affecting the code (if any). The authors considered six types of labels. Four are related to the most popular types of vulnerabilities they found, *i.e.,* CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), CWE-120 (Buffer Overflow), CWE-469 (Use of Pointer Subtraction to Determine Size), and CWE-476 (NULL Pointer Dereference).

One is a macro category identifying functions affected by other, less popular, vulnerabilities (*i.e.,* CWE-others). Finally, the last label identifies the clean functions. 2% of the functions were tagged with multiple vulnerabilities; 4% of the functions were tagged with one specific type of vulnerability; the rest of the functions (94%) were classified as not vulnerable, which makes it a highly unbalanced dataset. We excluded also from this dataset the functions tagged with multiple vulnerabilities for the reasons previously explained.

B. Code Representation

From each dataset (GH-DS, J-DS, R-DS) we extracted two sets of tuples. The first one, in the form $\langle function\_code, is\_vulnerable \rangle$, aims at experimenting the models in the scenario in which we want to identify vulnerable functions, but we are not interested to the specific type of vulnerability.

Raw Source Code (a)

```
int max(int num1, int num2){
  int result;
  if(num1 > num2)
    result = 1;
  else
    result = 0;
  return result;
}
```

R0 (b)

```
int Identif(int Identif, int Identif){
  int Identif;
  if(Identif > Identif)
    Identif = Literal;
  else
    Identif = Literal;
  return Identif;
}
```

R1 (c)

```
int Function(int Parm, int Parm){
  int Var;
  if(Parm > Parm)
    Var = IntegerLiteral;
  else
    Var = IntegerLiteral;
  return Var;
}
```

R2 (d)

```
int Function(int Parm, int Parm){
  int Var;
  if(Parm > Parm)
    Var = 1;
  else
    Var = 0;
  return Var;
}
```

R3 (e)

```
int Function1(int Parm1, int Parm2){
  int Var1;
  if(Parm1 > Parm2)
    Var1 = IntegerLiteral;
  else
    Var1 = IntegerLiteral;
  return Var1;
}
```

R4 (f)

```
int Function1(int Parm1, int Parm2){
  int Var1;
  if(Parm1 > Parm2)
    Var1 = 1;
  else
    Var1 = 0;
  return Var1;
}
```

Fig. 1: Code representations used in our study (b) to (f). Subfigure (a) shows a sample raw source code provided as input.

In the second, the tuples are instead in the form $\langle function\_code, vulnerability\_type \rangle$, to experiment the models in the scenario in which we want to classify the vulnerability type exhibited by a given function. We use the $non\_vulnerable$ label to identify functions not affected by any vulnerability.

We built five versions of each set of tuples (*i.e.,* binary and multi-class) representing the code in different ways, to study how the code representation affects the performance of the models. The generation of the representations was supported by a custom C/C++ lexer we wrote to extract the needed information (*i.e.,* the syntactic and semantic meaning of the tokens in each function). Using the raw source code as input for machine learning techniques is not an option especially due to the huge vocabulary of terms used in the identifiers and literals. Such a large vocabulary would hinder the goal of learning patterns in the data that characterize vulnerable functions. For this reason, we abstract the code and generate expressive yet vocabulary-limited representations, as done in previous applications of machine learning on code [60]–[62].

In the following, we describe each representation using as reference the source code snippet presented in Fig. 1-(a).

R0: Baseline. This is the most basic representation, since it provides minimum information in terms of code context. We include in R0 the C/C++ reserved keywords and punctuation symbols, while identifiers and literals are represented with the tokens `Identifier` and `Literal`, respectively. Fig. 1-(b) depicts the function in Fig. 1-(a) using R0 as representation.

R1: Augmented R0. As in R0, C/C++ keywords and punctuation symbols are kept, but identifiers are tokenized according to their type (*e.g.,* `Function`, `Parameter`, `Variable`). Also, for tokens related to literals, we explicitly label the ones related to strings as `StringLiteral` and those related to integer values as `IntegerLiteral`.

All the others are just kept as `Literal`. The rationale for this choice is that string and integer literals are involved in security vulnerabilities because, on the one side, strings can be used to represent permissions, access rules, and file paths; on the other side, integers are often used for buffer sizes, iterators indexes/limits, and offsets that play a role in different types of vulnerabilities. Fig. 1-(c) depicts the function in Fig. 1-(a) using R1 as representation.

R2: R1 + Integer values. This representation is almost the same as R1, but all one digit numeric literals are kept in this case. Fig. 1-(d) shows the R2 representation of the function used as running example. The rationale behind this representation is that integers are responsible for many types of vulnerabilities (*e.g.,* CWE-119, CWE-125, CWE-129, CWE-369, CWE-469, CWE-665, CWE-682, CWE-839) [63].

R3: R1 + Identifiers Numbering. Also R3 is built on top of R1. In this case, instead of listing all identifiers with their type, we keep track of the identifiers by numbering them, as done by Tufano *et al.* [62]. Assume that two variables are present in a function, and these variables are used several times inside the function body. With the R1 representation, all usages of these variables will be replaced with the `Var` token. In R3, all the instances of the first variable will be replaced with the `Var1` token, and all those of the second variable will use the `Var2` token (see Fig. 1-(e)). This representation aims at "informing" the classification model about the fact that the same identifier is used in different parts of the function body which could allow for identifying data flow dependencies.

R4: R2 + Identifiers Numbering. This is the representation providing more contextual information: We start from R2 and integrate the identifiers' numbering previously explained for R3. Fig. 1-(f) depicts this representation using as reference the example in Fig. 1-(a).

C. Data Cleaning

Before using the three datasets to train and evaluate the experimented models, we performed a transformation and cleaning process on each of them to (i) make the data treatable with DL/shallow models, and (ii) avoid possible pitfalls that are common in studies of machine learning on code (*e.g.,* duplicated functions due to forked projects [60], [64]).

First, for each dataset we created different versions using the five code representations shown in Fig. 1. This gave us five different datasets of function representations. Second, we addressed conflicting representation (*i.e.,* two samples with same code representation and different labels) and duplicates. In case of conflicting representations, all instances were removed. As for the duplicates, we removed all duplicates having the same raw source code representation and the same label (*i.e.,* type of vulnerability affecting them, if any), keeping only the first occurrence. This means that it is possible to have in our datasets two snippets having the same abstract representation, but not the same raw source code. Such a design choice is justified by the fact that the translation from raw source code to abstract representation is part of the classification pipelines used in ML implementations, and it is performed after the removal of duplicates.

Table III: Number of functions in each dataset for each representation after handling duplicates and conflicting representations.

	GH-DS	J-DS	R-DS
R0	164,919	56,073	1,117,137
R1	166,187	56,075	1,117,765
R2	167,280	56,083	1,117,925
R3	172,421	56,075	1,118,349
R4	173,498	56,083	1,118,488

Table IV: Number of functions in each dataset for each representation after data cleaning. Total number of functions (F), percentage of duplicates (D)

	GH-DS		J-DS		R-DS	
	F	D	F	D	F	D
R0	70,618	26.61%	28,572	36.18%	563,646	0%
R1	71,757	26.52%	28,572	34.97%	563,752	0%
R2	72,214	26.42%	28,572	34.93%	563,817	0%
R3	79,216	28.26%	28,572	34.63%	563,923	0%
R4	79,545	28.19%	28,572	34.58%	563,970	0%

Duplicates and conflicting instances can be different in the same dataset when using different representations. For example, two functions can be different using the more expressive R4 representation and equal when using the basic R0 representation. In this case, the duplicate will be removed from the dataset only when using the R0 representation. Table III summarizes the final number of functions in each dataset after this process. It is important to mention that after this process GH-DS is no longer balanced.

Third, for each dataset, we computed the distribution of number of tokens for all function representations it contains. Then, for each dataset representation, we excluded functions that are shorter than the first quartile (*e.g.,* for R1 J-DS 59 tokens) or longer than the third quartile (*e.g.,* for R1 J-DS 115 tokens). This has been done to exclude very short and very long functions from our dataset since the former provide very little information to characterize the presence of a vulnerability and the second are computationally expensive to process with DL models. Removing upfront very long functions, also prevents from truncating sentences (in our case, functions) to a fixed maximum length during the training, which would provide incomplete inputs to the model, something suboptimal when working with source code. After this step, the data cleaning process for binary classification ends, and the final number of functions (F) for this type of classification is reported in Table IV, which reports also the percentage of duplicates in the abstract representation.

Remember that duplicates can be in the abstract representation but not in the raw source code. The maximum times a single sample repeats is 18 for J-DS and 77 for GH-DS.

Finally, to have enough data points to train the multi-class models in charge of identifying the specific type of vulnerability affecting a given function, we kept in each dataset only the five most represented vulnerabilities and the samples belonging to the "non-vulnerable" class (see Table V).

Table V: Top five vulnerabilities in each dataset. *Russell has only four vulnerabilities, and two of them were merged

Dataset	Top Vulnerabilities
GH-DS	Deadlock, Race Condition, Null Pointer Dereference, Buffer Overflow, Dead Code
J-DS	CWE-401, CWE-762, CWE-590, CWE-122, CWE-121
R-DS*	CWE-other, CWE-120, CWE-476

Table VI: Number of functions in each dataset for each representation after data cleaning for multi-class classification. Total number of functions (F), percentage of duplicates (D).

	GH-DS		J-DS		R-DS	
	F	D	F	D	F	D
R0	57,475	33.19%	11,445	35.98%	563,646	0%
R1	57,947	32.76%	11,445	34.95%	563,752	0%
R2	58,390	32.70%	11,445	34.38%	563,817	0%
R3	60,305	31.00%	11,445	34.49%	563,923	0%
R4	60,768	31.00%	11,445	34.38%	563,970	0%

Note that this was only needed for GH-DS and J-DS, since R-DS does already focus on four possible categories of vulnerabilities. In R-DS one of the possible categories (CWE-119), after the cleaning process, had only five samples and we decided to merge it with another category (CWE-120), since representing similar vulnerabilities. Table VI summarizes the final number of functions for multi-class classification.

At the end of this process we obtained 30 different datasets. The first 15 datasets (five representations times three datasets) correspond to the binary classification, whereby the labels can only take two possible values (vulnerable or non-vulnerable). The other 15 datasets correspond to the multi-class classification and their labels represent a specific type of vulnerability or a non-vulnerable instance.

D. Classifiers

Given the four approaches (*i.e.,* GCP-AutoML, RF, CNN & RNN), five representations, three datasets, and two types of classification (binary and multi-class) used in our study, we built a total of 120 different models ($4 * 5 * 3 * 2$).

As previously mentioned, we started by training our shallow learning model (RF), the baseline (GCP-AutoML) and the two aforementioned deep learning models (CNN and RNN). Regarding GCP-AutoML, the training process is the same for both binary and multi-class classification and it consists of only two steps: (i) we upload a given dataset featuring a function representation (*e.g.,* R0 for J-DS) with its corresponding label (*e.g.,* vulnerable); and (ii) we trained a model with the GCP implementation of AutoML which uses NAS [53], [65]. NAS uses different optimization algorithms to find an optimal neural network architecture without requiring human expertise [53]. Thus, for the GCP-AutoML representing our baseline we did not need to define the type of model, the search space for the hyper-parameters, nor to process the input from text to sequence tokens. This is all automatically handled.

Concerning the RF, for binary and multi-class classification the model is the same, with the only difference being the labels in the training/test set.

Table VII: Hyper-parameters that were tuned for each type of model.

Model	Hyper-parameters tuned
RF	Number of estimators, Number of features to consider at every split, Maximum number of levels in tree, Function used to evaluate the quality of a split
CNN	Dimension of the dense embedding, number of Convolution layers, Number of output filters, Activation function for Convolution layer, Initializer for the kernel weights matrix, Stride Length of the convolution, Number of dense layers, Drop out rate, Activation function for each dense layer, Batch size
RNN	Dimension of the dense embedding, Number of LSTM layers, Activation function for recurrent layer,Number of units of the recurrent layer, Initializer for the kernel weights matrix, Regularizing factor for the kernel weights matrix, drop our rate, Number of dense layers, Dense Layers Activation function, Batch size, Optimizer's learning rate

We used a balanced class weighting to contrast imbalanced classes. For this model the input data is a Bag of words (BOW) of the specific representation and the output is a label classifying the instance. We used the Scikit learn [66] implementation and the hyper-parameters we tuned are reported in Table VII.

For RNN and CNN models, the input layer takes as input a token sequence extracted from the specific code representation and converts it into a fixed k-dimensional embedding representation. The k value is one of the hyper-parameters to tune. The output layer is a one-hot encoded vector that corresponds to a particular label. For the binary case the output layer has a sigmoid activation function and one neuron.

For the multi-class scenario it uses a softmax activation function and the number of neurons is equivalent to the number of prediction classes (*e.g.,* for R-DS the output layer has 4 neurons, but for J-DS and GH-DS it uses six neurons). The hidden layers between the input and output layers differ between these two models.

The RNN has LSTM layers after the embedding layer. The hyper-parameters we tune for this layer are: the number of units and the initialization of the kernels and the activation function. After this layer, the RNN uses a drop out layer to prevent overfitting followed by dense layers.

For the CNN, after the embedding layer, we have 1D dimensional convolutional layers, since we are working with one dimensional data (text), as opposed to other contexts in which more dimensions are needed (*e.g.,* to manipulate images). The hyper-parameters that we tune for this layer are: the number of filters, the kernel size and the activation function. After this layer, the CNN model includes a global max pooling layer, a drop out layer (to reduce the chance of overfitting) with a rate that can be tuned, and one or two dense layers depending on the hyper-parameter optimization process.

For both CNN and RNN, the learning rate and the batch size were also hyper-parameters to tune. We used the Adam [67] optimizer. For both models we used Keras [68]. The set of tuned hyper-parameters is reported in Table VII.

E. Analysis Method

We split each of the 30 datasets into training (60%), validation (20%), and test (20%) set using stratified random sampling. This strategy ensures that the labels frequency distribution is approximately equal between the sets and similar to the original distribution. For GCP-AutoML, we uploaded a *.csv* file with the following information for each sample: type of set (*i.e.,* train, validation, test) it belongs to, its abstract representation, and its label (*e.g.,* vulnerable). The datasets used for GCP-AutoML are the same used in our models, with the only difference that GCP-AutoML requires to remove the duplicates even at abstract representations.

Since we had two different families of approaches (*i.e.,* shallow and deep learning) which expect a different type of input (*i.e.,* BOW *vs* sequence of tokens), it was necessary to add a lightweight preprocessing to the data. For the RF, we generate, starting from the code representation, a TF-IDF BOW to weight the importance of each token in the BOW.

For the CNN and RNN models, it was necessary to convert each text representation into a sequence of tokens of the same length. All the instances in each dataset that were shorter than the upper limit were post padded: We pad sequences with a special token until it reaches a fixed length.

Training and validation sets were used to build the models and to tune the hyper-parameters of each combination of classifier and code representation, while the test set was used for assessing the performance of the models. When tuning CNN, RNN and RF we used a Bayesian-search [69], [70]. This type of optimization performs a directed search based on the surrogate function and an acquisition function. A surrogate function is a model that is used for approximating the objective function, while the acquisition function directs sampling to combinations of parameters where an improvement over the best observation is likely. For this process we used the Optuna [71] Python library.

In the binary classification scenario, the three models (*i.e.,* RF, CNN, RNN) were trained with cross entropy loss, while for multi-class classification we adopted the categorical cross entropy loss. In the tuning process we optimized accuracy for binary classification and categorical-accuracy for multi-class.

Once each model was tuned, its performance has been evaluated on the test set. We computed ***Accuracy*** and Matthews Correlation Coefficient (MCC) [72]–[74] for binary and multi-class scenarios. Accuracy evaluates the overall performance of the models while MCC was used to evaluate the performance with a metric that is less influenced by imbalanced data [75]–[77]. Further, we also compute Precision, Recall and F1 for the binary classification scenario, having vulnerable functions as the positive class. For the multi-class scenario we compute macro averages for Precision, Recall and F1. For the GCP-AutoML model we calculate the aforementioned metrics based on the confusion matrix that GCP returns for the test set.

IV. Results

We start by discussing the results achieved by the binary classifiers, having the goal of discriminating between vulnerable and non-vulnerable functions.

Then, we present the findings related to the multi-class classifiers, which classify functions as non-vulnerable or as affected by a specific type of vulnerability. The results achieved by the models for different representations of the code should not be directly compared, since small differences exist in these datasets due to the removal of duplicates. However, within the same representation and dataset, the performance of the different models can be compared. AutoML, due to its peculiarities, only serves as baseline to assess the performance a model only requiring a minimum human intervention is capable of achieving for this task. The results achieved by the experimented models can be found in Fig. 2.

A. Binary Classification

In binary classification, the models can be used to flag functions likely to be vulnerable, for a subsequent code review performed by developers. We discuss three possible scenarios: (I) Recall is favored over precision, meaning that the cost of reviewing the vulnerable functions flagged by the models is considered low by the developers (*i.e.,* a high number of functions to check flagged as potentially vulnerable is acceptable) while the potential cost of a missed vulnerable function is high (*i.e.,* it is not acceptable to release the code with a vulnerable function); (II) Precision is favored over recall, meaning that the manual inspection cost is considered high and the risk of missing a vulnerability is considered limited; (III) Balancing precision and recall is desirable, with metrics such as F1, MCC or Accuracy that should be maximized.

Before presenting the best models for each of these scenarios, it is important to discuss the results achieved for the J-DS dataset (top part of Fig. 2): All models exhibit extremely high performance, with the three DL-based being perfect predictors (*i.e.,* MCC=1) and the RF being very close (MCC$>$0.99).

These results, especially when compared with the ones achieved on the other two datasets, clearly highlight potential issues with the synthetic origin of this dataset, that includes artificially created instances of vulnerable functions. It is worth noting that this dataset has been used in previous works on vulnerability detection [11], [15]. From a manual inspection of its instances, we found that almost all non-vulnerable samples are characterized by the presence of the $static$ token, that makes the classification task trivial for the models. This poses doubt on the usefulness of this dataset for assessing the performance of vulnerability detection approaches.

Moving to the GH-DS dataset that, as explained in Section III, was built by using real instances of vulnerable functions mined from GitHub; the results achieved by the models with GH-DS, as expected, substantially drop. Overall, the best models are the RF and the CNN that achieve very similar performance. Considering the higher training cost of the CNN classifier, a RF could be preferred in many scenarios.

Also, while RF and CNN are undisputedly the best model in the scenario II (maximize precision) and III (maximize overall accuracy)*, this is not always the case for I (maximize recall).

*The only exception to this trend is the RNN with the R3 representation.

Binary Classifiers

		RF					AutoML					CNN					RNN				
Data	Rep.	P	R	F1	ACC	MCC	P	R	F1	ACC	MCC	P	R	F1	ACC	MCC	P	R	F1	ACC	MCC
J-DS	R0	100%	99.76%	99.88%	99.93%	0.9983	100%	100%	100%	100%	1	100%	100%	100%	100%	1	100%	100%	100%	100%	1
	R1	100%	99.82%	99.91%	99.95%	0.9987	100%	100%	100%	100%	1	100%	100%	100%	100%	1	100%	100%	100%	100%	1
	R2	100%	99.94%	99.97%	99.98%	0.9996	100%	100%	100%	100%	1	100%	100%	100%	100%	1	100%	100%	100%	100%	1
	R3	100%	99.82%	99.91%	99.95%	0.9987	100%	100%	100%	100%	1	100%	100%	100%	100%	1	100%	100%	100%	100%	1
	R4	100%	99.82%	99.91%	99.95%	0.9987	100%	100%	100%	100%	1	100%	100%	100%	100%	1	100%	100%	100%	100%	1
GH-DS	R0	62.37%	63.16%	62.77%	62.43%	0.2486	50.05%	37.54%	42.90%	50.59%	0.0093	62.72%	64.15%	63.43%	62.91%	0.2583	52.21%	54.66%	53.41%	52.19%	0.0437
	R1	63.23%	56.74%	59.81%	61.82%	0.2377	58.82%	0.45%	0.89%	50.61%	0.0115	65.71%	65.58%	65.65%	65.57%	0.3113	54.29%	16.66%	25.49%	51.25%	0.0359
	R2	61.01%	61.60%	61.31%	61.08%	0.2215	51.92%	0.60%	1.18%	51.09%	0.0045	61.20%	62.67%	61.92%	61.42%	0.2885	50.49%	92.91%	65.43%	50.85%	0.0296
	R3	63.46%	75.90%	69.12%	66.26%	0.3322	53.31%	87.55%	66.27%	56.50%	0.1810	61.71%	62.63%	62.17%	62.07%	0.2414	62.69%	97.30%	76.25%	69.84%	0.4762
	R4	62.62%	76.89%	69.03%	65.66%	0.3223	57.46%	88.60%	69.71%	62.23%	0.2963	63.18%	64.80%	63.98%	63.69%	0.2739	50.87%	35.52%	41.83%	50.84%	0.0161
R-DS	R0	78.66%	2.97%	5.72%	96.23%	0.1482	53.33%	0.41%	0.82%	96.08%	0.0443	74.14%	0.99%	1.95%	96.17%	0.0828	62.70%	1.82%	3.53%	96.17%	0.1022
	R1	79.89%	3.47%	6.65%	96.24%	0.1618	42.32%	10.28%	16.54%	95.99%	0.1942	72.58%	1.03%	2.04%	96.17%	0.0837	53.21%	2.67%	5.08%	96.15%	0.1128
	R2	76.09%	4.02%	7.64%	96.25%	0.1695	43.64%	5.63%	9.98%	96.09%	0.1462	61.19%	0.94%	1.86%	96.16%	0.0726	50.55%	3.15%	5.93%	96.14%	0.1190
	R3	76.97%	2.69%	5.20%	96.22%	0.1395	47.10%	1.88%	3.61%	96.10%	0.0881	59.49%	1.08%	2.12%	96.16%	0.0765	50.43%	2.69%	5.11%	96.14%	0.1098
	R4	75.76%	2.87%	5.54%	96.22%	0.1429	64.50%	3.33%	6.33%	96.18%	0.1406	62.07%	1.24%	2.43%	96.16%	0.0840	48.32%	2.64%	5.01%	96.14%	0.1062

Multi-Class Classifiers

		RF					AutoML					CNN					RNN				
Data	Rep.	P	R	F1	ACC	MCC	P	R	F1	ACC	MCC	P	R	F1	ACC	MCC	P	R	F1	ACC	MCC
J-DS	R0	99.91%	99.65%	99.78%	99.87%	0.9977	100%	100%	100%	100%	1	99.00%	98.69%	98.84%	99.52%	0.9917	99.79%	99.93%	99.86%	99.96%	0.9992
	R1	99.78%	99.72%	99.75%	99.91%	0.9985	100%	100%	100%	100%	1	99.69%	99.51%	99.60%	99.87%	0.9977	99.59%	99.44%	99.51%	99.83%	0.9970
	R2	99.98%	99.57%	99.77%	99.91%	0.9985	100%	100%	100%	100%	1	99.69%	99.65%	96.67%	99.87%	0.9977	99.63%	99.68%	99.66%	99.83%	0.9970
	R3	99.74%	99.37%	99.55%	99.78%	0.9962	99.79%	99.63%	99.71%	99.90%	0.9987	100%	100%	100%	100%	1	99.74%	99.49%	99.61%	99.83%	0.9970
	R4	99.69%	99.37%	99.52%	99.78%	0.9962	100%	100%	100%	100%	1	99.79%	99.93%	99.86%	99.96%	0.9992	99.57%	99.80%	99.68%	99.87%	0.9970
GH-DS	R0	65.10%	53.20%	57.38%	64.53%	0.4692	31.97%	26.57%	21.84%	50.76%	0.0524	65.38%	25.85%	27.34%	54.08%	0.2279	54.20%	41.28%	44.31%	57.58%	0.3345
	R1	61.76%	54.80%	57.72%	62.21%	0.4341	38.60%	29.69%	24.50%	50.83%	0.0464	57.04%	28.03%	30.92%	53.54%	0.2178	45.38%	34.41%	37.21%	54.67%	0.2679
	R2	62.73%	55.34%	58.41%	62.99%	0.4456	55.28%	26.03%	22.01%	50.50%	0.0653	55.01%	39.92%	44.35%	56.78%	0.3156	52.13%	47.74%	49.44%	57.24%	0.3604
	R3	60.12%	52.22%	55.44%	61.13%	0.4124	32.88%	27.13%	23.92%	50.26%	0.0412	54.41%	34.50%	38.14%	56.37%	0.2890	48.23%	38.55%	40.64%	54.54%	0.3017
	R4	60.93%	52.31%	55.83%	61.55%	0.4169	35.23%	26.11%	24.20%	48.77%	0.0433	48.06%	39.16%	41.90%	54.13%	0.2869	54.15%	33.07%	35.89%	57.71%	0.3104
R-DS	R0	82.02%	29.00%	31.78%	96.26%	0.1771	57.55%	32.75%	33.59%	96.19%	0.0883	37.44%	27.00%	28.00%	96.15%	0.0908	38.75%	26.76%	27.68%	96.16%	0.0878
	R1	86.43%	28.27%	30.57%	96.25%	0.1649	63.89%	38.85%	40.30%	96.16%	0.1184	37.80%	26.94%	27.92%	96.15%	0.0897	56.49%	28.10%	29.70%	96.17%	0.1195
	R2	83.24%	28.90%	31.68%	96.26%	0.1777	65.16%	41.29%	43.41%	96.18%	0.1472	53.53%	27.04%	28.19%	96.17%	0.1010	66.97%	27.40%	28.76%	96.19%	0.1150
	R3	74.83%	28.50%	30.89%	96.22%	0.1543	60.12%	40.83%	43.43%	96.15%	0.1569	65.58%	28.83%	31.27%	96.19%	0.1455	48.10%	27.96%	29.76%	96.15%	0.1140
	R4	81.43%	28.26%	30.46%	96.23%	0.1549	70.30%	43.24%	45.22%	96.22%	0.1434	39.31%	27.44%	28.73%	96.17%	0.1066	48.66%	28.23%	29.90%	96.16%	0.1186

Fig. 2: Top: Binary Classification metrics: Precision (P), Recall (R), F1, Accuracy (ACC), and MCC for each approach and dataset-representation, using vulnerable as the positive class; Bottom: Mutli-class Classification metrics: Macro Precision (P), Macro Recall (R), Macro F1, Accuracy (ACC), and MCC for each approach and dataset-representation

Indeed, here the RNN model and the AutoML model manage to achieve better results for specific code representations..

Concerning the third dataset (*i.e.,* R-DS) we also obtained more "realistic" results as compared to J-DS, with all models struggling to obtain high values for recall. Indeed, independently from the used model and code representation, the recall is usually lower than 5%. In this scenario, the RF seems to be the best solution. Indeed, while also for it the recall is extremely low, at least the shallow model is the one able to achieve the best precision values.

Concluding remarks. The results achieved by our baseline (AutoML), are inline or lower as compared to the other three experimented approaches. This serves as a sort of sanity check for our implementations, confirming how challenging the tackled problem is (*i.e.,* automated vulnerability detection) even in the simpler scenario (*i.e.,* binary classification).

Despite RF being representative of a shallow learning model, it achieved performance comparable or, in some cases, even better (see the R-DS) than the deep learning models.

The result achieved on J-DS can instead be basically ignored, if not for the message they give to the research community about the usage of this dataset in the assessment of vulnerability detection tools.

To summarize our findings, we also present in Fig. 3 a graphical representation conveying the number of times a model obtained the greatest score for each metric (*e.g.,* Precision) given a representation. Fig. 3 shows that (i) the RF is the better performing in terms of precision; and (ii) CNN is the most "balanced" across all considered metrics.

B. Multi-class Classification

In the multi-class classification (bottom part of Fig. 2), we have two main scenarios: (I) The first looks at the overall performance of the models (*i.e.,* accuracy), without considering the imbalanced nature of the subject datasets (*i.e.,* a vulnerability type might be more frequent than another); (II) In the second, by analyzing the MCC we can evaluate how well a model scores considering the unbalanced datasets we deal with.

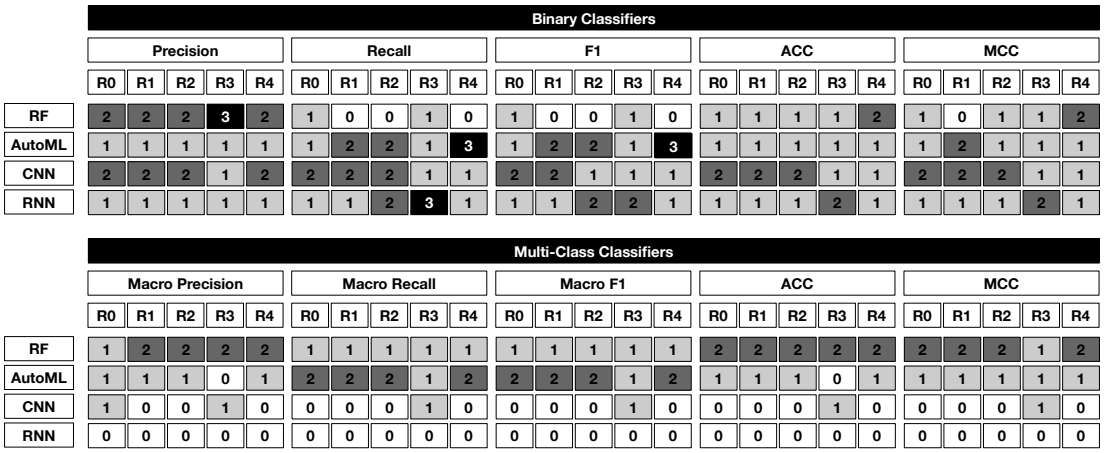

Fig. 3: Heatmap conveying the amount of times a model obtained the greatest score given a metric. Top: Binary Classification metrics; Bottom: Mutli-class Classification metrics.

The first scenario is useful to give higher weight to the majority class (non-vulnerable functions) when computing the performance of the model; the second looks for a model that is able to provide good performance across all classes. As already observed for the binary classification problem, the results achieved by all models on the J-DS dataset are extremely high, confirming the unchallenging classification task of this dataset.

Concerning GH-DS, RF is consistently the best model, achieving the greatest values for all the scores. RF also obtained very balanced macro-precision and macro-recall values, that differ less than 7%, being a model that does not sacrifice precision for recall or *vice versa*. Furthermore, RF performs well both when considering accuracy (>60%) and MCC (>0.4) as performance proxy.

Finally, also for the R-DS dataset, the RF was overall the best model in terms of both MCC and Accuracy, with the exception of AutoML with the R3 representation. However, the difference in performance in favor of AutoML is relatively small, while its higher training cost is substantial as compared to the RF. It is also worth noticing that all models on this dataset (R-DS) achieve a high accuracy in all the cases (>96%). This is due to the fact that this dataset is highly imbalanced (see Table II) and a constant classifier always predicting the functions as "non-vulnerable" would achieve, on this dataset a ∼95% accuracy. Thus the results in terms of MCC are more representative for this dataset.

Concluding remarks. Shallow learning using RF performed well across all different datasets, being the best model in the multi-class scenario. It is important to highlight that this finding holds for the specific DL-models, settings, datasets, and code representations we used. Our study does not claim the superiority of shallow over deep learning, but rather the fact that shallow learning baselines such as the RF, properly tuned, should always be considered as a valid baseline to compare with for vulnerability detection.

Also in this case, we conclude our results discussion presenting the bottom part of the heatmap in Fig. 3.

The RF confirms its superiority both in terms of accuracy and MCC, with the RNN not being the best model for any of the investigated representations and metrics.

V. Threats to Validity

Threats to **construct validity** concern the relation between the theory and the observation. When building GH-DS we identified vulnerability-fixing commits based on commit messages, assuming that the developer is fixing the vulnerability described in the commit message. However, it is known that parsing commit messages might imprecisely identify bug-fixing commits [55], [56] and, for similar reasons, vulnerability-fixing commits. To mitigate subjectivity bias, two authors independently analyzed a statistically significant sample of 384 commits identified by our heuristics as vulnerability-fixing commits. After solving conflicts, they found that 347 of them represented true positives (*i.e.,* correctly identified commits).

As previously explained, the results achieved by the models for different representations of the code should not be directly compared, since small differences exist in these datasets due to the removal of duplicates. However, within the same representation and dataset, the performance of the different models can be compared, and we did that by looking at a wide set of metrics assessing different aspects of the classification.

Threats to **internal validity** concern external factors we did not consider that could affect the variables and the relations being investigated. We applied stratified random sampling when splitting the dataset into training (60%), validation (20%), and test (20%) set to ensure that the labels frequency distribution was similar among the sets and to the original distribution.

An important factor that influences the models' performance is calibration of hyper-parameters, which has been performed as detailed in Section III. We are aware that additional tuning, possibly performed by using a different search strategy, could produce different and even better performances, especially for the DL models.

It is also important to highlight that a substantial difference exists between the tuning and training process performed for the three models subject of our study (*i.e.,* RF, RNN, and CNN) as compared to the AutoML baseline. For the former, we defined the search space for hyper-parameters tuning and we performed the preprocessing needed to transform our original abstract representations into a tokens sequence or a BOW representation.

As for AutoML, we just uploaded the original abstract representations to the GCP, with the latter taking care of modifying the input if needed, deciding also the hyper-parameters to be tuned. While AutoML reduces the human interaction needed in the process of finding a good model, it does not allow to get insights about the model's architecture and, for this reason, we had to treat it as a black-box.

Threats to **conclusion validity** concern the relationship between treatment and outcome. We adopted metrics allowing to obtain a clear overview of the performance provided by the models in the experimented scenarios. For example, in the case of multi-class classifiers, taking only the Accuracy into account does not provide a picture of how well the model works across all classes involved in the classification (*e.g.,* the different types of vulnerabilities). Furthermore, in multi-class classification it is important to document the type of average adopted to avoid a misunderstanding of the models performance: macro-average gives the same importance to each class being insensitive to class imbalance, while micro-average maximizes hits.

Our results concerning J-DS and R-DS cannot be directly compared with those reported in previous works sharing some of the datasets we used [11], [15]. This is due to several reasons. First, the models are not being trained with the same amount of samples (*i.e.,* number of functions). Second, the code granularity varies: Li *et al.* [15] use *code gadgets*, *i.e.,* a number of (not necessarily consecutive) lines of code that are semantically related to each other, as opposed to our function-level granularity. Third, the classification approaches differ from ours: While we studied binary and multi-class classification, Li *et al.* [15] train multiple binary classifiers, one for each type of vulnerability they studied, instead of a multi-class classifier. Finally, the adopted performance metrics differ among the discussed studies. For example the type of multi-class averaging adopted is not documented in those studies. Thus, our study should not be considered as a replication of previous work, but as complementary to past studies.

With respect to duplicate handling, we are aware and documented the presence of duplicates in our datasets due to the translation procedure we adopted from raw source code to the different abstract representations. Still, we removed duplicates at raw source code level, but intend to investigate in future how the performance of the approaches change when removing duplicated abstracted instances. In relation to duplicates handling for the AutoML baseline, since this utility is a black-box from the user point of view and it expects a dataset without duplicates we had to remove the duplicated abstracted instances to avoid receiving processing errors.

Threats to **external validity** concern the generalizability of our findings. We conducted our study with several variables involved, namely four approaches (*i.e.,* GCP-AutoML, RF, CNN & RNN), five code representations, three datasets, and two types of classification—binary and multi-class). This required building a total of 120 models ($4*5*3*2$). Also, our datasets are considerably different, including both synthetic instances and real-world vulnerable instances mined from open source projects. Overall, the three datasets account for $\sim$1.8M functions. However, all datasets focus on C/C++ code and report software vulnerabilities at function level. Therefore, the achieved results may not be valid for other languages and/or different granularity levels.

VI. Conclusions

We presented a large-scale empirical study aimed at analyzing the effectiveness of deep and shallow learning techniques for detecting software vulnerabilities in source code. Our study is based on the analysis of three datasets of C/C++ code reporting software vulnerabilities at the function granularity level, accounting for a total of $\sim$1.8M functions, of which $\sim$400k are vulnerable.

Our results show that there is large room for improvement in the field of automated vulnerability detection, independently from the usage of shallow or deep models. Indeed, our results show that shallow models such as the Random Forest can achieve competitive performance for the specific task we investigated.

Our future work will be mostly driven by the findings discussed in Section IV. First, we are planning to expand our study by building datasets featuring other programming languages in order to obtain more generalized conclusions. Second, while we adopted a simple code representation based on streams of tokens for the DL-based systems, more complex representations tailored for source code have been proposed recently (see *e.g.,* [78]); we plan to investigate the possible benefits brought by these representations in our models. We will also experiment with other models such as ensembles, BLSTM, GRU, and transformers [79]. Another avenue for future work is the analysis of the impact of different combinations of code granularities and representations in the detection models. Finally, we will extend the study to include the multi-label case, *i.e.,* code units annotated with more than one vulnerability.

VII. Data Availability

The data used in our study are publicly available [23].

Acknowledgment

Mazuera-Rozo and Bavota gratefully acknowledge the financial support of the Swiss National Science Foundation for the CCQR project (SNF Project No. 175513). Linares-Vásquez is partially funded by a Google Latin America Research Award (LARA) 2018-2021, and the Uniandes Vice-rectory for Research under the CI-120 call for grants.

REFERENCES

[1] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," ser. ICPC 2017.

[2] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," ser. ASE 2018, 2018.

[3] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *CoRR*, 2019. [Online]. Available: http://arxiv.org/abs/1901.01808

[4] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian, "Deepdelta: Learning to repair compilation errors," ser. ESEC/FSE 2019, 2019.

[5] H. Hata, E. Shihab, and G. Neubig, "Learning to generate corrective patches using neural machine translation," *CoRR*, 2018. [Online]. Available: http://arxiv.org/abs/1812.07170

[6] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," ser. ASE 2016, 2016.

[7] X. Gu, H. Zhang, and S. Kim, "Deep code search," ser. ICSE '18, 2018.

[8] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," ser. ICSE '19, 2019.

[9] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: How far are we?" ser. ASE 2018, 2018.

[10] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, ser. ICSE '18, 2018.

[11] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, "Automated vulnerability detection in source code using deep representation learning," *CoRR*, 2018.

[12] 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)*, 2017.

[13] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, S. Wang, and J. Wang, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *CoRR*, 2018.

[14] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, M. W. McConley, J. M. Opper, S. P. Chin, and T. Lazovich, "Automated software vulnerability detection with machine learning," *CoRR*, 2018.

[15] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proceedings 2018 Network and Distributed System Security Symposium*, 2018.

[16] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "Vuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[17] Y. Wang, Z. Wu, Q. Wei, and Q. Wang, "Neufuzz: Efficient fuzzing with deep neural network," *IEEE Access*, 2019.

[18] G. Lin, J. Zhang, W. Luo, L. Pan, O. De Vel, P. Montague, and Y. Xiang, "Software vulnerability discovery via learning multi-domain knowledge bases," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[19] NIST. Juliet test suite v1.3, 2017. [Online]. Available: https://samate.nist.gov/SRD/testsuite.php

[20] Clang. Clang. [Online]. Available: https://clang.llvm.org/

[21] Cppcheck. Cppcheck. [Online]. Available: http://cppcheck.sourceforge.net/

[22] D. A. Wheeler. Flawfinder. [Online]. Available: https://www.dwheeler.com/flawfinder/

[23] A. Mazuera-Rozo, A. Mojica-Hanke, M. Linares-Vásquez, and G. Bavota, "Replication package." https://tinyurl.com/ShallowOrDeep.

[24] M. Linares-Vásquez, G. Bavota, and C. Escobar-Velásquez, "An empirical study on android-related vulnerabilities," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017.

[25] A. Mazuera-Rozo, J. Bautista-Mora, M. Linares-Vásquez, S. Rueda, and G. Bavota, "The android OS stack and its vulnerabilities: an empirical study," *Empirical Software Engineering*, 2019.

[26] A. V. Barabanov, A. S. Markov, M. I. Grishin, and V. L. Tsirlov, "Current taxonomy of information security threats in software development life cycle," in *2018 IEEE 12th International Conference on Application of Information and Communication Technologies (AICT)*, 2018.

[27] S. Rafique, M. Humayun, B. Hamid, A. Abbas, M. Akhtar, and K. Iqbal, "Web application security vulnerabilities detection approaches: A systematic mapping study," in *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2015.

[28] S. Huang, H. Tang, M. Zhang, and J. Tian, "Text clustering on national vulnerability database," in *2010 Second International Conference on Computer Engineering and Applications*, 2010.

[29] M. R. Islam, M. F. Zibran, and A. Nagpal, "Security vulnerabilities in categories of clones and non-cloned code: An empirical study," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017.

[30] H. Homaei and H. R. Shahriari, "Seven years of software vulnerabilities: The ebb and flow," *IEEE Security & Privacy*, 2017.

[31] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, "The attack of the clones: A study of the impact of shared code on vulnerability patching," in *2015 IEEE Symposium on Security and Privacy*, 2015.

[32] D. Papp, Z. Ma, and L. Buttyan, "Embedded systems security: Threats, vulnerabilities, and attack taxonomy," in *2015 13th Annual Conference on Privacy, Security and Trust (PST)*, 2015.

[33] P.-A. V. J. C. Platon Kotzias, Leyla Bilge. (2019) Mind your own business a longitudinal study of threats and vulnerabilities in enterprises.

[34] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, "How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[35] Checkmarx. Checkmarx. [Online]. Available: https://www.checkmarx.com/

[36] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *USENIX ATC 2012*, 2012. [Online]. Available: https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker

[37] O. Chebaro, P. Cuoq, N. Kosmatov, B. Marre, A. Pacalet, N. Williams, and B. Yakobowski, "Behind the scenes in SANTE: a combination of static and dynamic analyses," *Automated Software Engineering*, 2013.

[38] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR. CHECKER: A soundy analysis for linux kernel drivers," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[39] L. Bello-Jiménez, A. Mazuera-Rozo, M. Linares-Vásquez, and G. Bavota, "Opia: A tool for on-device testing of vulnerabilities in android applications," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019.

[40] X. Yuan, O. Setayeshfar, H. Yan, P. Panage, X. Wei, and K. H. Lee, "Droidforensics: Accurate reconstruction of android attacks via multi-layer forensic logging," ser. ASIA CCS '17, 2017.

[41] D. J. Musliner, J. M. Rye, and T. Marble, "Using concolic testing to refine vulnerability profiles in fuzzbuster," in *2012 IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, 2012.

[42] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," 2017.

[43] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[44] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, "Software vulnerability prediction using text analysis techniques," ser. MetriSec '12, 2012.

[45] M. Ceccato, C. D. Nguyen, D. Appelt, and L. C. Briand, "SOFIA: an automated security oracle for black-box testing of SQL-injection vulnerabilities," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, 2016.

[46] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," ser. CCS '15, 2015.

[47] G. Lin, S. Wen, Q. L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," *Proceedings of the IEEE*, 2020.

[48] R. W. Selby and A. A. Porter, "Learning from examples: generation and evaluation of decision trees for software resource analysis," *IEEE Transactions on Software Engineering*, vol. 14, no. 12, pp. 1743–1757, 1988.

[49] A. A. Porter and R. W. Selby, "Evaluating techniques for generating metric-based classification trees," *Journal of Systems and Software*, 1990.

[50] L. C. Briand, V. R. Basili, and W. M. Thomas, "A pattern recognition approach for software engineering data analysis," *IEEE Transactions on Software Engineering*, 1992.

[51] X. He, K. Zhao, and X. Chu, "Automl: A survey of the state-of-the-art," 2020.

[52] C. Wong, N. Houlsby, Y. Lu, and A. Gesmundo, "Transfer learning with neural automl," ser. NIPS'18, 2018.

[53] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2017. [Online]. Available: https://arxiv.org/abs/1611.01578

[54] I. Grigorik. Gh archive. [Online]. Available: https://www.gharchive.org/

[55] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada*, 2008.

[56] K. Herzig, S. Just, and A. Zeller, ser. ICSE '13'.

[57] GitHub, "GitHub Compare API," https://developer.github.com/v3/repos/commits/#compare-two-commits, 2010.

[58] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 121–130.

[59] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine grained and accurate source code differencing," ser. ASE 2014.

[60] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, May 2013, pp. 207–216.

[61] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," ser. MSR '15, 2015.

[62] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, 2019, pp. 25–36.

[63] R. C. Seacord, *Secure Coding in C and C++*, 2nd ed. Addison-Wesley Professional, 2013.

[64] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," *CoRR*, vol. abs/1812.06469, 2018. [Online]. Available: http://arxiv.org/abs/1812.06469

[65] Q. Le and B. Zoph, "Using machine learning to explore neural network architecture," May 2017. [Online]. Available: https://ai.googleblog.com/2017/05/using-machine-learning-to-explore.html

[66] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013.

[67] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[68] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[69] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," ser. NIPS'12, 2012.

[70] A. H. Victoria and G. Maragatham, "Automatic tuning of hyperparameters using bayesian optimization," *Evolving Systems*, 2020.

[71] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[72] B. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme," *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 1975.

[73] P. Baldi, S. Brunak, Y. Chauvin, C. A. F. Andersen, and H. Nielsen, "Assessing the accuracy of prediction algorithms for classification: an overview ," *Bioinformatics*, 2000.

[74] J. Gorodkin, "Comparing two k-category assignments by a k-category correlation coefficient," *Computational Biology and Chemistry*, 2004.

[75] S. Boughorbel, F. Jarray, and M. El-Anbari, "Optimal classifier for imbalanced data using Matthews Correlation Coefficient metric," *PLOS ONE*, 2017.

[76] H. He and E. A. Garcia, "Learning from Imbalanced Data," *IEEE Transactions on Knowledge and Data Engineering*, 2009.

[77] L. A. Jeni, J. F. Cohn, and F. De La Torre, "Facing Imbalanced Data Recommendations for the Use of Performance Metrics," *International Conference on Affective Computing and Intelligent Interaction and workshops*, 2013.

[78] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019.

[79] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, pp. 6000–6010.