Learning To Predict Vulnerabilities From Vulnerability-Fixes: A Machine Translation Approach

Aayush Garg
aayush.garg@uni.lu
University of Luxembourg

Renzo Degiovanni
renzo.degiovanni@uni.lu
University of Luxembourg

Matthieu Jimenez
matthieu.jimenez@uni.lu
University of Luxembourg

Maxime Cordy
maxime.cordy@uni.lu
University of Luxembourg

Mike Papadakis
michail.papadakis@uni.lu
University of Luxembourg

Yves Le Traon
yves.letraon@uni.lu
University of Luxembourg

***Abstract*—Vulnerability prediction refers to the problem of identifying the system components that are most likely to be vulnerable based on the information gained from historical data. Typically, vulnerability prediction is performed using manually identified features that are potentially linked with vulnerable code. Unfortunately, recent studies have shown that existing approaches are ineffective when evaluated in realistic settings due to some unavoidable noise included in the historical data. To deal with this issue, we develop a prediction method using the encoder-decoder framework of machine translation that automatically learns the latent features (context, patterns, etc.) of code that are linked with vulnerabilities. The key idea of our approach is to learn from things we know, the past vulnerability fixes and their context. We evaluate our approach by comparing it with existing techniques on available releases of the three security-critical open source systems (Linux Kernel, OpenSSL, and Wireshark) with historical vulnerabilities that have been reported in the National Vulnerability Database (NVD). Our evaluation demonstrates that the prediction capability of our approach significantly outperforms the state-of-the-art vulnerability prediction techniques (Software Metrics, Imports, Function Calls, and Text Mining) in both recall and precision values (yielding 4.7 times higher MCC values) under realistic training setting.**

I. Introduction

Software vulnerabilities are a major concern in our ultra-connected world. A single occurrence has the power to harm millions of lives, e.g., through data leakage or via interruption of critical services. In 2017, the overall cost of cybercrime on the global economy was estimated to $600 billion [15].

A software vulnerability is usually defined as a flaw, a bug, or a weakness in a software system that can be leveraged by an attacker to steal information, gain access to a system or render it unavailable. While vulnerabilities can be thought of as specific types of software defects (or bugs), there are subtle and significant differences that make their identification considerably more complex and challenging than the problem of finding bugs [30], [39].

Firstly, their relatively small number in comparison to defects, reduces a lot of knowledge that one can build from the analysis of historical data. Secondly, their identification requires an attacker's mindset [28], which developers or code reviewers may not possess. Lastly, the continuous growth of codebases makes it difficult to investigate them entirely and track all code changes. For example, the Linux kernel, one of the projects with the highest number of publicly reported vulnerabilities, reached 27.8 million LoC (Lines of Codes) in the beginning of 2020.

Vulnerability prediction approaches were proposed to tackle these challenges and reduce the amount of effort that developers and code reviewers have to put on when testing or reviewing code to find vulnerabilities. These methods traditionally rely on a set of features and/or code properties that associate with vulnerabilities. For instance, the presence of vulnerabilities has been linked to high code churn [34], to the use of specific library imports and function calls [29], and the frequency of suspicious code tokens [39]. Unfortunately, building models around such features is challenging due to the small number of available vulnerability code instances, which limit the learning ability of the predictors [49].

Another important problem is the working assumption of previous work, which is that every component that has no vulnerabilities declared, at training time, is indeed non-vulnerable [24]. This assumption results in unavoidable noise in the training data and makes existing approaches perform poorly under realistic settings (training only on vulnerability instances available at release time) [24]. This indicates the need for more robust vulnerability prediction techniques.

We advance in this direction by developing a method that learns from validated data, i.e., we train only on the components that are known to be vulnerable and leave aside the (supposedly) non-vulnerable components. This way, we do not make any assumption on non-vulnerable data and bypass the key problem faced by previous work. To do so, we rely on the simple, language-agnostic but powerful machine translation technique [8], which we train on pairs of vulnerable and fixed component versions that are available at every project release time. The trained models can then be used to predict the

likelihood of a code fragment to be vulnerable.

Our approach learns from vulnerability fixes, learn transformations, by capturing the features related to the differences between vulnerable and fixed components, i.e., the actual fix context. Therefore, predictions are guided by the actual points of interest (the diff points) in the vulnerable code where the transformations should happen. This means that when our technique suggests a modification on some code part, this is a good indication that the code has characteristics similar to a vulnerable one (used for training). Moreover, machine translation based approaches have been shown to be effective in various fields such as introducing artificial faults (mutants) [40], automatically producing bug fixes [41], and identifying code clones [44]. Thus, machine translation based approach is a promising candidate for vulnerability prediction.

We empirically assess the effectiveness of our approach on available releases of three security-critical open source systems (Linux Kernel, Wireshark and OpenSSL) taken from the study of Jimenez *et al.,* [24]. Our evaluation demonstrates that the proposed prediction approach significantly outperforms the current state-of-the-art vulnerability prediction methods under both "clean" and "realistic" training data settings.

In particular, our results show that in the case where we train with clean data (case adopted by most of the previous work [24]), our approach improves all prediction performance metrics at the same time, *i.e.,* an improvement of 9% in precision, 135% in recall, 81% in F-measure, and 41% in MCC. In addition to these metrics, we also evaluate our approach on predicting "novel" vulnerable components specifically. This is a new metric that we introduce in this paper to help evaluate the extent to which vulnerability prediction generalizes. Under realistic training settings where data include some unavoidable noise [24], the average improvement achieved by our approach is even more remarkable (with a recall more than 22 times higher than the baselines).

In summary, we make the following contributions:

1) We present a vulnerability prediction method using machine translation.
2) We demonstrate that our approach significantly outperforms the existing ones through a large empirical study.
3) We corroborate that our approach remains robust when trained in realistic data settings that include unavoidable noise, where previous methods fail [24].

II. Background

A. Vulnerabilities

A security vulnerability as defined by the Common Vulnerability Exposures (CVE) [1] is "*a mistake in software that can be directly used by a hacker to gain access to a system or network*". Those mistakes are usually due to the inadvertence of a developer or insufficient knowledge of defensive programming. Still, vulnerabilities are of critical importance for software vendors that often offer bounties to find them and prioritize their resolution over other (less harmful) bugs to reduce potential business impact.

To help software engineers to make sure that their software is not vulnerable and build more knowledge on vulnerabilities, vulnerabilities are usually reported in publicly available databases. The National Vulnerability Database(NVD) [4] supported by the U.S. government is an example of such a database. It contains information on vulnerabilities, their severity, type, links to additional information, and is built on top of the CVE List dictionary that links each publicly reported vulnerability to a unique identifier.

B. Vulnerability Prediction Modeling

Prediction modeling aims at learning statistical properties of interest based on historical data. While the resulting models are usually suitable only for the project/application on which they have been trained on, the learning process is generic and applies on a specific set of features that associate with the property to predict. In the context of vulnerabilities, a prediction model can be used to classify software components as likely or unlikely vulnerable. Then, this information can be used to support the code review process. This task is somehow related to defect prediction, yet due to the sparsity of available examples, it is usually harder to predict vulnerabilities than defects [36].

C. Granularity Level

Depending on the target and on available data, prediction models can offer different granularity levels, such as line, function, component, etc. However, the granularity level should ultimately correspond to what the developers or code reviewers need. Thus, different granularities offer different tradeoffs. For instance, the line-level granularity seems appealing, but produces many false positives and lacks contextual information. On the other end of the spectrum, module-level granularity produces accurate results with the entire relative context to perform the inspection but only slightly reduces the inspection effort. A commonly accepted tradeoff is the component (file) level granularity as it has been vetted by Microsoft developers in a study from Morrison *et al.,* [28] and is used by most existing approaches.

Some studies also aim at the function level prediction [37], [48], but without providing any evidence on its utility. We thus consider the component level (i.e. code files) granularity, as it has been found to be actionable for industrial use [28] and provides a baseline for comparing our results with those reported by the relevant literature [24].

D. Machine Translation

We suggest a novel approach for vulnerability prediction using machine translation. Machine Translation can be considered as a transformation function $transform(X) = Y$ where the input $X = \{x_1, x_2, \ldots, x_n\}$ is a set of *entities* that represents a component to be transformed to produce the output $Y = \{y_1, y_2, \ldots, y_n\}$ which is a set of entities that represent a transformed (desired) component. In the training phase, the transformation function learns on the example pairs (X, Y) available in the training dataset. In our context, X contains

vulnerable entities (representing a vulnerable component) and Y contains fixed entities (representing the corresponding fixed component). The transformation function can be trained *not* to transform, i.e., to produce the same output as the input in cases where Y is the desired entity-set. This is achieved by training the function on the example pairs (Y, Y), i.e. $transform(Y) = Y$. In the case of vulnerability prediction modeling, this learned transformation will be used as our prediction model.

Among the several machine translation algorithms that have been suggested over the past years, we use the RNN Encoder-Decoder which is established and is used by many recent studies [38], [40].

E. RNN Encoder-Decoder architecture

The RNN Encoder-Decoder machine translation is composed of two major components: an RNN Encoder to encode a sequence of terms x into a vector representation, and an RNN Decoder to decode the representation into another sequence of terms y. The model learns a conditional distribution over an (output) sequence conditioned on another (input) sequence of terms: $P(y_1; \ldots; y_m | x_1; \ldots; x_n)$, where n and m may differ. For example, given an input sequence $x = Sequence_{in} = (x_1; \ldots; x_n)$ and a target sequence $y = Sequence_{out} = (y_1; \ldots; y_m)$, the model is trained to learn the conditional distribution: $P(Sequence_{out} | Sequence_{in}) = P(y_1; \ldots; y_m | x_1; \ldots; x_n)$, where x_i and y_j are separated tokens. A bi-directional RNN Encoder [8] (formed by a backward RNN and a forward RNN) is considered the most efficient to create representations as it takes into account both past and future inputs while reading a sequence [7].

III. Approach

The main objective of our approach is to automatically learn what indicates vulnerabilities (without any features definition and/or selection by human intervention) and use it to predict the presence of vulnerabilities in unseen code.

The key idea of our method is to train a machine translator (viz. an encoder-decoder sequence to sequence model) to identify vulnerable code, by feeding it with vulnerable code fragments and their corresponding fixes. Machine translators have been successfully used to translate natural text from one language to another, as they automatically recognize (i) the features of the language (to be translated) and (ii) the required translation (to the desired language). In our case, it is used to automatically identify the features of vulnerabilities without any investment of time and/or resources to define features.

It should be noted that we do not aim at fixing vulnerable code, but rather at identifying vulnerable code instances. The point here is that we use the translator to indicate the presence of vulnerabilities without considering what it produces as fixes. In other words, we leverage the ability of the translators to learn the vulnerabilities' context and not their exact location. We assert that since vulnerable code instances are scarce, the information gained from historical data is inevitably partial and incomplete. Therefore, it can be used to indicate the presence of vulnerabilities but not their actual location/pattern.

After training, one can input unseen code (which may or may not contain vulnerabilities) into the translator to check whether it is likely to have vulnerabilities. If the translator transforms the code then it can be concluded that the code is likely vulnerable. To avoid many false positives (that the translator transforms every input code fragment), we also train it to *not* transform non-vulnerable code fragments. Thus, we also feed it with input-output pairs, each of which is a copy of the same non-vulnerable code.

Figure 1 shows a more detailed view of the training process. Starting from vulnerable code components and their fixes, it involves the following activities: 1) decompose the components into code fragments; 2) identify which fragments are responsible for the vulnerability; 3) produce abstracted code fragments by removing irrelevant information (e.g. user-defined names); 4) configure and train the machine translator.

A. Decompose Components into Code Fragment (Functions)

Our approach takes as input a set of vulnerable code components together with their corresponding fixed components. We target program functions in order to have a fine grained information, while our predictions remain at the component level to account for the context of the functions and cases (vulnerability fixes) that can be fixes in different locations. A code-fix can be an addition and/or removal and/or modification of code. Since functions are the basic building blocks of a program, we use them to establish a function-level mapping between the vulnerable component and its fixed counterpart (based on the function headers).

Thus, we extract all the functions from both the vulnerable component and the fixed component and pair each before-fix function with the corresponding after-fix function. The functions that cannot be paired, i.e., have no counterpart in the other component, are discarded. This can happen due to the creation and/or deletion of function to fix a vulnerability, e.g., an additional function in the after-fix component which is not present in the before-fix component or vice-versa. Then, for each function pair, we keep the before-fix copy of the function (which can be vulnerable or non-vulnerable) and the after-fix function (which is non-vulnerable).

B. Categorize Functions as Vulnerable or Non-Vulnerable

We consider as vulnerable, following the current practice in this line of work, any function which was modified during the fix, i.e., the before-fix copy is different from its corresponding after-fix copy. Otherwise, we consider the function as non-vulnerable (not vulnerable to the specific vulnerability). When comparing before-fix to after fix copies, we ignore irrelevant syntactical changes, e.g., additional blank spaces and new lines. If there remain syntactical differences between the before-fix copy and the after-fix copy, we label the former as vulnerable and the latter as fixed. Otherwise, we keep only one of the two copies and label it as non-vulnerable.

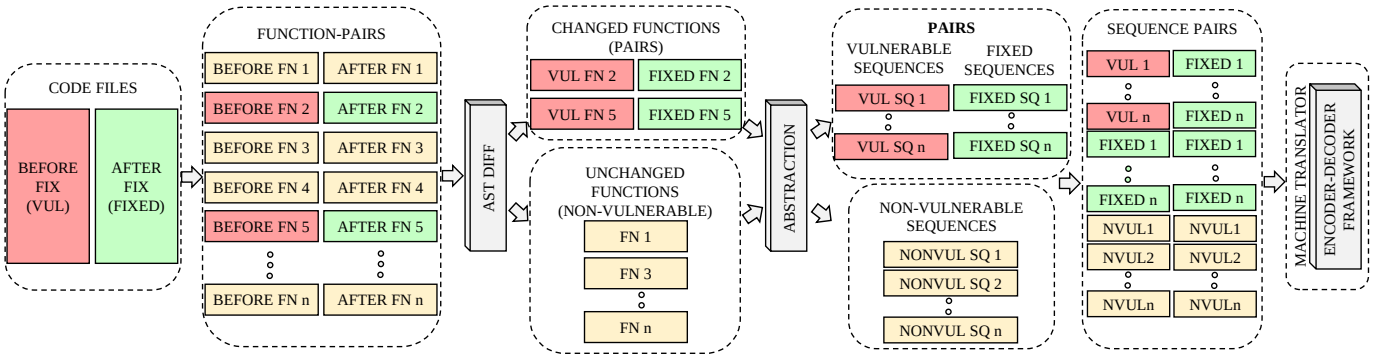

Fig. 1: Training process: Extracted functions from components are compared (before-fix with after-fix) and categorized in Changed (Vuln.–Fixed pair) and Unchanged functions. Further, these functions are abstracted and arranged in single sentences to get sequences (Vuln., Fixed, and Non-Vuln.). These sequence pairs are used to train the machine translation model.

C. Abstract the Irrelevant Information

A major challenge in dealing with raw source code is the huge vocabulary created by the abundance of identifiers and literals used in the code. Vocabulary, on such a large scale, hinders the goal of learning transformations of code as a neural machine translation task [40]. Thus, to reduce vocabulary size, we transform the source code of each function into an abstract representation by replacing user-defined entities with re-usable IDs.

Figure 2 shows a code snippet from a real function (Figure 2a) converted into its abstract representation (Figure 2b). The purpose of this abstraction is to replace any reference to user-defined entities (function name, type, variable name and string literal) by IDs that can be reused across functions (thereby reducing the vocabulary size). Thus, our abstraction approach first detects identifiers and string literals before replacing them with unique IDs. Additionally, comments and annotations are removed.

New IDs follow the regular expression `(F|T|V|L)_(num)`$^+$, where `num` stands for numbers $0, 1, 2, \ldots$ assigned in a sequential and positional fashion based on the occurrence of that entity. All the entities - user-defined *Function* names, *Type* names, *Variable* names, and *String Literals* are replaced with `F_num`, `T_num`, `V_num`, and `L_num`, respectively. Thus, the first function name found receives the ID `F_1`, the second function name receives the ID `F_2`, and so on. If any of these entities appear multiple times in a function, it will be replaced with the same ID.

Each function (pair) is abstracted in isolation (yielding abstracted function code), which means that the same IDs can be reused across functions without impacting our machine translation approach. Thus, ID references are not preserved across functions (e.g., `V_1` may refer to two different variable names from one function to another), which is key to reduce vocabulary size. For example, the name of the first function called in any pair is replaced with the ID `F_1`, regardless of the original function name.

In case of vulnerable functions, The before-fix copy is abstracted first and then the after-fix copy. The IDs are shared between the two copies (before-fix and after-fix) of the functions and new IDs are generated only for newly found *Function* names, *Type* names, *Variable* names, and *String Literals*.

Finally, any abstracted function code is rearranged in a single sentence to represent a sequence of space-separated entities (the representation supported by the machine translator). The sequences generated from vulnerable (before-fix), fixed (after-fix) and non-vulnerable functions are named vulnerable, fixed and non-vulnerable sequences, respectively. To limit the computation cost involved when training the translator, large sequences are split into multiple sequences of no more than *50* tokens each.

D. Build the Machine Translator

To build our machine translator, we train an encoder-decoder model that can transform a sequence (input to the model) to a desired sequence (output of the model). A representation of a sequence is similar to a sentence in a natural language that consists of words separated by spaces and ends with a full stop. Instead of words and full stop character, a sequence has tokens and the newline character. Thus, we train the encoder-decoder by feeding it with pairs of sequences. More precisely, we use three types of pairs: (i) each vulnerable sequence with its corresponding fixed sequence; (ii) each fixed sequence paired with itself; (iii) each non-vulnerable sequence paired with itself. The last two types of sequence pairs are essential to make the model learn to produce the *same* output as the input when fed with fixed or non-vulnerable sequences. Thereby, our approach avoids raising many false positives (i.e. wrongly predicting that some non-vulnerable sequences are vulnerable) while learning only from clean data (it does not use "non-vulnerable" components). In particular, learning not to modify fixed sequences helps the model differentiate vulnerable before-fix code from the corresponding (syntactically similar) fixed code.

E. Predict Vulnerabilities

To predict whether or not the unseen component contains vulnerabilities, we first decompose it into sequences following a similar process as depicted in Figure 1. Then, we feed the resulting sequences into the machine translator, yielding as

```c
void dev_load (struct net* netw, const char* name) {
 struct net_device* dev;
 rcu_read_lock();
 dev = dev_get_by_name_rcu(netw , name);
 rcu_read_unlock();
 if (!dev && capable(CAP_NET_ADMIN))
  request_module("%s", name);
}
```

(a) Actual Function

```c
void F_1 ( struct T_1 * V_1 , const char * V_2 ) {
 struct T_2 * V_3 ;
 F_2 ( ) ;
 V_3 = F_3 ( V_1 , V_2 ) ;
 F_4 ( ) ;
 if ( ! V_3 && F_5 ( V_4 ) )
  F_6 ( L_1 , V_2 ) ;
}
```

(b) Abstracted Function

Fig. 2: Abstraction: Actual Functions (left) are abstracted by replacing user-defined Function names, Type names, Variable names, and String Literals to F_num, T_num, V_num, and L_num, respectively to achieve Abstracted Function (right).

many output sequences. If one (or more) output sequences returned by the model was modified by the machine translator, we conclude that the component contains a vulnerability. Otherwise, we conclude that the component is non-vulnerable.

IV. Experimental Evaluation

A. Research Questions

Our approach aims to support code reviews by predicting vulnerable components in a new release based on the information learned from previous (historical) data, i.e., the previous project release. Therefore, our first research question regards the prediction performance of our technique. We evaluate this by training on all available vulnerabilities of one release and testing on the next release (for all available version pairs). Thus, we ask:

RQ1 How effective is our approach in predicting vulnerable components?

After assessing the prediction performance of our approach, we turn our attention on the existing state-of-the-art techniques. Hence, we investigate:

RQ2 How effective (in predicting vulnerable components) is our approach in comparison to the current state-of-the-art techniques?

In our approach, we train a model on the vulnerabilities of a release and test the trained model for the prediction of vulnerable components in the next release. We thus may have the knowledge that a component is vulnerable in a given release irrespective of the vulnerability detection date. As this vulnerable component may remain unfixed and repeat in the next version, it is essential to assess the learning potential of our trained models by evaluating the performance of existing vulnerable component prediction and comparing with the state-of-the-art techniques. Nonetheless, it is also imperative to assess the generalization capability of our trained models by evaluating their performance in predicting novel (unseen) vulnerable components and comparing with the state-of-the-art techniques. Hence, we ask:

RQ3 How effective is our approach in predicting novel and existing vulnerable components?

Until now, we considered that all vulnerable components of a given version are in the training set, as done by most of vulnerability prediction studies, e.g., [36], [48]. This analysis provides indications on what is the potential prediction ability (actual limits in predictions) of the approaches when assuming clean training data settings. Unfortunately, in practice, such data are unavailable and inflate the actual performance of the prediction models [24]. The actual performance in realistic settings is much lower due to the real-world labelling issues [24], i.e., vulnerabilities are frequently reported at a much later time than they are actually introduced, a fact that misleads the predictions (causing the classifiers to learn as non-vulnerable, the components that are in fact vulnerable). Thus, it is imperative to study *realistic* training settings, where a prediction model is trained only on those vulnerabilities that were detected between the release date of each version till the release date of the next version (for which we predict vulnerabilities). Our approach also does not make any assumption related to non-vulnerable components, thereby having the potential of being less sensitive to the real-world labelling issues. For these reasons, we also evaluate the approaches under realistic training settings. Therefore we ask:

RQ4 How effective (in predicting vulnerable components) is our approach in comparison to the current state-of-the-art technique under realistic training settings?

B. Data

To answer the research questions, we need projects with plenty of releases and a sufficient amount of historical data in order to train. Therefore, we consider three large security intensive open-source software systems in our evaluation – the Linux Kernel, the OpenSSL library, and the Wireshark tool. These systems are widely used, mature, and have a long history of releases and vulnerability reports, which is needed to perform release-based experiments.

Linux Kernel [3] is an operating system, integrated into billions of systems and devices, such as Android. Linux is one of the largest open-source code-bases and has a long history (since 1991), recorded in its repository. It is relevant for our evaluation since it has many security aspects and is among the projects with a higher number of reported vulnerabilities in NVD. *OpenSSL* [5] is a library implementing the SSL and TLS protocols, commonly used in communications. It is of critical importance as highlighted by the *Heartbleed* vulnerability, which made half of a million web servers vulnerable to attacks [2]. *Wireshark* [6] is a network packet analyzer mainly used for troubleshooting and debugging. The project is open source and is relevant for the study because it is integrated with most operating systems.

All the vulnerabilities (the vulnerable and fixed components) of the systems are gathered by *VulData7* [23]. *FrameVPM* [21] is a framework built to evaluate vulnerability prediction models. It collects the code files from GitHub repositories to train and evaluate prediction models. We use

TABLE I: For each system (Linux Kernel, Wireshark, and OpenSSL) in our dataset, we report the total number of versions, average number of components and vulnerable components in each version, and the ratio of vulnerable components.

System	#Versions	#Avg.Comp	#Avg.Vuln.Comp	%Vuln.
Linux Kernel	36	16456	456	3%
Wireshark	10	2012	134	7%
OpenSSL	10	664	59	9%

FrameVPM to gather the entire code-base of the systems employed in the experiments. Table I provides the details of our dataset.

C. Implementation and Model Configuration

During abstraction, we invoke the *srcML* tool [11] to convert source code into an XML format including tags to identify literals, keywords, identifiers, and comments. This helps in separating user-defined identifiers and string literals (the largest part of the vocabulary) from the language keywords (a limited set). Then, ID replacement is performed by a dedicated procedure that we implemented. To check whether before-fix copies and after-fix copies are different without considering irrelevant characters (i.e., white spaces and new lines), we input the XML produced by srcML into the *Gumtree Spoon AST Diff* tool. It is to be noted that our approach is not bound to the above-mentioned third-party tools. As an alternative, one can use any software that can identify user-defined entities and can check the difference while ignoring irrelevant characters.

We built our encoder-decoder model on top of *tf-seq2seq* [14], a general purpose encoder-decoder framework. To configure it, we learn on previous research applying machine translation to solve software engineering tasks (other than vulnerability prediction), e.g. [40], [41]. Hence, we rely on a bidirectional encoder as it generally outperforms unidirectional encoder. We use a Long Short-Term Memory (LSTM) network [19] to act as the Recurrent Neural Network (RNN) cell, which was shown to perform better than the common alternatives (simple RNNs or gated recurrent units) in other software engineering prediction tasks [9], [33]. Bucketing and padding are used to deal with the variable length of the sequences. To strike a balance between performance and training time, the combination used is 1-layer encoder and 2 layer-decoder both with 256 units.

To determine an appropriate number of training steps, we conducted a preliminary study involving a validation set (independent of both the training set and the test set that we use in our experimental evaluation). There, we train the model by iterations of 5,000 steps. At the end of each iteration, we check whether the prediction accuracy on the validation set improved. If that is the case, we pursue the training for another iteration (otherwise, we stop). We found out that the model stopped improving at 50,000 steps, which we use as the total number of training steps. Note that this order of magnitude is in line with previous research applying machine translation to solve software engineering prediction tasks, e.g. [40].

D. Experimental Settings

Our experimental evaluation is designed to evaluate the techniques under *clean* and *realistic* data settings during the training phase. We train a model on each release and test the trained model on the next release (future release) simulating a typical release-based vulnerability prediction evaluation scenario [24].

Clean Data Training - Used in RQs 1, 2 & 3: In these settings, a prediction model is trained using all the vulnerabilities (before-fix sequences transformation to after-fix sequences) of a release of a system (Linux Kernel / OpenSSL / Wireshark). The trained models are evaluated based on their predictions in the following release of the same system (e.g., trained on Linux Kernel release v4.0 and evaluated against v4.1). The components of the following release are converted into sequences that are input to the trained model to get the output sequences. Then, our approach compares the output sequences generated by the trained model with the input sequences. A component is considered vulnerable if any of the output sequences differ from the input sequences, otherwise considered as non-vulnerable. This training-testing process is repeated for all available releases. Since a model trained on the last release of a project does not have a future release to be evaluated against, for *n* releases of a project, *(n–1)* experiments are performed, e.g., we performed 35 experiments (35 models trained and evaluated) for the 36 releases of Linux Kernel.

Realistic Training - Used in RQ4: In contrast to the *clean* data training settings, in *realistic* training settings we consider the date of detection of a vulnerability. Based on the detection date, a decision is taken whether the vulnerability is included in the training dataset or not. A prediction model (of a system-release) is trained using only those vulnerabilities that were detected before the next release date. Then, the trained model is evaluated for its prediction in the following release of the same system.

E. Benchmarks for Vulnerability Prediction

To assess effectiveness, we compare our approach with the state-of-the-art in vulnerability prediction. For this task, we rely on *FrameVPM* [21], [24], a framework implementing a collection of state-of-the-art techniques, to create and evaluate the vulnerability prediction models based on the following methods:

Software metrics: Complexity metrics have been extensively used for defect prediction (e.g. [18]) and vulnerability prediction (e.g. [35], [34], [10]). It is based on the idea that complex code is difficult to maintain and test, and thus has a higher chance of having vulnerabilities than simple code. Software metrics' vulnerability prediction model is trained on features related to *complexity and coupling* (e.g., lines of code, cyclomatic complexity; nesting level of control constructs, etc); *code churn* (added, modified and deleted lines in the history of a component); and *developer activity metrics* (number of commits, number of developers modifying a component, etc).

Text Mining: It considers a source code component as a collection of terms associated with frequencies, also known as

Bag of Words (BoW), used for vulnerability prediction [31]. The source code is broken into a vector of code tokens, and the frequency of each token is then used as the features to build the vulnerability prediction model. Further improvements have been performed to significantly improve its performance, e.g., by pooling frequency values in different bins according to particular criteria to discretize BoW's features [25], [31].

Imports and Function Calls: The work of *Neuhaus et al.* [29] is based on the observation that the vulnerable components tend to import and call a particular small set of functions. Thus, the features of this simple prediction model are the components' imports and function calls. Following the suggested recommendations of *FrameVPM*, we use imports and function calls as separate sets of features. It trains one model based on imports and another model based on function calls, thus implementing one model per set of features.

F. Performance measurement

A confusion matrix stores the correct and incorrect predictions of the studied methods. Given a vulnerable component, if it is predicted as vulnerable, then it is a true positive (TP); otherwise, it is a false negative (FN). Given a non-vulnerable component, if it is predicted as non-vulnerable, then it is a true negative (TN); otherwise, it is a false positive (FP). The confusion matrix is used to compute the *Precision*, *Recall*, and *F-measure* scores, that quantitatively evaluate the prediction accuracy of vulnerability prediction models.

The mentioned traditional metrics do not take into account the true negatives and can be misleading, especially in the case of imbalanced data. Hence, we complement these with the *Matthews Correlation Coefficient (MCC)* [27], a reliable metric of the quality of prediction models [32]. It is generally regarded as a balanced measure that can be used even when the classes are of very different sizes, e.g. in case of Linux Kernel, 3% vulnerable components (positives) over 97% non-vulnerable components (negatives). *MCC* is calculated as:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$$

MCC returns a coefficient between 1 and -1. An MCC value of 1 indicates a perfect prediction, while a value of -1 indicates a perfect inverse prediction *i.e.,* a total disagreement between prediction and reality. MCC equals to 0 indicates that the prediction performance is equivalent to random guessing.

V. Experimental Results

A. Prediction with Clean Training data (RQ1)

Table II records the results for every experiment performed on the 56 releases (36 of Linux Kernel, 10 of Wireshark and 10 of OpenSSL) we study. As mentioned earlier, here the model is trained on a particular release and evaluated against the following (next) release of the system. The scores reported in the table correspond to the average and median of the scores that our approach achieves for all releases of the systems. Our approach obtained an overall average (median) of *MCC= 0.738* (0.761), *F-measure= 0.869* (0.883), *Precision= 0.914* (0.923), and *Recall= 0.838* (0.888) in prediction of vulnerable components in the next release of a project. In almost all the cases, the prediction models trained with the clean data achieved over 0.65 MCC (except 4 cases), over 0.75 F-measure (except 2 cases), over 0.8 Precision (except 1 case), and over 0.7 Recall (except 4 cases), which according to state of the art can be considered as actionable [36].

TABLE II: Prediction with clean training data (RQ1)

Release	MCC	F-measure	Precision	Recall
Linux Kernel				
v3.0	0.704	0.864	0.841	0.89
v3.1	0.716	0.869	0.821	0.924
v3.2	0.751	0.884	0.857	0.914
v3.3	0.696	0.862	0.819	0.909
v3.4	0.729	0.875	0.84	0.913
v3.5	0.716	0.856	0.936	0.788
v3.6	0.738	0.878	0.861	0.897
v3.7	0.667	0.85	0.817	0.886
v3.8	0.778	0.893	0.915	0.873
v3.9	0.694	0.861	0.827	0.897
v3.10	0.765	0.89	0.884	0.895
v3.11	0.851	0.928	0.932	0.923
v3.12	0.761	0.888	0.878	0.898
v3.13	0.715	0.869	0.821	0.923
v3.14	0.853	0.929	0.929	0.929
v3.15	0.775	0.894	0.891	0.898
v3.16	0.802	0.905	0.919	0.892
v3.17	0.809	0.909	0.909	0.909
v3.18	0.811	0.908	0.93	0.888
v3.19	0.72	0.871	0.839	0.906
v4.0	0.876	0.939	0.964	0.915
v4.1	0.862	0.933	0.938	0.929
v4.2	0.771	0.883	0.955	0.821
v4.3	0.835	0.919	0.942	0.897
v4.4	0.824	0.916	0.907	0.926
v4.5	0.793	0.9	0.924	0.877
v4.6	0.788	0.9	0.876	0.926
v4.7	0.793	0.902	0.905	0.899
v4.8	0.827	0.917	0.914	0.921
v4.9	0.799	0.904	0.904	0.904
v4.10	0.791	0.899	0.921	0.879
v4.11	0.746	0.882	0.866	0.898
v4.12	0.777	0.891	0.93	0.855
v4.13	0.79	0.896	0.941	0.856
v4.14	0.802	0.905	0.913	0.897
Wireshark				
v1.8.0	0.499	0.688	0.971	0.532
v1.10.0	0.577	0.774	0.923	0.667
v1.11.0	0.775	0.882	0.968	0.811
v1.12.0	0.577	0.759	0.953	0.631
v1.99.0	0.712	0.851	0.945	0.774
v2.0.0	0.589	0.778	0.933	0.667
v2.1.0	0.744	0.859	0.982	0.764
v2.2.0	0.667	0.829	0.929	0.748
v2.4.0	0.168	0.652	0.694	0.614
OpenSSL				
v0.9.3	0.833	0.909	1	0.833
v0.9.4	0.833	0.909	1	0.833
v0.9.5	0.833	0.909	1	0.833
v0.9.6	0.667	0.8	1	0.667
v0.9.7	0.71	0.83	1	0.71
v0.9.8	0.712	0.831	1	0.712
v1.0.0	0.71	0.844	0.964	0.75
v1.0.1	0.725	0.866	0.913	0.824
v1.0.2	0.667	0.8	1	0.667
Overall				
Average	**0.738**	**0.869**	**0.914**	**0.838**
Median	**0.761**	**0.883**	**0.923**	**0.888**

TABLE III: (RQ2) Comparison between state-of-the-art and our approach: Clean Training Data Setting - average(median)

Approach	MCC	F-measure	Precision	Recall
Software Metrics	0.497(0.53)	0.444(0.48)	0.852(0.92)	0.319(0.335)
Imports	0.475(0.49)	0.426(0.44)	0.827(0.88)	0.302(0.335)
Function Calls	0.524(0.555)	0.481(0.5)	0.841(0.89)	0.357(0.345)
Text Mining	0.491(0.5)	0.441(0.46)	0.842(0.92)	0.321(0.32)
Our	**0.738(0.761)**	**0.869(0.883)**	**0.914(0.923)**	**0.838(0.888)**

Answer to RQ1: The vulnerability prediction models built on our approach successfully predicts the vulnerable components with an average MCC score of 0.738.

B. Comparison with state-of-the-art techniques (RQ2)

We use *FrameVPM* [21] to apply the baseline techniques on all the components of a system release. The resulting prediction models are then evaluated in a similar manner as RQ1. Figure 3 shows (in box plot format) the performance comparison of our approach with the replicated approaches. Each box plot shows the distribution of a specific performance indicator (MCC / F-measure / Precision / Recall) for the related techniques per project.

We observe that our approach clearly outperforms the other techniques. Table III summarizes the overall performance of the techniques. Interestingly, our approach achieved much higher prediction performance in comparison to the existing state of the art techniques, in every criterion. The differences are statistically significant.[1] Table III shows that the technique *Function Calls* outperforms the other state-of-the art techniques (*Software Metrics*, *Imports* and *Text Mining*) with the highest average MCC of 0.524. Our approach even outperforms *Function Calls*, on an average by 41% in MCC and 81% in F-measure. It is worth mentioning that the average improvement offered by our approach in Precision is 9%, whereas in Recall, is 135% in comparison to *Function Calls*.

Answer to RQ2: When trained with clean data, our approach has significantly higher prediction ability (MCC score improvement of 41%) than the current state-of-the-art.

C. Novel and Existing Vulnerable Component Prediction (RQ3)

Table IV shows the average percentages of the existing vulnerable component correctly predicted by our approach and the state-of-the-art techniques across the 56 aforementionned releases of Linux Kernel, Wireshark and OpenSSL. The models that are based on our approach predict 92.79%, 69.48% and 87.19% of the existing vulnerable components on an average in Linux Kernel, Wireshark and OpenSSL project releases respectively. The percentages gained by our approach are higher than the state-of-the-art techniques by 34.14% for Linux Kernel releases, 8.72% for Wireshark releases and 22.61% for OpenSSL releases, indicating a higher learning potential in comparison to the existing techniques.

[1]We compared the MCC values, by using Wilcoxon sign-rank-test [45], and obtained a $p - value < 6.2\text{e}{-9}$ with Software Metrics, Function Calls and Text Mining , and a $p - value < 5.3\text{e}{-9}$ with Imports. We also compared the effect size of MCC values, by using the Vargha-Delaney A measure [42], and obtained a value of 0.05 with Software Metrics, 0.03 with Imports, and 0.07 with Function Calls and Text Mining, clearly indicating that our approach significantly outperforms related techniques.

TABLE IV: (RQ3) Comparison between previous techniques and our approach: Existing vulnerable component prediction.

Approach	Linux Kernel 36 releases	Wireshark 10 releases	OpenSSL 10 releases
Software Metrics	48.12%	54.84%	54.17%
Imports	48.12%	60.76%	50.00%
Function Calls	58.65%	52.69%	64.58%
Text Mining	57.14%	56.99%	64.58%
Our	**92.79%**	**69.48%**	**87.19%**

TABLE V: (RQ3) Comparison between previous techniques and our approach: Novel vulnerable component prediction.

Approach	Linux Kernel 36 releases	Wireshark 10 releases	OpenSSL 10 releases
Software Metrics	09.09%	15.48%	18.18%
Imports	50%	08.93%	23.08%
Function Calls	56.1%	60.00%	09.09%
Text Mining	45.45%	16.07%	18.18%
Our	**76.53%**	**91.03%**	**60.07%**

On the other hand, Table V shows the average percentages of the novel vulnerable component prediction. The trained models that are based on our approach predict 76.53%, 91.03% and 60.07% of the novel vulnerable components on an average in Linux Kernel, Wireshark and OpenSSL project-releases, respectively. The percentages gained by our approach are higher than the existing techniques by 20.43% for Linux Kernel releases, 31.03% for Wireshark releases and 36.99% for OpenSSL releases, reflecting higher generalization capability in comparison to the existing techniques.

It is worth noting that our approach obtains all the above mentioned percentages with an overall average MCC of 0.738, which is 41% higher than the MCC achieved by the state-of-the-art techniques.

Answer to RQ3: Both, the learning potential and the generalization capability of the models trained on our approach are remarkably higher than the state-of-the-art techniques.

D. Comparison with state-of-the-art techniques in a Realistic Training Setting (RQ4)

In realistic training settings, a model is trained only on the vulnerabilities of a release that were made public before the next release date of the system. This unavoidably introduces mislabelling noise [24].

Figure 4 shows that the performance of all the techniques is considerably reduced in the realistic training setting, as compared to the clean training data settings. This confirms the results reported by Jimenez *et al.,* [24]. Despite this

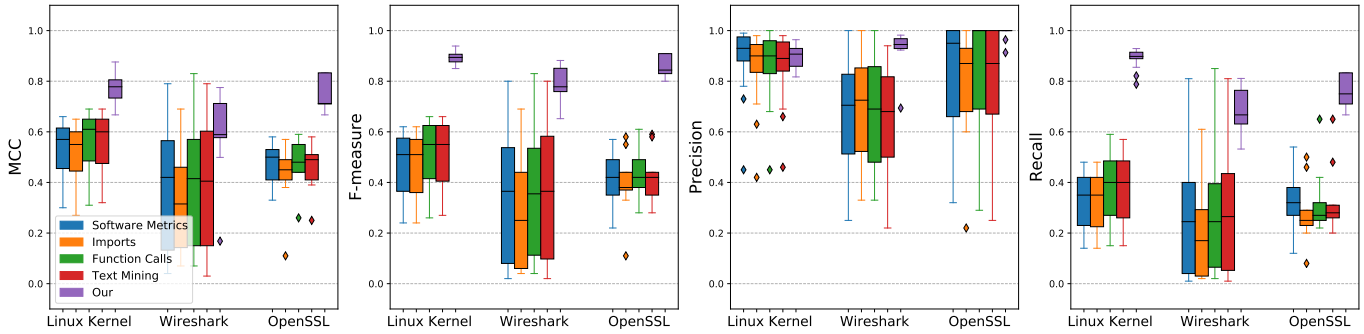

Fig. 3: Comparison with state-of-the-art techniques (RQ2): When trained with clean data, our approach outperforms state-of-the-art techniques with an average improvement in MCC, F-measure, Precision, and Recall of 41%, 81%, 9%, and 135% respectively.

TABLE VI: (RQ4) Comparison between state-of-the-art and our approach: Realistic Training Setting - average(median)

Approach	MCC	F-measure	Precision	Recall
Software Metrics	0.056(0.03)	0.032(0.01)	0.306(0.3)	0.019(0.01)
Imports	0.062(0.06)	0.035(0.02)	0.339(0.33)	0.021(0.01)
Function Calls	0.068(0.05)	0.041(0.02)	0.338(0.33)	0.025(0.01)
Text Mining	0.061(0.03)	0.04(0.01)	0.288(0.25)	0.026(0.01)
Our	**0.393(0.406)**	**0.687(0.677)**	**0.859(0.87)**	**0.582(0.557)**

drop in performance, our approach outperforms the existing techniques with a sizeable difference.

Again, the differences are statistically significant.[2]

Table VI shows the overall average and median performance statistics for each technique. We can observe that the technique *Function Calls* outperforms the other state-of-the art techniques (*Software Metrics*, *Imports* and *Text Mining*) with the highest average MCC of 0.068. Our approach even outperforms the technique *Function Calls* in all the performance measures with an average improvement of *4.7 times* in MCC and *15.76 times* in F-measure. It is interesting to note that the average improvement in Precision offered by our approach is *1.54* times over Function Calls; whereas in Recall, is 22 times, which is outstanding. This indicates that our approach has much higher accuracy in vulnerability prediction than the existing techniques in the realistic training setting as well.

> Answer to RQ4: Under the realistic training setting, a model based on our approach obtains significantly higher accuracy in vulnerability prediction (MCC score improvement by 4.7 times) than the current state-of-the-art techniques.

VI. Threats To Validity

Construct Validity: We use a publicly available tool (*VulData7* [23]) for data collection using the git commit messages and the CVE-NVD database. This process ensures the retrieval of known and fixed vulnerabilities but, undiscovered or unfixed vulnerabilities are ignored. This may result in false negatives with a potential impact on our measurements. However, given the size of Linux Kernel, Wireshark, and OpenSSL and the long history of vulnerability reports, we believe that it is unlikely to have many such cases.

Another concern originates from our choice to learn from the vulnerable and fixed pairs of components. Since our method has access to this information one could argue that the improved performance is due to this and not due to our method. To diminish this concern we also trained the previous methods on both vulnerable and fixed components but this resulted in negligible differences.

Internal Validity: This work only considers components written in C, but these are not the only project elements that can be linked to vulnerabilities. For instance, there are parts of the Linux kernel which are written using assembly code. However, since the great majority of the three systems is written in C, it limits the impact of this threat.

Additionally, we use a publicly available tool (FrameVPM [21]) to evaluate vulnerability prediction models. The tool may unintentionally not reimplement exactly the original approaches. To reduce this threat we inspected the code, parameters, and experiment decisions for the exact replication of the previous approaches. Since our results are in line with the previously published ones, we believe that this threat is of no particular importance.

Furthermore, following the suggestion of Shin et al. [34], the tool uses the three best metrics, according to information gain, to build the software metrics model. Still, there is a possibility that additional metrics could provide different results.

External Validity: Although the study expands its evaluation to three security-critical open source systems, the results may not generalize to other projects (e.g., Android). Additional studies are required to sufficiently take care of the generalization threat. Also, we split the methods in sequences of no more than *50* tokens each. Method-splitting in larger sequences may require more training time and computational resources but can lead to better results.

[2]We compared the MCC values using Wilcoxon sign-rank-test, and obtained a $p-value < 6.3e{-}9$ with Software Metrics, a $p-value < 5.9e{-}9$ with Imports, a $p-value < 7.7e{-}9$ with Function Calls and a $p-value < 5.6e{-}9$ with Text Mining. We also compared the MCC values with the Vargha-Delaney A measure, and obtained a value lower than 0.028 in every case, indicating that our approach significantly outperforms related techniques.

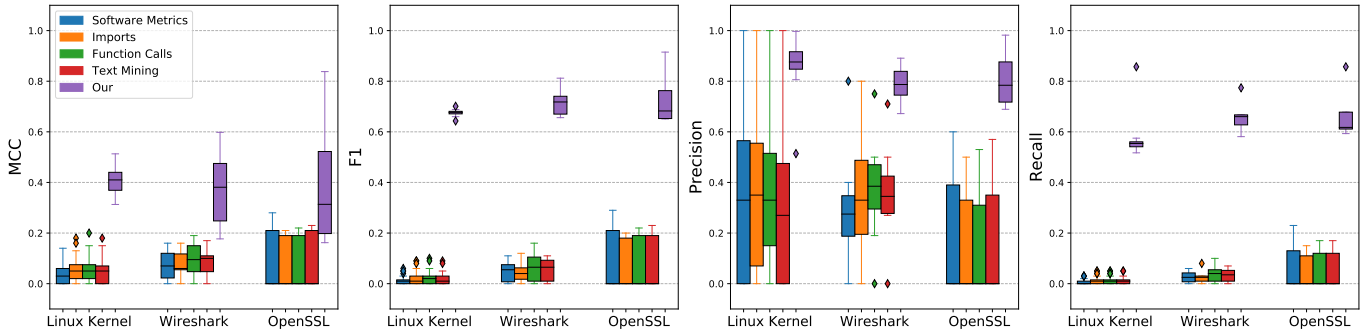

Fig. 4: Comparison with state-of-the-art techniques in a Realistic Training Setting (RQ4): Despite a reduced performance in a Realistic Training Setting, our approach significantly outperforms state-of-the-art techniques with an average improvement in MCC, F-measure, Precision, and Recall of 4.7, 15.76, 1.54, and 22 times respectively.

VII. Related Work

Early work in the area of vulnerability prediction has focused on defining features that could be linked to vulnerabilities and thus to be used to train learners. The first such work can be traced back to the study of Neuhaus *et al.,* [29], that investigated the use of libraries and function calls. Later, Shin *et al.,* [34], [36] and Zulkernine *et al.,* [10] investigated the use of code metrics such as complexity, code churn, and object oriented metrics.

These approaches although providing promising results were all using features designed based on human intuition. While powerful, such a way fails to take advantage of the big data and code available today. In view of this, Scandariato *et al.,* [31] advocated that the learners should find their own features without human intervention. To achieve this, they suggested an approach based on text mining where code is treated as text and the learner leans from Bags of words. Their results showed that text mining could outperform all previously introduced approaches [22].

Recently, deep learning techniques have been explored to automatically learn the required features to predict vulnerabilities. Li et. al [26] used Bidirectional LSTMs to train a vulnerability prediction model on *code gadgets*: semantically related lines of code. This technique was shown to be effective (under clean training settings) for analyzing two particular weaknesses, namely, buffer error vulnerabilities (CWE-119) and management error vulnerabilities (CWE-399). In contrast our approach trains the translation model on sequences extracted from the source code (methods) and does not target specific weaknesses.

Dam et. al [12] leveraged LSTM models to capture context relationships between different tokens in the source code to train a vulnerable component prediction classifier. Zhou *et al.,* and Suneja *et al.,* [37], [48] used Graph Neural Networks on program graph representations called Property Graphs [46] to predict vulnerabilities. Although promising, these approaches make the assumption that components with unreported vulnerabilities are non-vulnerable similar to what the other vulnerability prediction methods do. Therefore, their performance should be limited by the noisy data as shown by Jimenez *et al.,* [24]. Moreover, developing such methods require complex code analysis tasks which are impractical in many real-world cases, limiting the applicability of the methods. In contrast, our method is simple and easy to apply since it requires minimal information (preprocessing the variable and function names) to operate.

Machine learning has also been used in other software engineering prediction tasks. For instance, several works [13], [18], [43], [47] used machine learning models for defect prediction. Particularly, RNN models have been used for automatically fixing errors in C programs [17], for generating API usage sequences [16], and for fault localization [20]. Closer to our work, machine translation-based approaches have been successfully applied to automatically learn code features for detecting code clones [44], for learning how to mutate source code from bugs [40] and bug-fixing repairs [41]. Our approach constitutes, up to our knowledge, the first approach that proposes and evaluates a machine translation-based approach for vulnerability prediction.

VIII. Conclusion

This paper proposes a machine translation based approach to automatically learn the features to predict vulnerable components. Such predictions can be used to assist developers in code reviews and security testing. The important advantage of our approach is that it is completely automatic, it learns latent features (context, patterns, etc.) linked with vulnerabilities based on information mining from code repositories (in particular by analyzing historical vulnerability fixes and their context). We empirically evaluated the effectiveness of our approach following the methodological guidelines set by Jimenez et al. [24]. In particular, we compared our approach against the current state-of-the-art, on available releases of the three security-critical open source systems that were also used by [24], and showed that our approach outperforms existing techniques under both, clean and noisy (realistic) training data settings. On average, when trained in clean training data setting, our proposed approach achieved an overall improvement of 41% in MCC score. While the improvement in MCC score by 4.7 times achieved by our approach in realistic training setting, is even more remarkable.

REFERENCES

[1] Definition of vulnerability. https://cve.mitre.org/about/terminology.html, (accessed August 01, 2020).

[2] The heartbleed bug. https://heartbleed.com/, (accessed August 01, 2020).

[3] Linux kernal. https://www.kernel.org, (accessed August 01, 2020).

[4] National vulnerability database. https://nvd.nist.gov, (accessed August 01, 2020).

[5] Openssl. https://www.openssl.org, (accessed August 01, 2020).

[6] Wireshark. https://www.wireshark.org, (accessed August 01, 2020).

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2014.

[8] D. Britz, A. Goldie, T. Luong, and Q. Le. Massive Exploration of Neural Machine Translation Architectures. *ArXiv e-prints*, March 2017.

[9] Jason Brownlee. When to use mlp, cnn, and rnn neural networks. https://machinelearningmastery.com/when-to-use-mlp-cnn-and-rnn-neural-networks, 2018 (accessed August 01, 2020).

[10] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *J. Syst. Archit.*, 57(3):294–313, March 2011.

[11] M. L. Collard and J. I. Maletic. srcml 1.0: Explore, analyze, and manipulate source code. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 649–649, 2016.

[12] H. K. Dam, T. Tran, T. T. M. Pham, S. W. Ng, J. Grundy, and A. Ghose. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[13] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Softw. Engg.*, 17(4–5):531–577, August 2012.

[14] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[15] McAfee Center for Strategic and International Studies (CSIS). Economic impact of cybercrime - no slowing down. https://www.mcafee.com/enterprise/en-us/solutions/lp/economics-cybercrime.html.

[16] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 631–642, New York, NY, USA, 2016. Association for Computing Machinery.

[17] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 1345–1351. AAAI Press, 2017.

[18] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[19] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[20] Xuan Huo, Ming Li, and Zhi-Hua Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, page 1606–1612. AAAI Press, 2016.

[21] Matthieu Jimenez. FrameVPM: a framework to build and evaluate vulnerability prediction models. https://github.com/electricalwind/framevpm.

[22] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. An empirical analysis of vulnerabilities in openssl and the linux kernel. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 105–112. IEEE, 2016.

[23] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. Enabling the continous analysis of security vulnerabilities with vuldata7. In *Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation SCAM 2018, Madrid, Spain, September 23-24, 2018*, 2018.

[24] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. The importance of accounting for real-world labelling when predicting software vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 695–705, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Igor Kononenko. On biases in estimating multi-valued attributes. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, page 1034–1040, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[26] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[27] B.W. Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2):442 – 451, 1975.

[28] Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. Challenges with applying vulnerability prediction models. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, HotSoS '15, New York, NY, USA, 2015. Association for Computing Machinery.

[29] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 529–540, New York, NY, USA, 2007. Association for Computing Machinery.

[30] B. Potter and G. McGraw. Software security testing. *IEEE Security Privacy*, 2(5):81–85, 2004.

[31] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014.

[32] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, 2014.

[33] Apeksha Shewalkar, Deepika Nyavanandi, and Simone Ludwig. Performance evaluation of deep neural networks applied to speech recognition: Rnn, lstm and gru. *Journal of Artificial Intelligence and Soft Computing Research*, 9:235–245, 10 2019.

[34] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans. Softw. Eng.*, 37(6):772–787, November 2011.

[35] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, page 315–317, New York, NY, USA, 2008. Association for Computing Machinery.

[36] Yonghee Shin and Laurie Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18(1):25–59, February 2013.

[37] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim Laredo, and Alessandro Morari. Learning to map source code to software vulnerability using code-as-a-graph, 2020.

[38] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks, 2014.

[39] Yaming Tang, Fei Zhao, Yibiao Yang, Hongmin Lu, Yuming Zhou, and Baowen Xu. Predicting vulnerable components via text mining or software metrics? an effort-aware perspective. In *QRS*, pages 27–36. IEEE, 2015.

[40] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Learning how to mutate source code from bug-fixes. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep 2019.

[41] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4):19:1–19:29, 2019.

[42] András Vargha and Harold D. Delaney. A critique and improvement of the "cl" common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[43] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 297–308, New York, NY, USA, 2016. Association for Computing Machinery.

[44] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98, 2016.

[45] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[46] Fabian Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. *2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014.

[47] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26, 2015.

[48] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, 2019.

[49] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, page 91–100, New York, NY, USA, 2009. Association for Computing Machinery.