Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Machine learning for source code: vulnerability detection for C and Java

Author: Tina Marjanov (2576590)

1st supervisor:	prof. dr. ir. Fabio Massacci
daily supervisor:	dr. Ivan Pashchenko
2nd reader:	dr. Katja Tuma

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

December 21, 2021

Abstract

Machine learning methods are increasingly popular in detecting and correcting defects in source code. This thesis focuses specifically on vulnerability detection. We first provide a systematic literature review of recent approaches, breakthroughs and directions. We find a thriving, quickly developing field with numerous open questions and research directions. The review highlights –among others– two important questions: (i) reproducibility of defect detection/correction research, and (ii) extension of vulnerability detection systems past C/C++.

To explore the questions we first replicate one of the leading C/C++ vulnerability detection systems and perform extensive experiments by varying the parameters that were not explicitly defined. We are able to largely replicate the original study, but we do not achieve the same performance. We also show that performance is strongly affected by the choice of parameters, which further highlights the importance of transparency and clarity in reporting.

We then successfully adapt the pipeline into a Java vulnerability detection system. We show that we can find vulnerabilities in Java with the success comparable to the baseline. Our additional experimentation offers valuable insights into the importance of the dataset used for training and the challenge of detecting vulnerabilities in real source code.

Acknowledgements

The bulk of my master studies took place in the midst of uncertain and bleak times. It was not always easy, which makes people that stood by my side for the past years all the more valuable.

Thanks, Timo and Hrach, for being great friends and partners through the good and bad that the past years threw at us. Thank you Konstantinos, for bearing with me through my endless ramblings, brainstorming and debates. Thank you Ivan for your valuable insights and helping me through the more difficult stages of my research.

Finally, none of the work in this thesis would have been possible without the support and guidance of my supervisor, Fabio Massacci. Thank you for believing in me and being not only a supervisor, but a great mentor as well.

I learned a great deal while researching the material for this thesis, building my own system and experimenting with it. I hope my enthusiasm translated to the thesis and you enjoy reading it.

Tina Marjanov

Amsterdam, 21 December 2021

Contents

1

Introduction

Programming errors have been around for as long as programming itself; in 2017 alone, the cost of software failures reportedly reached $1.7 trillion and affected 3.6 billion people [62]. Yet, producing perfect code without mistakes is impossible for even the most experienced programmers. Unfortunately, the most elusive and difficult to find bugs can lead to severe consequences. OpenSSL's infamous Heartbleed vulnerability (CVE-2014-0160), for example, was caused by a single incorrect line of code and yet it made its way to thousands of web servers and gave attackers unparalleled access to sensitive information [69]. As we head towards a more and more software-reliant world, the need for quick and effective identification of such vulnerabilities is clear.

Traditionally, the detection of vulnerabilities was a resource–intensive endeavor, requiring great amounts of manual work and expertise. With the recent advances in deep learning and source code processing, availability of large corpora of source code, and easier access to big amounts of computational power, machine learning is beginning to take place as an attractive alternative to the traditional techniques. We provide evidence for the increasing popularity of machine learning techniques in vulnerability detection by conducting a systematic literature review.

One of the most notable breakthrough works that showed great promise in usage of deep learning for vulnerability detection in C programs is VulDeePecker [37]. While there has been significant work done on machine learning-aided vulnerability detection in C programs, Java has not yet received such attention, despite continuously appearing at the top of programming language popularity charts [10]. The main goal of this thesis is to design, implement and evaluate a machine learning system to detect vulnerabilities in source code of Java programs, much like VulDeePecker does in C.

With this in mind, the following goals and research questions are defined for this thesis:

- The exploration of state-of-the-art practices and techniques for vulnerability detection and closely related topics

  - What are the current best practices, challenges and future work directions in the domain of automatic defect detection and correction?

- The replication of a vulnerability detection system for C/C++ source code

  - Can the implementation and performance of VulDeePecker be replicated according to the information given in the original paper?

  - Which of the unspecified parameters and choices affect the performance and measurements of the replication?

- The creation of a vulnerability detection system for Java source code

  - Can the design of VulDeePecker be adapted and extended to support defect detection in source code of *Java* programs?

  - What performance can we achieve with Java vulnerability detection tool?

The contributions of this thesis can be summarized in the following points:

- We provide a concise review of applications of machine learning for detect detection and correction published between 2015 and 2020
- Using VulDeePecker replication as a case study, we show that performance cannot be perfectly replicated if all parameters of the pipeline are not explicitly disclosed. More specifically, resampling and reweighting heavily affect performance
- We successfully create a system for vulnerability detection in in Java; performance is comparable to published studies when using an artificial dataset, however the system underperforms when using a real dataset

The thesis is structured as follows. Chapter 2 introduces the relevant terminology and concepts necessary to understand the problem and the design of the vulnerability detection system. Chapter 3 follows with the presentation of the closely related works, promising directions and challenges. Chapter 4 presents the general design of a vulnerability detection system and discusses the implementation details. Chapter 5 and Chapter 6 present the results for replication of VulDeePecker and its adaptation to Java vulnerability detection system, respectively. Finally, we conclude with a discussion in Chapter 7.

2

Background

The following section provides the context, background information and terminology necessary to understand the remainder of the thesis.

2.1 Defects in source code

In the traditional literature on fault tolerance [32], an *error* is a part of an erroneous state which constitutes a difference from a valid state; a *fault* in the system is a defective value in the state of a component or in the design of a system; and a *failure* of a system occurs the first time the behavior of the system deviates from that required by its specification. More recently –both in practice and in the literature– these terms are often used loosely and interchangeably. We therefore adopt the recent terminology by Monperrus et al. [46]; the terms defect, error, bug, fault and similar will be, for the remainder of this thesis treated as synonyms, and refer to a "deviation between the expected behavior of a program execution and what actually happened". Depending on the type of defect found in a program, we distinguish:

i. *syntactic* defects refer to mistakes in the syntax of a program, i.e. the grammar and rules of the language. They are usually detected at compile or run-time and prevent the program from running at all and are the most well defined and arguably the easiest to target. Such problems -depending on the language in question- include missing brackets, semicolons, typos, missing indentations and similar.

ii. *semantic* defects refer to mistakes in the semantics of a program i.e. its meaning and intended behavior. They result in programs that do not behave as intended, but are not typically a security concern. Such problems include inconsistent method names,

variable misuse bugs, typing errors, API misuse, swapped arguments in functions and similar.

iii. *vulnerabilities* are a particular set of (typically) semantic defects that can compromise the security of a system. They include defects that are not obvious at runtime or show during normal execution, but can pose a security threat if abused by an attacker. Such problems include buffer overflows, integer overflows, cross-site scripting, use-after free etc.

The present thesis focuses on vulnerabilities. In order to facilitate the identification, increase exposure of vulnerabilities and simplify their classification, multiple systems for classification exist. Two of the most commonly used are CWE and CVE. CWE [45] stands for *common weakness enumeration* and classifies vulnerabilities by category - eg. basic cross-site scripting (CWE-80), SQL injections (CWE-89), improper validation of array index (CWE-129) and integer over-/under-flow (CWE-190, CWE-191). CVE [44], on the other hand, stands for *common vulnerabilities and exposures* and classifies the specific known instances of a vulnerability within some system or a product.

2.2 Defect detection and correction

As before, correction, repair, fixing, patching, recovery and similar will be used as synonyms and refer to "the transformation of an unacceptable behavior of a program execution into an acceptable one according to a specification" [46]. Such transformations range from adding, removing or replacing code, adding conditions, adding or removing method calls to type or naming changes. Traditionally, defects in source code have been discovered by means of static [12] (without executing the program) and dynamic [14] (by executing the program) analysis techniques. Such techniques, however, come with own set of challenges and drawbacks and cannot be always seamlessly integrated into the continuous integration and continuous delivery pipelines of the software projects [13, 51]. In this respect, machine learning techniques seem to become a very attractive alternative to the traditional software defect detection techniques.

Automated *error detection* is "a process of building classifiers to predict code areas that potentially contain defects, using information such as code complexity and change history. The prediction results (i.e., buggy code areas) can place warnings for code reviewers and allocate their efforts" [33]. There have been several error detection tools available over the recent years, two of the bigger ones being Google's Error Prone [1] and SpotBugs (formerly

known as FindBugs [30]). Such earlier works were usually frameworks on which checkers, consisting of manually defined heuristics, formal logical rules and test oracles containing ground truth, could be built. This made them both arduous to construct and slow to adapt. In addition to that, they would often only target a specific class of bugs despite requiring a considerable amounts of expert work and tend to generate many false positives.

To alleviate mentioned problems, some more recent attempts have been made to detect errors with minimal manual work, without oracle and without information about the intended behavior of the program. Such approaches rely on the fact that errors are anomalous behaviors and can be statistically distinguished from *correct* code [26]. A similar observation comes from recent work [29] in which the authors argue that source code has an inherent property called *naturalness*, meaning that code follows similar patterns as natural language and buggy code stands out in the same way a grammatically incorrect sentence does.

A logical next step from detection is automated *software correction*. It tries "to automatically identify patches for a given bug, which can then be applied with little, or possibly even without, human intervention" [19]. Compared to detection, correction is a more ambitious goal, which has only recently emerged as a research topic through the use of techniques previously applied to natural language. They do so by either learning to translate from pairs of incorrect and correct programs (as one would translate between two languages, i.e. English and Dutch) or learning from correct examples and translating programs that deviate from that (equivalent of learning one language and then correcting misuse of it).

Overall, similarities of source code to natural language, together with recent developments in machine learning methodology, computational power and wider availability of large corpora of real world open-source code –i.e. Big Code [55]– have made it possible to tackle both issues as a machine learning problem. It is not surprising that the most recent approaches borrow techniques from both natural language processing and deep learning. Such techniques can aid in finding interesting features and properties of programs and help distinguish erroneous patterns and potentially fix them.

2.3 Machine learning for source code

A typical machine learning pipeline consists of several important stages: data collection, data preparation, model training, and finally evaluation and deployment:

i. During the data collection stage, a sufficiently large and representative dataset for the task is constructed.

ii. Data preparation consists of cleaning and sometimes labeling, feature engineering and lastly splitting into (non-overlapping) subsets for training and testing. Ideally, the goal is to eliminate as much noise as possible to allow for better training. Additionally, it is important to select the most relevant features, which is often a non-trivial task.

iii. Model training phase is often computationally the most expensive. During the process, the training portion of the dataset is used to create such a model which will be able to distinguish erroneous code from correct one. Depending on the technique and type of model used, it is often necessary to adapt the parameters and retrain several models before achieving satisfactory results.

iv. Lastly, the model is evaluated on the test subset of data to determine if the model shows the desired behaviors when presented with unseen data. At this stage, the model should be able to detect programming defects and can be deployed to be used. Typically, the tool is monitored, maintained and improved also after deployment, but this is outside of the scope of this thesis.

2.3.1 Source code representation

As for step (ii), machine learning models are typically not capable of ingesting the source code in its original format, and therefore, the code is processed and transformed into some low level representation appropriate for ML model input (e.g. vectors for neural networks). To preserve the semantic and syntactic properties of the program, it is useful to consider some intermediate interpretation, capable of encoding such properties before feeding the program into the model. The three predominant approaches treat the source code as:

- *sequence of tokens*: the raw source code is split into small units (e.g. “int”, “func”, “(”, “)”, “{”, “}”) and presented to the model as such

- *abstract syntax tree* (AST): the abstract syntactic structure of source code is captured by a tree representation, with each node of the tree denoting a construct occurring in the source code

- *graph* encoding various semantic properties (e.g. control flow graph [4] or code property graph [70]): it can capture various syntactic or semantic relationships and properties of the source code through the edges and nodes of the graphs

2.3.2 Bidirectional Long Short-Term Memory

While there is a variety of machine learning approaches available to serve as the model for step (iii), neural networks have proven especially successful for dealing with source code. The aforementioned representations allow the code to be encoded as a sequence of vectors, which are easily ingested by the network, so that no additional feature engineering is necessary.

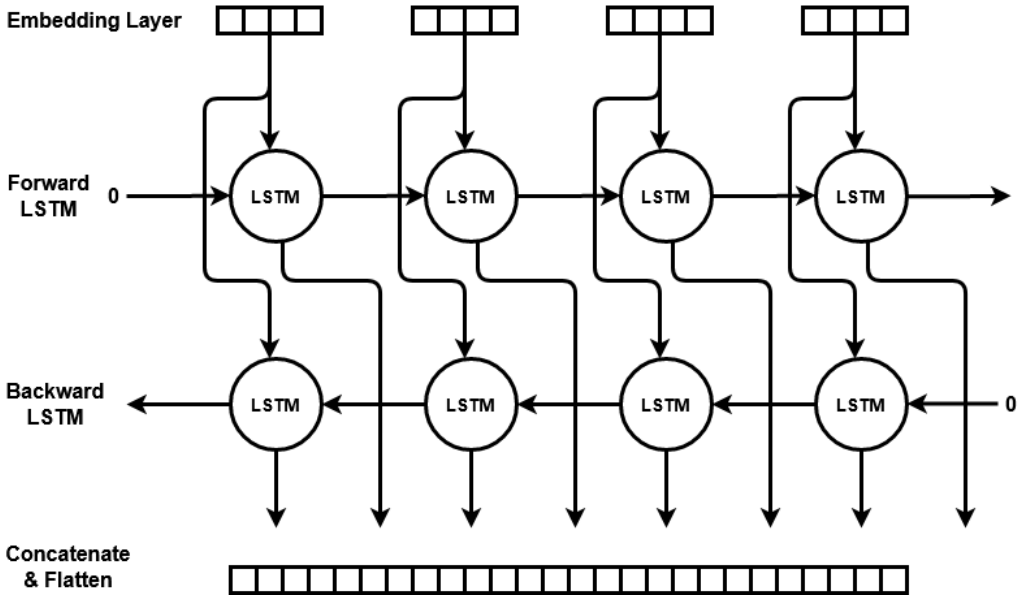

Figure 2.1: Architecture of a BLSTM

In this thesis, a type of neural network called Bidirectional Long Short-Term Memory (BLSTM) is used. In order to understand the architecture of BLSTM and what sets it apart from other options, we need to zoom out and understand the family of (recurrent) neural networks first.

A neural network is a connected network of simple computational nodes -artificial *neurons*-, connected to each other through edges with certain weights. As data passes through the network during training, the weights are incrementally adjusted so that the network can eventually tell whether a new piece of source code passing through is vulnerable or non-vulnerable. A neural network typically consists of an input and output layer and a number of (hidden) layers. In a recurrent network, the nodes are structured in such a way, that the outputs of nodes in one layer can be used as input for nodes in the following nodes. The architecture and the recurrent nature of computations makes it possible for the network to take into account historical information. In source code analysis, this means

that the network is in a way “aware" of the context and related code it has seen before.

Traditional recurrent neural networks, however, commonly run into the problem of vanishing gradient because of the multiplicative nature of the gradient with respect to the number of layers. This means that the network struggles with capturing of long term dependencies (i.e. the code it encountered less recently). Long Short-Term Memory is a variant of recurrent neural network that is not vulnerable to the vanishing gradient problem and therefore highly efficient at remembering long term dependencies. Bidirectional LSTM (Figure 2.1) is an extension that does not only consider the past, but also future context by adding an additional *backward* LSTM.

When specifying the architecture of a BLSTM, a number of choices can be made. First, the general geometry of the network is set by the number of layers and neurons. Next, neurons require an *activation function* through which the weighted sum is passed; the activation function then decides if the neuron activates and what is “fired” to the next neuron. Sometimes, training benefits from some purging of the network by probabilistically dropping some nodes - this probability is called the dropout rate. Lastly, a method to calculate loss while training (i.e. the metric to determine how good the network’s current prediction power is) can be specified in the form of loss function.

For the training of the model, several strategies, data normalization techniques and optimization algorithms exist. Important factors for training include the number of epochs (i.e. the number of times the algorithm trains on the whole dataset) and minibatch size (i.e. the size of batches in which the data is fed to the network.)

2.3.3 Performance metrics

Prediction of a machine learning model has four possible classification states: true positive (TP: cases correctly classified as true), true negatives (TN: cases correctly classified as false), false positives (FP: cases incorrectly classified as true) and false negatives (FN: cases incorrectly classified as false). In the case of bug detection, a true positive means that a buggy line of code was correctly classified as a bug, and a false positive that a non-buggy line of code was wrongly classified as a bug. The confusion matrix [66] can easily be generalized for multi-class problems by adapting true and false into relevant classes.

With these in mind, a number of measures of performance are often reported [20]. *Precision* is defined as $P = \frac{TP}{TP+FP}$ and shows what portion of classifications was actually correct. *True positive rate*, also known as *sensitivity* or *recall* is defined as $TPR = \frac{TP}{TP+FN}$ and shows what proportion of actual positives was classified correctly. *True negative rate*,

also known as *specificity* does the same, but for negatives: $TNR = \frac{TN}{TN+FP}$. *False positive rate* is defined as $FPR = \frac{FP}{FP+TN}$ and shows the proportion of false positives compared to all non-vulnerable samples. On the other hand, *false negative rate* is defined as $FNR = \frac{FN}{TP+FN}$ and shows the proportion of false negatives compared to all vulnerable samples. Finally, F_1 score combines precision and recall as follows: $F_1 = \frac{2(P*R)}{P+R}$.

Ideally, a tool should achieve high scores in precision and recall, while keeping false positives and false negatives low. Realistically, there is often a trade-off to be made between precision and recall, when an increase in one leads to a decrease of another. In other words, a tool might have to compromise with either a high false positive rate (low threshold to report a bug, at the cost of over-reporting and burying actual bugs in a pile of false positives) or high false negative rate (reporting only highly suspicious cases, but potentially missing some actual bugs). The best decision is generally application dependent and best done on per-case basis.

2.3.4 Machine learning challenges

Machine learning comes with a distinct set of challenges. Firstly, it is crucial to train the model on a *high quality dataset*. In general, this means a large enough dataset and a representative distribution of classes. For example, a model that is trained on a dataset that contains an equal number of buggy and non-buggy programs, might not perform best when used in a real setting where occurrence of bugs is significantly lower. When facing such challenges, resampling (undersampling or oversampling) of the data or class reweighting techniques are commonly applied. On the other hand, since model training is still a very resource-heavy task, training with too much data might take an unreasonably large amount of time. As mentioned before, training a good model is often about making an acceptable trade-off between different performance metrics by finding the right balance between the parameters. While there are some heuristics and values, that are commonly used as a starting point, the training phase will inevitably require some exploration and testing.

Next, the selection of relevant features is one of the most important tasks of machine learning. It is important to consider the amount of features –*more features is not necessarily better*– and what information about the code they carry. The most recent deep learning based approaches do not require manual feature selection, but rather take advantage of the ability of the model to learn the important features directly from the training data itself. In order to achieve this, the data might need to be simplified, at the danger of losing too much information. A common problem that surfaces when evaluating or when replicating

the results is *over-fitting*, meaning that the model too closely fits the training data and does not show same predictive power in production than it did during the training, often due to noisy data or over-complicated model. In such cases is important to consider the performance metrics on test data and stop the training before the model stops learning and starts merely "memorizing" the values it sees.

3

Systematic literature review of related works

3.1 Overview and methodology

The goal of the coming chapter is to examine and present a representative snapshot of the state-of-the-art research and identify the trends and gaps in both detection and correction of errors in source code. Starting from an agnostic starting point, we want to discover patterns without being biased by our own dispositions and conjectures. For this we leverage on the grounded theory [18] approach widely used in the empirical studies; this approach allows the hypotheses to emerge from the data. We uncover not only the latest state and approaches of Java vulnerability detection, but also closely related topics can offer valuable insights that might prove useful in the creation of our own Java vulnerability detection tool.

3.1.1 Inclusion and exclusion criteria

The initial set of 343 works was drawn from an online repository containing the state-of-the-art machine learning research on source code.[1] To reflect the state of the art techniques and considering that machine learning is rapidly evolving, we focus our review on the papers from 2015 onward (322 papers out of 343).

In addition to that, we only keep the papers on defect detection and correction, removing papers on other topics, such as code synthesis, prediction, recommendation, summarization and similar. We closely focus on the source code itself, and therefore, also exclude any

[1] Available on https://ml4code.github.io; the collection was created in the scope of [2] and is still maintained by the author.

papers discussing supporting techniques for defect detection such as testing, fuzzing, taint analysis, symbolic execution, defects in binary code etc. Finally, we exclude papers without proof of concept or papers that do not contain a full machine learning pipeline. Papers that share large parts of the pipeline and only adapt or discuss one part of it are treated as one, with only the most representative paper being included and discussed. After the removals, a set of 31 relevant papers has emerged.

To avoid biases and present a complete picture, we consider any additional relevant works referenced in the original set of papers and cross-reference the works with top hits in the domain from Google Scholar. The final list comprises 40 papers containing an end-to-end machine learning pipeline capable of either detecting or also correcting defects in source code.

3.1.2 Coding of the selected papers

To allow the discovery of emerging patterns from data (in this case, the set of selected papers), we first needed to identify the defining characteristics of an defect correction/detection tool, so that they can be annotated in each of the selected papers. The initial codes were heavily inspired by the characteristics discussed and defined by the surveys discussed in Chapter 2. The proposed codes were first used to annotate a small portion of the selected papers to test their suitability and completeness. We aimed to identify a set of codes that captures the most important differences between the studies, while ensuring that no part of the pipeline is left out. We performed this process iteratively until the code book became stable. Finally, after finalizing the full set of codes, we expanded the coding to the remainder of the papers.

The coding and the subsequent analysis including the creation of co-occurrence tables between categories and extraction of general statistics across the papers was performed using *Atlas.ti*.

3.1.3 The code book

The final set of code groups and their respective codes capture general information about the tools and the datasets used to train and test the model. Code groups related to the abilities of tools are:

- ◇ *Correction* refers to correction and detection ability of the tool as defined in section 2.1.

Table 3.1: Final code book

Code Group	Code	Description	Example
Correction	No	tool capable of only detecting defects	we present the design and implementation of a deep learning-based vulnerability detection system [37]
	Yes	tool capable of correcting defects	we present an end-to-end solution [...] that can fix multiple such errors in a program [21]
Defect type	Syntactic	tool targets syntax defects	algorithm [...] for finding repairs to syntax errors [6]
	Semantic	tool targets semantic defects	addressing the issue of semantic program repair [16]
	Vulnerability	tool targets vulnerabilities	System for Vulnerability Detection [37]
Representation	Tokens	source code represented as a sequence of tokens	the model treats a program statement as a list of tokens [54]
	AST	source code represented as an abstract syntax tree	representations of the abstract syntax trees (ASTs) [40]
	Graph	source code represented as a graph, capturing additional semantic information (CFG, DFG, ...)	this step generates a System Dependency Graph (SDGs) for each training program [74]
Language	Python	tool evaluated on source code written in Python	from the Introduction to Programming in Python course [6]
	C	tool evaluated on source code written in C/C++	Fixing Common C Language Errors [21]
	Java	tool evaluated on source code written in Java	we target Java source code [28]
	JavaScript	tool evaluated on source code written in JavaScript	broad range of bugs in JavaScript programs [17]
	C#	tool evaluated on source code written in C#	open source C# projects on GitHub [3]
Type	No bug	tool trained on only non-buggy source code	using language models trained on correct source code to find tokens that seem out of place [58]
	Bug + Fixed	tool trained on *paired* examples of buggy and fixed code	Given a pair (p; p^0) where p is an incorrect program and p^0 is its correct version [22]
	Bug + No bug	tool trained on *unpaired* examples of buggy and non-buggy code	dataset that contains 181,641 pieces of code [...] Among them, 138,522 are non-vulnerable and the other 43,119 are vulnerable [74]
Label	Yes	tool trained on labeled data	A program is labeled as "good" [...], "bad" [...], or "mixed" [...] [74]
	No	tool trained on unlabeled data	self-supervised learning with unlabeled programs [71]
Data Availability	Yes	dataset is publicly available	-
	No	dataset is not publicly available	-
Tool Availability	Yes	tool is publicly available	-
	No	tool is not publicly available	-
Details	Low	(almost) no detail given	–
	Middle	some details given, but not enough for full replication	–
	High	enough detail given to perfectly replicate the paper	–

◇ *Defect type* refers to the primary type of defect the tool targets, as defined in section 2.1. If a more advanced tool can simultaneously correct simpler mistakes (e.g. semantic defect tool fixing misplaced brackets –a syntax mistake), we classify it according to the most advanced type of defect it can target.

◇ *Representation* refers to the main representation of the source code that is fed to the model as defined in subsection 2.3.1. This does not include the further transforma-

tions inside the models, but rather the initial information that is presented to the model.

- ◇ *Language* refers to the language that the tool targets. More specifically, we refer to the training and testing datasets, in case the tool can act in a language-agnostic way.

Codes that capture information about the datasets include[1]:

- ◇ *Type* refers to the structure of the dataset and captures what types of examples are included in the dataset. It captures whether the datasets include buggy examples, and –if bugs are present– whether buggy and non-buggy examples are paired.
- ◇ *Label* captures whether the dataset in labelled or unlabelled.

Codes that capture the availability of source material and replicability include:

- ◇ *Dataset availability* captures whether the dataset used for training and testing is publicly available.
- ◇ *Tool availability* captures whether the tool created is open source and publicly available
- ◇ *Details* captures the level of details given about the tool's full pipeline including the data (e.g. amount, source, sampling/split etc.), preprocessing (e.g. cleaning, transformations, vectorization), implementation (e.g. tools, libraries, machine/hardware etc.), model architecture (e.g. model type, layers, activation function, dropout rate), training and testing (e.g. epochs, loss, batch size, learning rate, time required).[2]

Table 3.1 presents the final code book. It shows the identified code groups, the possible values for each of them, and illustrative examples taken from the source papers[3].

[1] We refer to *training data*. When training is performed on data that does not have the same structure as test subset, we describe the training data (e.g. correction tools that only train on non-buggy examples). Additionally, when training data is collected from a public dataset, but then modified in some way, we describe the modified version of data (e.g. an existing publicly available dataset of non-buggy code is injected with bugs).

[2] In coding the level of details, we take into account the variability in pipeline structures and parameters required for different models.

[3] An overview of the included studies together with the codes is also available on https://github.com/tmv200/ml4code/blob/main/sota.yaml.

Table 3.2: Studies

	General				Dataset			Reproducibility		
Tool	**Defect**	**Represent.**	**Method**	**Language**	**Size**	**Type**	**Label**	**D. avail.**	**T. avail.**	**Repr**
sk_p [54]	Sem	Token	RNN (LSTM), skipgram	Python	7×315-9,000 programs	NB	-	-	-	Low-mid
DeepFix [21]	Syn	Token	RNN	C	7,000 programs	B+F	✓	✓	✓	High
SynFix [7]	Syn	Token	RNN (LSTM)	Python	40,000 programs	NB	-	-	-	Low-mid
SSC [16]	Sem	AST	RNN, rule-based	Python	2,900,000 code snippets	B+F	✓	✓	-	Mid
Harer et al. [27]	Vul	Token	GAN	C	117,000 functions	B+F	✓	-	-	Low-mid
Ratchet [28]	Sem	Token	RNN (LSTM)	Java	35,137 pairs	B+F	✓	✓	✓	High
Sensibility [58]	Syn	Token	n-gram, RNN (LSTM)	Java	2,300,000 files	NB	-	✓	✓	High
SequenceR [11]	Sem	Token	RNN (LSTM)	Java	40,000 commits	B+F	✓	✓	✓	High
RLAssist [22]	Syn	Token	DRL, RNN (LSTM)	C	7,000 programs	B+F	✓	✓	✓	High
Liu et al. [41]	Sem	AST, Token	CNN, paragraph vector	Java	2,000,000 methods	B+F	-	✓	✓	High
DeepDelta [42]	Sem	AST	RNN (LSTM)	Java	4,800,000 builds	B+F	✓	-	-	Low-mid
Tufano et al. [63]	Sem	AST	RNN	Java	2,300,000 fixes	B+F	✓	✓	✓	High
VarMisuseRepair [64]	Sem	Token	RNN (LSTM), pointer network	Python	650,000 functions	B+F	✓	✓	-	Low
DeepRepair [68]	Sem	AST	RNN	Java	374 programs	B+F	✓	✓	✓	High
Hoppity [17]	Sem	AST	graph-NN, RNN (LSTM)	JavaScript	500,000 program pairs	B+F	✓	✓	✓	High
SampleFix [25]	Syn	Token	GAN, CVAE, RNN (LSTM)	C	7,000 programs	B+F	✓	✓	-	Mid
DLFix [36]	Sem	AST	RNN (tree-RNN)	Java	4,900,000 methods	B+F	✓	✓	✓	High
Graph2Diff [61]	Sem	AST	graph-NN (GGNN)	Java	500,000 fixes	B+F	✓	-	-	Low
DrRepair [71]	Syn	Token, Graph	graph-NN, RNN (LSTM)	C	64,000 programs	B+F	-	✓	✓	High

(a) Works which detect *and* correct defects

	General				Dataset			Replicability		
Tool	**Defect**	**Represent.**	**Method**	**Language**	**Size**	**Type**	**Label**	**D. avail.**	**T. avail.**	**Repr**
Wang et al. [67]	Sem	AST, Graph	DBN	Java	10×150-1046 files	B+F	✓	-	-	Low
DP-CNN [33]	Sem	AST	CNN, logistic regression	Java	7×330 files	B+F	✓	✓	-	Mid
POSTER [40]	Vul	AST	RNN (BLSTM)	C	6,000 functions	B+F	-	✓	✓	High
DeepBugs [53]	Sem	AST, Graph	NN	JavaScript	150,000 files	B+F	✓	✓	✓	High
VarMisuse [3]	Sem	AST, Graph	GGNN, GRU	C#	2,900,000 LoC	B+NB	-	✓	✓	High
VulDeePecker [37]	Vul	Token	RNN (BLSTM)	C	61,000 code gadgets	B+F	✓	✓	-	Mid
Russell et al. [56]	Vul	Token	CNN, BoW, RNN, random forest	C	1,270,000 functions	B+F	✓	✓	-	Mid
μVulDeePecker [74]	Vul	AST, Graph	RNN (BLSTM)	C	181,000 code gadgets	B+NF	✓	✓	-	Mid-high
Gupta et al. [23]	Sem	AST	tree-CNN	C	29×1,300 programs	B+F	✓	✓	✓	High
Habib and Pradel [24]	Syn	Token	RNN (BLSTM)	Java	112 projects	B+F	✓	-	-	Low-mid
Li et al. [35]	Sem	AST, Graph	RNN (GRU), CNN	Java	4,900,000 methods	B+F	✓	✓	✓	High
Project Achilles [57]	Vul	Token	RNN (LSTM)	Java	44,495 programs	B+F	✓	✓	✓	High
Li et al. [34]	Vul	Graph, Token	BoW, CNN	C	60,000 samples	B+NB	-	-	-	Mid
VulDeeLocator [38]	Vul	AST, Token	RNN (BRNN)	C	120,000 program slices	B+NB	✓	✓	✓	High
SinkFinder [8]	Vul	Graph, Token	SVM	C	15,000,000 LoC	NB	-	✓	-	Mid
OffSide [9]	Vul	AST	attention-NN	Java	1,500,000 code snippets	B+F	✓	✓	✓	High
AI4VA [59]	Vul	AST, Graph	graph-NN	C	1,950,000 functions	B+F	✓	✓	✓	High
Tanwar et al. [60]	Vul	AST	NN	C	1,270,000 functions	B+F	✓	-	-	Low
Devign [73]	Vul	all	graph-NN	C	48,000 commits	B+F	✓	✓	✓	High
Dam et al. [15]	Vul	AST, Token	RNN (LSTM)	Java	18×46-3450 files	B+NB	✓	-	-	Low-mid
SySeVR [39]	Vul	AST, Graph	RNN (BLSTM, BGRU)	C	15,000 programs	B+F	✓	✓	✓	High

(b) Works which *only* detect defects

Note to subtables: The studies are ordered first chronologically and then alphabetically (by author name) within each subtable.
Notes to columns:
(I) Defect abbreviations: Sem=Semantic, Syn=Syntactic, Vul=Vulnerabilities
(II) Method: We refer to the primary machine learning method used in the tool. When a tool experiments with several approaches we include all if they are presented and discussed equally, and skip the ones only mentioned in passing.
(III) Method abbreviations (alphabetically): (B)GRU=(Bidirectional) Gated Recurrent Unit, (B)LSTM=(Bidirectional) Long Short Term Memory, BoW=Bag of Words, CNN=Convolutional Neural Network, CVAE=Conditional Variational Auto-Encoder DBN=Deep Belief Network, GAN=Generative Adversarial Network, GGNN=Gated Graph Neural Network, NN=Neural Network, RNN=Recurrent Neural Network, DRL=Deep Reinforcement Learning SVM=Support Vector Machine
(IV) Type abbreviations: NB=No bug, B+NB=Buggy and non-buggy, B+F=Buggy and Fixed
(V) D. avail.= dataset availability, T. avail.= tool availability

3.2 Analysis and takeaways

In the following section, we discuss the most important observations about the approaches and emerging trends and patterns within and across the selected papers.

Table 3.2 provides an overview of the studies included in this review. Generally speaking, we can see an increase in publications since 2015, signaling a growing interest in the field. This is evident from the number of publications per year shown in Figure 3.1. This holds for both detection and correction studies. The slight drop in publications of defect correction studies could be the consequence of small sample size of reviewed studies or an actual shift towards defect detection papers.

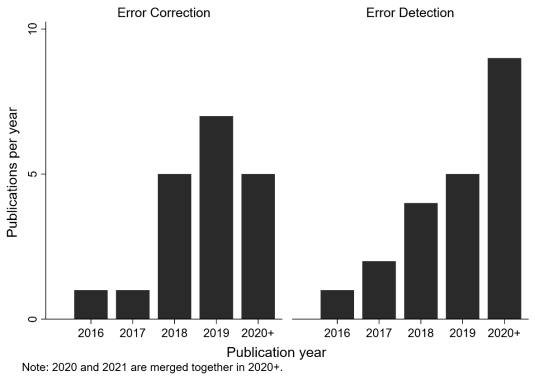

Figure 3.1: Histogram of publications per year

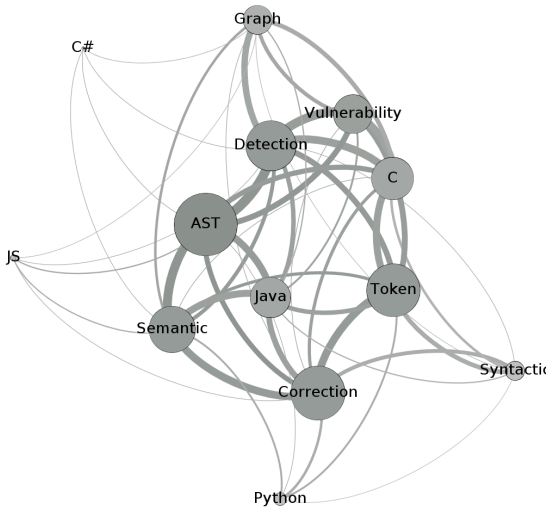

Figure 3.2: Co-occurrence graph

Overall, the examined papers exhibit wide variety in goals and priorities, which leads to a variety of approaches as discussed in the current section. For reasons of conciseness and brevity we leave out detailed description and low-level comparisons between the machine learning methods and focus on more general directions in approaches. Figure 3.2 is a co-occurrence network representing the relationships between the the characteristics of presented studies (codes), with more strongly related (co-occurring) concepts appearing closer to each other. The sizes of nodes and the thickness of edges represent how frequently a characteristic appears and the number of co-occurrences respectively. The next subsections discuss in more detail the findings across each category.

3.2.1 Detection vs. correction ability and defect types

Takeaway 1: *We find an almost equal split between the papers that focus only on detection and those also correcting defects.*

21 papers focus only on detecting defects, while 19 can also correct them. In terms of their evolution over time, research of both types seems to be growing fast as can be seen in Figure 3.1.

Takeaway 2: *The papers mostly address semantic defects and vulnerabilities, syntactic defects are less popular. Among them, vulnerabilities are only detected whereas semantic and syntactic defects are often also corrected.*

7 papers target syntactic defects, 15 vulnerabilities and 18 semantic defects. Correction studies target mostly semantic (12) and syntactic (6) defects, while the detection studies target mostly vulnerabilities (14) and semantic defects (6). Only a single correction study [27] targets vulnerabilities and one detection study [24] focuses on syntactic defects.

Since defect detection often targets more complex problems such as semantic bugs and vulnerabilities, many detection papers focus on a more narrow array of problems or try to narrow the granularity. As such, *DeepBugs* [53] only targets name-based semantic bugs, *SinkFinder* [8] examines security sensitive function pairs, while *OffSide* [9] looks for boundary condition mistakes.

Among the correction papers, [27] presented one of the first studies requiring no paired labeled examples for mapping from buggy to non-buggy domain. *Sensibility* [58] was one of the first studies focusing on correction of single token syntax defects *across domains*. *DeepRepair* [68] builds on the idea of redundancy, exploiting the fact that many programs contain seeds to their own repair. More advanced studies like *Hoppity* [17] use neural networks for source code embedding and graph transformations in order to correct semantic mistakes. *Graph2Diff* [61] and *VarMisuseRepair* [64] both use pointer networks, using pointers to locate the defect and a potential fix.

Takeaway 3: *Correction papers mostly use AST and tokens whereas detection studies use all the three representations.*

We can see a significant division in representation approaches between detection and correction studies. Firstly, the correction studies mostly use a single representation (17 out of 19), while detection studies more often use a combination of multiple approaches (12 out of 21). Among the defect correction papers, the most common representation is tokens (12), followed by AST (8). Only one correction study uses graph representation [71].

The split in representations is a bit more balanced among the detection-only papers: AST appears 15 times, graph 10 times, and tokens 9 times.

3.2.2 Source code representation

Takeaway 4: *The majority of the studies use either AST or token representation, with graph representation being the least used one. Despite the different representations, the input is commonly flattened when serving as input for a neural network.*

AST representation is used by 23 papers, token representation appears in 21 studies, and graph representation is used by 11 studies. The approaches can coexist, which is evident from the studies that combine several representations: 11 defect detection and 2 defect correction studies use some combination of the source code representations. The most common combination is AST – Graph (7), followed by AST – token (3) and Graph – Token (3). Zhou et al. [73] use a combination of all the three representations.

With deep learning rising compared to other machine learning techniques, the need for manually defined "traditional" features is falling. Instead, neural networks require input in the form of a vector. To achieve that, the previously described source code representations are commonly flattened into a vector, appropriate for neural network input [37, 41].

Takeaway 5: *Different representation seems preferable for addressing different types of defects.*

There seem to be different preferences in representation choice depending on the defect type targeted by a study. Syntactic defects almost exclusively use token representation (7), with a single paper adding graph representation [71]. On the other hand, papers aimed at semantic defects primarily use ASTs (14), followed by tokens (5) and graphs (4). The most variety in representation comes from the vulnerability finding papers. Those use AST and tokens equally often (9), with graphs used only slightly less commonly (6). Vulnerability finding studies also most commonly use a combination of more than one representation.

3.2.3 Languages

Takeaway 6: *The majority of examined studies target C and Java, with only a few papers aimed at other languages.*

Within the examined works, five programming languages are supported: C/C++ (17), Java (16), Python (4), JavaScript (2) and C# (1). Several of the featured studies aim to be language and syntax agnostic, but were only trained and tested on a specific language

(in those cases, we classify papers to that specific language). It is, however, commonly noted by the authors that the presented studies could be used on different languages with minimal changes to models and by retraining on a suitable dataset.

Takeaway 7: *We see a non-uniform distribution of goals across the examined languages, both in terms of correction ability as well as defect types targeted.*

Looking at the correction ability, we notice that the majority of C studies (12) only detect defects, while only five can correct defects. On the other hand, Java is more balanced, with seven detection and nine correction studies. JavaScript has one paper for correction [17] and one for detection [53]. All the four Python studies are capable of correction. Finally, the one examined C# paper [3] can detect defects. Overall, the two most commonly appearing are defect detecting C studies (12) and defect correcting Java studies (9).

In terms of defect types, most of the C-language studies target vulnerabilities (12), while the majority of Java papers target semantic defects (11). Python studies focus primarily on semantic defects (3), with one paper targeting syntactic defects. The two examined JavaScript studies as well as the only C# study target semantic defects. There is no Python, JavaScript or C# papers that focus on security vulnerabilities. Similarly, not a single JavaScript or C# paper aimed at detecting or correcting syntax defects.

3.2.4 Machine learning approaches/models

Takeaway 8: *Both defect detection and correction studies increasingly rely on neural networks. The most commonly used class of models is RNN.*

Defect correction studies heavily borrow from natural language translation, also often referred to as neural machine translation or sequence-to-sequence translation. This means that the majority of the models come from the same domain; more specifically recurrent neural networks (RNN) that appear 16 times out of 19 among defect correction papers. The most common method within the RNN family is the Long Short-Term Memory (LSTM) -11 studies-, which specifically targets the problem of long-term dependencies by allowing learning from context. The most recent papers highlight the usefulness of neural networks that are capable of understanding *contexts*, since the presence of a defect can highly depend on that [35]. Additionally, *attention* (focusing on the relevant parts of the code depending on the context) helps such neural networks learning long distance relations, which allows the systems to keep track of the context outside of a narrow code segment. It is worth mentioning that despite perceived uniformity, most studies add their own spin to the method, leading to diverse final implementations.

Among defect detection papers, nine use recurrent neural network, four use convolutional neural networks. Most of the remaining papers still rely on some member of the neural network family (e.g. spins on attention neural network, (gated) graph neural network, deep belief network etc.). Similar to defect correction studies, methods that can learn from context, such as Bidirectional Long Short-Term Memory (BLSTM) -5 papers- and Gated Recurrent Unit (GRU) –3 papers–, are popular due to their ability to take into account both future and past contexts [37]. Thus, there is only slightly more variety in the defect detection world, where the task can (but does not need to) be logically split into two: the embedding/feature extraction and the classification itself. While the former is mostly handled by a form of neural network, the latter invites more experimentation. Some of the classification methods include logistic regression[33], Bags of Words[34, 56], Random Forest[56], and Support Vector Machine[8]. Despite some outliers, the task of detection also seems to be heading in the neural network direction. The analysed papers commonly attribute this to the neural network's ability to operate without explicit feature formation, the ability to understand contexts and keep some form of memory over time; and neural network's suitability for handling texts and (a form of) language.

3.2.5 Datasets and results

Takeaway 9: *There is large disparity between datasets in terms of dataset size, data unit size, code complexity, realism, and source.*

The sizes of the datasets range from hundreds to millions of data units. The sizes of data units themselves (i.e. the source code to be fed into the model) also range from full program files to methods, functions, code gadgets or similar system-specific granularities. We notice that the granularity of data points mostly coincides with the output granularity, at which the tool is capable of spotting defects.

Additionally, the datasets differ in terms of complexity and realism. On the one hand, several studies relied on beginner student assignments, simple code segments or synthetically injected bugs (e.g. [7, 21]). On the other hand, several datasets consist of real open source projects collected from Github or big projects such as full Linux kernel source code (e.g. [8, 58]). We also notice, that authors of the examined works often source their data from publicly available datasets (14 papers), either with significant modifications or only as a subset of the original dataset. Examples of such datasets include SARD and NVD, Juliet Test Suite and Draper.

Takeaway 10: *The majority of datasets consist of bug-fix pairs.*

We notice three distinct patterns in dataset structure: datasets with bug-fix pairs (31), datasets of unrelated buggy and non-buggy examples (5) and datasets with no bugs (4). Datasets without bugs are mostly used to teach a model the correct use of the language, so that it is capable of discrimination and potentially translation when it encounters a code pattern it is unfamiliar with. The remaining two dataset patterns help to teach the model examples of *good* and *bad* behavior. The difference is that for defect correction, it is valuable to have examples of concrete fixes for a buggy example. This is most commonly achieved by either collecting version histories (commits with fixes) from publicly available repositories or artificially injecting bugs to correct code. In case of defect detection, it is not crucial to have such pairs, so several of the datasets include examples of bugs and correct code, but not necessarily on the same piece of code.

Takeaway 11: *There is little uniformity among studies' outputs - we find varying levels of granularity, but also some experimentation with additional feedback directions.*

In terms of the paper output, we notice a significant variety of detection granularity, ranging from simple binary classification (buggy vs. non-buggy program or file) to method, function or specific line of code. For example, [15] focuses on file level detection, *VulDeePecker* [37] works on code gadget granularity, and *Project Achilles* [57] on methods. Additionally, research has also been done on tools that can point out the specific type of defect they encounter. An interesting goal was set by Zou et al. [74]. The authors attempted to not only recognize whether there is a vulnerability with fine granularity, but also determine vulnerability type. There are similar differences between the correction studies that range from single token correction all the way to full code sections, sometimes as a single-step fix or sometimes as a collection of smaller steps with some form of correction-checking in between.

Takeaway 12: *There are significant differences in the quality of reporting of the dataset, preprocessing, and training and model details, which makes replication of a number of works difficult or impossible. Additionally, a relatively low number of papers have datasets and source code publicly available.*

We observe similar patterns in terms of dataset and tool availability from both correction and detection studies and will therefore address them jointly. 30 of the included studies make their dataset public, while slightly fewer –22– studies also provide their tool in an open-source fashion. All of the studies that provide the tool source code, also provide the dataset. On the other side, there is also a non-negligible number of papers that do not provide either –10 of the studies included.

When source material or the implementation is not available, we need to rely on the paper's own reporting of the implementation. Again, we find quite big disparities in the quality and level of detail. Since papers that provide both the dataset(s) and the tool source code can be considered highly detailed and therefore fully replicable by definition, we leave them out of further discussion.[1] Among the papers that provide the dataset, but no tool, we find the reporting is most commonly sufficient (mid score) for some level of replication, with some details or parameters unclear. We notice one outlier in each direction –[64] provides so little implementation detail that any level of reproduction is almost impossible, while [74] provides great level of implementation detail, despite not explicitly providing the source code. Perhaps unsurprisingly, we find the lowest level of detail reported by the papers that did not publicly provide the dataset or the tool.

3.3 Discussion

There are significant differences between the studies when it comes to supported languages, leading to different defect patterns and consequently representation choices. All of these seem to determine whether a tool will be able to automatically correct found bugs, or only detect them.

Arguably, the simplest defect type to catch is a syntactic one, with the vulnerabilities being the most challenging one. Seeing that most of the correction tools address the former, while detection tools largely address the latter, we can assume that an effective correction is more difficult to achieve. With several detection and correction tools targeting semantic defects, it is easy to assume that such defects lie in the middle in terms of difficulty.

As mentioned, we observe that patterns in source code representation seem to follow defect type patterns and, in turn, the detection or correction goals. We see that the defect correcting tools can achieve the intended goal through the use of simpler representations, while defect detecting tools use more advanced or combined representations. This further shows that tackling vulnerabilities and semantic defects is likely more challenging, so automatic correction on a large scale is not yet possible.

Sequence-of-tokens-based models are attractive because of their simplicity. Such an approach sees the source code as a flat sequence of elements, similar to words in natural language. They are especially useful for representing programs with syntactic defects in which constructing abstract syntax trees (AST) or control flow graphs is limited or not

[1] There seems to be no clear consensus on the definitions and the differences between replicability and reproducibility in machine learning setting, so we will use the two as synonyms.

possible due to severe syntax problems. The similarity to natural language makes it an attractive choice in sequence-to-sequence models where the goal is defect correction through the translation of the problematic sequence into a syntactically correct sequence.

Overall, token-level representation is the most popular choice for defect correction tools. The challenge of this approach is the selection of the appropriate granularity and range of tokens. Depending on the type of bug targeted, a model can benefit from simple stand-alone tokens, or from grouped and more structured representation (code gadgets, functions, or some other syntactic or semantic unit).

Syntactic representation considers the abstract syntax trees of the source code, allowing for a less flat view of the code. Such representations are of larger sizes and more difficult to construct, but can capture lexical and syntactic code properties. They are often combined with Recursive Neural Networks or Long Short Term Memory models. Their popularity lies mainly with the defect detection tools, especially semantic defect and vulnerability detection. While ASTs are good at capturing the structure of the code, they do not capture the semantics or large and complex pieces of code very well [72]. This is why ASTs are commonly supported by semantic representation capturing data and control flow information. The ability of graph models to capture more advanced semantic properties of code reflects itself in the use cases – they appear almost exclusively in tools targeting semantic defects and vulnerabilities.

Somewhat surprisingly, we observe a very unbalanced picture when it comes to the languages beyond C/C++ and Java. For example, we have found that C#, JavaScript, and Python lack the tools aimed at detecting and/or correcting security vulnerabilities. The possible reason we observed more studies aimed at C/C++ and Java, is that these languages are popular, well studied, and have large open databases of known defects (both bugs and security vulnerabilities). However, considering the ever growing popularity of C#, JavaScript, and Python, it becomes very important to develop the tools supporting them[1]. This also extends to other popular languages that did not appear in the study (e.g. PHP, Ruby etc.).

A look at the machine learning methods highlights the fact that the traditional ML approaches are more of a stepping stone towards a deep learning solution than solutions of their own. The reason likely lies in the fact that it is difficult to define the features (necessary for most other machine learning approaches) that will sufficiently capture the semantics of the program. The main benefit of deep learning is its ability to ingest the

[1] According to the PYPL index, Python is the most popular programming language of 2021 with JavaScript and C# taking third and fourth positions.

source code itself (after translation to appropriate representation) and create its own "features" to learn from.

Most commonly, we see that the complexity of the dataset used reflects the complexity of the task, which is to be expected. For example, one does not expect to find many syntax bugs in Linux kernel, nor does it make sense to look for complex vulnerabilities on a simple student program that does not even compile. The variety of datasets does however show the importance of selecting an appropriate dataset for the task and also highlights the vast difference in the desired behaviors of tools.

Similar to the datasets, it is useful to consider the full picture when discussing the tool output. It is not crucial to be given very specific output if the program consists of a dozen lines of code, whereas classifying a big project as vulnerable is next to useless if there is no way to determine where the problem lies. This is especially important for the practical application where the tools are supposed to be applied on the large number of real-world projects. Overall, the importance of smaller granularity and bigger precision is recognized and often highlighted throughout the works, with the trends moving towards the more specific tools.

Finally, we find big variety in terms of what the papers make public and in how well the process and details of the implementation are reported. This brings in the question reproducibility of the works. On the bright side, about half of the examined works are fully reproducible due to the full availability of the dataset and any related source code. Unfortunately, we also find that about one fourth of the papers suffers from low level of reporting and/or availability of core resource -the dataset. Seeing how machine learning is amid a reproducibility crisis [31], the issue should not be taken lightly.

3.4 Challenges and research directions

This review is motivated by the need to discover patterns in the rapidly evolving field of applications of machine learning in source code. Some of the challenges towards effective solutions (Table 3.4) include access to high quality training datasets, effective source code representation capable of semantic understanding, standardization, detection and correction across domains, catching application specific bugs (in regards to semantic defects) and expansion to more programming languages. We briefly elaborate on some of these challenges.

Future research in the domain should consider expansions to other commonly used programming languages. Among the languages included in the study, we have noticed a lack

Table 3.4: Key Takeaways

Finding	Obs.	Challenge
Missing detection or correction tools for some language-defect combinations	2, 7	Expand tools for all defect types to all languages
Variety of representation techniques, but struggling to capture deeper properties of code; over-simplistic embeddings	3, 4, 5	Advanced (semantic) representations and embeddings
Java and C/C++ most studied languages	6	Address Python, JavaScript etc.
Tool outputs not comparable	11	Formalize goals and measurements for tools, simplify output for developers
Vast differences in datasets	9	Collect and standardize high quality datasets across all defect types and languages
Problems with reproducibility	12	Normalize sharing of tool source code, datasets and increase quality/detail of reporting

of vulnerability detection approaches in Python, JavaScript and partly Java ss well. Additionally, more research is needed towards automatic correction of defects in Java and C/C++ programs. In particular, future research should work towards improving defect localization precision and a wider coverage of different defect types.

As mentioned, effective representation seems to be an active area of research, with more comprehensive approaches emerging, especially in the form of graph representations. A common go-to method for tools that do not invest into novel approaches seems to be the word2vec technique [43], which is primarily a simple token embedding technique. One then wonders why bother with the complex representations, just to flatten everything at the end of the pipe. We are already seeing (and expect to see) a further rise in similar, but more specialized __2vec-like vectorization techniques capable of capturing deeper properties of code.

Closely related to source code representation is the challenge of semantic understanding. A tool's ability to detect more complex semantic defects and vulnerabilities depends on its understanding of the source code. While syntax is finite, well defined, and therefore, easier to understand and capture, on the other hand, the semantics of programs are harder to capture. As more tools attempt to tackle complex types of defects, the need for advanced representation will further increase. In this respect, the graph-based representation capable

of capturing complex characteristics of the analysed programs (e.g. data or control flow) seems particular promising.

There is a big variety in terms of datasets, goals and testing. We believe the field would benefit from some degree of standardization, potentially in the form of a curated collection of open source datasets, together with some uniform goals for each of defect types along with a test suite and benchmarks. Since a tool's performance can heavily rely on the training data, stabilizing the dataset would allow more precise evaluation of the tool itself, rather than the training data. Formalization of goals (e.g. in terms of granularity, defect types) and results (speed, precision metrics) would also allow researchers in the field to get a clearer and more complete picture of the available tools. A less ambitious, but also useful goal would be standardization of terminology and measurements, which would allow comparisons when different datasets are used.

A relatively small number of tools working with unlabelled data points show that this is still a largely unexplored direction. It comes with the challenge of unsupervised learning, but at the same time unlocks access to large datasets of unlabeled corpora, eliminating the need for synthetic bug introduction or for manual labeling.

Finally, a relatively high number of paper that cannot be fully or at all reproduced. The field would benefit from stricter norms about dataset and source code sharing and –when that is not possible for whatever reason– detailed reporting of the full pipeline.

4

Design and implementation

The system implemented and evaluated in this thesis generally follows the pipeline initially introduced by [37] in their seminal paper. As part of the thesis, two variations of the proposed pipeline will be discussed. First we present an attempt to replicate the original VulDeePecker pipeline for detection of C/C++ vulnerabilities by staying close to the original design. Second, we follow up with an adaptation of said system to allow detection of Java vulnerabilities. The coming section discusses the general design of the system's pipeline and the shared implementation choices; we leave the discussion of version-specific implementation details, relevant changes and experimental treatments to the respective chapters.

4.1 Architecture of the vulnerability detection system

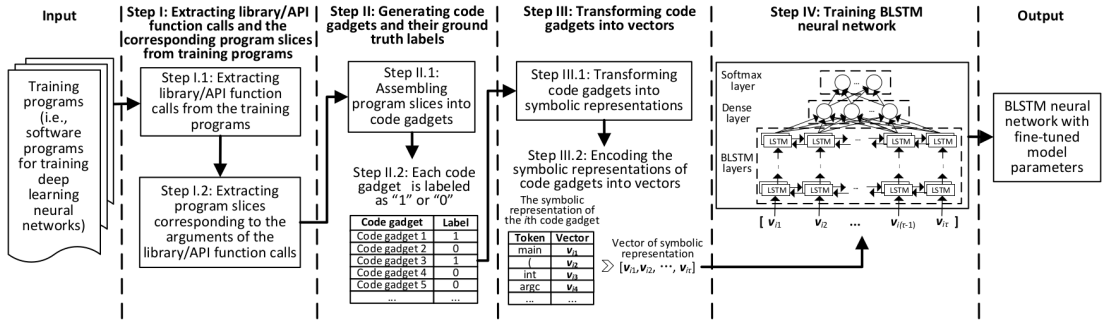

Figure 4.1: Learning phase of VulDeePecker as pictured in [37]

Conceptually, the vulnerability detection system operates in two phases: the model is first trained in the *learning phase*; after that it can be used to predict vulnerabilities in the *detection phase*. In practice the two phases share most of the pipeline, but with different

inputs and outputs. When the pipeline is used for the model training, the (training portion of) dataset is used as input, with the trained model itself being the output. Once trained, the input is a single unseen source code file which is eventually classified as vulnerable or non-vulnerable. Figure 4.1 pictures the general pipeline used for both phases.

The pipeline begins with the input of one (in detection phase) or a number (in learning phase) of program slices in a set programming language. The slices are then pre-processed and transformed from plain text format into vectors to be fed into the neural network (Steps II and III). The source code preparation and pre-processing steps proposed are novel in the vulnerability detection domain, yet somewhat straightforward. The plain text source code is first transformed into *gadgets*, which are labeled as vulnerable or non-vulnerable. A gadget is composed of "a number of (not necessarily consecutive) lines of code, which are semantically related to each other in terms of data dependency or control dependency" [37].

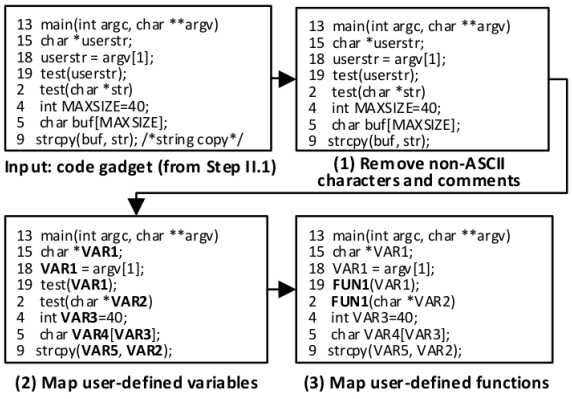

Figure 4.2: Illustration of Step III.1: transforming code gadgets into their symbolic representations as pictured in [37]

The gadgets are then cleaned up and transformed into symbolic representations. This includes removing any comments and non-ASCII characters as well as mapping of the user-defined variables and functions to symbolic names (eg. "VAR1", "FUN1", "FUN2"). Finally, the cleaned gadgets are transformed into vectors. This step includes the tokenization of the gadgets (as described in section 2.3), encoding of a number of said tokens using *word2vec* technique [43] and padding of corresponding vectors with zeroes to ensure uniform vector lengths.

Finally, the vectors are passed as input to a Bidirectional Long-Short Term Memory network for training or prediction. The system can predict the label at gadget-granularity.

4.1.1 Adaptation between languages

In principle the presented pipeline can act in a language-agnostic manner. This means than it should be able to predict vulnerabilities in any language, provided it was trained on an appropriate dataset. While the core of the system does not require many changes, there are parts of the pipeline that should be adapted to each language in order to allow successful training and produce useful results.

First and most importantly, an appropriate dataset of the desired language's source code needs to be available for the system to be trained on (Input and Step I). When constructing such a dataset, the language's structure and specifics need to be considered, so that appropriate code samples or slices can be extracted.

In addition to that, the input of the pipeline and the pre-processing may need adapting, depending on the language's syntax and semantics. More specifically, different languages will contain different keywords that should be preserved (not transformed into symbolic representations), data structures or concepts that do not appear in others (eg. object oriented languages), etc. While the system can still operate with some generic syntax set, adaptations are needed for optimal performance.

Finally, a new language (or even a different set of vulnerabilities in the same language) will likely require a different set of parameters in order to perform optimally. While this is most obvious when dealing with the model itself (Step IV), other parts of the pipeline might require adaptation as well (eg. gadget sizes, vectorization).

4.2 Implementation and overview of experimental setup

As the starting point of the implementation a public VulDeePecker replication attempt [65] was used, providing the ground work and facilitating the pre-processing stages of the replication. The pre-processing steps are relatively straightforward and were implemented as stated in the design section. Additionally, we fix parameters that were clearly described in [37]. The dropout rate is set to 0.5. The training is performed using minibatch stochastic gradient descent together with ADAMAX, using batch size of 64.

However, there are several parameters or choices that were either not addressed or insufficiently precise. We will, for the remainder of this thesis, refer to all of those choices as parameters. Such parameters relate to dataset balancing approaches, specific neural network hyper-parameters, activation, loss function and number of hidden layers in the final results. They will be explored and evaluated as part of the experimental treatments. In their respective chapters they are clearly stated and explained.

The system was implemented in Python, supported by Keras, Scikit and Pandas libraries. The full code for the system implemented in this thesis and the accompanying datasets is available on https://github.com/tmv200/ml4code-Java. The experiments were performed on a machine with NVIDIA GeForce GTX 1650 Ti GPU and Intel i7-10750H CPU operating at 2.60GHz. Depending on the treatment and parameter configuration, one model requires 5-60 minutes to train. The training is performed on 80% of the dataset, with the remaining 20% left for testing.

The reported results are based on 10 repetitions of the experiment. We report precision, true positive rate (sensitivity) and true negative rate (specificity). In addition to that -when relevant for comparison-, we report the same metrics that are available for VulDeePecker: false positive rate (FPR), false negative rate (FNR) and F1 score.

4.2.1 Datasets

	VulDeePecker	project KB	Juliet
Sample size (gadgets)	61,638	41,809	166,443
Vulnerability types (CWEs)	2	71*	112
Frequency of vulnerabilities	320-520	1-39	1-7015
Code properties	synthetic & real	real	synthetic & real

*Translated from their respective CVEs according to https://www.cvedetails.com/

Table 4.1: Overview of the datasets used

The replication of VulDeePecker and the related exploration of its parameters was performed using the same dataset of C/C++ source code that was created for and used in the original paper. It includes two types of vulnerabilities: CWE-399 (resource management error vulnerabilities) and CWE-119 (buffer error vulnerabilities). The dataset is a combination of two public datasets: SARD [48] and NVD [47], which include a combination of synthetic examples and real source code. It contains 520 programs with buffer error vulnerabilities and 320 resource management vulnerabilities. The programs produce 61,638 code gadgets.

For adaptation to Java, two datasets were used and separately evaluated: project KB [52] and Juliet [49]. Project KB dataset is part of a project aimed at facilitating vulnerability research and contains manually curated vulnerabilities found in real source code. It contains 71 different vulnerability types (CWEs), appearing between 1 and 39 times each, with a mean of 6 appearances per CWE. The most commonly appearing vulnerabilities

include: CWE-22 (39 appearances), CWE-611 (39), CWE-20 (34), CWE-502 (33) and CWE-79 (29). The programs included in the dataset produce 41,809 code gadgets.

Juliet, on the other hand is a popular dataset with mostly synthetic or academic vulnerability examples. In contains 112 distinct vulnerability types, appearing between 1 and 7015 times each, with a mean of 412 appearances per CWE. The most commonly appearing vulnerabilities include: CWE-190 (7015 appearances), CWE-191 (5612), CWE-129 (4392), CWE-89 (3660) and CWE-369 (3050). Once processed, the dataset produces 166,443 code gadgets.

The two Java datasets differ in a number of ways, which will allow us to examine the effects of different dataset properties on vulnerability detection in Java. Namely, project KB is a high quality dataset of real vulnerabilities, which comes at the cost of sample size. On the other hand, Juliet leads significantly in quantity of data, but pays in data realism, which may significantly affect external validity of the results. For better visualization, we include a vulnerable code sample from both datasets and a complete CWE list with number of appearances in the Appendix.

5

Vulnerability detection in C/C++ source code

5.1 Experimental treatments

The replication of VulDeePecker was implemented as follows: the stages, choices and parameters that are clearly specified in reference paper were fixed (as described in previous section), while the remainder was subject to testing. For the latter, the options, approaches and parameters that are commonly used in the deep learning community (as suggested in the paper) were considered. The following chapter presents an exploration of these choices in order to understand if and how they affect the performance of the tool.

The number of hidden nodes at each layer of the BLSTM network corresponds to the length of the padded vectors and is fixed to 300, with 50 tokens per vector. The training is performed in 4 epochs. While activation functions were not explicitly stated in the paper, its appendix contains mathematical discussion of the BLSTM network in which tanh and sigmoid functions are used as the activation and recurrent activation functions respectively. The two functions also most commonly appear as the default choice in the deep learning community. For this reasons, the two are used, with no alternative functions tested.

The set of unknown variables was narrowed down after some initial testing to identify the choices, appropriate for the design specified. The final set of choices that are systematically evaluated includes:

- *number of BLSTM layers*: we evaluate models with 1, 2, 3 and 4 layers (mentioned as best performing in the original paper, but unspecified what is the final configuration)

- *sampling*[1]: we evaluate models trained on the unchanged dataset and randomly undersampled negative examples to rebalance the dataset (50%[2] or 75% of the initial negative examples).
- *reweighting of classes*: we evaluate models with and without reweighting (as an alternative or as support to the resampling approach)
- *loss function*: we evaluate models with binary crossentropy and categorical crossentropy

In order to identify the factors that most significantly affect the results, 48 different models with all possible combinations of the 4 mentioned variables were trained and tested. Any additional parameters that could significantly affect the outcome of the experiments, but were clearly stated in the original paper are fixed (we leave the exploration and experiments with them for Java adaptation).

5.2 Parameter exploration results

5.2.1 Overview

Figure 5.1: Overview of model performances

Figure 5.1 offers an overview of the performance for all 48 tested model configurations. Overall, we see that the sensitivity (true positive rate) of models range from 65% up to more than 95%. At the same time, false positive rates range from 5% up to 40%.

We can also notice the typical "curve" in model performance, highlighting the trade-off between the true positive and true negative rates. In general, the models that achieve high sensitivity do so by more generously classifying piece of code as positive (vulnerable), resulting in higher false positive rate and vice versa. Table 5.1 confirms the observed relationships between the

[1]While not strictly a parameter of the model, it is a technique commonly used in unbalanced datasets, capable of affecting the performance and should therefore be tested.

[2]This produces approximately the same number of positive and negative examples.

metrics. In particular, we observe negative correlation between sensitivity and the remaining two metrics. For this reason, there is no single model that performs best in all metrics at the same time.

	Precision	Sensitivity	Specificity
Precision	1.000	-	-
Sensitivity	-0.786	1.000	-
Specificity	0.688	-0.927	1.000

Table 5.1: Correlations between performance metrics

For a more detailed picture of performance we can look at Figure 5.2. which shows the precision, true positive rate and true negative rate for each of the 48 tested models.[1] Each model is represented by a point on the graph, with different shape, fill and color representing resampling, reweighting and loss function, respectively. The observations are ordered by number of layers, separated by the dotted lines –first 12 models were built with 1 layer, next 12 with 2 etc.

In general, we can see noticeable differences in model performance across all three presented metrics. In terms of precision and specificity, we see that models without reweighting perform better. On the other hand, models with reweighting in general achieve the highest sensitivity. We can also see a slight dip in performance of the majority of models using 4 layers.

While some of the models achieve comparable scores for individual metrics, no single model does so across all metrics. It seems that the models that achieve the most consistent scores across all three metrics include some undersampling, but no reweighting (hollow triangles and some circles).

In order to identify some more nuanced patterns, we need to look more closely at each individual parameter. While it is unlikely that the findings can be generalized outside VulDeePecker or potentially even the current replication, they illustrate the volatility of the models as well as the importance of parameter selection and their precise reporting. In the coming section we further elaborate on each of the parameters and their effects on the overall performance.

[1] For brevity, we limit our discussion to those three metrics. For a complete presentation of all recorded metrics for all 48 models see Table B1 in the Appendix.

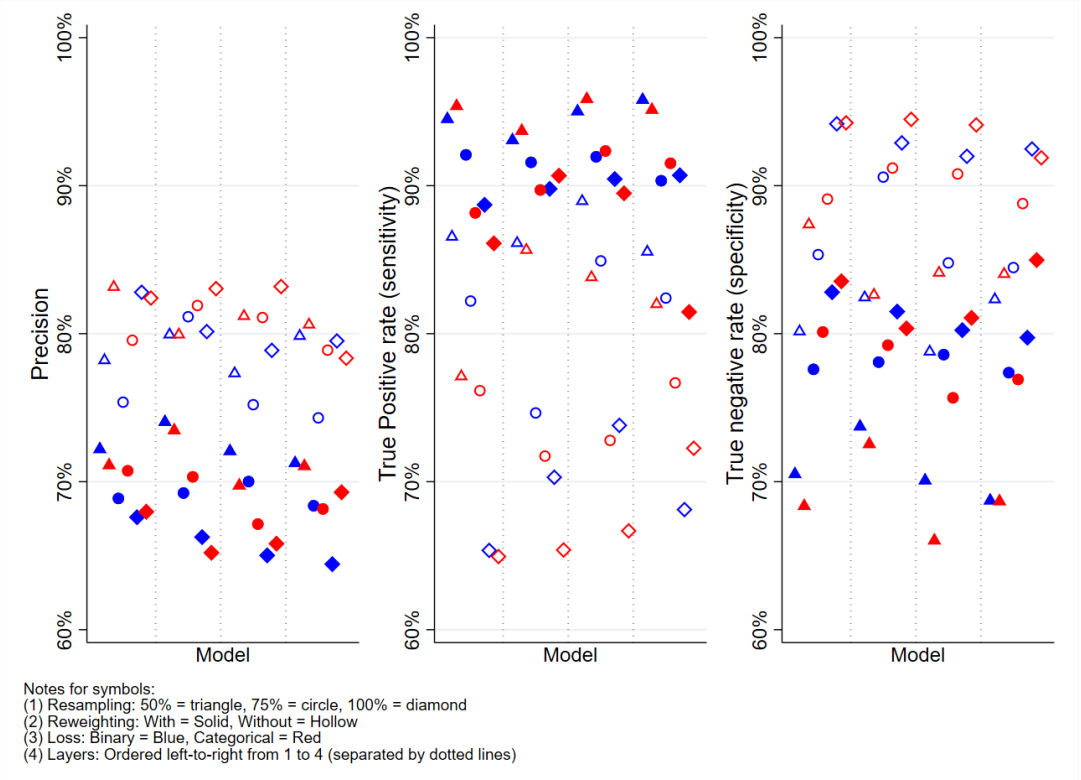

Figure 5.2: Precision, sensitivity and specificity for tested models

5.2.2 Effect of parameters

In this subsection we look at the effect of each parameter on the performance. First, we tabulate the raw averages for each level of the parameter and visualize the differences in boxplots. Next, we formally test whether each of the parameters has a statistically significant effect on each of the performance metrics. We do so by using an ANOVA test. ANOVA is a generalization of the t-test and it allows us to compare the means of more than two groups. The null hypothesis is that the means are all equal to each other. A p-value larger than 0.05 implies significant differences. We present the concise results in the coming section, for full overview see Table C1 in the Appendix. Finally, whenever the ANOVA test rejects the null hypothesis, it is interesting to see which of the means differ from the rest. To find out, we perform post-hoc pairwise comparisons while using the Bonferroni correction to control for multiple hypothesis testing.

The effect of number of layers

On average none of the metrics seem to be largely affected by the number of layers. We see the precision score fluctuating between 73.7% and 75.4%, true positive rate between

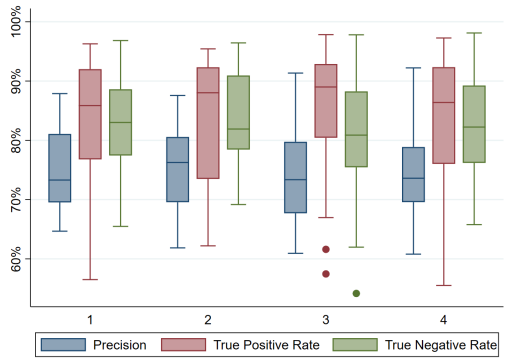

Figure 5.3: Effect of layers

Layers	#Runs	P		TPR		TNR	
		μ	σ	μ	σ	μ	σ
1	120	74.9	6.4	83.1	10.9	82.8	8.3
2	120	75.4	6.7	83.5	10.3	83.3	7.5
3	120	73.9	6.9	85.5	9.7	81.4	8.8
4	120	73.7	6.1	84.3	9.7	81.7	8.1

Table 5.2: Mean and standard deviation across layers

83.1% and 85.5%, and true negative rate between 81.4% and 83.3%. We can observe a small drop of precision and true negative rate scores and a slight growth of true positive rate scores with the higher number of layers. However, ANOVA tests find no statistically significant difference between the layer performances (p-values between 0.405 and 0.528).

The effect of dataset resampling

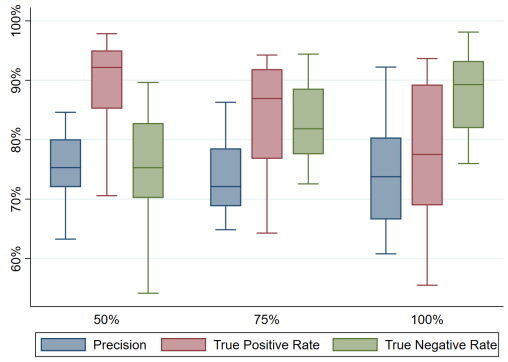

Figure 5.4: Effect of resampling

Layers	#Runs	P		TPR		TNR	
		μ	σ	μ	σ	μ	σ
50%	160	76.0	4.8	89.6	6.5	76.3	7.7
75%	160	73.8	5.8	84.3	8.5	83.0	6.2
100%	160	73.7	8.3	78.4	11.5	87.5	6.4

Table 5.3: Mean and standard deviation across resampling option

By resampling the dataset, we increase the ratio of vulnerable over non-vulnerable examples. The benefit, but also the risk of this approach is that the algorithm may become more of excessively likely to predict positive. This would translate to higher true positive rate and lower true negative rate. Our results confirm this conjecture. Without resampling, precision is 73.7%, true positive rate 78.4% and true negative rate 87.5%. With moderate resampling (75% of the original non-vulnerable samples), we achieve precision of 73.8%, true positive rate of 84.3% and true negative rate of 83.0%. With more aggressive undersampling, we achieve precision of 76.0%, true positive rate of 89.6% and true negative rate of 76.3%. Again, there is a trade-off between true positive rate and true negative rate.

The ANOVA tests show a very narrow effect of resampling on precision with only 2.5% of the total variance in precision explained by resampling. We find significant and substantial effects of resampling on true positive rate and true negative rate with 20.5% and 32.1% of the variance explained respectively. Pairwise comparisons reveal that means vary across all resampling choices. The raw averages, the boxplots, the ANOVA tests, and the pairwise comparisons all show that resampling heavily affects performance.

The effect of class reweighting

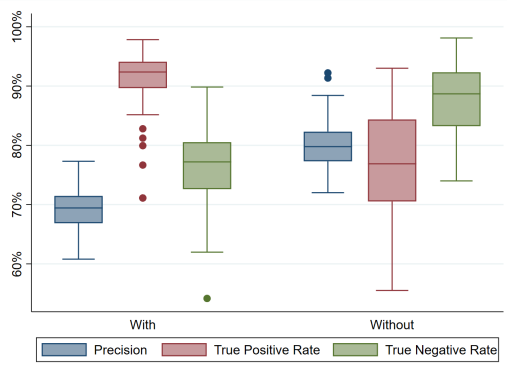

Figure 5.5: Effect of reweighting

Reweighting	#Runs	P		TPR		TNR	
		μ	σ	μ	σ	μ	σ
Without	240	79.8	4.1	76.8	8.9	88.0	5.5
With	240	69.1	3.4	91.4	4.4	76.5	6.1

Table 5.4: Mean and standard deviation across reweighting option

Instead of manually changing the dataset, we can feed the full dataset to the algorithm and allow it to reweight classes during the training phase. Without reweighting, we achieve average precision of 79.8%, true positive rate of 76.8% and true negative rate of 88.0%. With reweighting, precision is 69.1%, true positive rate 91.4% and true negative rate 76.5%. It seems that reweighting has an effect on all three metrics. ANOVA tests confirm the observations. Reweighting explains a large part of the variance in performance –between 49.9% and 67.2% depending on which metric we focus on.

The effect of loss function

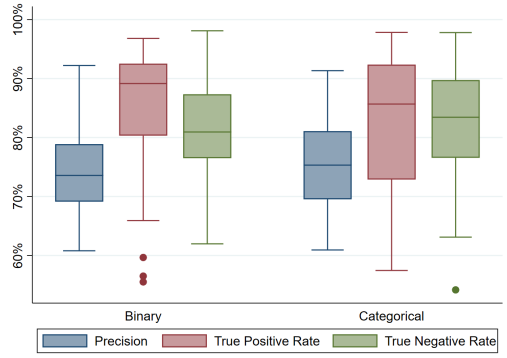

Figure 5.6: Effect of loss functions

Layers	#Runs	P		TPR		TNR	
		μ	σ	μ	σ	μ	σ
Bin	240	73.8	6.0	85.5	9.4	81.6	7.5
Cat	240	75.1	6.9	82.7	10.7	82.9	8.8

Table 5.5: Performance across loss functions

With binary crossentropy, we achieve average precision of 73.8%, true positive rate of 85.5% and true negative rate of 81.6%. With categorical crossentropy, the average precision is 75.1%, true positive rate 82.7% and true negative rate 82.9%. We do not observe significantly different performances between the two loss functions. ANOVA results confirm the limited role of the loss function. We find a significant effect only on true positive rate, but even there only 1.9% of the variance can be explained by the choice of loss function.

Interaction effects of parameters

It is worth noting that the standard deviations of certain metrics are noticeably larger in certain models than others (eg. true positive rate for resampling). A potential explanation for this could be the -both positive or negative- effects that the remaining parameters have in combination with resampling. Motivated by this conjecture, we looked at all six possible interactions effects between our parameters. We did so via two-way ANOVA tests. The tests revealed a significant interaction only between resampling and reweighting.

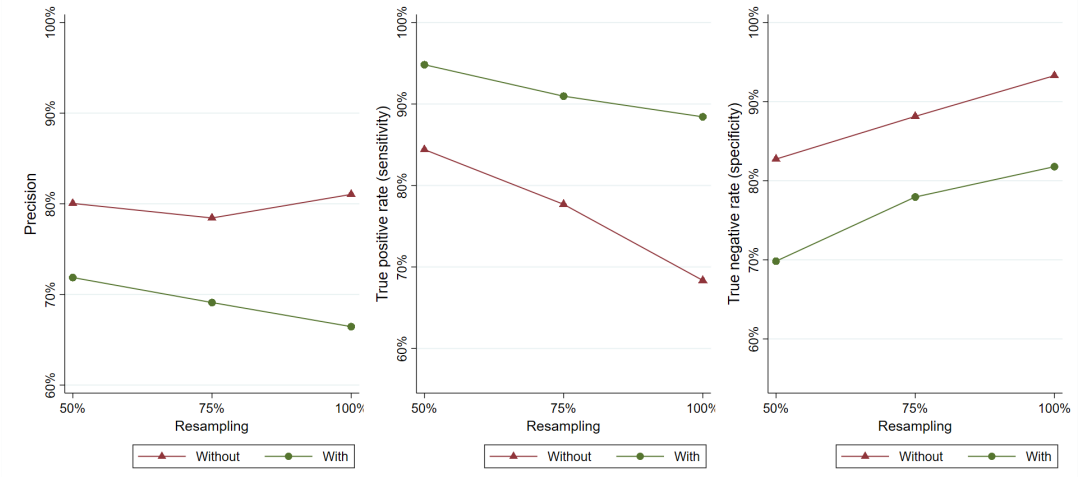

Figure 5.7: Interaction effects between resampling and reweighting

The effect is visualized in Figure 5.7. The graph shows the changes in performance across different resamplings. In each graph, we plot separately the changes with and without reweighting. Parallel lines imply no interaction effects. We clearly see this is not the case.

The strongest interaction effect is observed in precision. Using resampling *increases* precision if used together with reweighting, whereas precision is *decreased* if resampling is introduced without reweighting. The effects on true positive rate and true negative rate are smaller in magnitude. The direction of the effect does not change with or without reweighting. As noted in previous subsection, resampling overall increases true positive rate

and decreases true negative rate. The interaction with reweighting is the rate at which it happens. To illustrate, the gain in true positive rate is much faster without reweighting as it is without.

Takeaway 1: *The parameters that affect the performance the most are resampling and reweighting, while number of layers and loss function do not have a significant effect. Overall, we see positive correlation between precision and specificity and their negative correlation with sensitivity.*

5.2.3 Comparison with VulDeePecker

In order to best compare the tested models with VulDeePecker, we look at the metrics reported in the original paper: false positive rate (FPR), false negative rate (FNR), true positive rate (TPR), precision and F1 score. To an extent, this allows us to reason about the parameters potentially used in the original paper.

In terms of F1 score, the tested models range from 72% on the lower end up to 83% at the top. We observe that both FPR and FNR fluctuate between 5% and 35%. As mentioned, TPR moves between 55.5% and 97.8% and precision between 60.8% and 92.2%. For comparison, the VulDeePecker model, trained and tested on the same dataset, performed with FPR of 5.1%, FNR of 16.1%, TPR of 83.9%, precision of 86.9% and F1 score of 85.4%. While some of the models achieve comparable or even higher scores than VulDeePecker for individual metrics, no single model does so across all metrics.

According to the findings from the previous section, the two parameters that should be scrutinized are resampling of the dataset and class reweighting. We find that models that use reweighting tend to outperform VulDeePecker in terms of true positive rate, paying the price with higher false positive rate. This suggest the VulDeePecker might not use reweighting. On the other hand, we find that no dataset resampling roughly matches VulDeePecker's false positive rate, but does not match up in terms of F1 score. Instead, we find that resampling leads to metrics that are closest to VulDeePecker's results across the board. Therefore, it is possible that VulDeePecker uses some level of resampling.

To summarize, our results most closely match VulDeePecker's when undersampling of non-vulnerable examples and no class reweighting is used during training; the pattern seems to be consistent regardless of the number of layers or the loss function used.

For illustration purposes, we present one representative (handpicked) model with the described characteristic. Model #16 uses 2 layers, categorical crossentropy, no reweighting and was trained on resampled dataset where only 50% of the negative samples were used.

We record the following average performance: FPR of 17.4%, FNR of 14.3%, TPR of 85.5%, P of 80.0% and F1 of 82.7%. For comparison, VulDeePecker reported FPR of 5.1%, FNR of 16.1%, TPR of 83.9%, P of 86.9% and F1 of 85.4%. For a full overview of results, see Table B1 in the Appendix.

Takeaway 2: *Overall, the replication of VulDeePecker is possible to an extent, but the results cannot be reproduced fully. The choice of the model parameters and dataset details have a significant effect on the performance of the algorithm and should be disclosed in more detail.*

Takeaway 3: *The models that perform closest to VulDeePecker include some undersampling of negative samples and no reweighting. The choice of loss function and number of layers do not play a role in our testing.*

6

Vulnerability detection in Java source code

6.1 Experimental treatments

The Java adaptation follows the design specified in Chapter 4, with the input stages of the pipeline modified to support Java syntax. The remainder of the pipeline remains largely unchanged, apart from the parameters relevant for the experimentation. We fix the number of layers to 3 and use binary crossentropy for loss computation. We fix (in the current stage) the number of tokens per vector to 50 tokens. The remaining implementation details remain as described in previous sections. The exploration of parameters was performed using two dataset of Java source code: project KB and Juliet.

We use a number of treatments to explore two main questions: I) *Can we implement a Java vulnerability detection system and if so, how does it compare to the baseline (C detection system)*; and II) *How do different datasets perform and why*?

To answer the first question, we repeat the parameter exploration from Chapter 5, but on the Java datasets. Initial testing shows that the same parameters influence the results; we therefore limit the parameter exploration to sampling and class reweighting. To account for the potential differences in Java (compared to C/C++), we explore training for either 4 and 10 epochs. The final set of choices that are systematically evaluated includes:

- *sampling*: we evaluate models trained on the unchanged dataset and randomly undersampled negative examples to rebalance the dataset (25%[1], 50%[2] or 75% of the

[1]This produces approximately the same number of positive and negative examples for project KB (the only dataset tested with this parameter).

[2]This produces approximately the same number of positive and negative examples for Juliet.

initial negative examples).

- *reweighting of classes*: we evaluate models with and without reweighting (as an alternative or as support to the resampling approach)
- *training duration*: we evaluate models trained over 4 and 10 epochs

To explore the second question we vary 3 additional parameters. Each serves to explore a different potential underlying cause for differences in performance. Note that parallel (equivalent) experiments for both datasets are not always possible (eg. we cannot extend the sample size of the smaller dataset to match the bigger, we cannot experiment with CWE frequencies that are already very low) or outside of the scope of this thesis -i.e. we do not aim to perfectly optimize the performance of the better performing dataset, but rather explore the general feasibility of Java vulnerability detection and potential reasons for differences in performance. The set of parameters varied together with the dataset on which the testing was performed include:

- *sample size* [Juliet]: we evaluate models trained on the unchanged dataset and on the dataset reduced to match the sample size of project KB.
- *CWE frequency threshold* [Juliet]: we evaluate models trained on the dataset of all CWES, CWEs that appear more than 100 times, and more than 1000 times.
- *Number of tokens per vector* [project KB]: we evaluate models trained on vectors containing 50, 75 and 100 tokens.

Given the differences between the datasets, we believe the three parameters chosen should provide some evidence as to why the datasets perform the way they do. During the testing of each parameter, we fix the remaining ones to isolate the effects of the parameter in question.

6.2 Evaluation

6.2.1 Overview and parameter exploration

Figure 6.1 offers an overview of the performance for all 28 tested model configurations. The models trained on project KB dataset are pictured in blue, while models trained on Juliet are pictured in red. We can observe two distinct curves suggested by each of the dataset's models.

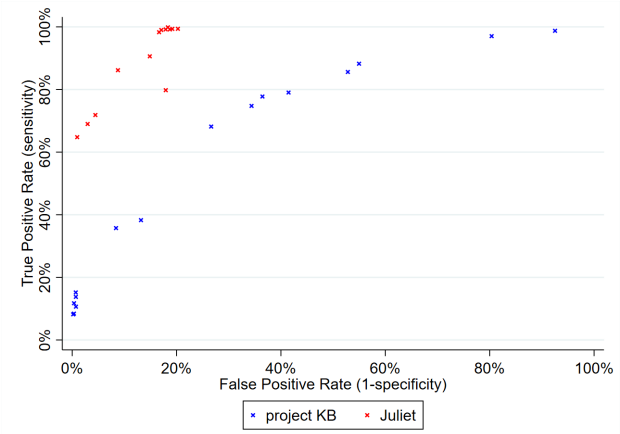

Figure 6.1: Overview of model performances

It is obvious straight away that the system trained on Juliet performs much better. Looking more closely at only Juliet trained models, we see that the true positive rates remain above 60%, with false positive rates up to about 20%. On the other hand, the true positive rates and false false positive rates of project KB trained models span across the entirety of the spectrum.

Similar to C/C++ models, we notice the typical "curve" in model performances, highlighting the trade-off between the true positive and true negative rates. Table 6.1 confirms the observed relationships between the metrics. Again, we observe negative correlation between sensitivity and the remaining two metrics, resulting in no single model that performs best across all metrics.

	Prec	Sen	Spec
Precision	1.000	-	-
Sensitivity	-0.934	1.000	-
Specificity	0.765	-0.929	1.000

	Prec	Sen	Spec
Precision	1.000	-	-
Sensitivity	-0.863	1.000	-
Specificity	0.858	-0.982	1.000

Table 6.1: Correlations between performance metrics for project KB (left) and Juliet (right)

For a more detailed picture of performance we can look at Figure 6.2 which shows the precision, true positive rate and true negative rate for each of the 28 tested models.[1] Each model is represented by a point on the graph, with different shape, fill and color representing resampling, reweighting and number of epochs, respectively. The models, trained on different datasets are separated by a vertical dotted line, with project KB trained models on the left and Juliet trained models on the right of each graph.

Again, we can see noticeable differences in model performance across all three presented metrics. The most obvious distinction comes from the dataset used in training. Models trained on project KB almost exclusively underperform or at best match their parametric counterparts trained on Juliet. We further explore the differences and potential reasons

[1] For a complete presentation of all recorded metrics for all 28 models see Table B2 in the Appendix.

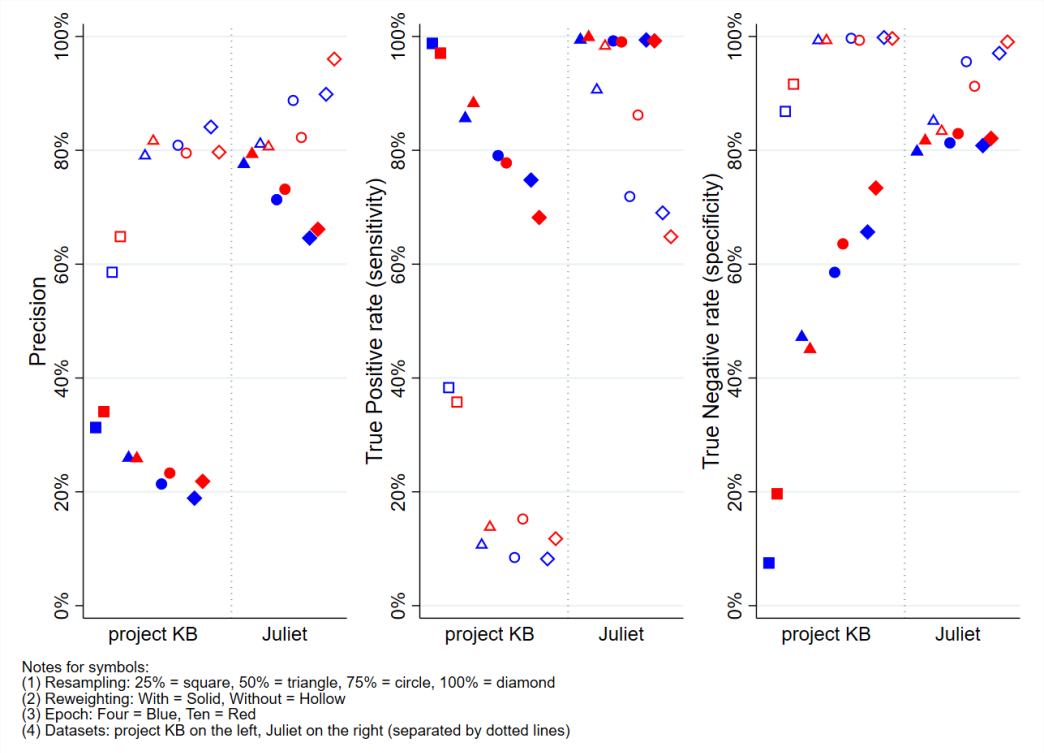

Figure 6.2: Precision, sensitivity and specificity for tested models

between the two datasets in the following section.

Takeaway 4: *The performance of a system for Java vulnerability detection matches or outperforms the baseline set for C/C++ vulnerabilities, but only with one of the two datasets tested.*

Takeaway 5: *The system trained on Juliet dataset significantly outperforms the system trained on project KB dataset.*

The effect of epochs

When looking at Figure 6.2, Table 6.3 and Table 6.4, we can see that among pairs of models with same parameters, the ones trained over 10 epochs more commonly outperform the ones trained over 4 epochs. The pattern is more evident when looking at precision and true negative rate. In terms of the true positive rate, the effect is actually reversed (on project KB trained models) or virtually indistinguishable (on Juliet trained models). Despite the direction of the effect based on visual inspection of the graphs, the differences are on aggregate not statistically significant (see Table C8 in Appendix).

The effect of resampling and reweighting

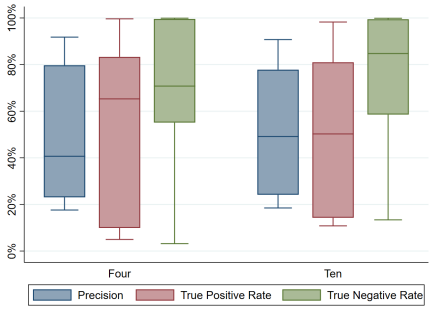

Figure 6.3: Effect of epochs on project KB trained models

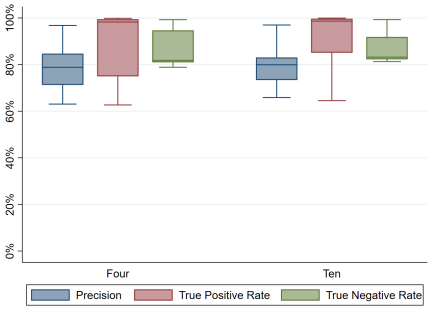

Figure 6.4: Effect of epochs on Juliet trained models

		PR		TPR		TNR	
Epochs	#Runs	μ	σ	μ	σ	μ	σ
4	80	50.0	27.6	50.5	36.6	70.6	31.3
10	80	51.3	26.5	51.0	34.1	73.9	28.6

		PR		TPR		TNR	
Epoch	#Runs	μ	σ	μ	σ	μ	σ
4	80	78.9	9.6	88.3	13.7	86.6	7.4
10	80	79.6	9.3	91.3	12.9	86.7	6.5

Table 6.2: Mean and standard deviation across epochs for project KB (left) and Juliet (right)

In terms of the remaining two parameters (resampling and reweighting), we notice similar patterns as in C/C++ system. More specifically, we find both to be statistically significant, with more aggressive undersampling and use of reweighting positively affecting true positive rate, while reducing precision and true negative rate. Due to the similarity with the observations in Chapter 5, we refrain from more detailed analysis; we instead provide additional graphical material in the Appendix.

Takeaway 6: *Reweighting and resampling have similar effect on performance of Java vulnerability detection system as they do on C/C++. Training the models for 10 epochs rather than 4, does not improve the performance.*

6.2.2 Dataset exploration

Seeing that there are large differences in performances of models trained on different datasets, we further explore some potential reasons. The conjectures tested are aligned with the datasets' underlying properties and the differences stemming from them, namely the sample size, frequencies of CWE occurrences and realism of the source code.

The effect of sample size

To explore whether the performance of project KB is worse due to the smaller sample size, we compare two treatments performed with Juliet: one with full sample size and one where the sample size is reduced to match the size of project KB.

		P		TPR		TNR	
Size	#Runs	μ	σ	μ	σ	μ	σ
Full	60	79.6	9.3	91.3	12.9	86.7	6.5
Reduced*	60	80.0	10.0	89.3	13.9	86.9	7.2

*Reduced size equals the full size of project KB dataset

Table 6.3: Mean and standard deviation across sample sizes for Juliet dataset

Table 6.3 shows the average performance and standard deviation for both versions, averaged over the remaining parameters. We do not see significant differences between the performance of the two on aggregate level. The observation is confirmed by comparing the means of two treatments using ANOVA test. With p-values ranging from 0.57 to 0.88, our experiments found no evidence for sample size affecting performance for either of the three metrics.

Due to the high standard deviation within each of the metrics when averaging the models (caused by resampling and reweighting variation), we perform an additional check on a randomly selected model with fixed resampling (75%) and reweighting (used). Interestingly, this way we find more evidence to support some effect of sample size (p=0.03 for precision, 0.67 for sensitivity and 0.01 for specificity). This suggests that higher sample size gives higher precision and true negative rate, while not losing true positive rate. While the effect can be picked up for a randomly selected model, it is not strong enough to surface on the aggregate level.

The effect of frequency of vulnerabilities

To explore if the performance depends on how commonly some vulnerability appears in the dataset, we perform 3 experiments on three variants of Juliet dataset. We train models using the full dataset (i.e. where certain vulnerabilities appearing only once or a handful of times), and the datasets where vulnerabilities that appear fewer than 100 or 1000 times were removed.

		P		TPR		TNR	
Threshold	#Runs	μ	σ	μ	σ	μ	σ
1	90	78.6	9.8	89.3	13.8	86.0	7.6
100	90	78.2	8.9	89.5	12.3	86.3	6.7
1000	90	76.9	8.6	90.4	13.6	85.6	7.1

Table 6.4: Mean and standard deviation across CWE threshold options for Juliet dataset

Similar to sample size, we do not pick up any significance on aggregate level (p-values

between 0.77 and 0.99). Interestingly, the same holds when testing the same single model configuration as before (p-values between 0.15 and 0.49). Having found no effect on either, we can reject any effect different vulnerability frequencies might have on performance.

The effect of number of tokens per vector

Lastly, we experiment with the number of tokens per vector. Given the bad performance of the project KB dataset, we want to explore if widening the window through which the system sees the code gadgets will improve its performance. We compare the default choice used throughout the thesis (50 tokens) with two additional options (75 and 100 tokens).

		P		TPR		TNR	
Tokens	#Runs	μ	σ	μ	σ	μ	σ
50	90	51.3	26.5	51.0	34.1	73.9	28.6
75	90	51.3	25.8	52.1	33.1	74.9	26.9
100	90	51.2	25.2	52.3	33.4	75.7	26.2

Table 6.5: Mean and standard deviation across different token size for project KB dataset

We see from Table 6.5 that the direction of the effect is in favor of higher number of tokens –there is a slight increase in sensitivity and specificity and no change in precision. However, the effect is not strong enough to be picked up by ANOVA testing on aggregate level, nor individual model level. We can therefore conclude that the increase in number of tokens per vector does not improve the performance of project KB trained models.

Takeaway 7: *The difference in sample size, frequency of vulnerabilities or number of tokens per vector do not explain the vast difference in performance between project KB and Juliet trained models.*

7

Discussion

Our experimentation with the parameters of VulDeePecker replication and Java adaptation showed significant differences between the models tested. This is interesting for a number of reasons.

First, we find that changing the parameters can significantly affect the performance of the system. More specifically, we see that changing dataset resampling and model reweighting options is an effective way to fine-tune the balance between the sensitivity and specificity. With the ratio between vulnerable and non-vulnerable examples heavily skewed towards non-vulnerable examples, stronger undersampling will lead to data with a more balanced number of both examples. This, in turn, makes the system more sensitive, but less specific. Reweighting aims to achieve a similar goal by letting the model adapt the weights of the classes, removing or reducing the need for dataset modifications. Models with reweighting show similar patterns –higher sensitivity, but lower specificity.

Having the means to affect the performance of the model, we should consider where on the spectrum of possibilities we want the final system to be. As is typical, the answer is case-specific. A more security-sensitive user might want to sacrifice the specificity to ensure that no vulnerability escapes, while a user with less resources might choose specificity over sensitivity. The answer –in part– also depends on how common vulnerabilities are in a typical project. In short, it is important to leave some choice of the final metrics and output to the user.

Lastly –and not only related to VulDeePecker–, the fact that the exploration of parameters was necessary highlights the need for more clear and detailed reporting, and sharing of source code and datasets of studies. While the dataset availability already places the original study among the more replicable ones, we could not perfectly replicate the results from the description and details given. This ties in with the current reproducibility crisis

of machine learning research [31, 50] many are pointing out. Significant efforts have been made to improve the situation (eg. OpenReview, PapersWithCode), but there is still much to improve.

In terms of the adaptation of the system to detect vulnerabilities in Java, we find promising results when training with Juliet. We were able to adapt the general pipeline and achieve performance equal to or in some cases better than VulDeePecker in C/C++. It is worth mentioning that the tests were performed on a dataset with many more vulnerability types compared to only two vulnerability types in C/C++ dataset. On the other hand, we fail to achieve comparable results with the second tested dataset –project KB. This serves as a reminder that a successful machine learning pipeline is nothing without an appropriate and realistic dataset.

The Java dataset-related results bring in question the feasibility of automatic vulnerability detection on *real* source code. While our results show that sample size and frequency of specific vulnerability appearance have little effect on performance, one glaring difference remains: the realism of source code. A synthetic dataset like Juliet consists of clean and illustrative examples, typically without additional unrelated code. On the other hand, real code will likely contain a number of unrelated lines of code before, after, and possibly within the vulnerable piece (when it's not a single line mistake). Increasing the number of tokens per gadget might alleviate this problem slightly, but an increase in gadget size might lead to more noise and potential training on features unrelated to the actual vulnerabilities.

Additionally, the overlap between the CWEs in Juliet and project KB is somewhat low. If we are to use the project KB dataset as a window into what vulnerabilities actually occur in real world setting, the realism of vulnerabilities included in Juliet dataset is put in question. Other works have noted similar patterns, raising concerns that the "vulnerabilities in Juliet are not very diverse, and many of them are of a synthetic nature that does not occur in real-world software projects" [5].

Overall, the results show promise for machine learning vulnerability detection for Java source code. However, many open challenges remain before similar systems can realistically be used as a constructive and useful part of a real-world production line.

7.1 Challenges and future work

The current thesis shows that the specified pipeline for vulnerability detection can indeed act in a language-agnostic way and can -as such- work with C/C++ as well as Java by

making relatively small adaptations to the core of the system. More research is needed to generalize such claims to other popular programming languages.

As the main focus of this thesis was to explore the feasibility of machine learning vulnerability detection on Java source code, we did not explore the full set of adaptations that could improve the performance of the pipeline. The experiments were therefore only performed on a pipeline using code gadgets, token representation, word2vec vectorization and a BLSTM neural network. Changes to any of the mentioned properties could lead to different results and should be further researched (as highlighted in the discussion of the literature study). Such experimentation could also lead to better performance when dealing with more complicated/realistic datasets. There is likely also much room for improvement on the model level –e.g. through pre-training of models on synthetic data before training on real world data.

The findings on the tradeoff between sensitivity and specificity open two new questions, namely what metrics should be prioritized in a vulnerability detection system and whether a collaboration of a human expert and a machine can bring up the performance of the full system. While the VulDeePecker study explores this idea, it does so by leveraging human expertise to improve the training data used. A possible extension of this research could explore which –higher sensitivity or higher specificity– is perceived more useful for the final users and which leads to better vulnerability detection overall (after human evaluation of the outputs).

In addition to that, we notice that the majority of the studies (this thesis included) highly focus on performance metrics and similar statistics, but largely leave the final user out of the picture. To make proposed systems more useful for the final user, more field and experimental research is needed on how the users perceive such systems, how useful they are compared to traditional approaches and how they can be improved.

The experimentation with the two Java datasets also highlights the great differences in performance depending on the data used. While the exploration offers some insight into potential reasons, it is in no way complete or exhaustive.

An example for this could be our failure to pick up any significant difference between the treatments with different vulnerability frequency thresholds. The reason for this could very well be that the vulnerabilities that only appear rarely, make up a relatively small portion of the data, therefore only affecting the performance in a negligible way. However, if this is true, we could argue that the same holds for project KB as well. Alternatively, it could also be that the relative frequency of some vulnerability (compared to the full size of the dataset) doesn't mater that much; rather there is some absolute number of times a certain

vulnerability needs to appear in order to be picked up and recognized by the system. Given the small sample size and the low frequency of pretty much all vulnerabilities in project KB dataset, it is possible that few or no CWEs reach that threshold. Our experiments were not designed to capture this.

Finally, more research is needed to confirm whether the low performance of project KB trained system is indeed because the dataset is composed of real source code and not "toy" examples. If that is the case, the field should aim to move away from proof of concept works, where synthetic data is used, and focus on real data, which could bring machine learning-powered vulnerability detection into the mainstream. This is not a trivial undertaking. For such efforts to be successful, bigger datasets of real source code are likely also needed, more advanced source code representation systems and potentially more specialized models.

References

[1] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. Building useful program analysis tools using an extensible java compiler. In *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation*, pages 14–23, Italy, 2012. IEEE. 4

[2] Miltiadis Allamanis, Earl Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51(4): 1–37, 2018. 11

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *Proceedings of the 6th International Conference on Learning Representations*, 2018. 13, 15, 19

[4] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970. 6

[5] Guru Bhandari, Amara Naseer, and Leon Moonen. *CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software*, page 30–39. Association for Computing Machinery, New York, NY, USA, 2021. 52

[6] Sahil Bhatia and Rishabh Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. In *Proceedings of the 2nd2nd Indian Workshop on Machine Learning*, 2016. 13

[7] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. Neuro-symbolic program corrector for introductory programming assignments. In *Proceedings of the 40th International Conference on Software Engineering*, pages 60–70. IEEE, 2018. 15, 20

[8] Pan Bian, Bin Liang, Jianjun Huang, Wenchang Shi, Xidong Wang, and Jian Zhang. Sinkfinder: Harvesting hundreds of unknown interesting function pairs with just one seed. In *Proceedings of the 28th Joint Meeting on European Software Engineering*

Conference and Symposium on the Foundations of Software Engineering, pages 1101–1113. ACM, 2020. 15, 17, 20

[9] Jón Arnar Briem, Jordi Smit, Hendrig Sellik, Pavel Rapoport, Georgios Gousios, and Maurício Aniche. Offside: Learning to identify mistakes in boundary conditions. In *Proceedings of the 42nd International Conference on Software Engineering Workshops*, pages 203–208. IEEE/ACM, 2020. 15, 17

[10] Stephen Cass. *Top Programming Languages 2021 Python dominates as the de facto platform for new technologies*, 2021. Available: https://spectrum.ieee.org/top-programming-languages-2021. 1

[11] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2019. 15

[12] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security and Privacy*, 2(6), 2004. 4

[13] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 332–343, 2016. 4

[14] Mark Curphey and Rudolph Arawo. Web application security assessment tools. *IEEE Security and Privacy*, 4(4), 2006. 4

[15] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, 47(1):67–85, 2021. 15, 21

[16] Jacob Devlin, Jonathan Uesato, Rishabh Singh, and Pushmeet Kohli. Semantic code repair using neuro-symbolic transformation networks. In *Proceedings of the 6th International Conference on Learning Representations*, 2018. 13, 15

[17] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *Proceedings of the 8th International Conference on Learning Representations*, 2020. 13, 15, 17, 19

[18] Barney G Glaser and Anselm L Strauss. *Discovery of grounded theory: Strategies for qualitative research*. Routledge, 2017. 11

[19] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019. 5

[20] Cyril Goutte and Eric Gaussier. A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In *European conference on information retrieval*, pages 345–359. Springer, 2005. 8

[21] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the 31st Conference on Artificial Intelligence*, pages 1345–1351. AAAI, 2017. 13, 15, 20

[22] Rahul Gupta, Aditya Kanade, and Shirish Shevade. Deep reinforcement learning for syntactic error repair in student programs. In *Proceedings of the 33rd Conference on Artificial Intelligence*, pages 930–937. AAAI, 2019. 13, 15

[23] Rahul Gupta, Aditya Kanade, and Shirish Shevade. Neural attribution for semantic bug-localization in student programs. In *Proceedings of the 33rd Conference on Neural Information Processing Systems*. ACM, 2019. 15

[24] Andrew Habib and Michael Pradel. Neural bug finding: A study of opportunities and challenges. preprint, arXiv, 2019. 15, 17

[25] Hossein Hajipour, Apratim Bhattacharyya, and Mario Fritz. Samplefix: Learning to correct programs by efficient sampling of diverse fixes. In *Proceedings of Workshop on Computer-Assisted Programming*. ACM, 2020. 15

[26] Sudheendra Hangal and Monica Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*. IEEE, 2002. 5

[27] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher Reale, Rebecca Russell, Louis Kim, and Peter Chin. Learning to repair software vulnerabilities with generative adversarial networks. In *Proceedings of the 32nd Conference on Neural Information Processing Systems*. IEEE, 2018. 15, 17

[28] Hideaki Hata, Emad Shihab, and Graham Neubig. Learning to generate corrective patches using neural machine translation. preprint, arXiv, 2018. 13, 15

[29] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016. 5

[30] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM sigplan notices*, 39 (12):92–106, 2004. 5

[31] Matthew Hutson. Artificial intelligence faces reproducibility crisis, 2018. 24, 52

[32] Peter Alan Lee and Thomas Anderson. Fault tolerance. In *Fault Tolerance*, pages 51–77. Springer, 1990. 3

[33] Jian Li, Pinjia He, Jieming Zhu, and Michael Lyu. Software defect prediction via convolutional neural network. In *Proceedings of the International Conference on Software Quality, Reliability and Security*, pages 318–328. IEEE, 2017. 4, 15, 20

[34] Xin Li, Lu Wang, Yang Xin, Yixian Yang, and Yuling Chen. Automated vulnerability detection in source code using minimum intermediate representation learning. *Applied Sciences*, 10(5):1692, 2020. 15, 20

[35] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages*, 3(162):1–30, 2019. 15, 19

[36] Yi Li, Shaohua Wang, and Tien N Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the 42nd International Conference on Software Engineering*, pages 602–614. ACM/IEEE, 2020. 15

[37] 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 *Proceedings of the Network and Distributed Systems Security Symposium*. IEEE, 2018. 1, 13, 15, 18, 20, 21, 27, 28, 29

[38] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. preprint, arXiv, 2020. 15

[39] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 2021. 15

[40] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, and Yang Xiang. Poster: Vulnerability discovery with function representation learning from unlabeled projects. In *Proceedings of the Conference on Computer and Communications Security*, pages 2539–2541. ACM, 2017. 13, 15

[41] Kui Liu, Dongsun Kim, Tegawendé Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st International Conference on Software Engineering*, pages 1–12. IEEE/ACM, 2019. 15, 18

[42] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. Deepdelta: learning to repair compilation errors. In *Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 925–936. ACM, 2019. 15

[43] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems*, 2013. 25, 28

[44] Mitre. *Common Vulnerabilities and Exposures*, 2021. Available: https://www.cve.org/. 4

[45] Mitre. *Common Weakness Enumeration*, 2021. Available: https://cwe.mitre.org/. 4

[46] Martin Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys*, 51(1):1–24, 2018. 3, 4

[47] NIST. *National Vulnerability Database*, 2000. Available: https://nvd.nist.gov/. 30

[48] NIST. *Software Assurance reference dataset*, 2006. Available: https://samate.nist.gov/SARD/. 30

[49] NIST. *National Vulnerability Database*, 2017. Available: https://samate.nist.gov/SRD/testsuite.php. 30

[50] Will Douglas Heavenarchive page. AI is wrestling with a replication crisis, 2020. Available: https://www.technologyreview.com/2020/11/12/1011944/artificial-intelligence-replication-crisis-science-big-tech-google-deepmind-facebook-openai/. 52

[51] Ivan Pashchenko, Riccardo Scandariato, Antonino Sabetta, and Fabio Massacci. Secure software development in the era of fluid multi-party open software and services. In *2021 IEEE/ACM rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 91–95. IEEE, 2021. 4

[52] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and C´edric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proceedings of the 16th International Conference on Mining Software Repositories*, May 2019. 30

[53] Michael Pradel and Koushik Sen. Deepbugs: a learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(147):1–25, 2018. 15, 17, 19

[54] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. sk_p: a neural program corrector for moocs. In *Companion Proceedings of the 2016 SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 39–40. ACM, 2016. 13, 15

[55] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *Proceedings of the 42nd Annual Symposium on Principles of Programming Languages*, page 111–124. ACM SIGPLAN-SIGACT, 2015. 5

[56] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *Proceedings of the 17th International Conference on Machine Learning and Applications*, pages 757–762. IEEE, 2018. 15, 20

[57] Nicholas Saccente, Josh Dehlinger, Lin Deng, Suranjan Chakraborty, and Yin Xiong. Project achilles: A prototype tool for static method-level vulnerability detection of java source code using a recurrent neural network. In *Proceedings of the 34th International Conference on Automated Software Engineering Workshop*, pages 114–121. IEEE/ACM, 2019. 15, 21

[58] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. Syntax and sensibility: Using language models to detect and correct syntax errors. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*, pages 311–322. IEEE, 2018. 13, 15, 17, 20

[59] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim Laredo, and Alessandro Morari. Learning to map source code to software vulnerability using code-as-a-graph. preprint, arXiv, 2020. 15

[60] Anshul Tanwar, Krishna Sundaresan, Parmesh Ashwath, Prasanna Ganesan, Sathish Kumar Chandrasekaran, and Sriram Ravi. Predicting vulnerability in large codebases with deep code representation. preprint, arXiv, 2020. 15

[61] Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. Learning to fix build errors with graph2diff neural networks. In *Proceedings of the 42nd International Conference on Software Engineering Workshops*, pages 19–20. IEEE/ACM, 2020. 15, 17

[62] Tricentis. *Tricentis Software Fail Watch Finds 3.6 Billion People Affected and $1.7 Trillion Revenue Lost by Software Failures Last Year*, 2018. Available: www.tricentis.com/news/tricentis-software-fail-watch-finds-3-6-billion-people-affected-and-1-7-trillion-revenue-lost-by-software-failures-last-year. 1

[63] 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 Transactions on Software Engineering and Methodology*, 28(4):1–29, 2019. 15

[64] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural program repair by jointly learning to localize and repair. In *Proceedings of 7th International Conference on Learning Representations*, 2019. 15, 17, 22

[65] VDPython. *VDPython*, 2019. Available: https://github.com/johnb110/VDPython/. 29

[66] Sofia Visa, Brian Ramsay, Anca L Ralescu, and Esther Van Der Knaap. Confusion matrix-based feature selection. *MAICS*, 710:120–127, 2011. 8

[67] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of 38th International Conference on Software Engineering*, pages 297–308. IEEE/ACM, 2016. 15

[68] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*, pages 479–490. IEEE, 2019. 15, 17

[69] Chris Williams. *Anatomy of OpenSSL's Heartbleed: Just four bytes trigger horror bug*, 2014. Available: www.theregister.co.uk/2014/04/09/heartbleed_explained. 1

[70] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014. 6

[71] Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. In *Proceedings of the 37st International Conference on Machine Learning*, pages 10799–10808. IEEE, 2020. 13, 15, 17, 18

[72] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019. 23

[73] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proceedings of the 33rd Conference on Neural Information Processing Systems*. ACM, 2020. 15, 18

[74] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. μvuldeepecker: A deep learning-based system for multi-class vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*, 2019. 13, 15, 21, 22

Appendix

A Supporting material for Java datasets

```java
public class CWE190_Integer_Overflow__int_max_add_54e
{
    public void badSink(int data ) throws Throwable
    {

        /* POTENTIAL FLAW: if data == Integer.MAX_VALUE, this will overflow */
        int result = (int)(data + 1);

        IO.writeLine("result: " + result);
    }

    /* goodG2B() - use goodsource and badsink */
    public void goodG2BSink(int data ) throws Throwable
    {

        /* POTENTIAL FLAW: if data == Integer.MAX_VALUE, this will overflow */
        int result = (int)(data + 1);

        IO.writeLine("result: " + result);
    }

    /* goodB2G() - use badsource and goodsink */
    public void goodB2GSink(int data ) throws Throwable
    {

        /* FIX: Add a check to prevent an overflow from occurring */
        if (data < Integer.MAX_VALUE)
        {
            int result = (int)(data + 1);
            IO.writeLine("result: " + result);
        }
        else
        {
            IO.writeLine("data value is too large to perform addition.");
        }
    }
}
```

Listing A1: Vulnerable code sample taken from Juliet dataset (CWE-190)

```
...
public class HtmlFormFromFileController {

    private static final String TEMP_HTML_FORM_FILE_PREFIX = "html_form_";

    /** Logger for this class and subclasses */
    protected final Log log = LogFactory.getLog(getClass());

    @RequestMapping("/module/htmlformentry/htmlFormFromFile.form")
    public void handleRequest(Model model, @RequestParam(value = "filePath", required =
        false) String filePath,
                              @RequestParam(value = "patientId", required = false)
                                  Integer pId,
                              @RequestParam(value = "isFileUpload", required = false)
                                  boolean isFileUpload,
                              HttpServletRequest request) throws Exception {

        if (log.isDebugEnabled())
            log.debug("In reference data...");

        model.addAttribute("previewHtml", "");
        String message = "";
        File f = null;
        try {
            if (isFileUpload) {
                MultipartHttpServletRequest multipartRequest = (
                    MultipartHttpServletRequest) request;
                MultipartFile multipartFile = multipartRequest.getFile("htmlFormFile");
                if (multipartFile != null) {
                    //use the same file for the logged in user
                    f = new File(SystemUtils.JAVA_IO_TMPDIR, TEMP_HTML_FORM_FILE_PREFIX
                            + Context.getAuthenticatedUser().getSystemId());
                    if (!f.exists())
                        f.createNewFile();

                    filePath = f.getAbsolutePath();
                    FileOutputStream fileOut = new FileOutputStream(f);
                    IOUtils.copy(multipartFile.getInputStream(), fileOut);
                    fileOut.close();
                }
            } else {
                if (StringUtils.hasText(filePath)) {
                    f = new File(filePath);
                } else {
                    message = "You must specify a file path to preview from file";
                }
            }

            if (f != null && f.exists() && f.canRead()) {
                model.addAttribute("filePath", filePath);

                StringWriter writer = new StringWriter();
                IOUtils.copy(new FileInputStream(f), writer, "UTF-8");
                String xml = writer.toString();

                Patient p = null;
                if (pId != null) {
                    p = Context.getPatientService().getPatient(pId);
                } else {
                    p = HtmlFormEntryUtil.getFakePerson();
                }
                HtmlForm fakeForm = new HtmlForm();
                fakeForm.setXmlData(xml);
                FormEntrySession fes = new FormEntrySession(p, null, Mode.ENTER,
                    fakeForm, request.getSession());
                String html = fes.getHtmlToDisplay();
                if (fes.getFieldAccessorJavascript() != null) {
                    html += "<script>" + fes.getFieldAccessorJavascript() + "</script>";
                }
                model.addAttribute("previewHtml", html);
                //clear the error message
                message = "";
            } else {
                message = "Please specify a valid file path or select a valid file.";
            }
        }
        catch (Exception e) {
            log.error("An error occurred while loading the html.", e);
            message = "An error occurred while loading the html. " + e.getMessage();
        }

        model.addAttribute("message", message);
        model.addAttribute("isFileUpload", isFileUpload);
    }
}
```

Listing A2: Vulnerable code sample taken from project KB dataset (CVE-2017-12795/CWE-20)

Table A1: CWEs and their occurrences in project KB dataset

Occurrences	CWE-ID
20-39	CWE-22, CWE-611, CWE-20, CWE-502, CWE-79, CWE-200, CWE-287
19-10	CWE-352, CWE-74, CWE-264, CWE-295, CWE-835, CWE-863, CWE-770
9-5	CWE-119,CWE-444, CWE-94, CWE-400, CWE-399, CWE-918, CWE-89, CWE-522, CWE-284, CWE-601
<5	CWE-284, CWE-601, CWE-362, CWE-19, CWE-319, CWE-862, CWE-640, CWE-326, CWE-184, CWE-78, CWE-310, CWE-254, CWE-613, CWE-255, CWE-330, CWE-338, CWE-269, CWE-384, CWE-668, CWE-755, CWE-77, CWE-417, CWE-88, CWE-434, CWE-732, CWE-113, CWE-297, CWE-829, CWE-345, CWE-1021, CWE-1188, CWE-674, CWE-667, CWE-470, CWE-787, CWE-346, CWE-332, CWE-532, CWE-776, CWE-312, CWE-798, CWE-327, CWE-552, CWE-494, CWE-347, CWE-610, CWE-401, CWE-915

Table A2: CWEs and their occurrences in Juliet dataset

Occurrences	multicolumn1cCWE-ID
3000-7015	CWE-190, CWE-191, CWE-129, CWE-89, CWE-369
1000-2999	CWE-789, CWE-400, CWE-113, CWE-197, CWE-134, CWE-80
500-999	CWE-606, CWE-643, CWE-15, CWE-90, CWE-78, CWE-36, CWE-23, CWE-470, CWE-319, CWE-83, CWE-601, CWE-81
50-500	CWE-690, CWE-476, CWE-563, CWE-259, CWE-398, CWE-506, CWE-546, CWE-477, CWE-510, CWE-315, CWE-256, CWE-566, CWE-321, CWE-193, CWE-511, CWE-328, CWE-681
<50	CWE-327, CWE-396, CWE-617, CWE-209, CWE-459, CWE-526, CWE-325, CWE-338, CWE-390, CWE-382, CWE-483, CWE-486, CWE-586, CWE-533, CWE-395, CWE-698, CWE-481, CWE-523, CWE-759, CWE-535, CWE-572, CWE-253, CWE-484, CWE-379, CWE-597, CWE-252, CWE-605, CWE-614, CWE-584, CWE-329, CWE-336, CWE-598, CWE-378, CWE-478, CWE-613, CWE-534, CWE-539, CWE-615, CWE-549, CWE-114, CWE-226, CWE-760, CWE-482, CWE-570, CWE-571, CWE-383, CWE-835, CWE-833, CWE-404, CWE-581, CWE-499, CWE-568, CWE-397, CWE-580, CWE-582, CWE-500, CWE-772, CWE-764, CWE-561, CWE-609, CWE-491, CWE-775, CWE-674, CWE-585, CWE-607, CWE-765, CWE-832, CWE-111, CWE-667, CWE-579, CWE-248, CWE-600

B Full results

This section contains the raw averages for all models we tested in all experiments of this thesis.

Table B1: Full results for the C/C++ vulnerability detection models

Model	Lay	Sam	Loss	Rew	A(%)	P(%)	TPR(%)	TNR(%)	FPR(%)	FNR(%)	F1(%)
VDP	?	?	?	?	?	86.9	83.9	?	5.1	16.1	85.4
1	1	50	bin	✓	81.2	72.2	94.5	70.5	29.5	5.5	81.8
2	1	50	bin	-	83.0	78.2	86.5	80.1	19.9	13.5	82.0
3	1	50	cat	✓	80.5	71.1	95.4	68.4	31.6	4.6	81.3
4	1	50	cat	-	82.8	83.2	77.1	87.4	12.6	22.9	80.1
5	1	75	bin	✓	82.7	68.9	92.1	77.6	22.4	7.9	78.8
6	1	75	bin	-	84.2	75.4	82.2	85.3	14.7	17.8	78.5
7	1	75	cat	✓	82.9	70.7	88.2	80.1	19.9	11.8	78.3
8	1	75	cat	-	84.6	79.6	76.2	89.1	10.9	23.8	77.4
9	1	100	bin	✓	84.5	67.6	88.7	82.8	17.2	11.3	76.7
10	1	100	bin	-	85.9	82.8	65.4	94.2	5.8	34.6	72.5
11	1	100	cat	✓	84.3	68.0	86.1	83.5	16.5	13.9	75.9
12	1	100	cat	-	85.8	82.4	64.9	94.2	5.8	35.1	72.4
13	2	50	bin	✓	82.3	74.0	93.1	73.7	26.3	6.9	82.4
14	2	50	bin	-	84.1	79.9	86.1	82.4	17.6	13.9	82.7
15	2	50	cat	✓	82.0	73.5	93.7	72.5	27.5	6.3	82.2
16	2	50	cat	-	84.0	80.0	85.5	82.6	17.4	14.3	82.7
17	2	75	bin	✓	82.8	69.2	91.6	78.1	21.9	8.4	78.8
18	2	75	bin	-	85.0	81.1	74.6	90.6	9.4	25.4	77.6
19	2	75	cat	✓	82.9	70.3	89.7	79.2	20.8	10.3	78.6
20	2	75	cat	-	84.4	81.9	71.7	91.2	8.8	28.3	76.2
21	2	100	bin	✓	83.9	66.3	89.8	81.5	18.5	10.2	76.2
22	2	100	bin	-	86.4	80.1	70.3	92.9	7.1	29.7	74.8
23	2	100	cat	✓	83.3	65.2	90.7	80.4	19.6	9.3	75.8
24	2	100	cat	-	86.1	83.0	65.4	94.5	5.5	34.6	73.0
25	3	50	bin	✓	81.2	72.1	95.0	70.1	29.9	5.0	81.9
26	3	50	bin	-	83.3	77.4	88.8	78.8	21.2	11.0	82.5
27	3	50	cat	✓	79.3	69.7	95.9	66.0	34.0	4.1	80.7
28	3	50	cat	-	84.0	81.2	83.6	84.1	15.9	16.2	82.5
29	3	75	bin	✓	83.3	70.0	92.0	78.6	21.4	8.0	79.5
30	3	75	bin	-	84.8	75.2	84.9	84.8	15.2	15.1	79.7
31	3	75	cat	✓	81.5	67.1	92.3	75.7	24.3	7.7	77.7
32	3	75	cat	-	84.5	81.1	72.8	90.8	9.2	27.2	76.7
33	3	100	bin	✓	83.2	65.0	90.4	80.2	19.8	9.6	75.6
34	3	100	bin	-	86.8	78.9	73.8	92.0	8.0	26.2	76.2
35	3	100	cat	✓	83.5	65.8	89.5	81.1	18.9	10.5	75.8
36	3	100	cat	-	86.2	83.2	66.7	94.1	5.9	33.3	73.4
37	4	50	bin	✓	80.8	70.3	96.4	68.7	32.9	3.6	81.3
38	4	50	bin	-	83.8	77.7	90.7	82.3	21.0	9.3	83.7
39	4	50	cat	✓	80.5	70.1	96.1	68.7	33.0	3.9	81.1
40	4	50	cat	-	83.1	80.6	81.6	84.0	15.9	18.4	81.0
41	4	75	bin	✓	81.9	68.1	90.0	77.4	22.9	10.0	77.4
42	4	75	bin	-	83.7	72.9	85.9	84.5	17.2	14.1	78.8
43	4	75	cat	✓	82.0	68.6	90.5	76.9	22.3	9.5	78.0
44	4	75	cat	-	84.6	80.3	73.7	88.8	9.8	26.3	76.8
45	4	100	bin	✓	82.9	65.3	89.9	79.7	19.3	10.1	75.7
46	4	100	bin	-	85.5	76.1	70.1	92.5	8.9	29.9	73.0
47	4	100	cat	✓	84.0	68.6	81.2	85.0	15.7	18.8	73.6
48	4	100	cat	-	86.2	77.3	75.4	91.9	9.0	24.6	76.3

Notes:
(I) Parameters: Lay=layers, Sam=Sampling, Loss=Loss function, Rew=Reweighting
(II) Loss abbreviations: bin=binary crossentropy, cat=categorical crossentropy
(III) Metrics: A=Accuracy, P=Precision, TPR=True positive rate, TNR=True negative rate, FPR=False positive rate, FNR=False negative rate, F1=F1 score

Table B2: Studies

Model	Dataset	Size	Thr	Epoch	Tok	Res	Rew	A(%)	P(%)	TPR(%)	TNR(%)	F1(%)
1	KB	fullKB	1	4	50	0.25	1	34.77	31.30	98.78	7.50	47.52
2	KB	fullKB	1	4	50	0.25	0	72.34	58.58	38.31	86.84	44.21
3	KB	fullKB	1	4	50	0.5	1	53.95	25.98	85.63	47.21	39.73
4	KB	fullKB	1	4	50	0.5	0	83.74	79.06	10.68	99.30	18.69
5	KB	fullKB	1	4	50	0.75	1	61.11	21.38	79.07	58.56	33.63
6	KB	fullKB	1	4	50	0.75	0	88.34	80.88	8.47	99.69	15.12
7	KB	fullKB	1	4	50	1	1	66.53	18.89	74.79	65.65	30.12
8	KB	fullKB	1	4	50	1	0	91.01	84.11	8.23	99.83	14.95
9	KB	fullKB	1	10	50	0.25	1	42.78	34.09	97.07	19.65	50.42
10	KB	fullKB	1	10	50	0.25	0	74.94	64.84	35.78	91.62	45.74
11	KB	fullKB	1	10	50	0.5	1	52.65	25.87	88.27	45.06	39.86
12	KB	fullKB	1	10	50	0.5	0	84.30	81.64	13.81	99.31	23.55
13	KB	fullKB	1	10	50	0.75	1	65.34	23.30	77.79	63.57	35.85
14	KB	fullKB	1	10	50	0.75	0	88.87	79.51	15.23	99.33	25.22
15	KB	fullKB	1	10	50	1	1	72.88	21.85	68.20	73.38	32.85
16	KB	fullKB	1	10	50	1	0	91.20	79.67	11.78	99.66	20.45
17	KB	fullKB	1	10	75	0.25	1	46.30	35.53	95.50	25.35	51.69
18	KB	fullKB	1	10	75	0.25	0	75.04	66.76	35.07	92.06	44.96
19	KB	fullKB	1	10	75	0.5	1	55.83	27.42	86.75	49.24	41.42
20	KB	fullKB	1	10	75	0.5	0	84.57	71.62	20.59	98.20	31.71
21	KB	fullKB	1	10	75	0.75	1	66.44	24.16	77.72	64.84	36.72
22	KB	fullKB	1	10	75	0.75	0	88.83	76.46	16.62	99.09	26.56
23	KB	fullKB	1	10	75	1	1	70.88	21.28	73.02	70.66	32.88
24	KB	fullKB	1	10	75	1	0	91.23	86.97	11.87	99.79	20.80
25	KB	fullKB	1	10	100	0.25	1	44.98	34.98	97.22	22.73	51.42
26	KB	fullKB	1	10	100	0.25	0	76.01	66.79	40.87	90.97	50.17
27	KB	fullKB	1	10	100	0.5	1	58.92	28.16	85.22	53.32	42.26
28	KB	fullKB	1	10	100	0.5	0	84.41	67.62	22.78	97.53	33.64
29	KB	fullKB	1	10	100	0.75	1	68.22	25.06	77.69	66.88	37.87
30	KB	fullKB	1	10	100	0.75	0	88.91	81.61	14.09	99.53	23.95
31	KB	fullKB	1	10	100	1	1	74.32	22.95	68.05	74.99	34.06
32	KB	fullKB	1	10	100	1	0	91.21	82.41	12.12	99.63	20.79

(a) Full results for the Java vulnerability detection models trained on project KB dataset

Model	Dataset	Size	Thr	Epoch	Tok	Res	Rew	A(%)	P(%)	TPR(%)	TNR(%)	F1(%)
1	Juliet	fullJ	1	4	50	0.5	1	87.88	77.57	99.42	79.77	87.14
2	Juliet	fullJ	1	4	50	0.5	0	87.41	81.11	90.64	85.15	85.47
3	Juliet	fullJ	1	4	50	0.75	1	87.02	71.33	99.22	81.30	83.00
4	Juliet	fullJ	1	4	50	0.75	0	88.01	88.76	71.88	95.58	79.22
5	Juliet	fullJ	1	4	50	1	1	85.66	64.60	99.39	80.83	78.30
6	Juliet	fullJ	1	4	50	1	0	89.76	89.85	69.02	97.06	77.76
7	Juliet	fullJ	1	10	50	0.5	1	89.21	79.35	99.89	81.68	88.44
9	Juliet	fullJ	1	10	50	0.75	1	88.08	73.16	99.04	82.94	84.16
10	Juliet	fullJ	1	10	50	0.75	0	89.64	82.26	86.20	91.25	84.17
11	Juliet	fullJ	1	10	50	1	1	86.57	66.14	99.24	82.11	79.38
12	Juliet	fullJ	1	10	50	1	0	90.14	96.03	64.82	99.06	77.40
13	Juliet	fullKB	1	10	50	0.5	1	88.24	78.11	99.39	80.39	87.47
14	Juliet	fullKB	1	10	50	0.5	0	88.95	81.19	95.50	84.34	87.69
15	Juliet	fullKB	1	10	50	0.75	1	87.57	72.23	99.22	82.11	83.60
16	Juliet	fullKB	1	10	50	0.75	0	89.13	85.90	78.99	93.89	82.26
17	Juliet	fullKB	1	10	50	1	1	86.44	65.97	98.95	82.04	79.16
18	Juliet	fullKB	1	10	50	1	0	89.92	96.60	63.57	99.19	76.64
19	Juliet	35k	1	10	50	0.5	1	87.73	77.33	99.55	79.41	87.03
20	Juliet	35k	1	10	50	0.5	0	88.06	80.91	93.85	83.99	86.60
21	Juliet	35k	1	10	50	0.75	1	86.17	69.89	99.71	79.81	82.17
22	Juliet	35k	1	10	50	0.75	0	88.42	84.15	78.95	92.86	81.26
23	Juliet	35k	1	10	50	1	1	86.14	65.49	98.79	81.68	78.77
24	Juliet	35k	1	10	50	1	0	89.68	93.74	65.01	98.37	76.62
25	Juliet	35k	100	10	50	0.5	1	88.33	78.10	99.69	80.34	87.58
26	Juliet	35k	100	10	50	0.5	0	88.12	82.62	90.42	86.49	86.25
27	Juliet	35k	100	10	50	0.75	1	86.71	70.83	99.25	80.83	82.66
28	Juliet	35k	100	10	50	0.75	0	88.07	82.78	79.48	92.09	80.93
29	Juliet	35k	100	10	50	1	1	85.67	64.63	99.18	80.92	78.26
30	Juliet	35k	100	10	50	1	0	89.77	90.12	68.93	97.10	77.76
31	Juliet	35k	1000	10	50	0.5	1	87.90	76.98	99.72	79.96	86.88
32	Juliet	35k	1000	10	50	0.5	0	88.12	77.64	98.94	80.85	87.01
33	Juliet	35k	1000	10	50	0.75	1	86.85	70.30	99.50	81.19	82.39
34	Juliet	35k	1000	10	50	0.75	0	88.64	84.30	78.01	93.40	80.90
35	Juliet	35k	1000	10	50	1	1	85.79	64.02	99.25	81.27	77.83
36	Juliet	35k	1000	10	50	1	0	89.39	88.52	67.06	96.89	75.98

(b) Full results for the Java vulnerability detection models trained on Juliet dataset

Notes:
(I) Parameters: Thr=threshold for minimum frequency for a CWE, Tok=Tokens, Res=Resampling, Rew=Reweighting
(II) Sample size explanation: fullKB=full size of project KB dataset, fullJ=full size of Juliet dataset, 35K=35.000 code gadget samples
(III) Metrics: A=Accuracy, P=Precision, TPR=True positive rate, TNR=True negative rate, F1=F1 score

C Additional tables and figures

C.1 Vulnerability detection in C source code

This subsection presents in a concise table the ANOVA results supporting the conclusions from subsection 5.2.2.

		P	TPR	TNR
Layers	p	0.41	0.59	0.53
	ES	1.2%	0.8%	0.9%
Resampling	p	0.05*	0.01*	0.01*
	ES	2.5%	20.5%	32.1%
Reweighting	p	0.01*	0.01*	0.01*
	ES	67.2%	51.9%	49.9%
Loss	p	0.12	0.03*	0.225
	ES	1.0%	1.9%	0.6%

Notes:
p refers to p-value of the ANOVA test
ES is the percentage of total variance explained
Stars (*) indicate significance at 5% confidence level

Table C1: Summary of ANOVA results for C/C++

C.2 Vulnerability detection in Java source code

This subsection presents additional results supporting the conclusions from subsection 6.2.1. We first present visual evidence in the form of boxplots of performance metrics across resampling and reweighting (separately for project KB dataset and for Juliet dataset). We also present the means and standard deviations that facilitate the comparisons. Finally, we present a synopsis of the ANOVA tests formally testing for significant differences.

	P		TPR		TNR	
Res	μ	σ	μ	σ	μ	σ
25%	47.2	15.7	67.5	32.0	51.4	39.6
50%	53.1	28.6	49.6	38.5	72.7	28.2
75%	51.2	30.2	45.1	34.3	80.3	19.9
100%	51.1	31.8	40.7	32.0	84.6	16.2
#Runs	40		40		40	

	P		TPR		TNR	
Res	μ	σ	μ	σ	μ	σ
50%	79.7	1.8	97.1	4.7	82.5	2.5
75%	78.9	7.5	89.1	11.8	87.8	6.1
100%	79.1	14.6	83.1	16.9	89.8	8,6
#Runs	40		40		40	

Table C2: Mean and standard deviation across resamplings for project KB (left) and Juliet (right)

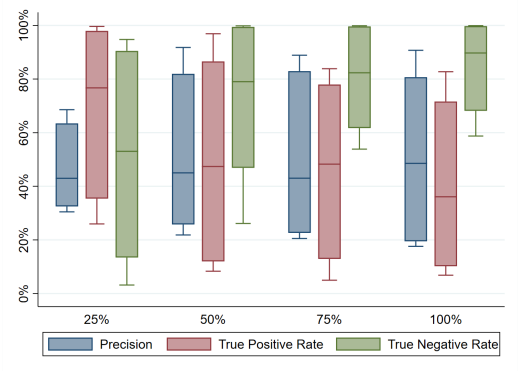

Table C3: Effect of resampling on models trained with project KB dataset

	P		TPR		TNR	
Rew	μ	σ	μ	σ	μ	σ
With	25.3	5.1	83.7	11.0	47.6	22.9
Without	76.0	11.0	17.8	12.6	96.9	5.5
#Runs	80		80		80	

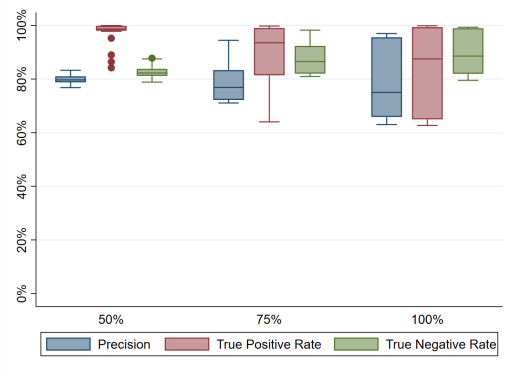

Table C4: Effect of resampling on models trained with Juliet dataset

	P		TPR		TNR	
Rew	μ	σ	μ	σ	μ	σ
With	72.0	5.5	99.4	0.5	81.4	1.1
Without	86.4	6.4	80.1	12.9	91.9	6.2
#Runs	60		60		60	

Table C5: Mean and standard deviation across reweighting for project KB (left) and Juliet (right)

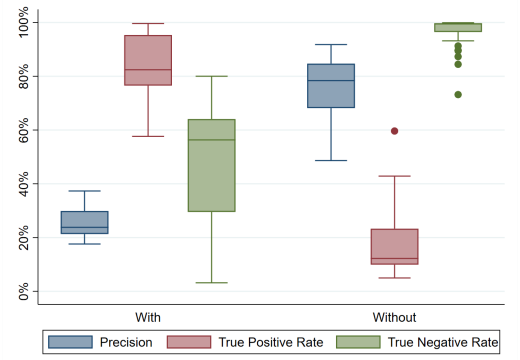

Table C6: Effect of reweighting on models trained with project KB dataset

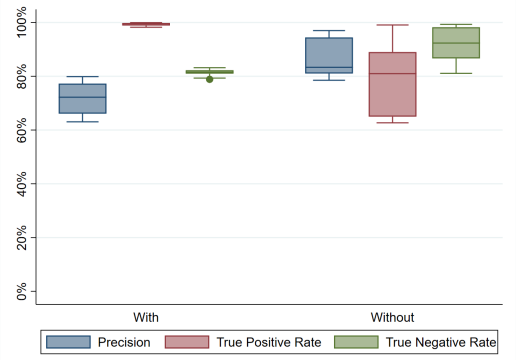

Table C7: Effect of reweighting on models trained with Juliet dataset

Table C8: ANOVA results for Java adaptation

		P	TPR	TNR
Epochs	p	0.83	0.95	0.62
	ES	0.1%	0.1%	0.3%
Resampling	p	0.92	0.08	0.001*
	ES	0.7%	8.4%	18.5%
Reweighting	p	0.01*	0.01*	0.01*
	ES	89.9%	88.8%	69.3%

(a) ANOVA results for KB trained models

		P	TPR	TNR
Epochs	p	0.77	0.39	0.95
	ES	0.2%	1.3%	0.1%
Resampling	p	0.97	0.01*	0.01*
	ES	0.1%	18.8%	20.1%
Reweighting	p	0.01*	0.01*	0.01*
	ES	60.2%	53.0%	58.6%

(b) ANOVA results for Juliet trained models

Notes for both tables: p refers to p-value of the ANOVA test, ES is the percentage of total variance explained, and stars (*) indicate significance at 5% confidence level