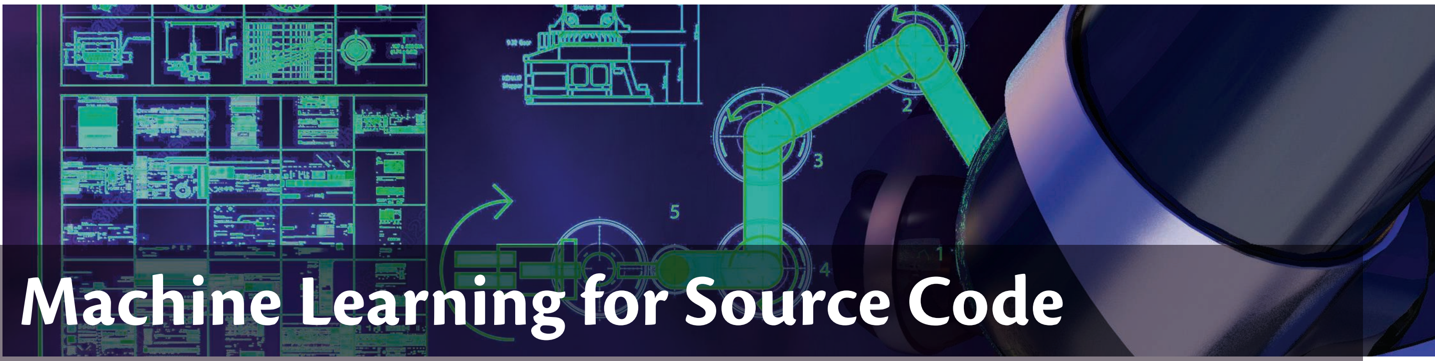

Machine Learning for Source Code Vulnerability Detection: What Works and What Isn't There Yet

Tina Marjanov | Vrije Universiteit Amsterdam and University of Cambridge
Ivan Pashchenko | TomTom
Fabio Massacci | University of Trento and Vrije Universiteit Amsterdam

We review machine learning approaches for detecting (and correcting) vulnerabilities in source code, finding that the biggest challenges ahead involve agreeing to a benchmark, increasing language and error type coverage, and using pipelines that do not flatten the code's structure.

Traditionally, defects in source code have been discovered by means of static and dynamic analysis techniques. However, static analysis techniques are known to generate a high number of false positive (FP) findings,[1] while dynamic analysis tools are designed to underestimate the number of defects in a program and therefore are prone to false negatives (FNs). Moreover, static analysis techniques might require significant computation resources, while dynamic analysis tools increase the size and execution time of a program. Hence, such techniques cannot always be seamlessly integrated into the continuous delivery pipelines of today's software projects.[2,3]

In this respect, machine learning (ML) techniques seem to have become a very attractive alternative to traditional software defect detection and correction techniques. In this article, we discuss a representative snapshot of the state-of-the-art research on the detection and correction of security defects through ML techniques. Given the rapidly increasing interest in ML applications in source code, several studies have started to apply ML for bug prediction. Some earlier reviews are presented in Table 1.

Digital Object Identifier 10.1109/MSEC.2022.3176058
Date of current version: 18 August 2022

From Detection to Correction

To frame our work, we adopt the recent terminology by Monperrus et al.[4] For simplicity, we formulate detection as classification, while repair is formulated as generation. Realistically, not all approaches fit this precisely; for example, Hoppity [P15] uses the classification of graph edit types to do repair.

Automated defect detection is "a process of building classifiers to predict code areas that potentially contain defects, using information such as code complexity and change history."[5] There have been several defect detection tools available in recent years, two of the bigger ones being Google's Error Prone and SpotBugs (formerly known as FindBugs). 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. They required a considerable amount of expert work and generated many FPs.

A logical next step from detection is automated correction, which tries "to automatically identify patches for a given bug that can then be applied with little, or possibly even without human intervention."[6] Compared to detection, correction is a more ambitious goal, which has only recently emerged as a realistic research topic through the use of techniques previously applied to natural language. It often operates either by learning to translate from pairs of incorrect and correct programs (as one would translate between two languages, e.g., English and Dutch) or learning from correct examples and translating programs that deviate from that.

If we consider autonomous vulnerability detection and/or correction the end goal, we can see research in syntactic (i.e., the grammar of code) and semantic (i.e., the meaning of code) error detection and correction as stepping stones toward the end goal. Additionally, even if a tool is not primarily aimed at traditional vulnerabilities, syntactic and semantic errors can introduce vulnerabilities to code, which is why we discuss the three error types equally. Table 2 provides a concise dictionary of the relevant terms.

Enter ML for Source Code

Recent years have seen the emergence of ML for finding vulnerabilities. A typical ML pipeline consists of

Table 1. The surveys on ML, defect detection, and correction.

Authors	Venue	Survey type
Malhotra	*Applied Soft Computing* (2014)	ML techniques for software fault prediction, comparing the performance of ML to statistical techniques
Ghaffarian and Shahriari	*ACM Computing Surveys* (2018)	Traditional ML and data mining techniques for vulnerability detection
Allamanis et al.	*ACM Computing Surveys* (2018)	ML used for source code and natural language translations
Ji et al.	IEEE Conference on Dependable and Secure Computing (2018)	Autonomous cyberreasoning systems for detection, patching, and exploiting software vulnerabilities
Monperrus	*ACM Computing Surveys* (2019)	Automatic program repair techniques
Singh and Chaturvedi	International Conference on Soft Computing: Theories and Applications (2020)	Deep learning techniques for vulnerability detection
Lin et al.	*Proceedings of the IEEE* (2020)	Vulnerability detection tools using deep neural networks
Shen et al.	*Security and Communication Networks* (2020)	Vulnerability detection, program repair, and defect prediction methods that include binary code
Zeng et al.	*IEEE Access* (2020)	Deep learning software vulnerability discovery approaches

Table 2. The defect types.

Term	Definition
Defect	Also known as *errors*, *bugs*, and *faults*, defects are deviations between a program's expected behavior and what actually happens.
Syntactic defects	These are mistakes in the syntax of a program, i.e., the grammar and rules of the language. They are usually detected at compile time and runtime and prevent a program from running at all. Such problems, depending on the language, include missing brackets and semicolons, typos, indentation problems, and so on.
Semantic defects	These are 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 primarily a security concern. Such problems include inconsistent method names, variable misuse bugs, typing errors, application programming interface misuse, swapped arguments in functions, and so on.
Vulnerabilities	Vulnerabilities form a particular set of semantic defects that can compromise the security of a system. Such problems include buffer overflows, integer overflows, cross-site scripting, use-after-free, and so on.

several important stages: data collection and preparation, model training, and, finally, evaluation and deployment. (We assume that the reader is familiar with these steps, but to make the article self-contained, we report them in Table 3.) Typically, the tool is monitored, maintained, and improved after deployment, but this is outside the scope of this article.

ML models are generally not capable of ingesting the source code in its original format, so the code is processed and transformed into some low-level representation appropriate for ML model input [e.g., vectors for neural networks (NNs)]. To preserve the semantic and syntactic properties of the program, it is useful to consider some intermediate interpretation that is capable of encoding such properties before feeding the program into the model. The three predominant approaches treat the source code as follows (roughly inspired by Chakraborty et al.[7] and Shen and Chen[8]):

- *Sequence of tokens*: The raw source code is split into small units (e.g., "int," "func," "{," and "}") and presented to the model as such.
- *Abstract syntax tree (AST)*: The syntactic structure of the program is captured by a tree representation, with each node of the tree denoting a construct occurring in the source code.
- *Graphs*: These can capture various syntactic and semantic relationships and properties of the source code through the edges and nodes of the graphs (e.g., control flow graphs[9] and code property graphs[10]).

The three classes are a simplification for the purpose of synthesis. In practice, many of the tools use versions that blend the lines between representations. Additionally, the classes do not reflect the full picture but rather the most widely used approaches.

ML comes with a distinct set of challenges that need to be considered to produce reliable and useful results. First, it is crucial to train the model on a high-quality data set. In general, this means a large enough, realistic data set with a representative distribution of classes. For example, a model trained on a data set that contains an equal number of buggy and nonbuggy programs might not perform well when used in a real setting where the occurrence of bugs is significantly lower or different types of bugs occur. Problems with a data set can be mitigated to some extent in the later stages of the pipeline, but a strong data set is preferred.

Additionally, a common problem that surfaces when evaluating and replicating the results is overfitting, meaning that the model too closely fits the training data and does not show previous predictive power, often due to noisy data and overcomplicated models. Finally, the selection of relevant features is one of the most important tasks of ML. It is important to consider the number of features—having 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 important features directly from the training data themselves.

The prediction of an ML model has four possible classification states, i.e., the confusion matrix: true positives (TPs), true negatives (TNs), FPs, and FNs. In our case, a TP could mean a buggy line of code that is correctly classified as a bug, and an FP could be a nonbuggy line of code that is wrongly classified as a bug.

Methodology

Our goal is to examine and present a representative snapshot of the state-of-the-art research and identify

Table 3. The ML pipeline.

Stage	Description
Data collection	A sufficiently large and representative data set for the task is constructed.
Data preparation	Data preparation consists of cleaning and sometimes labeling, feature engineering, and, finally, splitting into (nonoverlapping) 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 nontrivial task.
Model training	The training portion of the data set is used to create a model that will be able to distinguish erroneous code from correct code and optionally propose candidate corrections. 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. Training is frequently the longest and computationally most expensive part.
Evaluation	The model is evaluated on the test subset of data to determine if it exhibits the desired behaviors when presented with unseen data. At this stage, the model should be able to detect and optionally correct programming defects and can be deployed to be used.

the trends and gaps. Adopting an agnostic starting point, we want to discover patterns without being biased by our own dispositions and conjectures. For this, we leverage the grounded theory approach widely used in empirical studies; this method allows hypotheses to emerge from the data.

An initial set of 343 works was drawn from an online repository containing ML research on source code; the collection was created in the scope of Allamanis et al.[11] and is actively maintained (https://ml4code.github.io). To reflect the state-of-the-art techniques and considering that ML is rapidly evolving, we focus our review on papers from 2015 onward (322 papers out of 343). Additionally, we keep only the papers on defect detection and correction that use static analysis of the source code. We therefore exclude papers on topics such as synthesis, prediction, recommendation, summarization, and so on as well as those discussing supporting techniques for defect detection, including testing, fuzzing, taint analysis, symbolic execution, defects in binary code, and so forth.

Finally, we exclude papers without a proof of concept or full ML pipeline. Papers that share large parts of the pipeline and adapt or discuss only one part of it are treated as one with the most representative paper included and discussed. After the removals, a set of 31 relevant papers emerges. 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 from Google Scholar. The final list consists of 40 papers containing an end-to-end ML pipeline capable of either detecting or correcting defects in source code (see Table 4).

To facilitate the discovery of emerging patterns from data (i.e., the set of selected papers), we need to identify the defining characteristics of a defect correction/detection tool. Our initial codes were heavily inspired by the characteristics discussed by related literature (see Table 1). The codes were first used to annotate a small portion of the selected papers to test their suitability and completeness. Then we synchronized the resulting codebook among all the researchers involved in the study to identify a set of codes that captured the most important differences among the studies while ensuring that no part of the ML pipeline was left out. We performed this process iteratively until the codebook became stable. Finally, after finalizing the full set of codes, we expanded the coding to the remainder of the papers. The coding and subsequent analysis were performed using Atlas.ti.

Code groups related to the abilities of tools include the following:

- *Correction* refers to the correction and detection ability of the tool.
- *Defect type* refers to the primary type of defect the tool targets. If a more advanced tool can simultaneously correct simpler mistakes (e.g., a semantic defect tool fixing misplaced brackets, which is 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. This does not include further transformations inside the models but rather the initial information presented to the model.
- *Language* refers to the language the tool targets. If a tool can act in a language-agnostic way, we refer to the language of the data set that is tested.

Code groups that capture information about the data sets include the following:

- The type captures whether the data sets include buggy examples and, if bugs are present, whether buggy and nonbuggy examples are paired.
- The label captures whether the data set is labeled or unlabeled.
- Realism captures whether the programs and errors in the data set are taken from real applications or synthetically produced.
- Availability captures whether the data set and/or tool are publicly available.

It is important to note that the type, label, and availability of the data set refer to the training data. When training is performed on data that do not have the same structure as the test subset, we describe the training data (e.g., correction tools that train only on nonbuggy examples). Additionally, when training data are collected from a public data set but then modified in some way, we describe the modified version (e.g., a publicly available data set is injected with bugs). Table 5 presents the final codebook. It shows the identified code groups, the possible values for each, and illustrative examples taken from the source papers (an overview of the studies is also available on https://github.com/tmv200/ml4code/blob/main/sota.yaml).

Analysis of Recent Works

Table 6 provides an overview of the studies included in this review. Generally speaking, we can see (Figure 1) an increase in publications since 2015, signaling growing interest in the field. This holds for both detection and correction studies. Overall, the examined papers exhibit wide variety in goals and priorities, leading to a wealth of different approaches (Figure 2 and Table 7).

Table 4. The analyzed papers (correction).

[P1]	P. Yewen, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, “sk_p: A neural program corrector for MOOCs,” in *Proc. Companion 2016 SIGPLAN Int. Conf. Syst., Program., Languages Appl., Softw. Humanity*, ACM, 2016, pp. 39–40, doi: 10.1145/2984043.2989222.
[P2]	G. Rahul, S. Pal, A. Kanade, and S. Shevade, “DeepFix: Fixing common C language errors by deep learning,” in *Proc. 31st Conf. Artif. Intell.*, 2017, pp. 1345–1351, doi: 10.5555/3298239.3298436.
[P3]	S. Bhatia, P. Kohli, and R. Singh, “Neuro-symbolic program corrector for introductory programming assignments,” in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 60–70, doi: 10.1145/3180155.3180219.
[P4]	J. Devlin, J. Uesato, R. Singh, and P. Kohli, “Semantic code repair using neuro-symbolic transformation networks,” in *Proc. 6th Int. Conf. Learn. Representations*, 2018, pp. 1–11.
[P5]	J. Harer *et al.*, “Learning to repair software vulnerabilities with generative adversarial networks,” in *Proc. 32nd Conf. Neural Inf. Process. Syst.*, 2018, pp. 7944–7954, doi: 10.5555/3327757.3327890.
[P6]	H. Hata, E. Shihab, and G. Neubig, “Learning to generate corrective patches using neural machine translation,” 2018, *arXiv:1812.07170*.
[P7]	E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral, “Syntax and sensibility: Using language models to detect and correct syntax errors,” in *Proc. 25th Int. Conf. Softw. Anal., Evol. Reeng.*, 2018, pp. 311–322, doi: 10.1109/SANER.2018.8330219.
[P8]	Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, “Sequencer: Sequence-to-sequence learning for end-to-end program repair,” *IEEE Trans. Softw. Eng.*, vol. 47, no. 9, pp. 1943–1959, 2019, doi: 10.1109/TSE.2019.2940179.
[P9]	R. Gupta, A. Kanade, and S. Shevade, “Deep reinforcement learning for syntactic error repair in student programs,” in *Proc. 33rd Conf. Artif. Intell.*, 2019, pp. 930–937, doi: 10.1609/aaai.v33i01.3301930.
[P10]	K. Liu *et al.*, “Learning to spot and refactor inconsistent method names,” in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 1–12, doi: 10.1109/ICSE.2019.00019.
[P11]	A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian, “DeepDelta: Learning to repair compilation errors,” presented at the 27th Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., 2019, pp. 925–936, doi: 10.1145/3338906.3340455.
[P12]	M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, “An empirical study on learning bug-fixing patches in the wild via neural machine translation,” *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 1–29, 2019, doi: 10.1145/3340544.
[P13]	M. Vasic, A. Kanade, P. Maniatis, D. Bieber, and R. Singh, “Neural program repair by jointly learning to localize and repair,” in *Proc. 7th Int. Conf. Learn. Representations*, 2019, pp. 1–12.
[P14]	M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, “Sorting and transforming program repair ingredients via deep learning code similarities,” in *Proc. 26th Int. Conf. Softw. Anal., Evol. Reeng.*, 2019, pp. 479–490, doi: 10.1109/SANER.2019.8668043.
[P15]	E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, “Hoppity: Learning graph transformations to detect and fix bugs in programs,” in *Proc. 8th Int. Conf. Learn. Representations*, 2020, pp. 1–17.
[P16]	H. Hajipour, A. Bhattacharyya, and M. Fritz, “SampleFix: Learning to correct programs by efficient sampling of diverse fixes,” in Proc. Workshop Comput.-Assisted Program., ACM, 2020, pp. 1–10.
[P17]	Y. Li, S. Wang, and T. N. Nguyen, “DLFix: Context-based code transformation learning for automated program repair,” in *Proc. 42nd Int. Conf. Softw. Eng.*, ACM/IEEE, 2020, pp. 602–614, doi: 10.1145/3377811.3380345.
[P18]	D. Tarlow *et al.*, “Learning to fix build errors with graph2diff neural networks,” in *Proc. 42nd Int. Conf. Softw. Eng. Workshops*, IEEE/ACM, 2020, pp. 19–20, doi: 10.1145/3387940.3392181.
[P19]	M. Yasunaga and P. Liang, “Graph-based, self-supervised program repair from diagnostic feed-back,” in *Proc. 37st Int. Conf. Mach. Learn.*, 2020, pp. 10,799–10,808.

(*Continued*)

Table 4. The analyzed papers (detection). (*Continued*)

[P20]	S. Wang, and T. Liu, and L. Tan, “Automatically learning semantic features for defect prediction,” in *Proc. 38th Int. Conf. Softw. Eng.*, IEEE/ACM, 2016, pp. 297–308, doi: 10.1145/2884781.2884804.
[P21]	J. Li, P. He, J. Zhu, and M. Lyu, “Software defect prediction via convolutional neural network,” in *Proc. Int. Conf. Softw. Qual., Rel. Secur.*, 2017, pp. 318–328, doi: 10.1109/QRS.2017.42.
[P22]	G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, “POSTER: Vulnerability discovery with function representation learning from unlabeled projects,” in *Proc. Conf. Comput. Commun. Secur.*, ACM, 2017, pp. 2539–2541, doi: 10.1145/3133956.3138840.
[P23]	M. Pradel and K. Sen, “DeepBugs: A learning approach to name-based bug detection,” *Proc. ACM Program. Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018, doi: 10.1145/3276517.
[P24]	M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to represent programs with graphs,” in *Proc. 6th Int. Conf. Learn. Representations*, 2018, pp. 1–17.
[P25]	Z. Li *et al.*, “VulDeePecker: A deep learning-based system for vulnerability detection,” in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.
[P26]	R. Russell et al., “Automated vulnerability detection in source code using deep representation learning,” in *Proc. 17th Int. Conf. Mach. Learn. Appl.*, 2018, pp. 757–762, doi: 10.1109/ICMLA.2018.00120.
[P27]	D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, “μ-VulDeePecker: A deep learning-based system for multi-class vulnerability detection,” *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 5, pp. 2224–2236, 2019, doi: 10.1109/TDSC.2019.2942930.
[P28]	R. Gupta, A. Kanade, and S. Shevade, “Neural attribution for semantic bug-localization in student programs,” in Proc. 33rd Conf. Neural Inf. Process. Syst., ACM, 2019, 11,884-11,894.
[P29]	A. Habib and M. Pradel, “Neural bug finding: A study of opportunities and challenges,” 2019, *arXiv:1906.00307*.
[P30]	Y. Li, S. Wang, T. N. Nguyen, and S. V. Nguyen, “Improving bug detection via context-based code representation learning and attention-based neural networks,” *Proc. ACM Program. Languages*, vol. 3, no. OOPSLA, pp. 1–30, 2019, doi: 10.1145/3360588.
[P31]	N. Saccente, J. Dehlinger, L. Deng, S. Chakraborty, and Y. Xiong, “Project achilles: A prototype tool for static method-level vulnerability detection of java source code using a recurrent neural network,” in *Proc. 34th Int. Conf. Autom. Softw. Eng. Workshop*, IEEE/ACM, 2019, pp. 114–121, doi: 10.1109/ASEW.2019.00040.
[P32]	X. Li, L. Wang, Y. Xin, Y. Yang, and Y. Chen, “Automated vulnerability detection in source code using minimum intermediate representation learning,” *Appl. Sci.*, vol. 10, no. 5, p. 1692, 2020, doi: 10.3390/app10051692.
[P33]	Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, “VulDeeLocator: A deep learning-based fine-grained vulnerability detector,” *IEEE Trans. Dependable Secure Comput.*, early access, 2020, doi: 10.1109/TDSC.2021.3076142.
[P34]	P. Bian, B. Liang, J. Huang, W. Shi, X. Wang, and J. Zhang, “SinkFinder: Harvesting hundreds of unknown interesting function pairs with just one seed,” presented at the 28th Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., ACM, 2020, pp. 1101–1113, doi: 10.1145/3368089.3409678.
[P35]	J. A. Briem, J. Smit, H. Sellik, P. Rapoport, G. Gousios, and M. Aniche, “OffSide: Learning to identify mistakes in boundary conditions,” in *Proc. 42nd Int. Conf. Softw. Eng. Workshops*, IEEE/ACM, 2020, pp. 203–208, doi: 10.1145/3387940.3391464.
[P36]	S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, and A. Morari, “Learning to map source code to software vulnerability using code-as-a-graph,” 2020, *arXiv:2006.08614*.
[P37]	A. Tanwar, K. Sundaresan, P. Ashwath, P. Ganesan, S. K. Chandrasekaran, and S. Ravi, “Predicting vulnerability in large codebases with deep code representation,” 2020, *arXiv:2004.12783*.
[P38]	Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” in *Proc. 33rd Conf. Neural Inf. Process. Syst.*, ACM, 2020, pp. 10,197–10,207, doi: 10.5555/3454287.3455202.
[P39]	H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, “Automatic feature learning for predicting vulnerable software components,” *IEEE Trans. Softw. Eng.*, vol. 47, no. 1, pp. 67–85, 2021, doi: 10.1109/TSE.2018.2881961.
[P40]	Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, “SySeVR: A framework for using deep learning to detect software vulnerabilities,” *IEEE Trans. Dependable Secure Comput.*, early access, 2021, doi: 10.1109/TDSC.2021.3051525.

Table 5. The codebook.

Code group	Code	Description	Example
Correction	No	Tool capable only of detecting defects	Design and implementation of a deep learning-based vulnerability detection system [P25]
	Yes	Tool capable of correcting defects	End-to-end solution ... that can fix multiple such errors in a program [P2]
Defect type	Syntactic	Tool targets syntax defects	Algorithm ... for finding repairs to syntax errors [P3]
	Semantic	Tool targets semantic defects	Addressing the issue of semantic program repair [P4]
	Vulnerability	Tool targets vulnerabilities	System for vulnerability detection [P25]
Representation	Tokens	Source code represented as a sequence of tokens	Model treats a program statement as a list of tokens [P1]
	AST	Source code represented as an abstract syntax tree	Representations of ASTs [P22]
	Graph	Source code represented as a graph capturing additional semantic information (control flow graphs, data flow graphs, and so on)	Generates a system dependency graph for each training program [P27]
Language	Python	Tool evaluated on source code written in Python	From an introduction to programming in python course [P3]
	C	Tool evaluated on source code written in C/C++	Fixing common C language errors [P2]
	Java	Tool evaluated on source code written in Java	Targeting Java source code [P6]
	JavaScript	Tool evaluated on source code written in JavaScript	Broad range of bugs in JavaScript programs [P15]
	C#	Tool evaluated on source code written in C#	Open source C# projects on GitHub [P24]
Type	No bug	Tool trained on only nonbuggy source code	Using language models trained on correct source code to find tokens that seem out of place [P7]
	Bug + fixed	Tool trained on paired examples of buggy and fixed code	A pair (p; p^0), where p is an incorrect program and p^0 is its correct version [P9]
	Bug + no bug	Tool trained on unpaired examples of buggy and nonbuggy code	Data set that contains 181,641 pieces of code; 138,522 are nonvulnerable (i.e., not known to contain vulnerabilities) and 43,119 are vulnerable [P27]
Label	Yes	Tool trained on labeled data	A program is labeled as "good," ..."bad," ... or "mixed" [P27]
	No	Tool trained on unlabeled data	Self-supervised learning with unlabeled programs [P19]
Realism	Real	Data set consists of mostly real programs	Javascript code change commits collected from Github [P15]
	Semireal	Data set consists of semirealistic code: real code injected with synthetic bugs, or simpler/beginner code with real mistakes	Corpus of open source Python projects with synthetically injected bugs [P4] and C programs written by students for 93 different programming tasks [P2]
	Synthetic	Data set consists of mainly synthetic/academic code	Juliet Test Suite, with 81,000 synthetic C/C++ and Java programs with known security vulnerabilities [P31]
Availability	Yes	Data set and/or tool are publicly available	—
	No	Data set and/or tool are not publicly available	—

For conciseness, we leave out detailed descriptions and low-level comparisons among the approaches and focus on more general directions.

Detection Versus Correction Ability and Defect Types

Observation 1. We find an almost equal split among the papers that focus only on detection and those also correcting defects. Twenty-one papers focus on detecting defects, while 19 can also correct them. In terms of the defects' evolution over time, research into both types seems to be growing fast, as evident in Figure 1. The slight drop in publications of defect correction studies could be the consequence of a small sample size or an actual shift toward defect detection.

Observation 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. Seven papers target syntactic defects, 15 focus on vulnerabilities, and 18 concentrate on semantic defects. Correction studies target mostly semantic (12) and syntactic (six) defects, while detection studies target mostly vulnerabilities (14) and semantic defects (six). Only a single correction study [P5] targets vulnerabilities, and one detection study [P29] focuses on syntactic defects.

Since defect detection often targets more complex problems, such as semantic bugs and vulnerabilities, many detection papers focus on a narrower array of problems or try to narrow the granularity. As such, DeepBugs [P23] targets only name-based semantic bugs, SinkFinder [P34] examines security-sensitive function pairs, and OffSide [P35] looks for boundary condition mistakes.

Among the correction papers, [P5] presented one of the first studies requiring no paired labeled examples for mapping from the buggy domain to the nonbuggy one. Sensibility [P7] was one of the first studies focusing on the correction of single token syntax defects across domains. DeepRepair [P14] builds on the idea of redundancy, exploiting the fact that many programs contain seeds to their own repair. More advanced studies, such as Hoppity, [P15] use NNs for source code embedding and graph transformations to correct semantic mistakes. Graph2Diff [P18] and VarMisuseRepair [P13] both use pointers to locate the defect and a potential fix.

Source Code Representation

Observation 3. The majority of the studies use either ASTs or token representation, with graph representation being the least used. Despite the different representations, the input is commonly flattened when serving as input for an NN. AST representation is used by 23 papers, token representation appears in 21 studies, and graph representation is employed by 11 studies. The approaches can coexist, which is evident from studies that combine several representations: 12 defect detection and two defect correction studies use some combination. The most common combination is AST–graph (seven), followed by AST–token (three), and graph–token (three). Zhou et al. [P38] use a combination of all the three representations. With deep learning rising compared to other ML techniques, the need for manually defined "traditional" features is falling. Instead, NNs require input in the form of a vector. To achieve that, the previously described source code representations are commonly flattened into a vector (vectorized) [P10], [P25].

Observation 4. Correction papers mostly use ASTs and tokens, whereas detection studies use all three representations. We can see a significant division in representation approaches between detection and correction studies. Among the defect correction papers, the most common representation is tokens (12), followed by ASTs (eight). Only one correction study uses graph representation [P19]. The split in representations is a bit more balanced among the detection-only papers: ASTs appear 15 times, graphs 10 times, and tokens nine times.

Observation 5. Different representations seem preferred by researchers for addressing varying types of defects, depending on the defect type targeted by a study. Tools targeting syntactic defects almost exclusively use token representation (seven), with a single paper adding graph representation [P19]. Papers aimed at semantic defects primarily use ASTs (14), followed by tokens (five) and graphs (four). The most variety in representation comes from the vulnerability finding papers. Those use ASTs and tokens equally often (nine), with graphs employed slightly less frequently (six). Vulnerability finding studies also most commonly use a combination of more than one representation.

Languages

Observation 6. The majority of the 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 (four), JavaScript (two) and C# (one). Several of the featured studies aimed to be language and syntax agnostic but were trained and tested only on a specific language. It is, however, commonly noted that such studies could be used on different languages through

Table 6. The studies and their codes.

	General				Data Set				Availability	
Tool name	Defect	Representation	Method	Language	Size	Type	Realism	Label	Data	Tool
Works That Detect and Correct Defects										
sk_p [P1]	Semantic	Token	RNN (LSTM) and skip-gram	Python	7 × 315-9,000 programs	NB	semireal	—	—	—
DeepFix [P2]	Syntactic	Token	RNN	C	7,000 programs	B + F	Semireal	✓	✓	✓
SynFix [P3]	Syntactic	Token	RNN (LSTM)	Python	75,000 programs	NB	Semireal	—	—	—
SSC [P4]	Semantic	AST	RNN and rule based	Python	2,900,000 code snippets	B + F	Semireal	✓	✓	—
Harer 2018 [P5]	Vulnerability	Token	GAN	C	117,000 functions	B + F	Synthetic	✓	—	—
Ratchet [P6]	Semantic	Token	RNN (LSTM)	Java	35,137 pairs	B + F	Real	✓	✓	✓
Sensibility [P7]	Syntactic	Token	n-Gram and RNN (LSTM)	Java	2,300,000 files	NB	Semireal	—	✓	✓
SequenceR [P8]	Semantic	Token	RNN (LSTM)	Java	35,000 samples	B + F	Real	✓	✓	✓
RLAssist [P9]	Syntactic	Token	DRL and RNN (LSTM)	C	7,000 programs	B + F	Semireal	✓	✓	✓
Liu 2019 [P10]	Semantic	AST, token	CNN and paragraph vector	Java	2,000,000 methods	B + F	Real	—	✓	✓
DeepDelta [P11]	Semantic	AST	RNN (LSTM)	Java	4,800,000 builds	B + F	Real	✓	—	—
Tufano 2019 [P12]	Semantic	AST	RNN	Java	2,300,000 fixes	B + F	Real	✓	✓	✓
VarMisuseRepair [P13]	Semantic	Token	RNN (LSTM) and pointer network	Python	650,000 functions	B + F	Real	✓	✓	—
DeepRepair [P14]	Semantic	AST	RNN	Java	374 programs	B + F	Semireal	✓	✓	✓
Hoppity [P15]	Semantic	AST	GNN and RNN (LSTM)	JavaScript	500,000 program pairs	B + F	Real	✓	✓	✓
SampleFix [P16]	Syntactic	Token	GAN, CVAE, and RNN (LSTM)	C	7,000 programs	B + F	Semireal	✓	✓	—
DLFix [P17]	Semantic	AST	RNN (tree RNN)	Java	4,900,000 methods	B + F	Real	✓	✓	✓
Graph2Diff [P18]	Semantic	AST	GNN (GGNN)	Java	500,000 fixes	B + F	Real	✓	—	—
DrRepair [P19]	Syntactic	Token, Graph	GNN and RNN (LSTM)	C	64,000 programs	B + F	Semireal	—	✓	✓

Works That Detect Defects

Wang 2016 [P20]	Semantic	AST, graph	DBN	Java	10 × 150–1,046 files	B + F	Real	✓	—	—
DP-CNN [P21]	Semantic	AST	CNN and logistic regression	Java	7 × 330 files	B + F	Real	✓	✓	—
POSTER [P22]	Vulnerability	AST	RNN (BLSTM)	C	6,000 functions	B + F	Real	—	✓	✓
DeepBugs [P23]	Semantic	AST, graph	NN	JavaScript	150,000 files	B + F	Semireal	✓	✓	✓
VarMisuse [P24]	Semantic	AST, graph	GGNN and GRU	C#	2.9 million LoC	B + NB	Real	—	✓	✓
VulDeePecker [P25]	Vulnerability	Token	RNN (BLSTM)	C	61,000 code gadgets	B + F	Synthetic	✓	✓	—
Russell 2018 [P26]	Vulnerability	Token	CNN, BoW, RNN, and random forest	C	1.27 million functions	B + F	Real + synthetic	✓	✓	—
*μ*VulDeePecker [P27]	Vulnerability	AST, graph	RNN (BLSTM)	C	181,000 code gadgets	B + NF	Synthetic	✓	✓	—
Gupta 2019 [P28]	Semantic	AST	Tree CNN	C	29 × 1,300 programs	B + F	Semireal	✓	✓	✓
Habib 2019 [P29]	Syntactic	Token	RNN (BLSTM)	Java	112 projects	B + F	Synthetic	✓	—	—
Li 2019 [P30]	Semantic	AST, graph	RNN (GRU) and CNN	Java	4.9 million methods	B + F	Real	✓	✓	✓
Project Achilles [P31]	Vulnerability	Token	RNN (LSTM)	Java	44,495 programs	B + F	Synthetic	✓	✓	✓
Li 2020 [P32]	Vulnerability	Graph, token	BoW and CNN	C	60,000 samples	B + NB	Synthetic	—	—	—
VulDeeLocator [P33]	Vulnerability	AST, token	RNN (BRNN)	C	120,000 program slices	B + NB	Synthetic	✓	✓	✓
SinkFinder [P34]	Vulnerability	Graph, token	SVM	C	15 million LoC	NB	Real	—	✓	—
OffSide [P35]	Vulnerability	AST	Attention NN	Java	1.5 million code snippets	B + F	Semireal	✓	✓	✓
AI4VA [P36]	Vulnerability	AST, graph	Graph NN	C	1.95 million functions	B + F	Real + synthetic	✓	✓	✓
Tanwar 2020 [P37]	Vulnerability	AST	NN	C	1.27 million functions	B + F	Real	✓	—	—
Devign [P38]	Vulnerability	All	Graph NN	C	48,000 commits	B + F	Real	✓	✓	✓
Dam 2021 [P39]	Vulnerability	AST, token	RNN (LSTM)	Java	18 × 46–3,450 files	B + NB	Real	✓	—	—
SySeVR [P40]	Vulnerability	AST, graph	RNN (BLSTM and BGRU)	C	15,000 programs	B + F	Synthetic	✓	✓	✓

RNN: recurrent NN; LSTM: long short-term memory; GAN: generative adversarial network; DRL: deep reinforcement learning; CNN: convolutional NN; GNN: graph NN; CVAE: conditional variational autoencoder; GGNN: gated GNN; DBN: deep belief network; BLSTM: bidirectional LSTM; GRU: gated recurrent unit; BoW: bag of words; BRNN: bidirectional RNN; SVM: support vector machine; BGRU: bidirectional GRU; NB: no bug; B: buggy; F: fixed.

The studies are first ordered chronologically and then alphabetically (by author name) within the top and bottom halves of the table.

In the "Method" column, we refer to the primary ML approach used in the tool. When a tool experiments with several approaches, we include all of them if they are presented and discussed equally and skip the ones mentioned only in passing.

minimal changes to the models and by retraining on a suitable data set.

Observation 7. We see a nonuniform distribution of goals across the examined languages both in terms of correction ability as well as targeted defect types. Looking at correction ability, we notice that the majority of C studies (12) only detect defects, while five can correct them. Java is more balanced, with seven detection and nine correction studies. JavaScript has one paper for correction [P15] and one for detection [P23]. All four Python studies are capable of correction. Finally, the one examined C# paper [P24] can detect defects. Overall, the two most common are defect detecting C studies (12) and defect correcting Java studies (nine).

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 (three), with one paper targeting syntactic defects. The two examined JavaScript studies as well as the only C# study target semantic defects. There are no Python, JavaScript, or C# papers that focus on security vulnerabilities. Similarly, no JavaScript or C# paper detects or corrects syntax defects.

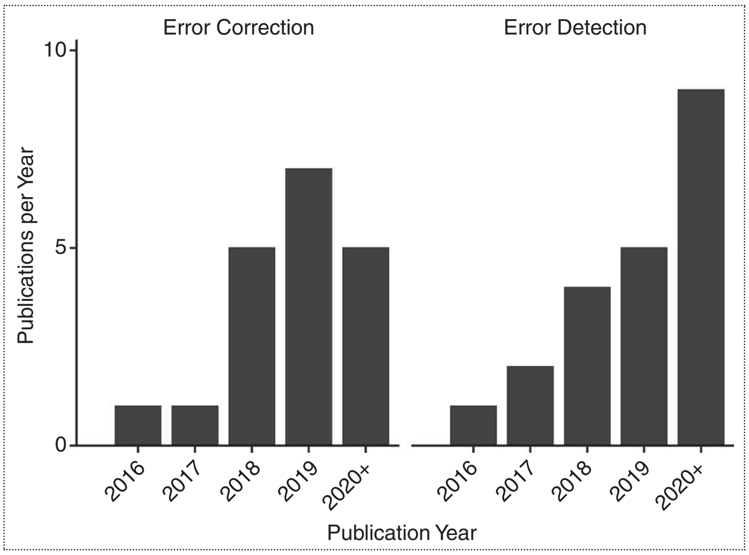

Figure 1. A histogram of publications per year. Note that 2020 and 2021 are merged as 2020+.

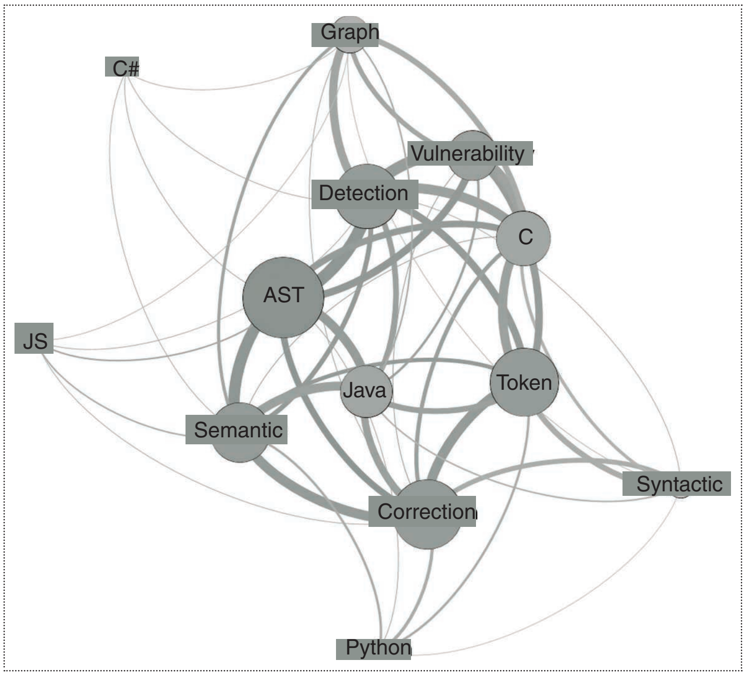

Figure 2. A co-occurrence graph of tool characteristics.

ML Approaches/Models

Observation 8. Both defect detection and correction studies increasingly rely on NNs. The most commonly used model is the recurrent NN (RNN). Defect correction studies heavily borrow from natural language translation, often referred to as *neural machine translation* or *sequence-to-sequence translation*. This means that the majority of the models comes from the same domain, more specifically, RNNs that appear 16 times out of 19 among defect correction papers. The most common method within the RNN family is long short-term memory (LSTM)—11 studies—which specifically targets the problem of long-term dependencies by enabling learning from context.

The most recent papers highlight the usefulness of NNs that are capable of understanding contexts since the presence of a defect can highly depend on that [P30]. Additionally, attention (focusing on the relevant parts of the code, depending on the context) helps such NNs learn long-distance relations to keep track of issues outside 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 RNNs, and four use convolutional NNs. Most of the remaining papers still rely on some member of the NN family [e.g., attention NNs, (gated) graph NNs, deep belief networks, and so on]. Similar to defect correction studies, methods that can learn from context, such as bidirectional LSTM—five papers—and the gated recurrent unit—three papers—are popular due to their ability to take into account both future and past contexts [P25].

There is only slightly more variety in the defect detection world, where the task can (but does not need to) be logically split in two: embedding/feature extraction and classification. While the former is mostly handled by a form of NN, the latter invites more experimentation. Some of the classification methods include logistic regression, bags of words, random forests, and support vector machines. Despite some outliers, the task of detection also seems to be heading in the NN direction. The analyzed papers commonly attribute this to the NN's ability to operate without explicit feature formation, capacity to understand contexts and keep some form of memory over time, and suitability for handling texts and (a form of) language.

Data Sets

Observation 9. There is large disparity among data sets in terms of data set size and data unit size. The sizes of the data sets range from hundreds to millions of data units. Data units themselves (i.e., the source code snippets fed into the model) also range from full program files to methods, functions, code gadgets, and similar paper-specific granularities. We notice that the granularity of data points commonly coincides with the output granularity at which the tool is capable of spotting defects.

Observation 10. There are significant differences in source code complexity, realism, and origin. On the one hand, we have real source code (18 studies), often collected from Github and open source projects. On the other hand, we find eight papers that use primarily synthetic data sets, which consist of shorter and cleaner code samples with "textbook" examples of errors. The remaining data sets fall somewhere in the middle, consisting of either real source code with artificially injected errors (four) or simple code segments and student assignments with genuine mistakes (eight). Two studies, Russell et al. [P26] and Suneja et al., [P36] separately train and evaluate on both real and synthetic data. Regardless of the realism, the studies often source their data from publicly available data sets and previous studies. Such data sets include the Software Assurance Reference Dataset, National Vulnerability Database, Juliet Test Suite, and Draper.

Observation 11. Correction tools mostly use real and semireal data, while detection tools use both real and synthetic data. Additionally, tools targeting vulnerabilities mostly employ synthetic

Table 7. The co-occurrence table.

		Correction		Error			Representation			Language					Real		
		No	Yes	Semireal	Synthetic	Vulnerability	AST	Graph	Token	C	C#	Java	JavaScript	Python	Real	Semireal	Synthetic
Correction	No (21)			6	1	14	15	10	9	12	1	7	1	0	9	3	7
	Yes (19)			12	6	1	8	1	12	5	0	9	1	4	9	9	1
Error	Semireal (18)	6	12				14	4	5	1	1	11	2	3	13	5	0
	Synthetic (7)	1	6				0	1	7	4	0	2	0	1	0	6	1
	Vulnerability (15)	14	1				9	6	9	12	0	3	0	0	5	1	7
Representation	AST (23)	15	8	14	0	9				8	1	11	2	1	14	5	3
	Graph (11)	10	1	4	1	6				7	1	2	1	0	5	2	3
	Token (21)	9	12	5	7	9				11	0	7	0	3	7	7	6
Language	C (17)	12	5	1	4	12	8	7	11						4	5	6
	C# (1)	1	0	1	0	0	1	1	0						1	0	0
	Java (16)	7	9	11	2	3	11	2	7						11	3	2
	JavaScript (2)	1	1	2	0	0	2	1	0						1	1	0
	Python (4)	0	4	3	1	0	1	0	3						1	3	0
Real	Real (18)	9	9	13	0	5	14	5	7	4	1	11	1	1			
	Semireal (12)	3	9	5	6	1	5	2	7	5	0	3	1	3			
	Synthetic (8)	7	1	0	1	7	3	3	6	6	0	2	0	0			

data sets; semireal data are common with syntactic errors and real data with semantic errors. Among the correction tools, real and semireal data sets are used equally often—nine times—with only one study using synthetic data sets. Detection-only tools primarily use realistic data (nine), but synthetic data sets are also popular (seven). Semirealistic data are the least popular among the error detection tools, with only three occurrences.

We also notice distinct patterns of data sets used for different types of errors. Synthetic data are almost exclusively used in tools targeting vulnerabilities (seven). Semireal data are mostly harnessed in studies related to syntactic errors (six) and semantic errors (five) and less in studies related to vulnerabilities (one). Finally, real data are employed in semantic error (13) and vulnerability (five) studies but not in syntactic error studies.

Observation 12. The majority of data sets consists of bug fix pairs. We notice three distinct patterns in data set structure: data sets with bug fix pairs (31), data sets of unrelated buggy and nonbuggy examples (five), and data sets with no bugs (four). The latter 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 an unfamiliar code pattern. The remaining two patterns help 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 commonly achieved by either collecting version histories (commits with fixes) from public repositories or artificially injecting bugs to correct code. In case of defect detection, it is not crucial to have such pairs, so several data sets include examples of bugs and correct code but not necessarily on the same piece of code.

Output and Performance

Observation 13. There is little uniformity among studies' outputs in terms of granularity and error types a tool can target. We notice a significant variety of detection granularity, ranging from simple binary classification (buggy versus nonbuggy) to method, function, and specific lines of code. For example, Dam et al. [P39] focus on file-level detection, VulDeePecker [P25] works on code gadget granularity, and Project Achilles [P31] concentrates on methods. An interesting goal was set by Zou et al. [P27] The authors attempted not only to recognize whether there was a vulnerability with fine granularity but also determine the vulnerability type. There are similar differences among the correction studies that range from single token correction all the way to full code sections, sometimes as a single-step fix or as a collection of smaller steps with some form of correction checking in between.

There are also differences in how many different error types a tool can handle. Some tools are trained and tested on a smaller set of vulnerability types, which makes them narrow but comparatively high performing. Examples of such tools include SinkFinder, [P34] which looks for vulnerabilities in function pairs, such as lock/unlock; OffSide, [P35] which focuses on boundary conditions; and VulDeePecker, [P25] which targets buffer and resource management error vulnerabilities. On the other hand, some tools target a wide range of errors, potentially at some performance cost. SySeVR, [P40] for example, targets 126 vulnerability types, and Project Achilles [P31] focuses on 29. It is worth mentioning that some tools train separate models for each type of error and evaluate a piece of code by passing it through each of the trained models separately to determine the probability of each of the vulnerabilities.

Observation 14. There are significant inconsistencies in the reporting of performance metrics. Studies using real data sets seems to perform worse than those using synthetic data sets. We find that the studies differ greatly in their reporting of performance. The most commonly reported metrics include recall (reported in some form by 22 studies), the F1 score (16), accuracy (15), and precision (11). While detection-only tools tend to be more diligent in their reporting, the correction tools more commonly frame their results simply as "we could fix *x* out of *y* errors" without providing more detail. We find additional inconsistencies even among the studies that report the same metrics: some relay only the best performance, others provide average values, and others convey the full range.

Taking all this into account, it is uninformative, if not misleading, to directly compare performance across the papers. However, setting aside all nuances, we can cautiously draw some rough patterns from the metrics reported. Specifically, we find that studies using synthetic data sets generally report higher metrics regardless of the other study properties (around 80–90% for all mentioned metrics), while studies using real and semireal data perform significantly worse (their accuracy and recall rarely exceed 60–70%), have wider ranges, and sometimes dip all the way down to 0–20%. Given the previously identified relations among correction ability, error type, and representation, the performance across those categories is also affected by the realism of the data set.

An interesting insight into the effects of data set realism is provided by Russell et al. [P26] and Suneja et al., [P36] who train and test their pipelines separately on real and synthetic data sets. The two studies enable us to get a glimpse at the behavior of the same tool when faced with different types of source code. Both papers exhibit the same pattern we observed: the F1 score is significantly lower when realistic data are used. More specifically, the studies report F1 scores of 50–60% on real data and 70–90% for synthetic data.

Discussion

There are significant differences among the studies when it comes to the error types that are targeted, leading to different defect patterns and, consequently, representation choices. All 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 vulnerabilities being the most challenging. Seeing that most of the correction tools address the former, while detection tools largely address the latter, we can assume that effective correction is more difficult to achieve. With several detection and correction tools targeting semantic defects, we speculate that such defects lie in the middle in terms of difficulty.

“Arguably, the simplest defect type to catch is a syntactic one, with vulnerabilities being the most challenging.”

We can find additional support for such observations when looking at data set realism. Fully synthetic data sets are used primarily by vulnerability detection tools, suggesting that is not yet possible to detect realistic vulnerabilities “in the wild.” It is worth noting that some of the vulnerability detection tools use real-world projects and successfully catch vulnerabilities, but this cannot be effectively done on a large scale and without a high number of false classifications.

Tools targeting syntactic errors use semirealistic data, in particular, simple code snippets written by students for beginner programming courses and that have genuine but simple mistakes. The use of such data sets seems only natural, as syntax problems are common with beginner programmers, who cannot yet catch and correct their mistakes. Finally, we find the use of real and semireal data with the tools aimed at semantic errors. The semireal data sets that were used mostly consist of realistic source code injected with artificial errors.

We see that the complexity of the used data set reflects common use cases as well as the complexity of the targeted error type, 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 in a student program that does not even compile. It seems that the performance goes hand-in-hand with the realism of the code. Generally, we find better performance with tools using synthetic data, even when the goal is more challenging (e.g., dozens of different vulnerability types). A similar pattern has been documented by Chakraborty et al.[7] More research is required to confirm such patterns, but present evidence highlights how crucial the use of appropriate, realistic, and well-labeled data is. The field should be wary of high-performance reports, especially when synthetic data sets are used, and work instead toward more realistic goals that will make tools practical in the real world.

Similar to the data sets, it is useful to consider the full picture when discussing 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 practical applications where the tools are applied on a large number of real-world projects. Overall, the importance of lower granularity and higher precision is recognized and often highlighted, with the trends moving toward more precise tools.

Patterns in source code representation seem to follow defect type patterns and, in turn, the detection and 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 and 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. They are especially useful for representing programs with syntactic defects in which constructing ASTs and control flow graphs is limited or not 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 by translating a problematic sequence into a syntactically correct one.

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 that is targeted, a model can benefit from simple, stand-alone tokens and from grouped and more structured representation (code gadgets, functions, or some other syntactic or semantic unit).

Syntactic representation considers the ASTs of the source code, enabling a less flat view of the code. Such representations are larger in size and more difficult to construct but can capture lexical and syntactic code properties. They are often combined with recursive NNs and LSTM models. Their popularity lies mainly with 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 and large and complex pieces of code very well.[12] 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 found that C#, JavaScript, and Python lack tools aimed at detecting and correcting vulnerabilities. The possible reason for prevalence of 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. This also extends to other popular languages that did not appear in the study.

A look at the ML methods highlights the fact that traditional ML approaches are more of a stepping stone toward a deep learning solution than solutions of their own. The reason likely lies in the fact that it is difficult to define the features that will sufficiently capture the semantics of the program. The main benefit of deep learning is its ability to ingest the source code itself (in an appropriate format) and create its own "features" to learn from.

Challenges and Future Directions

This article is motivated by the need to discover patterns in the rapidly evolving field of ML for source code. Some of the challenges toward effective solutions (Table 8) include access to and use of high-quality training data sets with realistic, representative, and correctly labeled data; effective source code representation capable of semantic understanding; standardization in terms of goals and reporting; detection and correction across domains; and catching application-specific bugs (in regard to semantic defects) and high FP rates. We briefly elaborate on some of these challenges.

There is significant variety in terms of data sets, goals, testing, and performance reporting. We believe the field would benefit from some degree of standardization, potentially in the form of a curated collection of open source data sets, together with some uniform goals for each defect type along with a test suite and benchmarks. Since a tool's performance can heavily rely on the training data, stabilizing the data set would enable more precise evaluation of the tool itself rather than the training data. Such data sets would ideally consist of realistic source code with representative errors and high-quality labeling to increase the usability of the tools in the real world. The formalization, or at the very least, clear reporting, of goals (e.g., in terms of granularity and defect types) would also enable researchers in the field to get a clearer and more complete picture of the available tools.

Finally, there is a need for clearer and more complete reporting of performance. One step in the right direction could be the reporting of the four basic metrics (TP, TN, FP, and FN), which facilitate the calculation of the remaining metrics. However, at the end

Table 8. The key takeaways.

Finding	Observation	Challenge
Missing detection or correction tools for some language–defect combinations	2 and 7	Expand correction and detection tools for all defect types
Variety of representation techniques but struggling to capture deeper properties of code; oversimplistic embeddings	4, 3, and 5	Advanced (semantic) representations and embeddings
Java and C/C++ most studied languages	6	Expand to more languages
Tool outputs not comparable	13 and 14	Formalize goals and metrics for tools and simplify output for developers
Vast differences in data sets and performance	9, 10, and 11	Collect, standardize high-quality, realistic, and representative data sets across all defect types and languages

of the day, such metrics tell us little about the usability of a tool to its intended users—the developers—who should be more closely involved in the testing and evaluation. Future research in the domain should also consider expansions to other commonly used programming languages, improve defect localization precision, and provide 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 word-to-vector technique,[13] which is primarily a simple token embedding technique. One then wonders: Why bother with all the complex representations 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 *x*-to-vector-like vectorization techniques capable of capturing deeper properties of code and, as is the current trend, finding overfitting with the particular data set that is used.

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, 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, graph-based representations capable of capturing complex characteristics of the analyzed programs seem particularly promising.

Finally, the relatively small number of tools working with unlabeled data points shows 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 data sets of unlabeled corpora, eliminating the need for synthetic bug introduction and manual self-labeling. ■

Acknowledgments

This work was partly supported by the European Union, under the Horizon 2020 research and innovation program (Assurance and Certification in Secure Multi-Party Open Software and Services project grant 952647), and The Netherlands' Sectorplan program.

References

1. H. Shen, J. Fang, and J. Zhao, "Efindbugs: Effective error ranking for findbugs," in *Proc. 2011 4th IEEE Int. Conf. Softw. Testing, Verification Validation*, pp. 299–308, doi: 10.1109/ICST.2011.51.
2. I. Pashchenko, R. Scandariato, A. Sabetta, and F. Massacci, "Secure software development in the era of fluid multi-party open software and services," in *Proc. 2021 IEEE/ACM 43rd Int. Conf. Softw. Eng., New Ideas Emerg. Results (ICSE-NIER)*, pp. 91–95, doi: 10.1109/ICSE-NIER52604.2021.00027.
3. M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 332–343, doi: 10.1145/2970276.2970347.
4. M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 1–24, 2018, doi: 10.1145/3105906.
5. J. Li, P. He, J. Zhu, and M. Lyu, "Software defect prediction via convolutional neural network," in *Proc. Int. Conf. Softw. Quality, Rel. Secur.*, 2017, pp. 318–328, doi: 10.1109/QRS.2017.42.
6. C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019, doi: 10.1145/3318162.
7. S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Trans. Softw. Eng.*, vol. 1, no. 1, pp. 1–17, 2021, doi: 10.1109/TSE.2021.3087402.
8. Z. Shen and S. Chen, "A survey of automatic software vulnerability detection, program repair, and defect prediction techniques," *Secur. Commun. Netw.*, vol. 2020, no. 1, pp. 1–16, 2020, doi: 10.1155/2020/8858010.
9. F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970, doi: 10.1145/390013.808479.
10. F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. 2014 IEEE Symp. Secur. Privacy*, pp. 590–604, doi: 10.1109/SP.2014.44.
11. M. Allamanis, E. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 1–37, 2018, doi: 10.1145/3212695.
12. J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proc. 2019 IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, pp. 783–794, doi: 10.1109/ICSE.2019.00086.
13. T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. 26th Int. Conf. Neural Inf. Process. Syst.*, 2013, pp. 3111–3119, doi: 10.5555/2999792.2999959.

Tina Marjanov is a research assistant at the University of Cambridge, Cambridge, CB3 0FD, The United Kingdom. Her research interests include cybersecurity, privacy, and machine learning. Marjanov received an M.S. in computer science from Vrije Universiteit Amsterdam and the University of Amsterdam. Contact her at marjanov.tina@gmail.com.

Ivan Pashchenko is a security manager at TomTom, Amsterdam, 1011 AC, The Netherlands. His research interests include threat intelligence, open source software security, and machine learning for security. Pashchenko received a Ph.D. from the University of Trento. In 2017, he was awarded a Second Place Silver Medal at the Association for Computing Machinery/Microsoft Student Research competition in the graduate category. He was the UniTrento main contact in the Continuous Analysis and Correction of Secure Code work package for the Horizon 2020 Assurance and Certification in Secure Multi-Party Open Software and Services project. Contact him at ivan.pashchenko@tomtom.com.

Fabio Massacci is a professor at the University of Trento, Trento, 38123, Italy, and Vrije Universiteit Amsterdam, Amsterdam, 1081 HV, The Netherlands. His current research interest is in empirical methods for cybersecurity of sociotechnical systems. Massacci received a Ph.D. in computing from the Sapienza University of Rome. He participates in the Cyber Security for Europe pilot and leads the Horizon 2020 Assurance and Certification in Secure Multi-Party Open Software and Services project. For his work on security and trust in sociotechnical systems, he received the Ten Year Most Influential Paper Award at the 2015 IEEE International Requirements Engineering Conference. He is a Member of IEEE. Contact him at fabio.massacci@ieee.org.

Digital Object Identifier 10.1109/MSEC.2022.3199549