Editors: Fabio Massacci, fabio.massacci@ieee.org | Eric Bodden, eric.bodden@uni-paderborn.de | Antonino Sabetta, as@sabetta.com

Why Is Static Application Security Testing Hard to Learn?

Padmanabhan Krishnan and Cristina Cifuentes | Oracle Labs
Li Li | Beihang University
Tegawendé F. Bissyandé and Jacques Klein | University of Luxembourg

In this article, we summarize our experience in combining program analysis with machine learning (ML) to develop a technique that can improve the development of specific program analyses. Our experience is negative. We describe the areas that need to be addressed if ML techniques are to be useful in the program analysis context. Most of the issues that we report are different from the ones that discuss the state of the art in the use of ML techniques to detect security vulnerabilities

While issues such as relevant datasets and representation of program semantics are common, our focus is on enhancing vulnerability detection by combining static analysis and ML approaches.

Static application security testing (*SAST*) is a methodology that statically examines source code to find security flaws that make the application susceptible to attack. SAST is popular because it can detect security vulnerabilities already in the early stages of the software development lifecycle. The static analysis can be integrated into a continuous integration/continuous delivery pipeline, thus automating the checks during the build process. While this is effective for deploying existing analyses, the process of developing new analyses is still manual: whenever a new defect or vulnerability type needs to be supported, an expert in static analysis needs to extend the existing framework to detect the new vulnerability type. This can be laborious and time-consuming, as one has to check that the new analysis has the desired accuracy, that it does not introduce any regressions to other analyses already deployed in production, and that it does not adversely affect the performance of the deployed products. Ideally, one would want to automate the generation of such analyses to make them available faster.

ML has been applied to perform security analysis tasks that are currently performed using static analyzers.[15] In particular, ML techniques have been used to learn to solve SAST problems. Actually, ML techniques have already become popular in the context of mimicking specific program analyses, such as symbolic execution. Although the results in such domains are impressive, they are, unfortunately, not generalizable to automatically learn static checkers, especially for security analysis. Even deep learning techniques have focused on relatively simple defect types, such as incorrect operators or assignments. Based on our experience in using different ML techniques for vulnerability detection, here we describe our insights into why it is hard to learn to solve SAST problems.

Learn to Solve Static Analysis Tasks: Our Initial Attempts

In our past work, we have combined program analysis with ML, aiming to enable the ML technique to learn from existing program analysis approaches to improve future program analysis. In our experience, any simple combination of the two techniques does not work. ML by itself merely echoes its result with additional errors. Indeed, it often requires a program analysis to run in the first place and recognizes the output of the analysis as being reliable.[3] If we do not involve program analysis, the ML-based classifier can only use the input source code or its direct intermediate representation, like opcodes, to represent the program's semantics, such as loading data to a variable or calling a function (both considered as a sequence of words) to train and subsequently predict vulnerable code. Unfortunately, the tokens of the source code, such as opcodes, expressions, and statements, are generic

Digital Object Identifier 10.1109/MSEC.2023.3287206
Date of current version: 11 September 2023

 Copublished by the IEEE Computer and Reliability Societies 1540-7993/23©2023IEEE

syntactic constructs that alone do not represent the semantics of vulnerable code when directly applied to ML approaches.[2] While there are existing works that represent the code sequence with more advanced data structures, such as trees or graphs, e.g., using abstract syntax trees or program dependence graphs,[13] these statically extracted representations are not sufficient also to capture semantics related to the program's runtime behavior, such as values of expressions and operations on the heap. Whether ML techniques and lightweight program analysis techniques can be combined to have a technique that is comparable to a custom program analysis technique is still an open question.

For such techniques to be useful in practice, they have to work on real-world codebases, not toy programs. Our experience in using ML-based approaches on codebases with several millions of lines of code, such as Open Solaris to detect vulnerabilities in the C code, was not successful.[3] Since there is no ground truth in such large codebases, it is nontrivial to compute an F-score to evaluate the actual capability of the ML approaches. In the experiments we ran, the ML technique generated 250 times the number of potential vulnerability reports compared to a static analysis tool. However, the generated reports were all false positives, providing no useful information to the developers. This evidence experimentally shows that it is indeed nontrivial to learn ML approaches to handle real-world vulnerability detections.

Another key distinguishing feature between program analysis techniques and ML-based techniques is explainability. Program analysis techniques typically generate an abstract trace, also known as a *potential witness path*, derived from data-flow analysis to explain why a value generated at a program point can have a detrimental effect on an operation at another program point. While explainable ML is an active area of research, its focus is on generating explanations of the characteristics of the model that resulted in the observed output. No ML is able to produce abstract traces of program behaviors. This is related to the fact that the ML models do not capture the execution semantics of programs (i.e., runtime behavior). There are ML techniques that accept traces as input, but none of them, as yet, can generate traces from programming language models.

Whether ML techniques and lightweight program analysis techniques can be combined to have a technique that is comparable to a custom program analysis technique is still an open question.

Understanding the Limitations of ML Applied to Static Analysis

In this section, we summarize, at a high level, the reasons the ML-based techniques fare poorly. These results are based on our experience of exploring such techniques in different domains, including misuse of cryptographic application programming interfaces (APIs) in Java programs and detecting memory-related issues in C programs.

Labeling Issue: Learning Through Crowd-Sourcing Solutions Is Not Feasible

The usual ML problem of having sufficient labeled data is a challenge. Creating such labeled data of a large corpus of code with annotated vulnerabilities can hardly be automated. It is unclear whether creating such labeled data is any cheaper than writing a specific program analysis. Solutions, such as crowd-sourcing (e.g., learning from existing benchmarks collected by different teams from different code repositories), which works for very simple problems, do not typically work in the context of security analysis: in fact, it can be hard to get consensus on vulnerable code purely through such crowd-sourced datasets (e.g., such datasets per se may suffer from quality issues[14]). Our work on the misuse of cryptographic APIs[4] shows that the level of expertise required to identify proper and improper uses is quite high. While researchers have shown that it is possible to learn rules from code changes to fix incorrect cryptographic API usages, it is nevertheless hard to automate the process to achieve a comparable set of rules manually summarized by humans. By checking the updates of API usage in the evolution of 40,000 real-world Android apps, we have experimentally found that cryptographic APIs are widely misused in practice. Such misuses are not even regularly fixed by app developers.

This labeling issue also relates to a "definition issue," where several security-related concepts are not well-defined. In contrast with other traditional classification or detection tasks on which artificial intelligence techniques perform extremely well, several important security concepts are difficult to define properly, and they are often context-dependent. Researchers and analysts still require a lot of effort and expertise to check if a given warning is actually a malicious piece of code or a vulnerability.

Semantics Issue: Learning Code Semantics Is Hard

While pretrained models like CodeBERT[1] and Graph CodeBERT offer

promising new code representations (i.e., new embeddings for code), they still only capture the structural aspects of the code. They do not quite capture the runtime semantics of the code, especially for arithmetic expressions and operations on arrays. Other approaches like JSNice[5] do guess the semantics, but that is only in the context of variable renaming; i.e., they do not deal with the semantics of the instructions in the code. Yamaguchi et al.[13] propose a novel code representation approach called *code property graph* that merges abstract syntax trees, control flow graphs, and program dependence graphs into a joint data structure, representing the semantics more comprehensively. Their approach, however, requires performing complicated program analysis already (e.g., to build data-flow analysis) and it is hard to retain context-sensitivity information (which has been considered important for purely static program analysis approaches).

Assessment Issue: In the Lab Versus in the Wild

It is not rare to read papers proposing a new ML-based approach to solve a given security problem, for instance, malware detection, showing impressive performance scores, sometimes up to 0.99.[8] We have shown in in Allix et al.[6] that most of these approaches suffer from assessment issues. Indeed, many approaches are assessed with what we call *in the lab* validation scenarios, i.e., a combination of 10-fold cross-validation and a limited dataset. We demonstrated the limitations of such a validation scenario. In particular, we showed that 10-fold cross-validation on the usual sizes of datasets presented in the literature is not a reliable performance indicator for realistic malware detectors "in the wild." With Tesseract, Pendlebury et al.[7] confirmed our findings and introduced the notions of *spatial bias* (distributions of training and testing data that are not representative of a real-world deployment) and *temporal bias* (incorrect time splits of training and testing sets). In the context of program semantics, focusing only on specific datasets like big data clone benchmarks,[10] semantic clone bench,[11] or using examples from GoogleCodeJam[12] seems to yield good results. But these articles do not investigate the case when a model is generated on one benchmark and is used on a different set of benchmarks. Thus, is it not possible to estimate the generalizability of the approaches.

Understanding the Dataset's Diversity Is Challenging

Another important aspect of the evaluation of learning-based techniques is the diversity of data within the dataset (e.g., to what extent has the dataset covered the landscape of the concerned problems). For instance, in the context of vulnerability detection, the commonly used dataset contains only code fragments that are related to vulnerabilities. Hence, any evaluation that uses only that dataset is potentially misleading. It is important to use datasets that also have nonvulnerability-related code fragments. Furthermore, the number of nonvulnerability-related code fragments must be much higher than vulnerability-related code fragments. This will determine if the proposed technique is actually applicable in practice. That is, the technique must be able to distinguish vulnerability-related code from nonvulnerability-related code where most of the code is not vulnerable.

Lack of Explainability

Program analysis approaches usually yield warnings with relevant data-flow traces and even change recommendations that are often useful for users to understand the problem or fix the issues. This level of explainability as to why the program analysis determined that a particular statement in the code is an issue is not available when performing ML classifiers, they only report there is likely a vulnerability but do not explain why it is regarded as such. Therefore, we argue that, in order to make ML approaches more useful in practice, it is important to develop explainable ML techniques.

It is indeed hard to train ML-based security static checkers. We have identified four main reasons that make learning static security checkers challenging: labeling, semantics, assessment issues, and explainability. The labeling issue can be overcome by putting more effort into building reliable artifacts, sharing annotated datasets, releasing tools, etc. This is still too rarely done in the security community. The semantics issue can be addressed by developing new advanced code representation techniques, for instance, by embedding semantically rich information such as value-flow graphs. Regarding the assessment issue, we strongly invite researchers to adequately assess their approach to match practical and realistic constraints. Finally, the lack of explainability is a tough area of research where we invite researchers to develop techniques to generate traces from programming language models.

Moreover, while it is nontrivial to automatically learn to generate fully functional SAST approaches, we argue that it might still be feasible to generate partial solutions, e.g., only using ML to generate modules (i.e., type inference module of a static analysis approach) that are actually suitable for ML approaches. These partial modules could then be integrated into program analysis approaches to enable better performance, which cannot be achieved using program analysis techniques alone. Our fellow researchers have recently demonstrated the feasibility of implementing that.[9] They have proposed an approach that leverages deep learning techniques to infer types for Python programs

and then integrates the outcomes into a program analysis approach to validate and refine the results. Static analyzers could further leverage this ML-generated type data to support more advanced program analyses, such as context-aware data-flow analysis. We invite the research community to further explore this exciting research direction. ■

References

1. Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Findings Assoc. Comput. Linguistics, EMNLP*, 2020, pp. 1536–1547, doi: 10.18653/v1/2020.findings-emnlp.139.
2. T. Chappell, C. Cifuentes, P. Krishnan, and S. Geva, "Machine learning for finding bugs: An initial report," in *Proc. IEEE Workshop Mach. Learn. Techn. Softw. Qual. Eval. (MaLTeSQuE)*, 2017, pp. 21–26, doi: 10.1109/MALTESQUE.2017.7882012.
3. Y. Zhao, X. Du, P. Krishnan, and C. Cifuentes, "Buffer overflow detection for C programs is hard to learn," in *Proc. Companion (MLPL) ISSTA/ECOOP*, 2018, pp. 8–9, doi: 10.1145/3236454.3236455.
4. J. Gao, P. Kong, L. Li, T. F. Bissyandé, and J. Klein, "Negative results on mining crypto-API usage rules in android apps," in *Proc. IEEE/ACM 16th Int. Conf. Mining Softw. Repositories (MSR)*, 2019, pp. 388–398, doi: 10.1109/MSR.2019.00065.
5. V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from 'Big Code'," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 111–124, Jan. 2015, doi: 10.1145/2775051.2677009.
6. K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State, and Y. Le Traon, "Empirical assessment of machine learning-based malware detectors for android: Measuring the gap between in-the-lab and in-the-wild validation scenarios," *Empirical Softw. Eng.*, vol. 21, no. 1, pp. 183–211, Feb. 2016, doi: 10.1007/s10664-014-9352-6.
7. F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating experimental bias in malware classification across space and time," in *Proc. 28th USENIX Security Symp. (USENIX Security)*, Santa Clara, CA, USA, 2019, pp. 729–746.
8. Y. Liu, C. Tantithamthavorn, L. Li, and Y. Liu, "Deep learning for android malware defenses: A systematic literature review," *ACM Comput. Surv.*, vol. 55, no. 8, pp. 1–36, Dec. 2022, doi: 10.1145/3544968.
9. Y. Peng et al., "Static inference meets deep learning: A hybrid type inference approach for Python," in *Proc. 44th Int. Conf. Softw. Eng. (ICSE)*, 2022, pp. 2019–2030, doi: 10.1145/3510003.3510038.
10. J. Svajlenko, I. Keivanloo, and C. Roy, "Big data clone detection using classical detectors: An exploratory study," *J. Softw., Evol. Process*, vol. 27, no. 6, pp. 430–464, Jun. 2015, doi: 10.1002/smr.1662.
11. F. Al-Omari, C. K. Roy, and T. Chen, "SemanticCloneBench: A semantic code clone benchmark using crowd-source knowledge," in *Proc. IEEE 14th Int. Workshop Softw. Clones (IWSC)*, 2020, pp. 57–63, doi: 10.1109/IWSC50091.2020.9047643.
12. W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *Proc. SANER*, 2020, pp. 261–271, doi: 10.1109/SANER48275.2020.9054857.
13. F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. Security Privacy*, 2014, pp. 590–604, doi: 10.1109/SP.2014.44.
14. Y. Zhao et al., "On the impact of sample duplication in machine-learning-based android malware detection," *ACM Trans. Softw. Eng. Methodology*, vol. 30, no. 3, pp. 1–38, May 2021, doi: 10.1145/3446905.
15. T. Marjanov, I. Pashchenko, and F. Massacci, "Machine learning for source code vulnerability detection: What works and what isn't there yet," *IEEE Security Privacy*, vol. 20, no. 5, pp. 60–76, Sep./Oct. 2022, doi: 10.1109/MSEC.2022.3176058.

Padmanabhan Krishnan is the director of research at Oracle Labs, Brisbane, QLD 400, Australia, leading a team that is working on developing suitable tools to detect and remediate security vulnerabilities. His research interests include program analysis, application security, and formal methods. Krishnan received a B.Tech. from the Indian Institute of Technology of Kanpur, India and a Ph.D. in programming languages from the University of Michigan, USA. He is a senior member of the Association of Computing Machinery and a Senior Member of IEEE. Contact him at paddy.krishnan@oracle.com.

Cristina Cifuentes is vice president of software assurance at Oracle Labs, Brisbane, QLD 400, Australia, leading a global team focusing on making application security and software assurance, at scale, a reality. Her research interests include software assurance. Cifuentes received a Ph.D. in computer science from the Queensland University of Technology. She was the founding Director of Oracle Labs, Australia in 2010 and has over 20 years of industrial experience, holds 15+ US patents, and has published over 50 peer-reviewed publications. Contact her at cristina.cifuentes@oracle.com.

Li Li is a professor with the School of Software, Beihang University, 100191 Beijing, China. His research interests include mobile software engineering and intelligent software engineering. Li received a Ph.D. in software engineering from the University of Luxembourg. He is a Senior Member of IEEE. Contact him at lilicoding@ieee.org.

Tegawendé F. Bissyandé is a chief scientist, associate professor at the University of Luxembourg, L-1359 Luxembourg, Luxembourg, where he conducts research on program debugging and repair at the Interdisciplinary Centre for Security, Reliability, and Repair. His research interests include program repair and software analytics. Bissyandé received a Ph.D. in computer science from the University of Bordeaux, France. He is a Member of IEEE. Contact him at tegawende.bissyande@uni.lu.

Jacques Klein is a full professor in software engineering and software security within the Interdisciplinary Centre for Security, Reliability, and Trust at the University of Luxembourg, L-1359 Luxembourg, Luxembourg. His main research interests are software security, software reliability, and data analytics. Klein received a Ph.D. in computer science from the University of Rennes, France. He is a Member of IEEE. Contact him at jacques.klein@uni.lu.

Lessons Learned on Machine Learning for Computer Security

Daniel Arp | Technische Universität Berlin and University College London
Erwin Quiring | ICSI and Ruhr University Bochum
Feargus Pendlebury | University College London
Alexander Warnecke | Technische Universität Berlin
Fabio Pierazzi | King's College London
Christian Wressnegger | KASTEL Security Research Labs and Karlsruhe Institute of Technology
Lorenzo Cavallaro | University College London
Konrad Rieck | Technische Universität Berlin

We identify 10 generic pitfalls that can affect the experimental outcome of AI driven solutions in computer security. We find that they are prevalent in the literature and provide recommendations for overcoming them in the future.

Artificial intelligence (AI) and machine learning have enabled remarkable progress in science and industry. This advancement has naturally also impacted computer security, with nearly every major vendor now marketing AI-driven solutions for threat analysis and detection. Similarly, the number of research papers applying machine learning to solve security tasks has literally exploded.

These works come with the implicit promise that learning algorithms provide significant benefits compared with traditional solutions. In recent years, however, different studies have shown that learning-based approaches often fail to provide the promised performance in practice due to various restrictions ignored in the original publications.[1,2,3,4] In this article, we want to ask, Are there *generic* pitfalls that can affect the experimental outcome when applying machine learning in security? If so, how can researchers avoid stepping into them?

Why Should I Care?

As a thorough researcher, one might tend to think, "This can

Digital Object Identifier 10.1109/MSEC.2023.3287207
Date of current version: 11 September 2023