Research Statement

1 Overview

With the development of information technology, artificial intelligence, and the internet of things, the cost and complexity of software systems keep increasing and it is more and more difficult for developers to maintain software quality and reliability. In 2018, poor software quality cost the global economy over US$1.7 trillion dollars. Software is an integrant part of every market sector and on average, companies spend more than 75% of their software development budget on quality control [6]. As a result, there is a high demand for effective automated bug detection tools to improve software quality and dependability.

My main research area focuses on using machine learning to improve software reliability and dependability (i.e., automatic program repair [4, 11], bug detection [7], vulnerability and malicious browser extension detection [17, 16], and software architecture recovery [10, 9]). During that time, I found that AI libraries, and deep learning libraries in particular, are often buggy, with bugs that are often hard to find with standard techniques. Furthermore, most researchers often misunderstand how much variance in accuracy and fairness DL algorithms can have. This decided me to move to a second research topic where I propose new techniques to improve the reliability of DL libraries. Under this second topic, I proposed new solutions to test DL systems using differential testing [13] and equivalent models [18], and quantified DL systems' variance in accuracy [14] and fairness [15].

My research interests are software engineering, artificial intelligence, program repair, software reliability, and automated software testing. My research contributions included:

- New deep-learning approaches that combine a new context-aware architecture and ensemble learning to automatically generate software patches in multiple programming languages [11]. We further extend it by including programming language syntax knowledge in the Deep Learning model using a large language model (GPT), and a new code-aware search strategy [4].
- Two new automatic approaches for testing deep learning systems. The first one uses differential testing and anomaly propagation tracking [13] while the second one leverages new graph equivalence rules [18] to automatically find bugs in deep-learning libraries.
- A new automatic approach that uses error clustering, image similarity algorithms, and delta-debugging to detect and localize bugs in electronic documents and readers [7].
- A new approach that leverages accurate software dependency for automatic software architecture recovery [9, 10].
- Two studies on variance in accuracy [14] and fairness [15] caused by software nondeterminism in DL systems.

2 Current Research

My current research lies at the cross-road of software engineering and AI. First, I've proposed DL approaches to automatically improve software reliability and dependability, focusing on program

repair, bug detection, and software architecture recovery. Second, I've been developing new testing approaches to find bugs in DL systems and libraries.

2.1 AI for software Engineering

Automatic Program Repair (APR) [11, 4]: One of the main approaches for automatic program repair is the generate-and-validate (G&V) method. G&V) APR techniques typically rely on hard-coded rules, thus only fixing bugs following specific fix patterns. These rules require a significant amount of manual effort to discover and are hard to adapt to different programming languages. I addressed this challenge by proposing CoCoNuT [11], a new G&V technique. CoCoNuT uses ensemble learning on a new context-aware neural machine translation (NMT) architecture to automatically fix bugs in multiple programming languages. To better represent the context of a bug, we introduce a new context-aware NMT architecture that represents the buggy source code and its surrounding context separately. CoCoNuT also takes advantage of the randomness in hyper-parameter tuning to build multiple models that fix different bugs and combine these models using ensemble learning. Our evaluation on six popular benchmarks for four programming languages shows that CoCoNuT fixed 493 bugs, including 329 bugs that were fixed by none of the 23 APR techniques we compared with.

We further extend CoCoNuT with CURE [4] by (1) proposing a new APR architecture and pre-training a programming language model (GPT) on 4.04 million methods to capture code syntax and developer-like source code and (2) proposing a new beam-search strategy to integrate domain knowledge (i.e., which identifiers are valid) into the patch generation process. Overall, this extension of CoCoNuT outperforms all existing approaches on the Defects4J and QuixBugs Java benchmarks fixing 57 and 26 bugs respectively.

Beyond program repair, AI can be used to help automate and improve many other software reliability tasks. This is demonstrated by the additional work below where I used AI to find bugs [7], recover software architecture [9, 10], and detect malware [16, 17].

Studying and Detecting bugs in PDF readers and Files: [7] Electronic documents are widely used to store and share information such as bank statements, contracts, articles, maps and tax information. Many different applications exist for displaying a given electronic document, and users rightfully assume that documents will be rendered similarly independently of the application used. However, this is not always the case, and these inconsistencies, regardless of their causes—bugs in the application or the file itself—can become critical sources of miscommunication. In this work, we present a study on the correctness of PDF documents and readers. First, we manually investigated a large number of real-world PDF documents and find that such inconsistencies are common—13.5% PDF files are inconsistently rendered by at least one popular reader. We then propose an approach to detect and localize the source of such inconsistencies automatically. We evaluate our automatic approach on a large corpus of over 230K documents using 11 popular readers (Acrobat Reader, Chrome, Evince, etc.) and our experiments have detected 30 unique bugs in these readers and files, 18 of which got fixed or confirmed by developers.

Software Architecture Recovery [9, 10] Many techniques have been proposed to automatically recover software architectures from software implementations, however, little work has been done on improving the input of such techniques. In this work, we first propose to leverage more accurate symbol dependencies and study their impact on the accuracy of architecture recovery techniques. We also evaluate other rarely considered factors (e.g., granularity level, dynamic binding graph construction) to understand their impact on architecture recovery techniques. We found that while using accurate symbol dependencies improves recovery quality, most architecture

recovery techniques did not perform better than the simple baseline we proposed, suggesting that more accurate recovery techniques are needed.

Malware Detection [16, 17]: Unsafely coded browser extensions can compromise the security of a browser, making them attractive targets for attackers as a primary vehicle for conducting cyber-attacks. We proposed a model-based approach to detect vulnerable and malicious browser extensions. Our evaluation indicated that the approach not only detects known vulnerable and malicious extensions but also identifies previously undetected extensions with a negligible performance overhead.

2.2 Software Engineering Approach for more Reliable AI

All the work above extensively uses machine learning libraries and while developing these new AI-based techniques, I found that AI libraries are often buggy, with bugs that are often hard to find with standard techniques. Furthermore, most researchers often misunderstand how much variance in accuracy and fairness DL algorithms can have. This decided me to move to a second research topic where I propose new techniques to improve the reliability of DL libraries.

Differential Testing in DL libraries [13]: Deep learning (DL) systems are widely used in domains including aircraft collision avoidance systems, Alzheimer's disease diagnosis, and autonomous driving cars. Existing DL testing work focuses on testing the DL models, not the implementations of the models. One key challenge of testing DL libraries is the difficulty of knowing the expected output of DL libraries given an input instance. Fortunately, there are multiple implementations of the same DL algorithms in different DL libraries. Thus, we propose a new technique, CRADLE, that detects bugs in DL libraries using cross-implementation inconsistency checking and localize faulty functions using anomaly propagation tracking. In our evaluation in popular DL libraries (TensorFlow, Theano, and CNTK), CRADLE detected 12 bugs and 104 unique inconsistencies, and highlights functions relevant to the causes of inconsistencies for all 104 unique inconsistencies.

Testing DL libraries using equivalent models: While the differential testing approach we proposed in [13] is effective, there are bugs that cannot be detected using such approaches because they are included in functionalities available in only one library. If a non-crash bug occurs in such functionality, none of the existing DL testing approaches can easily find it. Therefore we propose a technique called EAGLE that leverages equivalent graphs; i.e., graphs that use different APIs (e.g., different optimizations or data format) from the same library but produce equivalent output given the same input. Specifically, we designed 17 new equivalence rules to create equivalent graphs to test DL libraries. Applying these rules to PyTorch and TensorFlow, we generated 1,933 pairs of equivalent graphs, finding 20 bugs, including 9 previously unknown.

Quantifying Accuracy and Fairness variances [14, 15]: Deep learning algorithms use nondeterminism to improve training performances, therefore, training the same network twice on the same training data will result in slightly different models. Our work is the first to (1) define the sources of nondeterminism in deep learning systems, (2) measure the impact of nondeterminism on both accuracy [14] and fairness [15], and (3) quantify the awareness of deep learning researchers and practitioners on the nondeterminism problem. First, we found that even with fixed-seed, there is still up to 2.9% accuracy difference and up to 52.4% per class accuracy difference between training runs. Furthermore, we found that this nondeterminism also affects the fairness of DL systems and that most debiasing techniques (techniques used to make a DL system fairer) increase the training variance caused by nondeterminism.

3 Future Research Directions

I look forward to continuing research improving software quality and reliability. There are two directions I would like to continue doing research. First, I would like to develop new AI (deep learning, source code encoding, tree-based neural networks, etc.) and big data (mining software repository) techniques to improve automatic software engineering, as well as to help developers better understand and use existing techniques. Second, I would like to propose new software engineering techniques to improve the reliability of AI systems.

3.1 AI for improved Automatic Software Engineering

Constraint-based DL for APR: While using NMT and Deep learning for APR showed promise, there is much domain knowledge and constraints that are known to the developer (or the compiler) that are not directly passed to the deep learning model. While some of this knowledge can be learned from previous bug or real world code [4], the model (1) might not learn all of these constraints and (2) waste time learning constraints from historical data that are already known. It would be much more efficient to feed such domain knowledge (variables need to be declared before being used, source code syntax, etc.) as constraints to the model instead of having the model learn it from historical data. I propose to build a new deep learning architecture that is able to take as input, not only historical data, but additional constraints that can guide the training and the source code generation [3, 8]. Using such an approach would allow the model to focus on learning how to fix the bug instead of spending time learning baseline knowledge such as correct code syntax. This type of domain-knowledge-guided deep learning approach could apply to other software engineering tasks such as defect prediction or source code completion.

Actionable defect prediction and APR: Most work on defect prediction and APR directly apply existing machine learning algorithms (e.g., support vector machine, random forest, deep belief neural network), without much consideration about the characteristic of the task or how the output can be used by developers. For example, most defect prediction approaches simply output whether the tested code snippet is buggy, without additional information regarding how the decision was made. This is generally not very useful for developers as it does not produce any relevant information to either (1) identify the bug (e.g., test case triggering the bug) or (2) fix the bug directly. The goal of this project is to focus on usability of these approaches and provide their output in a meaningful and actionable way to software developers.

I will design a new deep-learning model that is specifically focused on providing useful information to the user, making it more likely for the developer *to act* on the tool's result. This can be done by producing user friendly attention maps [11] that can give the developers some insight on the reason why a specific fix is generated or why a specific code snippet is detected as buggy by the network. Furthermore, text generation models could be trained to generate explanations using existing historical data such as bug reports or commit messages. These explanation could provide clues to the developer to help him understand the output of the model. This project would be complemented by user study to understand how such additional elements (attention map or generated explanation) help developers in their task to find, understand, and fix bugs.

Another way to make these tools more actionable is to give more space to the developer in these new techniques. I would do that by first doing some user studies to better understand if developers find current APR tools useful in practice, then I would propose a more developer-centered approach to APR, by introducing a developer-feedback loop to the test generation. For example, a developer could choose "key" tokens or variable that the correct patch must include

in order to guide the patch generation, helping DL techniques to fixed more bug with additional developer feedback.

Better software repository mining and data cleaning for better automatic software engineering: In the last few years, the amount of software data increased significantly (GitHub reached 1.1 billion contributions in 2018 [21]). This is extremely beneficial for automatic software engineering since we can use this data to train more powerful models. While such data is available, there are many challenges to mine, clean and analyze such large amounts of data for software engineering. Existing work [2, 1] generally relies only on a small subset (between tens of thousands to one million instances) of available data which is orders of magnitude below the amount of data used in other domains (e.g., the CommonCrawl dataset used in NLP contains 380M English sentences and 8.894 billion words [5]). In addition, SE datasets are very noisy and not much effort is done to automatically clean them. Either existing work uses simple automatic approaches such as keyword search in commit messages [23, 19, 20] or an extremely expensive manual process [12]. The first part of this project is to significantly improve the quality and the quantity of data mined from GitHub by leveraging advanced algorithms (e.g., PageRank) on the many characteristics and relationships between commits (authors, projects, dates, associated issues, etc.). Leveraging such information would allow mining a large amount of data while reducing the noise from data extracted from GitHub. The second part of the project would consist of using the extracted data (up to hundreds of millions of instances) on existing machine-learning-based SE tasks to measure the influence of (1) data quality and (2) data quantity at a scale that has not been studied before.

3.2 Software Engineering Approach for more Reliable AI

AI systems are difficult to test. Yet, more and more projects integrate AI models. While many techniques exist for testing software, they cannot directly be applied to test AI systems. As a result, it is difficult to assert the reliability of AI systems, even if they are used in critical environments. While some work has been done in applying simple code coverage techniques or adversarial learning, not much has been done. This research direction aims at developing new approaches to ensure the reliability and dependability of AI systems.

Automatic Testing of AI Systems Automatically testing AI systems is challenging because the expected output is not always known. There has been a lot of work on testing and fuzzing deep learning model inputs (e.g. adversarial learning) but not much work on automatically testing the source code that is used to build deep learning models. Our recent work [13] was the first to focus on testing the source code of deep learning libraries and the results were promising. We can continue this line of work with several projects. Finding bugs during the training process is a natural follow up. Compared to our recent work, this has the additional challenge that training is a non-deterministic process, hence we would need to quantify the variations (e.g., using variance or developing a more suitable metrics) to know which variations reveal bugs in the training process. Second, recent deep learning library (TensorFlow 2.0 and Pytorch) act more and more like compilers (The glow compiler was developed for optimizing Pytorch neural networks). In compilers, such optimizations can contain many bugs [22] and need to be extensively tested. The goal of this project is to develop a fuzzing approach that automatically generates deep learning architectures that should be correct to test such optimizers. The main challenge would be to extract constraints from source code regarding the compatibility between the different layers of the deep learning architecture.

References

[1] A. Agrawal and T. Menzies. Is better data better than better data miners?: on the benefits of tuning smote for defect prediction. In *Proceedings of the 40th International Conference on Software engineering*, pages 1050–1061. ACM, 2018.

[2] 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 Transactions on Software Engineering*, 2019.

[3] Z. Hu, X. Ma, Z. Liu, E. H. Hovy, and E. P. Xing. Harnessing deep neural networks with logic rules. In *ACL (1)*, 2016.

[4] N. Jiang, T. Lutellier, and L. Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021.

[5] P. Koehn, F. Guzmán, V. Chaudhary, and J. Pino. Findings of the wmt 2019 shared task on parallel corpus filtering for low-resource conditions. In *Proceedings of the Fourth Conference on Machine Translation (Volume 3: Shared Task Papers, Day 2)*, pages 54–72, 2019.

[6] H. Krasner. The cost of poor quality software in the us: A 2018 report. 2018.

[7] T. Kuchta, T. Lutellier, E. Wong, L. Tan, and C. Cadar. On the correctness of electronic documents: studying, finding, and localizing inconsistency bugs in pdf readers and files. *Empirical Software Engineering*, 23(6):3187–3220, 2018.

[8] T. Li and V. Srikumar. Augmenting neural networks with first-order logic. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

[9] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger. Comparing software architecture recovery techniques using accurate dependencies. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 69–78. IEEE, 2015.

[10] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, and R. Kroeger. Measuring the impact of code dependencies on software architecture recovery techniques. *IEEE Transactions on Software Engineering*, 44(2):159–181, 2017.

[11] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114, 2020.

[12] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017.

[13] H. V. Pham, T. Lutellier, W. Qi, and L. Tan. Cradle: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1027–1038. IEEE, 2019.

[14] H. V. Pham, S. Qian, J. Wang, T. Lutellier, J. Rosenthal, L. Tan, Y. Yu, and N. Nagappan. Problems and opportunities in training deep learning software systems: an analysis of variance. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 771–783, 2020.

[15] S. Qian, H. Pham, T. Lutellier, T. Hu, J. Kim, L. Tan, Y. Yu, J. Chen, and S. Shah. Are my deep learning systems fair? an empirical study of fixed-seed training. In *Under submission at NeurIPS 2021*, 2021.

[16] H. Shahriar, K. Weldemariam, T. Lutellier, and M. Zulkernine. A model-based detection of vulnerable and malicious browser extensions. In *2013 IEEE 7th International Conference on Software Security and Reliability*, pages 198–207. IEEE, 2013.

[17] H. Shahriar, K. Weldemariam, M. Zulkernine, and T. Lutellier. Effective detection of vulnerable and malicious browser extensions. *Computers & Security*, 47:66–84, 2014.

[18] J. Wang, T. Lutellier, S. Qian, H. Pham, and L. Tan. Eagle: Creating equivalent graphs to test deep learning libraries. In *under submission at ICSE 2022*. IEEE, 2022.

[19] S. Wang, T. Liu, J. Nam, and L. Tan. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, 2018.

[20] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016.

[21] J. Warner. Thank you for 100 million repositories. In *https://github.blog/2018-11-08-100m-repos/*. GitHub, 2018.

[22] Q. Zhang, C. Sun, and Z. Su. Skeletal program enumeration for rigorous compiler testing. In *ACM SIGPLAN Notices*, volume 52, pages 347–361. ACM, 2017.

[23] H. Zhong and Z. Su. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 913–923. IEEE Press, 2015.